UNIVERSITY OF SOUTHAMPTON

FACULTY OF PHYSICAL SCIENCES AND ENGINEERING

Electronics and Computer Science

User-Experience-Aware System Optimisation for Mobile Systems

by

Alexander S. Bischoff

Thesis for the degree of Doctor of Philosophy

January 2016

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF PHYSICAL SCIENCES AND ENGINEERING
Electronics and Computer Science

Doctor of Philosophy

USER-EXPERIENCE-AWARE SYSTEM OPTIMISATION FOR MOBILE SYSTEMS

by Alexander S. Bischoff

This thesis considers the concept of Quality of Experience (QoE) in the context of mobile electronic consumer devices, such as smartphones. The modern smartphone is expected to deliver a high level of user experience across a wide variety of tasks, whilst remaining as power efficient as possible. Commonly, mobile devices undergo runtime optimisation to achieve the required level of performance, with the energy consumption being a secondary concern. In this thesis, we stress that it is vital to not focus on the raw performance of the device, but instead to concentrate on the needs and desires of the end user. This approach ensures that the end-user is satisfied at all times, and that the power consumption for a given level of user experience is minimised. Hence, we advocate user-experience-aware system optimisation.

We introduce the concept of Quality of Experience, which has traditionally been used only in the telecommunications industry, to mobile system optimisation. We develop user experience models in the form of utility functions, and use these to translate low-level metrics into the delivered user experience. Upon these models we build simple, yet effective, QoE-aware Central Processing Unit (CPU) and Graphics Processing Unit (GPU) governing algorithms which adjust the performance and power consumption at runtime to meet user experience requirements. When creating our algorithms, we first analyse and characterise the operation of both CPU and GPU workloads. Specifically, we investigate how the level of compute-boundedness or memory-boundedness of CPU workloads affects frequency scalability, as well as determining how the available bandwidth and core count for a GPU affects the rendering performance. We combine both gem5-based simulation driven analysis and hardware-based verification in order to validate our QoE-aware governing algorithms. Additionally, we validate the operation of our algorithms using a variety of common mobile workloads. As part of this work, we have also extended the gem5 simulator to allow use to investigate the potential for fine-grained Dynamic Voltage and Frequency Scaling (DVFS) adjustment, and use this as a platform to investigate the operation of the Linux CPUFreq governors used on modern mobile platforms.

Contents

List of Figures

List of Tables

Declaration of Authorship

I, Alexander S. Bischoff , declare that the thesis entitled *User-Experience-Aware System Optimisation for Mobile Systems* and the work presented in the thesis are both my own, and have been generated by me as the result of my own original research. I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- parts of this work have been published as: Bischoff et al. (2013)

Signed:...

Date:..

Acknowledgements

During the course of this research, 3.5 years were spent full time at ARM Ltd. in Cambridge, under the supervision of Dr. Andreas Hansson in the Research and Development department. First, and foremost, the author would like to express his thanks to Andreas Hansson, who provided expert industrial guidance throughout the entirety of the placement period. A significant proportion of the ideas and results expressed in this dissertation were the direct results of discussions with Andreas Hansson, and the overall quality and direction of this work would have been drastically different without his valued advice. The author would also like to thank numerous other colleagues and friends at ARM, who have not only contributed ideas and partaken in discussions, but have also provided moral support during harder periods. These include, in no particular order, René de Jong, Dan Nevill, Roxana Ruşitoru, Stephan Diestelhorst, Radhika Jagtap and Andreas Sandberg, amongst others who remain unnamed for the sake of brevity.

Second of all, the author would like to thank the University of Southampton, and his supervisor, Prof. Bashir M. Al-Hashimi for his advice and the opportunity to study for a PhD. The author is grateful to Bashir M. Al-Hashimi for finding the time to discuss the work amidst an ever-increasing workload at the university. Additionally, the author wishes to express his thanks to Dr Geoff V. Merrett, who took over some of the supervisory duties during busier times. Finally, the author would like to thank Dr. Rob Maunder who provided feedback on the intermediate reports completed during the course of this research, and hence helped to steer the direction of the research.

The author would also like to thank his family and friends, who provided support and re-assurance during the most difficult of times. It was only with the motivation that his family provided that the author was able to undertake the difficult task of completing a PhD. Thank you for the moral support given over the past four years.

As a final note, the work contained in this dissertation has been completed as part of a collaboration between the University of Southampton and ARM Ltd. The work has been funded by both the EPSRC, in the form of CASE award, as well as by ARM, without which it would not have been possible to complete.

Nomenclature

ADB	Android Debug Bridge
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
CPI	Cycles Per Instruction
CPU	Central Processing Unit
DPM	Dynamic Power Management
DVFS	Dynamic Voltage and Frequency Scaling
ESQoS	End-to-end Service Quality of Service
FIFO	First In, First Out
FLOPS	Floating-point Operations Per Second
FPS	Frames Per Second
FR-FCFS	First-Ready First-Come-First-Served
FS	Full System
GPU	Graphics Processing Unit
HMP	Heterogeneous Multi-Processing
IO	Input/Output
IP	Intellectual Property
IPC	Instructions Per Cycle
ISA	Instruction Set Architecture
ISP	Image Signal Processor
ITT	Inter-Transaction Time
LIFO	Last In, First Out
LRG	Least Recently Granted
MOESI	Modified Owned Exclusive Shared Invalid
MOS	Mean Opinion Score
NIC	Network Interface Controller
NoC	Network on Chip
OS	Operating System
PESQ	Perceptual Evaluation of Speech Quality
PMU	Performance Monitoring Unit
QoE	Quality of Experience
QoP	Quality of Perception

QoS	Quality of Service
RTL	Register-Transfer Level
SE	System-call Emulation
SLA	Service-Level Agreement
SQoS	System Quality of Service
US	User Interface
VPU	Video Processing Unit

Chapter 1

Introduction

Mobile phones and tablets have become ubiquitous in the modern society with 66% of UK adults owning a smartphone (Ofcom, 2015), and 93% of UK adults owning a mobile phone. As the adoption rate of smartphones and tablets increases, users rely on these devices for more and more of their daily computing needs, shifting away from bulky desktop computers and laptops. The modern smartphone offers many of the same capabilities as a laptop computer, and can be used for standard calls, video calls, to browse the Internet, taking photos and videos and even playing games. Due to this wide variety of use cases, modern mobile devices are complex devices comprised of many interoperating components coordinated by a full-fledged operating system and applications. It is the function of the software to ensure that all components work together to provide the desired service to the end user. Furthermore, it is vital that the user experience of such services is as high as possible.

User experience is inherently a subjective metric, which makes it extremely difficult to measure without explicitly asking a user how good a device or service is performing in their eyes. User experience is affected by the limits of human perception, as well as the preconceptions of the individual. For example, sufficiently short delays in response to a user-triggered action are perceived as instantaneous (Nielsen, 2009), and therefore deliver the best user experience. However, longer delays result in a reduction in user experience, with increasing delay time resulting in greater dissatisfaction. Additionally, the level of user disruption if affected by the conditioning of the end user (Fiedler et al., 2010), and those exposed to longer delays can often be more tolerant of these. The subjective nature of user experience makes this a hard metric to measure and act upon at run time. Therefore, a typical assumption is that higher performance results in higher levels of user experience. This drives vendors to strive to deliver the highest possible level of performance in an attempt to provide the best user experience possible.

Maximising the performance of mobile devices has a large impact on the energy consumption, reducing the device lifespan and increasing the device temperature. User experience is not only affected by the performance of the device, but also by the time between charges and the surface temperature. Hence, aiming to ensure high user experience by delivering the highest possible performance is a flawed approach, and in fact results in decreased user experience under the wrong circumstances. Hence, it is important to understand the level of user experience delivered by the mobile device, and to use this information when governing the performance of the system. We consider the short-term user experience and aim to maximise this whilst reducing energy consumption. Whilst we do not consider the long-term impacts of our decisions, our work forms the initial first steps towards user experience aware system optimisation.

A typical mobile system is comprised of many different components, which all compete for shared resources, such as the memory system and the energy stored in the battery. It is vital that background activities do not noticeably affect the operation of foreground tasks, and therefore components and tasks must be prioritised accordingly. These priorities control the level of service received by each component by the memory system, and which components are allowed to consume more of the precious energy stored in the battery. The allocation and management of these resources is known as Quality of Service (QoS). A typical QoS implementation consists of a set of hardware mechanism which can be used to trade the level of service received by each component dynamically at run time, based on either pre-defined static policies, or more advanced run-time management. QoS sets the order in which components fail to receive their required level of service in the event that not all components can be satisfied. It is imperative that the QoS strategies are correctly set up to ensure sufficient device performance and high user experience.

Commonly, it is possible to adjust the performance and energy consumption of components, such as the CPU, at run time. This often takes the form of Dynamic Voltage and Frequency Scaling (DVFS), where the frequency and corresponding voltage supplied to a component are adjusted based on demand for high performance or to reduce the energy consumption. This mechanism allows the system to trade efficient, low performance modes of operation for inefficient, but high performance modes of operation based on the workload and user demands. By providing a range of different frequencies, it is possible to choose a configuration that delivers just-enough performance, whilst consuming the least amount of energy whilst doing so. However, picking the correct frequency is a complex operation, which is affected by the capabilities of the hardware, the applications running and the level of user experience demanded by the end user.

All components in a mobile system must work together to ensure that the system operates as intended, and that the end-user is satisfied. Two of the most important system components are the CPU, which run the operating system as well as a large portion of most workloads, and the GPU, which is used to offload rendering from the CPU. Should the CPU underperform, then applications run more slowly, and the system becomes less responsive. In the case of the GPU, if the frame rate at which the GPU renders is too low, then the user will see a set of static images in the place of smooth motion. Both of these strongly affect the delivered user experience. The CPU and GPU are also large consumers of system energy (Carroll and Heiser, 2010). For example, Carroll et al. show that during video playback, aside from the display, the CPU and GPU consume the highest power in the system. Therefore, we focus our analysis on these two vital components, and investigate their operation in both standalone configurations, as well as in a complete system context.

In this thesis, we present generalised user experience models, which relate the performance of the device to the quality of the user experience. These models present the user experience in terms of utility, i.e., on a scale of 0 to 1, which allows multiple models to be combined to determine the system- level user experience. We relate the user experience to low-level metrics which can be used to predict the user experience at run time. Our user experience models are used as part of Quality of Experience mechanisms, which are a counterpart to the traditional Quality of Service mechanisms used in modern devices. We combine our user experience models with studies regarding the efficiency of the CPU and GPU to allow us to adjust the system-level performance at run time, whilst preserving the level of user experience. This results in decreased energy consumption, and longer device run time between charges. We analyse the efficiency of the CPU and GPU in isolation and in a system context, for a range of workloads. We compare our results to those achieved by using typical governing strategies found in modern devices. Additionally, we combine simulation-based approaches with high observability with studies focusing on real hardware.

Throughout this work, we present results gathered from simulation of, or execution on, ARM architecture based hardware. Whilst we have only explored these optimisation strategies for the ARM architecture, they are largely architecture agnostic and can be applied to a wide range of systems and system architectures.

1.1 Research Statement

The research detailed in this work, and the future work in detailed in Chapter 7, is working towards the following goal:

> Link **Quality of Service** and **Quality of Experience** to provide **automated run-time control** of the system configuration in order to achieve a **desired, user-oriented, level of performance** whilst **minimising energy usage** for commonly used mobile applications.

1.2 Research Questions

This work answers the following research questions:

1. Can a quantitative relationship between low-level metrics and user experience be defined which permits realistic estimation of user experience at run time?
2. From a low-level perspective, which level of service - in terms of latency and bandwidth - do typical components such as CPUs and GPUs require from the shared memory system?
3. How does CPU frequency affect performance and user experience for a spectrum of workload scenarios, ranging from compute-bound to memory-bound?
4. Across a range of typical graphics workloads, what effect does the scaling of GPU core count and available memory bandwidth have in terms of performance and user experience?
5. Making various assumptions about user preferences - which we will term user experience models - how can CPUs, GPUs as well as complete systems be run-time optimised in such a way that the end-user remains satisfied?

1.3 Experimental Setup

Throughout this work we used a combination of experiments run in simulation and ones run on real hardware. Simulation allows us to explore system modifications that are not possible with real hardware, as well as providing greater levels of observability. Additionally, simulation allows a large number of design choices to be explored in parallel. Real hardware, on the other hand, gives much more realistic results than simulation, and therefore is employed to validate our QoE-Aware CPU, GPU and advanced system-level governors. These are covered in detail in Sections 5.2.1, 6.4 and 6.6, respectively.

1.3.1 Full-System Simulation

There are a vast number of full-system simulators which allow the user to boot an operating system and evaluate system performance. However, each simulator is different and is designed to explore different areas of CPU hardware and software design. These simulators can range from high-performance software simulators which are used for early software development and debugging (Romdan, 2008) to highly-accurate simulators designed to investigate architectural and micro-architectural modifications (Binkert et al., 2011). Due to these differences, it is vital to choose the correct simulator for the task at hand.

In this work we investigate both hardware and software changes, and consider low-level traffic as well as higher level user experience. It is therefore crucial that the simulator is highly-accurate when simulating, and is able to respond correctly when, for example, the CPU frequency is changed as part of the DVFS mechanism. This requires realistic CPU models, memory system models and high-quality models of other system components required to run real workloads. The system must also be representative from a software point of view, and must run unmodified operating systems and applications. This ensures the most realistic set of results which are representative of real hardware and allow general conclusions to be drawn.

A simulator which meets all of these criteria is gem5 (Binkert et al., 2011), and we choose this as our full-system simulator in this work. The gem5 simulator is able to simulate a number of different ISAs including ARM and x86. It comes with detailed CPU models, which can be tuned to more closely represent real commercial CPUs, as well as a detailed memory system and other components. It supports both full-system and system-call-emulation modes, the latter of which allows applications to be run in isolation without an OS.

The gem5 simulator has been proven as a realistic full-system simulator by Butko et al. (2012). The work by Butko et al. focuses on comparing the performance of a real ARM Cortex A9 with the performance resulting from a corresponding gem5 simulation. Their results show that the performance estimations from gem5 correlate will with real hardware, with the exception of memory intensive workloads. This was attributed to the lack of a DRAM controller model, which has since been added to the simulator by Hansson et al. (2014), thereby solving this issue. DVFS support has been added to the simulator by Spiliopoulos et al. (2013), and fully supports the Linux *CPUFreq* framework, which allows the operating system to control the voltage and frequency of different in-system components.

The gem5 simulator has been used for a large volume of research in the computer architecture field. This research ranges from characterising smartphone applications such as the web browser (Gutierrez et al., 2011) to QoS for smartphone NoCs (Feng, 2012). gem5 has also been integrated with other simulators such as DRAMsim2 (Rosenfeld et al., 2011) by Wang et al. (2013). The simulator has been widely accepted by the community and has been empirically proven as there have been over 140 publications to date (Gem5.org, 2015).

An alternative ARM-ISA compliant simulator is ARM FastModels (ARM, 2014), which offers very fast simulation in the region of 100 to 500 MIPS (Romdan, 2008), but it is only a functional model of the hardware and does not make any guarantees on the accuracy for architectural or performance exploration. ARM FastModels is designed to assist in the development of software prior to the hardware platform being available, as well as providing debug facilities not present in the hardware itself. However, due to the lack of performance guarantees, it is unsuitable for our purposes.

Simics (Magnusson et al., 2002) is a full-system simulator which is designed to allow fast simulation of binaries without modification. However, the models used in the simulator are not as advanced or as accurate at those used in gem5 as they are designed for software development. Therefore, Simics is not suitable for our purposes. SimpleScalar (Austin et al., 2002) is a system simulator which provides detailed architectural models of many components found in a typical system which are cycle accurate. However, the models and simulator are no longer maintained and are therefore are not as relevant as those found in gem5, so again this simulator is not suitable.

1.3.1.1 Simulator Configuration

As discussed above, for system simulations we use the gem5 full-system simulator. This detailed simulator is capable of simulating different CPU models, cache hierarchies and memory system types, as well as running whole operating systems, driver stacks and workloads with little to no modifications. The simulator allows full observability of the simulated system, and can be extended to add additional hardware models, adjust the operation of those already present or increase visibility for a specific part of the system. We use gem5 to both profile the operation of different workloads, as well as for fine-grained analysis of their operation. Additionally, we use this simulator to test the operation of governing algorithms without the need to modify the software stack of the guest operating system. Unless stated otherwise, we run gem5 in a single core configuration with an ARM Cortex A15-like CPU model. For our Android simulations we simulate a system with 2 GB of DDR3 memory. We run Android 4.4.4 with a 3.14 Linux kernel based on the linux-linaro-tracking git repository (Linaro, 2015). We have extended gem5 using a proprietary GPU model which allows us to measure the impact of our decisions on GPU performance. We measure and profile GPU performance using this model in Section 6.1.

1.3.1.2 GPU Modelling

In addition to a sophisticated full-system simulator with a detailed memory system and CPU models, it is important to have a realistic GPU model which produces real traffic and therefore has accurate effects on the simulated memory system. The time taken to render GPU frames must scale with the complexity of the frame as this allows the GPU performance to be measured. In addition, the GPU model itself must be configurable such that the frequency and number of GPU cores affects the time to render the frame and the amount of bandwidth requested from the memory.

The number of currently available GPU simulators is limited, and many are focused on evaluating the performance of a workload on a specific GPU architecture, and fail to take into account the rest of the system. GPGPU-sim (2012) is a GPU simulator which is designed to provide a realistic platform to investigate the performance of different GPU architectures and the software running on these. GPGPU-Sim is designed to explore the performance of GPU architectures when executing CUDA (Nvidia, 2008) or OpenCL (Stone et al., 2010), and has not been designed to deal with graphics benchmarks, unlike the GPU model used in this work. GPGPU-sim has been used by Bakhoda et al. (2009) to investigate why some CUDA applications do not achieve the expected performance when run on a GPU as opposed to a CPU, i.e., why they fail to scale as the number of concurrent threads is increased. GPGPU-sim has been integrated with PTLsim (Yourst, 2007), an x86 simulator, by Zakharenko (2012).

In contrast, our study uses a cycle-approximate proprietary GPU model which not only produces realistic traffic patterns, but also dynamically responds to memory system changes. Our work also considers the trade-offs between available GPU bandwidth and overall user experience for a CPU-GPU system. In addition to this model, we implement a detailed GPU trace player which is able to reproduce the traffic characteristics of the more detailed GPU model to a high degree, whilst reducing the simulation overhead. We describe the GPU trace player in Section 6.1. Then enables us to complete longer simulations that would otherwise be possible. Our GPU model is able to replay GPU traces from mobile workloads, and does not just focus on CUDA or OpenCL. Therefore, it is able to realistically represent the types of traffic found in a typical modern mobile system.

Hong and Kim (2010) present us with a power and performance model for a GPU which illustrates that once a GPU saturates the maximum bandwidth that the memory system can supply, then it is pointless to increase the number of cores in the GPU as performance does not increase, and sometimes worsens. By limiting the number of running GPU cores, the paper shows that it is possible to achieve the highest possible GPU performance for a particular application, whilst saving power by shutting down unused cores. It is also shown that some GPU applications scale non-linearly as the number of cores is increased, and therefore have a lower optimal operating point. Non linear scaling has been observed as part of this work (see Chapter 4), as some frames do not scale with the number of cores due to the high bandwidth requirements. Our work expands on this as it also investigates CPU performance penalties due to contention from high volume GPU traffic.

In the situation that the memory is not capable of satisfying the large bandwidth requirements for the GPU, Wang (2011) demonstrate that fewer processing cores must be used. This demonstrates for various GPU applications using GPGPU-sim, and they suggest that the number of active GPU threads must be chosen based on per-thread

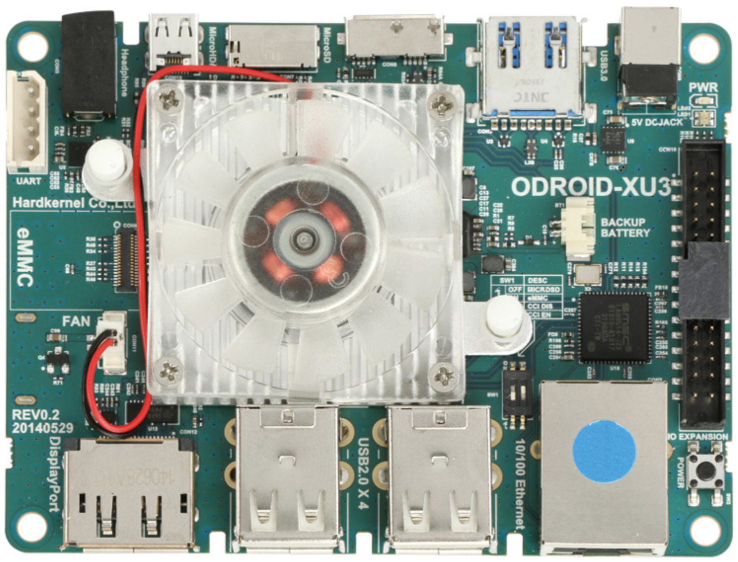


Figure 1.1: Odroid XU3 by Hardkernel. Image from Pollin.de (2015)

resource requirements. This ties in with our work in Chapter 6 in which we investigate how the GPU requirements change as the number of active threads is altered.

1.3.2 Real Hardware

When running experiments on real hardware, we use the Hardkernel Odroid XU3 (Hardkernel, 2015b) development board, which is shown in Figure 1.1. This platform is based on the Samsung Exynos 5422, which contains two clusters of four CPU cores in a Big.LITTLE configuration. Specifically, it includes four ARM Cortex A7 cores, and four ARM Cortex A15 cores. Additionally, the development board includes an ARM Mali T628 GPU and 2 GB of LPDDR3 memory. This allows us to investigate the operation of different CPU types, as well as profile the performance of the memory system and GPU under different configurations. Finally, the board supports the use of hardware performance counters which allow us to monitor the operation of various components without significantly perturbing the operation of the system. We run Android 4.4.2 on the XU3, and use a modified 3.10 Linux kernel based on the kernel supplied by Hardkernel specifically for the Odroid XU3 (Hardkernel, 2015a). We use the Odroid XU3 extensively in Chapters 5 and 6 to validate results gathered through simulation on real hardware.

1.3.3 Workloads

Throughout this work we use a set of workloads to both stress and evaluate the performance of system components and the system as a whole. These workloads are used to determine when the system is operating efficiently, as well as investigating why this is the case. We run a set of workloads which allow us to test various aspects of and components in the system. For this reason, we use workloads which stress the CPU, the GPU, the memory system and the system as a whole. Each of the CPU workloads is analysed in detail in Chapter 4, and therefore we only provide a high-level overview of each workload here. Similarly, the GPU workloads are analysed in Chapters 5 and 6.

Our workloads have been chosen so that they cover a wide range of use cases. We include both artificial workloads, as well as realistic workloads which are representative of what a real end-user would run on a mobile device. Whilst some of our workloads are artificial, they have been chosen to exercise the corner cases, such as extremely compute-bound or memory bound workloads, i.e., Dhrystone (Weicker, 1989) and Memcpy (ARM, 2015b), respectively. These represent the two extremes and assist us in both determining where those extremes lie, as well as validating our experimental strategies. We also run a number of benchmarks which lie in between these two extremes, such as BBench (Gutierrez et al., 2011), AnTuTu (Antutu, 2015) and RLBench (RedLicense Labs, 2012). Whilst these are benchmarks, they force the system to perform similar action to those one would see when a user interacts with the device. BBench, a web browser workload, is an excellent example of this. We also run a set of graphics benchmarks, such as The Chase (Unity, 2015), which strongly exercise the GPU, as well as stressing the rest of the system. The Chase is a demo of a rendering engine, and therefore is a good example of what one would find it a typical mobile game, minus the direct user control. Therefore, the user experience delivered when running The Chase is representative of that when the user is playing a 3D game. We provide additional details about the workloads in the following sections.

1.3.3.1 Compute-Bound Workloads

Throughout this work we use two workloads which are close to being purely compute-bound. These workloads are Dhrystone (Weicker, 1989) and AndEBench (Levy, 2012). These workloads scale close to linearly with the CPU frequency, and therefore can be used to ensure that governing algorithms are operating correctly. For example, due to the close-to-linear scaling, a CPU governor must run the CPU at the highest frequency.

Dhrystone is a cross platform CPU benchmark, which measures the CPU integer performance. A Dhrystone score is presented in terms of DMIPS, or Dhrystone-MIPS, and measures the number of Dhrystone operations which could be completed each second.

AndEBench in an Android-only benchmark, which measures the native and Java performance for both single- and multi-core systems. The workload is very compute-bound for both the native and the Java execution modes, and therefore provides an Android-specific measure of the CPU performance.

1.3.3.2 Memory-Bound Workloads

In the previous section, we briefly discussed two compute-bound workloads. These workloads scale linearly with the CPU frequency, and therefore benefit from a higher frequency. In this section we discuss a memory-bound workload, memcpy (ARM, 2015b), which does not scale significantly with the CPU frequency, and instead achieves a similar level of performance irrespective of the CPU frequency. Memcpy measures the time taken to copy a portion of memory to another location in the memory. Due to the relatively high memory access latencies, this workload does not scale with frequency, and purely measures the memory bandwidth. For energy efficiency reasons, this workload must be run at a low frequency.

1.3.3.3 Graphics Workloads

In this work, we also use two graphics workloads. These workloads rely on on the complete system as they make use of the CPU, GPU and memory system. Specifically, if any of these components is under-performing, then the workload suffers as a result. However, these workloads rely primarily on the GPU performance, and are hence presented as such. For graphics workloads, it is imperative that the achieved number of frames rendered each second is sufficiently high to appear smooth to the end user. Therefore, for these workloads, we can measure the FPS in order to gauge the performance of the workload.

The first workload we use is The Chase (Unity, 2015), a demo for the Unity rendering engine. This workload is a looping demo designed to show off the features of the Unity rendering engine. As the workload loops, and always runs through the same sequence, it allows different energy-saving strategies to be compared directly as the workload remains the same. This workload is run for a fixed period of time, and the number of frames executed is compared to get a score for the benchmark.

The second graphics workload we use is Transporter (Geomerics, 2015b), which is a rendering demo for the Geomerics Enlighten (Geomerics, 2015a) rendering engine. This engine is able to compute the lighting for a scene in real time on the CPU, and then renders the scene on the GPU. Therefore, the workload has both a well-define CPU-bound portion and a GPU-bound portion. Hence, both CPU and GPU performance must be sufficiently high to produce an acceptable level of performance from the workload. As with The Chase, we measure the achieved number of frames rendered over a fixed period of time to score the workload performance.

1.3.3.4 Complete System Workloads

Finally, we cover the workloads which we believe assess the complete system performance. These workloads require high levels of performance from each of the system components, and therefore will highlight when parts of the system are not performing adequately.

The first workload we discuss is BBench (Gutierrez et al., 2011), a browser based benchmark. BBench loads a subset of the most visited websites using an operating system's native browser. BBench measures the system performance by rendering a set of the most commonly loaded web pages using the native system browser. For each page, the page is loaded, and the benchmark scrolls to the bottom of each page in fixed-size steps at constant rate. The time taken to load and scroll through each page is measured and a geometric mean time across all pages is calculated and presented as the overall benchmark score. In general, BBench is run for multiple iterations, and the geometric mean is calculated across all warm runs, i.e., excluding the first time each page is loaded and scrolled.

The second complete-system benchmark we use is Antutu (Antutu, 2015), which measures the CPU, GPU, memory and IO performance of a device. Antutu consists of multiple smaller benchmarks which measure the performance of each aforementioned system component in turn. Antutu presents a score for each sub-benchmark, as well as an overall score which is the sum of all sub-benchmark scores. This allows the benchmark to both give a high level evaluation of the device, and to allow the user to compare the different sub-benchmarks to gain an understanding of which parts of the system are under-performing. As the benchmarks profile different parts of the system, there are phases where it is compute-bound, phases where it is memory-bound and phases where it is neither. Therefore, the workload provides an interesting use case to evaluate different governing strategies.

Finally, we use RLBench (RedLicense Labs, 2012) to measure the system performance. SQLite performance is important when running the Android operating system, as it heavily relies on SQLite databases behind the scenes. Studies have linked the SQLite performance of an Android system to the user experience delivered by it (Kim et al., 2012). RLBench measures the SQLite database performance. In order to do this, it assesses the time for different SQLite transactions to take place. These transactions involve adding and removing data to the database at different granularities, and performing different SQL queries on the database. As was the case with Antutu, this benchmark has phases where it is either compute- or memory-bound, depending on the type of transactions being performed.

1.3.3.5 Automating Workload Execution

We make use of an open source tool called Workload Automation (ARM, 2015a) to run the above mentioned workloads on both the gem5 simulator, and on real hardware. Workload Automation interfaces with a system over the Android Debug Bridge (ADB), which provides an interface to interact with the device by executing commands directly on the device from a secondary machine. Workload Automation uses this interface to set up the device, transfer workloads, execute workloads and extract the workload results from the device. One of the most powerful features of Workload Automation is the ability to set up an agenda, which allows the user to configure a set of workloads and corresponding system settings to execute. Therefore, Workload Automation is a powerful tool to run configuration sweeps to determine how the workload respond to different system configurations. We use Workload Automation extensively throughout this work. In addition to using Workload Automation with real hardware, we also use workload automation with gem5. This allows us to run Android workload in gem5, without manually setting up the simulated disk image.

1.4 Thesis Outline

In the following chapters, the content is organised as follows. Figure 1.2 shows the layout of this thesis. In Chapter 2 we present the related literature, and contrast this to our own work. Here, we present the motivation that has led to this work, as well as providing a succinct summary of our research questions and goals. In this section we also discuss both the gem5 simulation framework, which is used heavily throughout in order to analyse both workload behaviour and to test dynamic optimisation strategies, as well as the workloads used to analyse device performance.

Chapter 3 describes the concepts behind measuring the user experience delivered by a device. We consider the different types of user experience that contribute to the overall experience delivered by a device, and provide models for a subset of these. These models allow us to relate measurable, low-level system metrics to user experience scores, which are used for run time system optimisation. Hence, these models form the basis for many of the optimisation strategies presented in the subsequent chapters.

After establishing the different types of user experience, and how to determine the level of experience delivered at run time, we move onto CPU efficiency in Chapter 4. This chapter considers the operation of the CPU under various constraints, and presents the conditions under which a CPU is inefficient. We consider a set of workloads, and

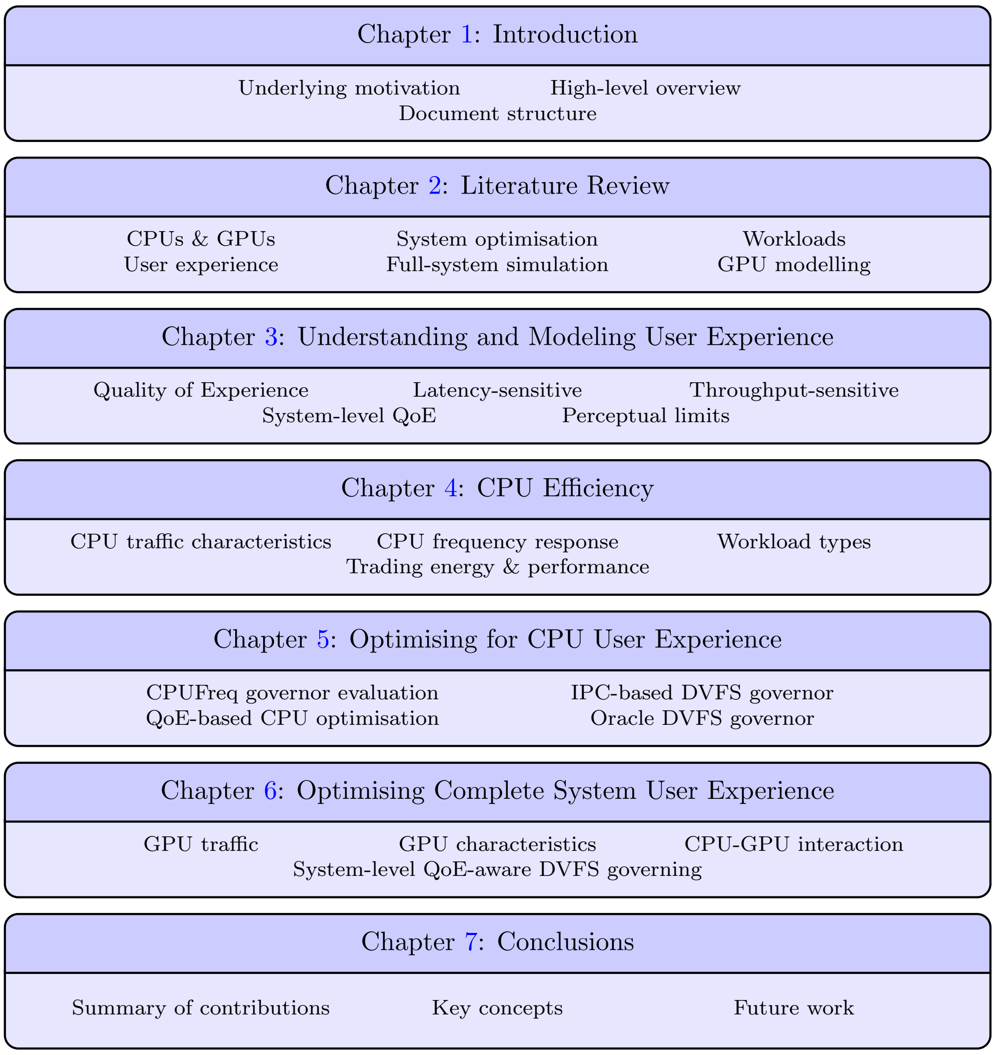


Figure 1.2: Thesis outline

demonstrate how the performance and energy consumption of these changes as the CPU frequency is adjusted in both simulation and on real hardware. We use this information to classify different types of workload, and describe the strategies that must be employed at run time to ensure that the CPU is able to operate efficiently. Finally, we present the optimisation space for set of applications that are compute-bound, memory-bound and graphically intensive, and demonstrate that it is possible to reach the same performance point at a range of different energy consumption levels when performing DVFS. This demonstrates that a dynamic frequency governor must strive to make good decisions in order to achieve the desired level of performance at as low energy consumption as possible.

Chapter 5 is concerned with dynamic run time optimisation for the CPU, and builds upon the previous two chapters. We begin by analysing the decisions made by typical DVFS governors found on modern mobile devices, and compare these to a simple Oracle governor. We follow this by presenting a more advanced CPU governor which considers more than the CPU load when making DVFS decisions, and instead focuses on utilising IPC as measured from hardware performance counters to make run time decisions. We profile a set of applications to determine the relationship between the workload performance, user experience and the measured IPC, and use this information at run time to make DVFS decisions. We demonstrate the ability to achieve lower energy consumption than both the Ondemand and Conservative DVFS governors in both simulation and on real hardware.

In our final contribution chapter, Chapter 6, we consider the whole system, rather than focusing on the CPU alone. The chapter begins by demonstrating how a GPU behaves under various constraints. Specifically, we consider how the number of active GPU cores and available memory bandwidth affect the performance of a GPU using a simulation model of the device. We then demonstrate how the additional memory congestion generated by a GPU affects the system performance, user experience and energy consumption, even when running typical CPU-bound workloads. Using this knowledge, we introduce some simple GPU governing strategies which reduce the GPU energy consumption, whilst preserving the user experience. Finally, we extended the IPC based governor presented in Chapter 5 to take into account the strain placed on the memory system, and hence reduce the CPU energy consumption when CPU performance is limited by other in system components. We demonstrate the potential energy savings using real hardware.

Finally, in Chapter 7 we conclude this work, providing a summary of the key contributions found throughout the document. Additionally, we discuss the potential additional work which is able to build upon our findings.

Additionally, we provide two appendices. The first appendix, Appendix A, provides further details on the forking infrastructure developed in order to investigate fine-grained CPU DVFS. This appendix provides both a high-level governing algorithm overview, as well as implementation details. The second appendix, Appendix B, lists additional algorithms and source code used throughout the thesis. These algorithms are listed for completeness, as is the source code.

Chapter 2

Literature Review

In this chapter we present a literature review which covers related research, and contrast it to our work. We aim to highlight the current state of the art for mobile system performance and energy optimisation. We begin by considering user experience, a key component of our work. We discuss the concept of Quality of Experience. Next, we focus on works found in the literature which try to estimate the user experience delivered to the end user. We then provide an overview of strategies which attempt to either optimise for user experience, or ensure that the level of user experience delivered is sufficient. Next, we consider proposed system optimisation techniques found in the literature, which focus on providing high performance and energy efficiency, concurrently, and in general do not consider the user's experience. We concentrate on CPU and GPU optimisation, as these are two key components to which pay particular attention in this thesis.

2.1 Measuring User Experience

We begin the literature review by considering definitions used for Quality of Experience within the industry and academia. Whilst we believe that we are among the first to use the concept of QoE to optimise a mobile or embedded system, the term has been used within the telecommunications industry for a number of years. The International Telecommunications Union (ITU) define the term Quality of Experience as:

> The overall acceptability of an application or service, as perceived subjectively by the end-user.

First of all, it is worth noting that they consider QoE as a subjective measure. This highlights that it is important to understand how a particular user or group of users will rate the operation of a device or service, as opposed to treating it as an objective measure of performance. Whilst this definition of QoE makes it extremely hard to

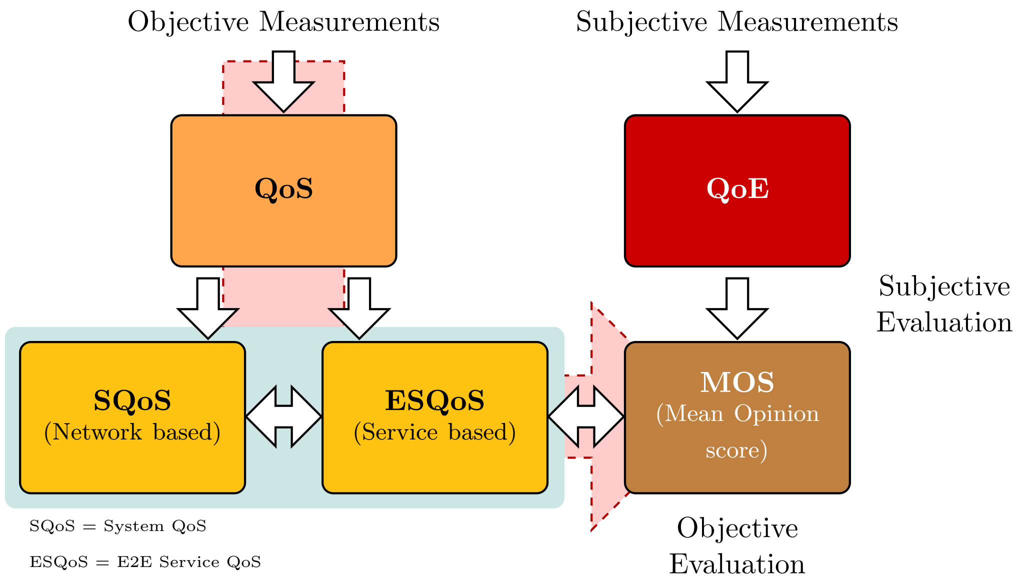


Figure 2.1: Relationship between QoS, SQoS, ESQoS, QoE and MOS as presented by Soldani et at. (Soldani, 2010). Re-drawn from aforementioned publication.

measure and quantify, it is important to understand that it is a term that is based on human perceptions and expectations, rather than an absolute measure.

Lopez et al. (2006) present a QoE-based strategy for adaptive multimedia streaming over IP based networks, i.e., the Internet or similar networks. As part of this, they use the following definition of QoE:

> Quality of experience (QoE) has been defined as an extension of the traditional quality of service (QoS) in the sense that QoE provides information regarding the delivered services from an end-user point of view.

Their definition of QoE is similar to the way we use the term. We consider QoE as an additional layer which builds upon the already existing Quality of Service (QoS) mechanisms, and provides a means to control these based on user experiences, rather then low level Service Level Agreements (SLA). The advantage of using QoE over traditional performance-based optimisation is that it helps the system to determine if the end user will be satisfied with the result, and hence provides scope for far greater energy savings than would otherwise be available to the system.

Soldani (2010) presents a link between QoS and QoE which includes the System Quality of Service (SQoS), and the performance of the delivered services, ESQoS. We present a replication of their QoS and QoE mapping in Figure 2.1. SQoS is defined as how well the interconnect and components are operating in terms of bandwidth and latency, whilst ESQoS is defined as the performance of a service running on the system, such as the time taken to load a web page. The latter is something which has been shown to correlate well with user experience. These metrics are then linked to get a Mean Opinion Score (MOS), which is in the range of 1 to 5, where 1 means bad and 5 means excellent.

In order to evaluate the performance of the system, it is important to be able to gauge the impact in both terms of low level network metrics, as well as higher level user-oriented performance. Fiedler et al. (2010) present a possible model for deriving a quantitative QoE from the lower level QoS metrics. This allows them to provide an estimate for the user satisfaction given the performance of the lower level system components. This model makes the assumption that there is an exponential relationship between the two performance metrics. This is verified with a set of experiments using a metric called Perceptual Evaluation of Speech Quality(PESQ), which estimates how well an end-user perceives the quality of reproduced speech, and web-page loading times. The relationship between QoS and QoE is claimed to have three stages. If there is a mild disturbance in the QoS, then there is no effect on the QoE. As this disturbance increases the user starts to notice the effect on the system performance and becomes increasingly unhappy with the service delivered. Once the QoS disturbance increases to a critical level, the user becomes very dissatisfied with the performance of the system, and potentially gives up.

Ghinea and Thomas (2005) define the term Quality of Perception (QoP) which consists of two terms: the user's ability to assimilate knowledge from a video, QoP_U, and the user's satisfaction with the video as a whole, QoP_S. They evaluate the QoP of 12 video clips for a total of 72 users using 3 different settings for the frame rate and 2 different settings for the colour depth of the video. They observe that the utility of the video clip, i.e. the user's ability to understand the contained information, did not deteriorate significantly when the quality of the video itself was reduced significantly. They did however find that the user satisfaction dropped significantly when both the colour depth and frame rate of the video was reduced, although simply reducing one of the two had limited impact on the satisfaction.

Apteker et al. (1995) undertook a study to determine the relationship between the quality of a video and the acceptability. They define the term *Watchability* which is compared of various aspects of the quality of the video and audio streams, and synchronisation between the two. This term is analogous to the term QoE in terms of video quality. However, it is worth noting that there are many factors, such as frame rate, colour depth and resolution which are fixed at the time that the video is encoded. Other factors which affect the QoS are the successful decoding of the video stream, i.e. blockiness or dropped frames, and audio stream, as well as ensuring that the audio and video remain synchronised.

Shye et al. (2009) logged the user activity on G1 Android phones and used this to determine which hardware components have the largest impact on the battery life of the device. They also analysed the usage patterns of the device to determine how to best reduce the power consumption of the device. After determining that the display and the CPU of the device were the largest energy consumers, they introduced a gradual reduction of screen brightness and adjusted the CPU frequency governor in order to reduce the power consumption in a less noticeable way that just reducing the brightness or frequency in one go. Using this technique they estimate that over 10% of the energy was saved with minimal impact on the user experience as the users stated a preference for the gradual reduction of brightness and frequency. It is worth noting that this approach is deterministic and does not adjust dynamically.

A survey of 255 users by Falaki et al. (2010) demonstrates the diversity of applications and smartphone use cases. The work suggests that providing mechanisms to adjust the device according to the desired user experience. They look at the user behaviour as a mechanism to predict future energy usage, and suggest that it can be used as part of scheduling decisions to reduce the overall power consumption. They advocate the need for run-time, machine-learning based system control which is able to take into account the user behaviour, as well as control the performance of the individual IPs in the system.

There are a number of works which focus on predicting the Mean Opinion Score (MOS) of a device or service by developing a model of human perception. A MOS is a rating in the range of 1 to 5 based on user feedback, where 1 is the lowest rating, and 5 is the highest. For example, the work by Rix et al. (2001b) focuses on developing PESQ, which attempts to evaluate the human opinion of audio quality based on a model of human hearing. This type of model is vital in order to determine how the end user will evaluate the device, system or service as they are able to provide a quantitative representation of the user's opinion of a delivered service.

In this dissertation, we present models for different types of user experience, focusing specifically on user interactions wherein the user must wait for an action to complete, i.e. they are latency sensitive, or ones where the user experience is dependent on the rate of a particular action being completed, i.e. a throughput sensitive task. Our models are heavily based on the work presented by Fiedler et al. (2010) for the latency-sensitive user experience, and Krause et al. (2008) for the throughput sensitive workloads. We detail their work in Sections 3.2.1 and 3.2.2 and hence do not go into significant detail in this section.

2.2 Optimising for User Experience

We now consider approaches presented in the literature which focus on optimising for user experience specifically. Li et al. (2013) utilise user experience to reduce system power consumption. Their work, SmartCap, investigates the impact of the CPU frequency on the user experience. By eliminating the highest DVFS operating point, the authors demonstrate a minimal reduction in perceived performance and a significant energy reduction for certain mobile applications, such as games and the web browser. Their work presents a significant first step to QoE-aware DVFS control, but only investigates a single-step DVFS reduction and does not consider the GPU in the optimisation.

Application Defined Computing is presented by Jagatheesan and Li (2013), which is a concept where the hardware is tuned based on the applications running. Whilst their concept is designed to apply to the system as a whole, they focus on the DRAM. They are able to adjust the DRAM scheduling to increase performance and reduce energy consumption based on application requirements. In addition, they account for user requirements, although their work does not focus on user experience specifically. Their work has not been applied to a complete system.

Techniques to measure QoE in the context of communication rather than system optimisation which is the focus of this paper are presented by Kuipers et al. (2010). They look at the delivered QoE for audio and video, amongst others, and investigate ways to quantify the subjective measure. QoE is applied to mobile broadband in the work by Schatz et al. (2011), and they use this to determine the acceptability data services delivered over a mobile broadband connection. These works demonstrate the importance of understanding how well a particular device or service is performing in a user context.

QoE has been investigated in the context of telecommunications and networks. Collange et al. (2008) present a methodology which allows passive estimation of QoE for large scale networks, and demonstrate their approach with ADSL traffic traces. They use the packet loss for the ADSL traces to try and estimate the user experience. However, as their approach is designed to work with lossy networks, it is less suited for SoC interconnects as these are usually designed not to lose packets.

2.2.1 Utility-Based Optimisation

In this section, we consider approaches that use a form of time-based utility function as part of system optimisation. The essence of a utility function is that it translates a time delay for a particular application or service into utility. Specifically, if a process delivers maximal utility prior to a deadline, and zero utility after the deadline, the utility function will be a step from 1 to 0 with the transition occurring at the deadline. This is commonly employed for real-time systems to determine when different processes should be scheduled to ensure that strict deadline requirements are met. We use a form of utility functions in Chapter 3 when creating our user experience models. These are used to translate time and rates into user experience, and form the core of our optimisation approaches.

As was the case with QoE, utility functions have been employed for large scale networks, and the delivery for multimedia services. A key example is the work by Mu et al. (2008)

which uses network utility functions derived from user studies to adjust network QoS for different multimedia applications. They consider the impact of delay, jitter, packet load and bandwidth limitations on different multimedia applications, including Voice Over IP and video streaming. In our work we consider the impact of delay for latency sensitive workloads, as well as processing rate which is affected by the available resources. The authors demonstrate that they are able to use their derived models to determine why the experienced quality of a multimedia stream declines. In our work, on the other hand, we use the utility functions to predict the user experience, and hence allow us to maintain a desired level of user experience, whilst minimising energy.

Alia et al. (2007) present a user experience model based on utility. This model is used to understand the context of the service being delivered to the end user and adapt it accordingly. Their approach focuses strongly on multimedia delivery, with a strong emphasis on maintaining uninterrupted service delivery. For example, they allow a user to switch the device they are using to watch a multimedia stream, and use the information about device capabilities and the type of stream to choose the appropriate stream quality. Additionally, they consider the case where the user is streaming the device via a less capable network, resulting in lower usable bandwidth. Therefore, they are required to understand device characteristics, user requirements, as well as the perceived quality of streams for different devices. We focus on mobile applications on a single device, and hence do not need to deal with the migration of devices. However, many mobile applications have vastly different characteristics, and often a mobile application will deliver numerous services at the same time. Therefore, we need to understand the types of user experience delivered by an application, as well as how good the experience is at any point in time.

2.3 Optimising for Performance and Energy Efficiency

Following our overview of optimisation and modelling strategies for Quality of Experience, we consider strategies which focus on the optimisation of specific system components. We consider the CPU and GPU specifically as these are both major contributors to performance, as well as the system energy consumption. As such, we focus heavily on these components through this dissertation.

2.3.1 CPU

In this section, we consider optimisation approaches that focus on the CPU. We begin by considering Continuously Adaptive DVFS, which is introduced by Spiliopoulos et al. (2011). The authors present a framework to adjust the DVFS scaling for applications which are memory-bound. They monitor the low-level operation of the application and adjust the DVFS scaling based on CPU stalls and memory latency. Another work which detects when execution is compute-bound or memory-bound is presented by Hsu and Feng (2005). This is similar to our approach as we monitor the number of Instructions Per Cycle (IPC), which implicitly includes the stalls, and the miss latency. However, these approaches do not focus on QoE, and specifically focus on the CPU.

Yuan and Nahrstedt (2003) present GRACE-OS which adjusts the CPU DVFS configuration in order to reduce idle time and thus runs tasks at lower frequencies to reduce overall energy consumption. By reducing the amount of time that the system is idle, whilst still meeting the task deadlines they are able to reduce the energy consumption for the CPU. It is worth noting that this approach does not always reduce the system energy as other system components may consume a large amount of power, and hence it is better to run the CPU at a higher frequency, and then power down. In our work we demonstrate similar characteristics about systems, and show that in many cases the CPU frequency can be drastically reduced without impacting the performance of the workload.

An online regression model is used to predict the required performance in the next time slot, and the DVFS scaling is adjusted accordingly by Choi et al. (2004). Additional machine learning based approaches are presented in Shen et al. (2012) and Jung and Pedram (2010). The former presents a reinforcement-leaning-based DVFS governor which adjusts the scaling based on temperature, energy and performance. The latter uses a pre-computed table of DVFS states, and a supervised learning algorithm chooses the correct DVFS scaling. However, this is not demonstrated in a real implementation. The approach taken in our work uses simpler models which are able to deliver similar energy reductions in a realistic and realisable environment.

Statistical models to predict web page demand and energy consumption are used by Zhu and Reddi (2013) in order to choose whether to schedule execution on a big or a little core, and at which frequency. They employ offline analysis techniques to determine the compute requirements for each web page, which allows them to create a model predicting the compute requirements for a web page. They employ this model to determine if a particular web page should run on a bigger, more powerful and power hungry core, or a smaller, energy-efficient core, based on a target render time. Whilst their approach allows them to save a significant amount of energy, it focuses specifically on the task of rendering web pages. We, on the other hand, consider a variety of applications, and consider the CPU and GPU, as well as the interactions between these components.

2.3.2 GPU

We now move on to strategies which attempt to optimise the performance of the GPU specifically, or include the GPU as part of their overall optimisation strategy. We begin by considering the work by Pathania et al. (2014), who devise a combined CPU-GPU governor which adjusts both the CPU and GPU DVFS configuration in order to provide a minimum frame rate for mobile gaming. Traditionally, a GPU would not support DVFS, and would therefore rely on a run-fast-then-sleep approach, generally controlled by the GPU drivers. The authors demonstrate that they are able to reduce the energy consumption of the system as a whole by considering both the DVFS of the CPU and the GPU at the same time. In our work, we also consider the system as a whole, focusing on the CPU and GPU specifically, as the memory traffic generated by one component can significantly impact the performance of the other. Our work builds upon the findings of Pathania et al. by also considering the user experience, as opposed to raw performance, when making DVFS adjustments.

Jeong et al. (2012) present run-time QoS adjustment for a GPU. They begin by proposing using the number of tiles for a GPU frame to determine the rendering progress, then use this to determine if the GPU is ahead of schedule, on time, or behind. When ahead of time the GPU priority is lowered to below that of the CPU, whereas it is raised otherwise. This ensures that the GPU meets its deadlines while keeping the CPU latency as low as possible. Their work, however, does not consider how the GPU behaves under bandwidth constraints, or how the performance of the CPU is affected when running a full-system workload. We consider both of these aspects in Chapter 6, and demonstrate that the bandwidth available to the GPU can first of all strongly affect the performance of the GPU itself, but can drastically reduce the performance of other system components. This is especially significant for latency-sensitive components such as a CPU.

Hong and Kim (2010) present us with a power and performance model for a GPU which illustrates that once a GPU saturates the maximum bandwidth that the memory system can supply, then it is pointless to increase the number of cores in the GPU as performance does not increase, and sometimes worsens. By limiting the number of running GPU cores, the paper shows that it is possible to achieve the highest possible GPU performance for a particular application, whilst saving power by shutting down unused cores. It is also shown that some GPU applications scale non-linearly as the number of cores is increased, and therefore have a lower optimal operating point. Non linear scaling has been observed as part of this work (see Chapter 6), as some frames do not scale with the number of cores due to the high bandwidth requirements. Our work expands on this as it also investigates CPU performance penalties due to contention from high volume GPU traffic.

In the situation that the memory is not capable of satisfying the large bandwidth requirements for the GPU, Wang (2011) demonstrates that fewer processing cores should be used. This is demonstrated for various GPU applications using GPGPU-sim, and they suggest that the number of active GPU threads should be chosen based on per-thread resource requirements. This ties in with our work in Chapter 6 in which we investigate how the GPU requirements change as the number of active threads is altered.

2.4 Concluding Remarks

In this chapter we have provided an overview of the related works found in the literature. We considered various approaches to measuring and acting upon user experience in the literature. These includes approaches which combined the already present QoS mechanisms with user experience information in the form of Quality of Experience (QoE), and investigated their effect on user-supplied MOS scores. QoE is a term which has been used in the literature, but is less often found when considering mobile devices, and instead is applies to communication networks, and the supply of end-to-end multimedia services. In this thesis, we take QoE, and use it to optimise mobile systems at run time. This allows us to get the required level of user experience, whilst consuming as little power as possible. As part of our contribution, we provide a pair of parameterised QoE utility models, which translate temporal measures of performance into the resulting user experience.

Chapter 3

Understanding and Modelling User Experience

Understanding user experience is key to optimising systems for user behaviour, as well as ensuring that the end user is presented with a satisfactory experience. As it is almost impossible to directly measure the user experience, without explicitly asking the users how a service was perceived, proxies for the user experience must be used. It must be possible to calculate the user experience at run time before the task has completed as this allows the system configuration to be adjusted to the task at hand, as opposed to optimising for future instances of the same or a similar task. This results in a higher overall experience.

In this chapter we begin to answer our first research question: *Can an quantitative relationship between low-level metrics and user experience be defined which permits realistic estimation of user experience at run time?* We define QoE utility functions, which allow us to translate the low-level performance of the device into an estimate for user experience. Whilst we do not conduct user studies, we base our models on key works presented in the literature. Our models aim to predict the instantaneous user experience at a given point in time, and assume that the workload does not change significantly in the long run. We leave the investigation of long-term user experience optimisation as future work.

Throughout this work we use QoE, which is an objective estimate of the user experience. We begin by defining QoE in Section 3.1. We then define the different types of QoE workload, and explain how we measure and predict the QoE for each in Section 3.2. In Section 3.3, we explain how to calculate the QoE for the complete system, as oppose to for a service or a particular workload. Finally, we present models for web browsing user experience and GPU rendering user experience in Section 3.4.

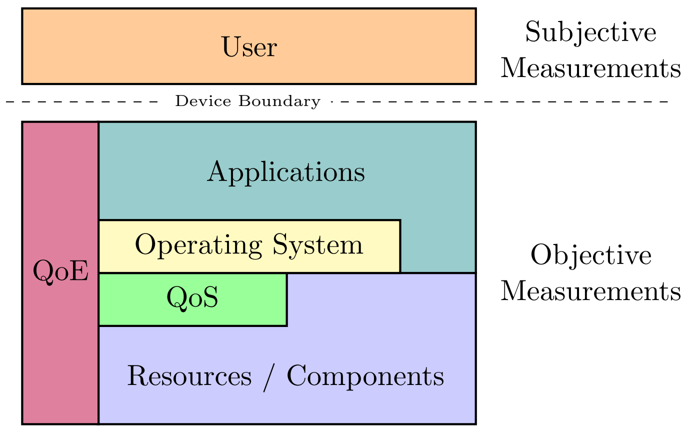


Figure 3.1: Relationship between QoS, QoE, user requirements and system resources.

3.1 Quality of Experience

When consumers use a device, they have certain expectations such as the device's ability to play a particular game, long battery life, or simply being responsive. As long as the device is able to live up to these expectations, the user remains satisfied. If, however, the device is unable to satisfy the user's expectations the user becomes increasingly dissatisfied. The user will rate the device very poorly for large disturbances in the delivered service. This has been highlighted by Fiedler et al. (2010), who artificially generated QoS disturbances for web browsing and monitored user satisfaction. Therefore, great care needs to be taken when designing the system and corresponding QoS schemes that try to ensure a high quality service without disruption.

QoS schemes provide a mechanism for dividing up the available system resources, such as energy, run time and component frequency, to ensure that each component in the SoC, such as a CPU or a GPU, gets at least the minimum resources it requires, e.g., high bandwidth or low latency. As the load on the system increases beyond that which can be sustained, QoS sets the order of resource allocation failure, and the more important devices will receive their required resources first and at the expense of less important components. It is, however, no simple task to determine which components are the most important since this changes dynamically as the services delivered by the system change, and therefore require run-time adjustment.

CPU governors adjust the DVFS state based on the type of governor and the performance of the CPU (The Linux Kernel Archives, 2014; Pallipadi and Starikovskiy, 2006). They are efficient at targeting high performance or extreme power efficiency. However, they are inadequate at preserving high user experience whilst operating at the minimum power required to do so. As we discuss in Chapters 5 and 6, the complexity of targeting the optimal operating point reduces when user experience is utilised in the place of low-level metrics. Chapter 5 demonstrates that low-level metrics are a good indicator of how optimally the system is performing, whilst Chapter 6 combines this with knowledge about the user experience.

In Chapter 5 we show that there is merit in explicitly using user experience for system optimisation. Hence, we need a set of measurable system level metrics which can be linked to the user experience. Specifically, a set of them needs to be defined which are able to quantify the user experience and provide an estimation of the experienced performance. We call this set of metrics QoE, which is a term used frequently by the telecommunications industry when referring to a delivered end-to-end service, such as streaming video (ETSI TR 102 274, 2010; Ghinea and Thomas, 2005; Collange et al., 2008). More recently, applications are being viewed as delivering a service or set of services (Hirsch et al., 2006), and it is to the delivered services which we wish to apply QoE. QoE does not appear to have been explored in the context of system-level optimisation.

The relationship between QoS and QoE is presented in Figure 3.1 and illustrates how QoE can be used for system optimisation. The dashed box shows the system boundary and illustrates that the subjective requirements come from outside the system, i.e., from the user of the device. QoE is a measure of how the user perceives and reacts to the services provided by the device. QoE is defined as being in range 0-1, where 0 means completely dissatisfied and 1 means completely satisfied. QoE is used as a part of a feedback loop to measure the performance of the services provided by the system in a user-centric manner. It is then used to influence the resource allocation, which in turn alters the performance of the delivered services. A similar feedback loop is used for QoS. QoS is able to act upon objective requirements such as the distribution of bandwidth but is unaware of the subjective, user-specific requirements - QoE is designed to encompass precisely these. Understanding how the end user interprets device performance enables the system to make sensible, user-centric resource allocation decisions which can be combined with those taken by traditional QoS mechanisms to ensure proper operation, and high user satisfaction. Fiedler et al. (2010) present a link between the time taken to load a web page and the mean opinion score (MOS) for a group of users. MOS is a user experience metric in the range of 1 to 5, where higher scores are better, which is obtained by asking the users themselves to rate the interaction. Therefore, it is an accurate representation of user experience for a sample of users, but it cannot be used easily for run time optimisation, unlink QoE. This provides us with an established link between QoS and QoE for a real scenario and shows how a disturbance can have a pronounced effect on the user experience.

It is worth noting that there is no single, universal measure of user experience as this depends heavily on the types of applications running (Brooks and Hestnes, 2010). As an example, a GPU-intensive task, such as a 3D game, will require a sufficiently high frame rate to ensure that the user is satisfied with the performance. This is quite different to, say, a file retrieval task or the display of a still picture, where frame rate is an irrelevant concept. Therefore, whilst there is a set of measurable metrics which give an indication of the user's satisfaction and the quality of their experience, those of relevance vary as the applications running on the device change.

In order for systems to be able to optimise their performance for user experience, multiple requirements must be fulfilled:

1. A set of metrics which are correlated with the user experience but remain measurable from within the confines of the device must be determined.
2. Metrics which are of relevance must be selected at run time based on either observed traffic or explicit notification from a higher level within the system.

Each application run on a device uses a different subset of components and provides different services to other applications. Therefore, it is not possible to define a single QoE metric which can cover all workloads. For this reason, applications need to be subdivided into services, which then need to be classified such that the QoE per service can be calculated. This service-level QoE can then be combined to give the QoE for the application or complete system.

The user experience delivered by a device can be divided into two main categories. These categories are *temporal*, i.e., how long does it take to complete an action, or based on the *quality* of a delivered service. Temporal QoE is concerned with how long it takes a device to complete a specific operation, or if a device is capable of delivering a service at a minimum rate, e.g., *N times per second*. Qualitative QoE is focused more of the quality of the input data, such as the bit-rate for video and audio streams, or texture quality and polygon count for 3D games. Whilst there is some overlap between these two categories — dropped video frames can cause the quality of a video to decrease, for example — we focus on temporal QoE in this work as it is more closely related to device performance itself. Quality is an application layer QoE control, which is outside the scope of this thesis, which considers architecture layer QoE control.

The next section considers the different types of temporal QoE, and define a set of QoE-utility functions which allow the QoE delivered by the service to be quantified. This allows QoE to be used for user-aware system optimisation. Additionally, we provide a discussion of other types of QoE in Section 3.2.4.

3.2 Types of Workload

In general, unless an application delivers only a specific service, such as playing music, it is difficult to determine the user experience. This is because the application is based on a combination of different services, which all contribute to the user experience. These services may be of equal importance and relevance to the end-user, or the user may only

Service	Type of QoE
Audio	Throughput
Video	Throughput
Application Loading	Latency
Web Page Rendering	Latency
Downloading a File	Latency
3D Gaming	Throughput
Word Processing	Latency

Table 3.1: Services and the type of QoE they provide

care about a single one of them. If we take the example of streaming an online video, then both the video and audio must be decoded with minimal interruptions, and this must be completed within a specific time period. Both services must be of satisfactory quality in order to ensure that the user experience is high. Therefore, the QoE delivered by each service must be calculated independently, and must then be combined to give the QoE for the application or device.

Due to the complexity of modelling the QoE for a multi-service application, we subdivide the application into services, which are classified based on their operation. As stated previously, we only focus on *temporal* QoE in this work, and therefore, we define two main types of service. We define services to be either *latency sensitive* or *throughput sensitive* based on the how the service is being delivered. Latency sensitive workloads need to complete work within a short time period in order to provide a high level of user experience. On the other hand, throughput sensitive workloads need to complete work items as a minimum rate to deliver an acceptable user experience. Table 3.1 lists different services and the types of QoE they deliver.

In addition to latency- and throughput-sensitive services, we also need to define a third type of QoE-utility function. Compute workloads, such as benchmarks which measure the performance of the system when completing various tasks, do not contribute to the user experience *per se*. Ideally, compute workloads complete as quickly as possible, and therefore shorter run-times are favourable over longer ones.

We define QoE-utility functions for the three types of workload in the following sections.

3.2.1 Latency-Sensitive Workloads

Latency-sensitive workloads aim to complete within a certain time period, or latency. Due to the way that humans perceive delays, there is a minimum time below which the QoE saturates as humans are either not able to perceive it, or because it is sufficiently short such that the user does not care. Once this minimum time has been reached, the QoE starts to decrease as the user begins to notice and become dissatisfied with the delay.

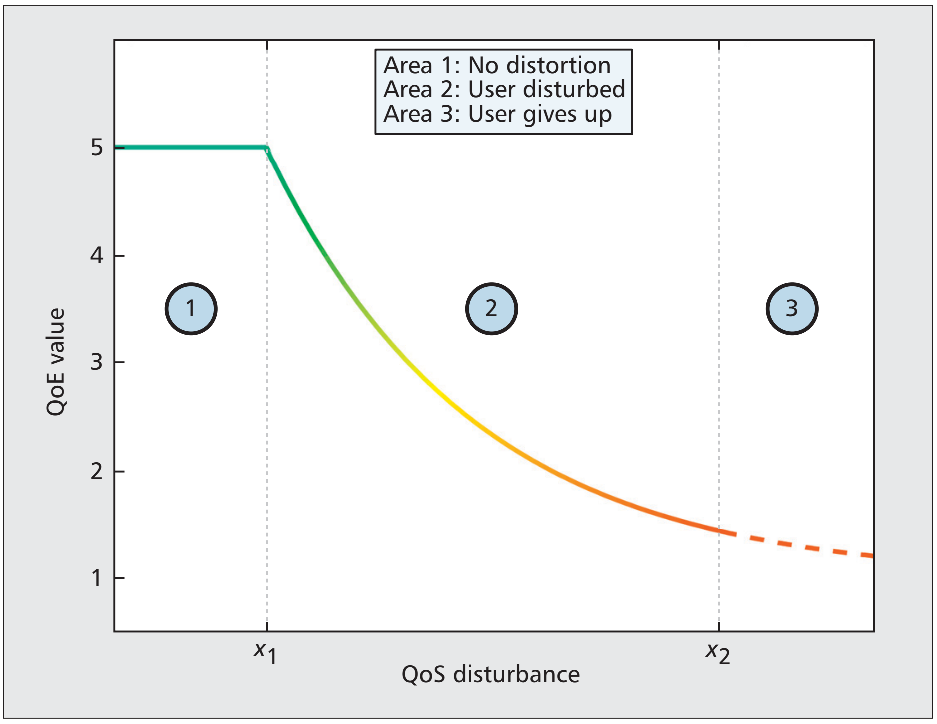


Figure 3.2: *General shape of the mapping curve between QoS and QoE* as presented by Fiedler et al. (2010). The figure has been directly copied from their publication without alteration.

Examples of workloads that are latency-sensitive include loading applications, where shorter load times are favourable, rending web pages, and even seeming minor interactions such as feedback when a user presses a button on a device. The device should feel as if it is directly responding to the input from the user, rather than providing a slow, unresponsive service. Nielsen (2009) provides rough guidelines for time delays for user interfaces, and states that delay of less than 0.1 s appear instantaneous to the user, whilst 1 s delays become noticeable and 10 s delays become disruptive.

Related work (Fiedler et al., 2010; ITU-T, 2005) demonstrates that the relationship between the time taken to complete a task and the user experience is roughly logarithmic or inverse exponential. We present the relationship defined by Fiedler et al. (2010) in Figure 3.2. They define three key areas for user interaction. In the first, labelled *1* in the figure, the user perceives no distortion, and hence is maximally satisfied with the service they receive, although there is a small QoS disturbance. In *2* the user becomes aware of the disturbance, and becomes increasingly dissatisfied with the service delivered as the QoS disturbance increases. Finally, in *3* the user is extremely disturbed by the level of service received, and gives up, abandoning all interaction. In their publication, Fiedler et al. present an exponential model for the relationship between QoS disturbance and the QoE perceived by the user, but also present a logarithmic model, which demonstrates a close fit too. Therefore, we define a similar, but logarithmic model for our latency QoE utility function.

Figure	t_0	t_x	QoE_{t_x}	λ
A	1 s	10 s	0.1	0.25
B	1 s	10 s	0.005	0.75
C	0 s	10 s	0.5	0.069

Table 3.2: Parameters used to calculate QoE for Figure 3.3

For our work, the following representation for latency-sensitive QoE is chosen:

$$QoE_L = \begin{cases} 1 & t{<}t_0 \\ e^{-\lambda(t-t_0)} & \text{otherwise} \end{cases}$$

where QoE_L is the QoE for the latency workload, t is the time taken to complete the work, t_0 is the latency below which maximum QoE is achieved and λ is a constant used to control the slope of the QoE degradation. This model allows us to capture both the range of delays which can not be perceived by the end-user, as well as the rapid degradation in experience once that threshold is passed. As λ controls the slope of the trade off between the time taken and the QoE observed, the value must be carefully chosen. In order to assist in determining a sensible value for λ, we derive the following relationship:

$$\lambda = \frac{ln(QoE_{t_x})}{t_x - t_0}$$

where QoE_{t_x} is the desired QoE at time t_x. Using this relationship, a QoE value at a specific time can be used to determine the value of lambda, and hence the slope of the QoE utility function.

Example latency QoE utility functions are shown in Figure 3.3, and the parameters are shown in Table 3.2. The parameters are varied such to show their influence on the time-QoE relationship, and do not represent any use cases in particular. Figures 3.3 (A) and 3.3 (B) demonstrate the effect of t_0 on the utility function, and demonstrate the effect of QoE_{t_x} of the shape of the function. Figure 3.3 (C) demonstrates a very shallow trade off, which could be used for minimally latency-sensitive workloads.

3.2.2 Throughput-Sensitive Workloads

Throughput-sensitive workloads are required to complete work items at a minimum rate in order to provide a satisfactory user experience. These workloads include the GPU rendering frames, video decoding and audio decoding. Provided that the required rate is met, then throughput-sensitive workloads deliver a high QoE, whilst the QoE drops down when the rate cannot be met. For example, a GPU rendering at or above the

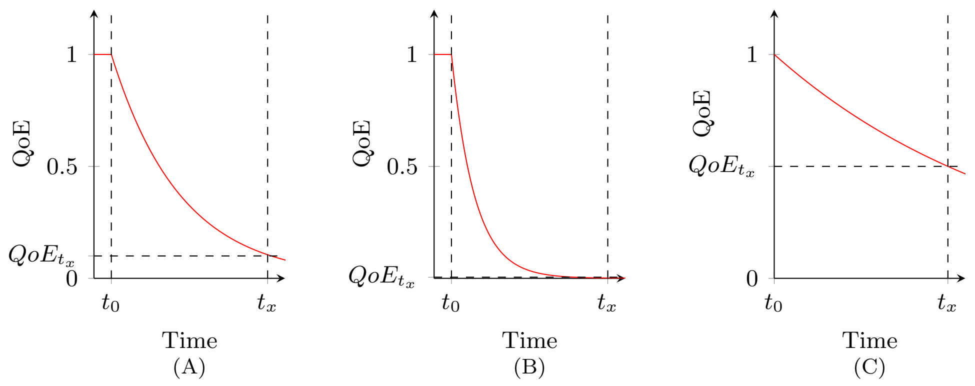


Figure 3.3: Calculating QoE for a latency workload

refresh rate of the display controller will deliver the maximum QoE that the particular system can deliver. However, should the GPU frame rate drop below this threshold, then the QoE will begin to decrease, especially when the end user begins to perceive the GPU frames as a set of distinct images rather than fluid motion. Such a relationship has been shown by Krause et al. (2008) whose results we present in Figure 3.4. They conduct a series of experiments in which they ask a set of 17 users to judge if videos appear smooth in motion, or if they are perceived as a series of still images. They conduct this experiment for three different types motion captured in the videos: slow, medium and fast. They demonstrate that a sufficiently high frame rate is required for an end user to judge motion in a video clip as smooth. For medium and high rates of motion, they show a very steep relationship between the frame rate, and the end-user ratings.

As the QoE for throughput-sensitive workloads tends to 0 for low rates, and tends to 1 for high rates, we model the QoE utility using a sigmoid function. This sigmoid is shifted such that the y-intercept is not at a rate of zero. Therefore, the QoE utility function for throughput-sensitive workloads is given by:

$$QoE_T = \frac{1}{1+e^{-x}}$$

$$x = \frac{m - \frac{1}{t_r}}{s}$$

where QoE_T is throughput QoE, t_r is the time taken to complete one iteration, m is the midpoint of the sigmoid (where is crosses 0.5) and s is a scaling factor used to control the slope of the sigmoid. The value of s is calculated using the derived equation:

$$s = \frac{R_{target} - m}{ln\left(\frac{1-QoE_{target}}{QoE_{target}}\right)}$$

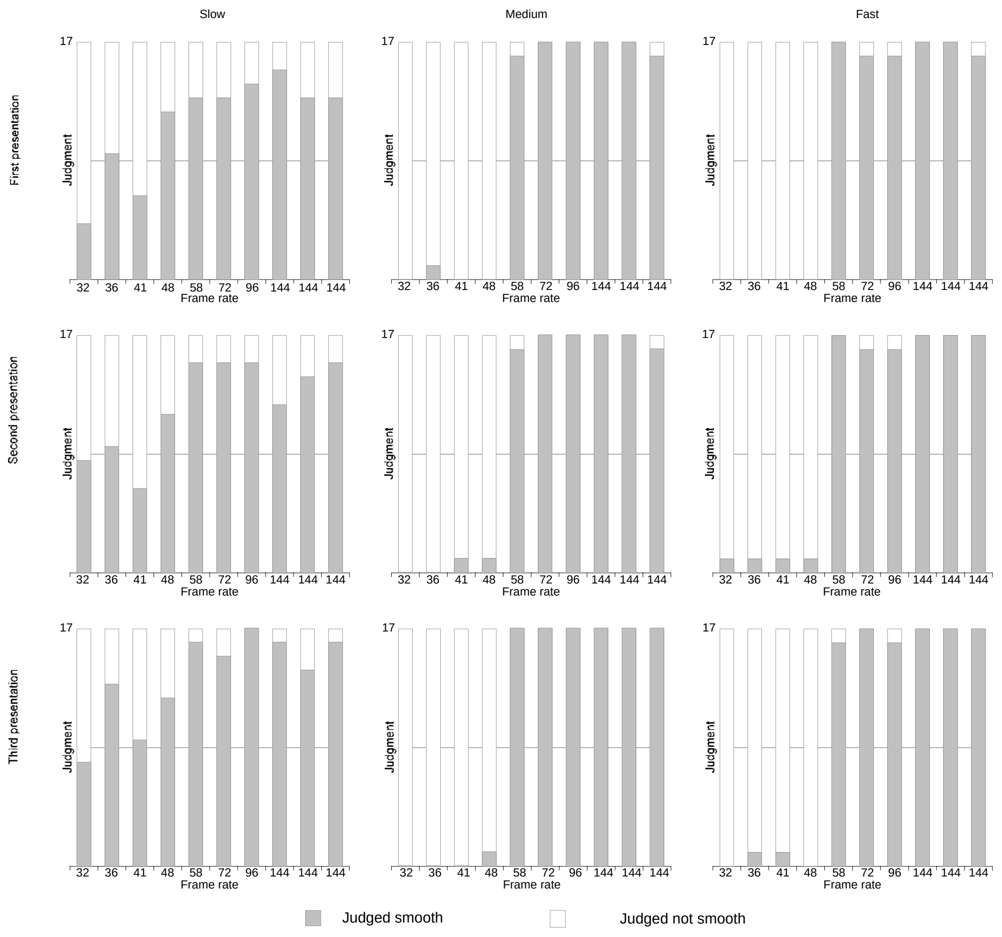


Figure 3.4: *Summed results of the judgements in the user experiment for each judgement attempt and each speed* as presented by Krause et al. (2008). The figure has been directly copied from their publication without alteration.

Figure	m	R_{target}	QoE_{target}	s
A	30	60	0.99	-6.53
B	50	80	0.99	-6.53
C	30	60	0.9	-13.65

Table 3.3: Parameters used to calculate QoE for Figure 3.5

where $Rate_{target}$ is the rate and QoE_{target} is the QoE at the target rate.

Example throughput-sensitive workload QoE utility functions are shown in Figure 3.5, and the corresponding parameters are shown in Table 3.3. As before, the parameter values chosen do not represent any use case in particular, but instead serve as example configurations. Figures 3.5 (A) and (B) demonstrate the effect of m and R_{target} on the utility function. Figure 3.5 (C) demonstrates how the QoE_{target} parameter influences the shape of the utility function.

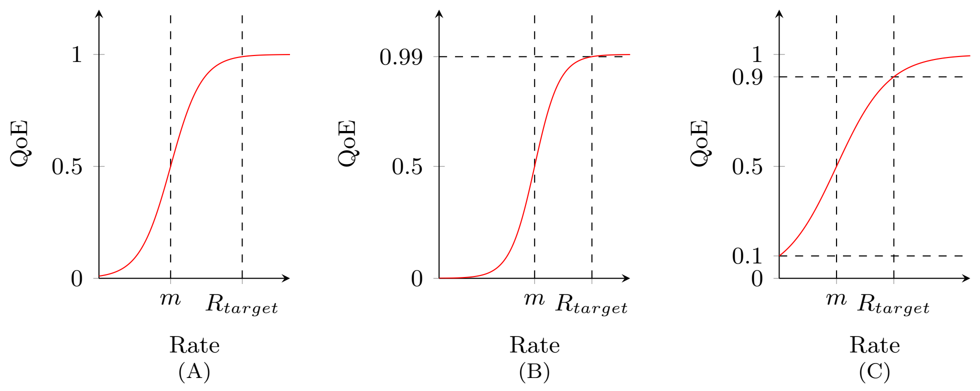


Figure 3.5: Calculating QoE for a throughput workload

3.2.3 Compute Workloads

In general, compute workloads do not contribute to the user experience, and instead attempt to measure the limits of performance or battery life, or simply complete a computation. In the case of a benchmark, which is generally a compute workload, the workload provides a score based on the capability of a device to complete a specific type of operation, e.g., computing π and measuring the time taken to do so. For these sorts of workloads higher scores are favoured and therefore the workload should complete as rapidly as possible. As these types of benchmarks do not directly provide a level of user experience, it becomes very difficult to determine when their performance is satisfactory.

Given that compute workloads generally do not contribute to the user experience, and that those that do have a deadline can be modelled as a latency-sensitive workload, we propose to run compute workloads as rapidly as possible. In the case where there is no deadline at all, it makes sense to throttle back the execution in order to optimise for energy efficiency. Ideally, such workloads should be run when the device is not in use by the end-user to ensure that their experience remains unaffected.

3.2.4 Other Types of Workload & User Experience

We have considered latency-sensitive user experience, throughput-sensitive user experience, as well as considering compute workloads, which simply need to complete as quickly as possible to satisfy the end user. In this section we consider the other types of user experience for completeness.

We begin by considering the surface temperature of the device. It is well known that the surface temperature of the device affects the user experience (Egilmez et al., 2015). This actually is comprised of two components. The first is the maximum surface temperature reached and the second is the rate of surface temperature rise, i.e., the amount of time until the device becomes uncomfortably hot for the end user. The surface temperature of the device is affected by both design time decisions, as well as the running applications and the run time decisions made by the various system governors. When a device is designed, the thermal capacity of the materials is taken into account, and the designers settle on a Thermal Design Power (TDP). This is the amount of power that the device is designed to dissipate. As long as the device power dissipation remains below this level, then it should continue to operate normally, but if this is exceeded for a long enough period of time, then the performance of the device must be throttled in order to reduce the temperature. The approach taken by Egilmez et al. (2015) involves adjusting the frequency of components in response to the surface temperature to ensure a satisfactory user experience. Additionally, the materials used to assemble the mobile device must be chosen to give a sufficiently high TDP and as low a surface temperature as possible. This means that the temperature of the device not only affects the experience of the user directly, but it can also affect the performance of the device, resulting in further reduced user experience. Whilst we do not consider thermal device characteristics as part of this work, it is worth noting that we focus on providing the desired level of user experience at the lowest energy consumption possible. This results in a reduction is power dissipation, and therefore can influence both the time until an uncomfortable surface temperature is reached, as well as reducing the maximum temperature reached.

Another type of user experience is quality-based user experience. This type of user experience is influenced by the bit rate used to encode audio and video streams. For example, when playing back a two otherwise identical videos where one is encoded at a substantially lower bit rate than the other, the video will the lower bit rate will often be rated more poorly than the other. The same applies to audio, which is why scientists and engineers use the PESQ model to determine the resulting quality of an audio stream (Rix et al., 2001a). There are various works which adjust the quality of a video or audio stream to save energy, and to meet computation deadlines (Choi et al., 2002; Ge and Qiu, 2011; Shafik et al., 2015). For example, a typical video is encoded with a series of key frames, which contain the complete image, and frames which only store the relative differences when compared to the previous frame. By skipping one or more non-key frames it is possible to save energy with only a minimal reduction in quality, assuming sufficiently high rates of key frames, and little motion within the video itself. In this work we do not consider this type of optimisation. First of all, for many input sources it is difficult to determine the resulting user experience without explicitly asking the end user how good the quality was. This makes it hard to optimise for at run time. Secondly, we focus on system level optimisation, and aim to optimise irrespective of the workload.

The design of a User Interface (UI) also strongly affects the end user's experience, as a complex non-intuitive UI can result in a poor user experience. A typical user interface should be clear, and must be easy to understand in as short an amount of time as possible. If the user is required to spend a long time figuring out the intricacies of the UI, then they will rapidly become irritated and will stop using the application or device. This is especially important when the user is trying to input data, such as when using the keyboard. This was studied in detail by Page (2013) for a variety of mobile software keyboards on Android. They compared the rate of data entry for each keyboard and concluded that a combination of simple input gestures and prediction achieved the highest data entry rate. Similar principles apply when a user views web pages, or any time a user must perform actions to instruct the device to complete a task (Roto, 2006; Shrestha, 2007). Additionally, even aspects such as the colour scheme must be taken into consideration, as the wrong choice of colours can convey the wrong message, or may simple make the UI hard to read and use. This type of user experience is something that we again do not consider. This is something we leave up to the application and operating system designers, as it is not something we are able to optimise for at run time.

At this stage, we would also like to consider long-term user experience. Our models and experiments focus on the user experience delivered over a short period of time, and do not consider the long-term impact of our decisions. For example, the battery life of a device is strongly affected by the amount of energy consumed when running various tasks on the device. If a particular task runs for too long or consumes too much energy, then the battery life of the device will be significantly shortened. Should the battery life of the device become too short, then the end user must charge the device in order to continue using it. This by itself has a user experience impact. Therefore, the device should deliver good-enough user experience whilst ensuring that the device itself is able to run for a long-enough period of time between charges. Our models, experiments and methodologies present a significant first step towards run-time user experience optimisation, but focus on the QoE at a specific point in time, and do not include forecasting. Therefore, we leave the long-term user experience optimisation as future work.

3.3 Determining System-Level QoE

Section 3.2 defines the relationship between workload performance and QoE for latency- and throughput-sensitive workloads, as well as discussing compute workloads. These QoE utility functions can be used to determine the QoE delivered by a particular service or application, but do not give the QoE for the complete system. Therefore, these component- or service-level QoE metrics need to be combined in order to determine the system-level QoE.

Naively, determining the system-level QoE appears as a trivial task. However, care must be taken in order to ensure that this relationship is correctly modelled. First of all, it does not make sense to model the system-level QoE by taking the arithmetic mean of the component QoE values. This is due to the fact that taking the arithmetic mean does not clearly reflect when a single component is behaving poorly. If, for example, a single component such as the display controller was performing poorly, then this would not be reflecting in the arithmetic mean. Similarly, it would not make sense to calculate total QoE as a sum of all component values as it would be difficult to determine if a single component is behaving poorly, or is many components were behaving sub-optimally.

The service-level QoE is defined such that it is in the range 0 - 1, where 0 indicates a completely unacceptable QoE and 1 indicates the highest QoE possible. Therefore, system level QoE should also be in the range 0 - 1. Therefore we propose a multiplicative approach to calculating the system-level QoE. Not only does this preserve the range of the component QoE values, but also clearly highlights when one more more components are behaving inadequately. The overall system-level QoE is calculated as a product of individual component QoE values:

$$QoE_{sys} = \prod_{\forall c \in C} QoE_c$$

where QoE_{sys} is the overall QoE for the complete system, QoE_c is the QoE for component c and C is the set of all components in the system. This definition of system-level QoE ensures that a single component delivering inadequate QoE is correctly reflected in the overall QoE. However, this definition will also highlight when multiple components are behaving unacceptably and are impacting the QoE. The calculation of system-level QoE also needs to take into account components that are inactive and must not include these in the QoE. This can be done by removing the terms from the calculation, or, more simply, setting the QoE of all inactive components to 1. The multiplicative QoE calculation ensure that the QoE of 1 for inactive components will have no impact on the system QoE.

3.4 Measuring User Experience for CPUs and GPUs

This section builds on the QoE utility functions described above, and specifies a set of values for the utility functions. These values are chosen to tune the QoE utility functions to be representative of real user experiences for web browsing and graphics workloads running on the CPU and GPU respectively.

3.4.1 Calculating QoE for the Web Browser

As part of this work, we look at the time taken to render a set of web-pages, which has been linked to user experience in various studies (Fiedler et al., 2010; ITU-T, 2005).

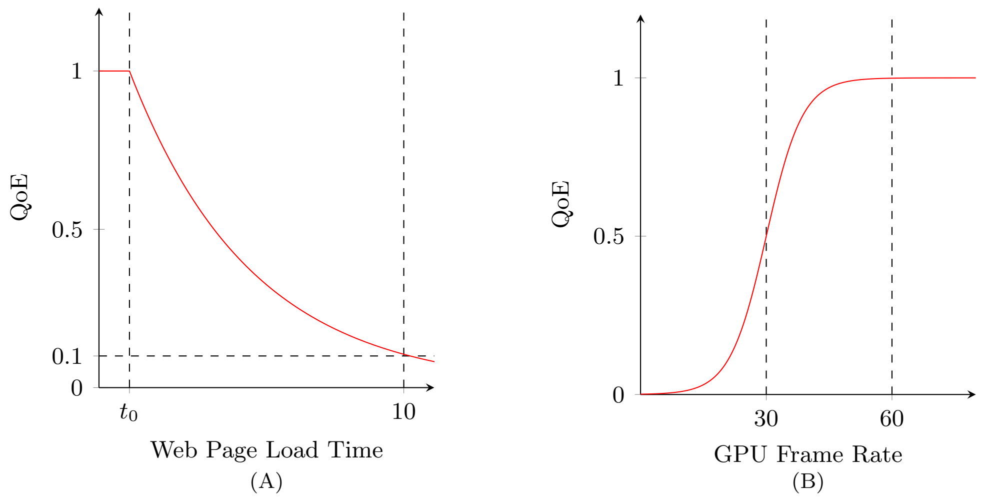


Figure 3.6: Relationship between web page load time and QoE (left) and frame rate and QoE for the GPU (right)

These studies have demonstrated that the relationship between QoE and web page load time is approximately logarithmic or inverse exponential. We use the model defined in Section 3.2.1 to model the QoE utility for web page rendering. Specifically, the QoE delivered whilst rendering web pages is given by:

$$QoE_{CPU} = \begin{cases} 1 & t{<}t_0 \\ e^{-\lambda(t-t_0)} & \text{otherwise} \end{cases}$$

where QoE_{CPU} is the QoE for the CPU, t is the time taken to render the web page, t_0 is the render time below which maximum QoE is achieved and λ is a constant used to control the slope of the QoE degradation. In order to determine λ we derive the following relationship:

$$\lambda = \frac{ln(QoE_{10})}{10 - t_0}$$

where QoE_{10} is the desired QoE at a render time of 10 seconds. The parameters used for calculating the CPU QoE are presented in Table 3.4 (A) and the utility function is illustrated in Figure 3.6 (A). This value is likely to vary between different demographics but, for the purpose of this study, we assume that web page render times of less than 1 second deliver the highest QoE. This value agrees with the work of Nielsen (2009), as well as being of the same order of magnitude as that presented in Ibarrola et al. (2009); Nah (2004). We assume a value of 0.1 for QoE_{10} as this is a close approximation of the findings of Ibarrola et al. (2009) (for expert-level users). These figures give us a λ value of 0.2559.

(a)

CPU QoE Parameters	
t_0	1 s
QoE_{10}	0.1 s
λ	0.2559

(b)

GPU QoE Parameters	
m	30 Hz
FPS_{target}	60 Hz
QoE_{target}	0.99
s	4.35

Table 3.4: Parameters used to calculate QoE for a CPU and a GPU

This model of QoE can be applied to any workload where work must be completed before a deadline is reached, i.e., the workload is *latency sensitive*. Therefore, this model can be applied whenever the user must wait for an event to complete. Examples include loading applications, saving files and waiting for downloads to complete, amongst others. However, care must be taken to tune the parameters for each particular workload.

3.4.2 Calculating QoE for GPU-Based Graphics Workloads

We model the QoE of the GPU using a shifted sigmoid function which can be tuned to give the desired QoE-utility relationship. This is chosen as the GPU provides *throughput sensitive* QoE, as explained in Section 3.2.2. The FPS-QoE relationship is given by:

$$QoE_{GPU} = \frac{1}{1 + e^{-x}}$$

$$x = \frac{m - \frac{1}{t_{frame}}}{s}$$

where QoE_{GPU} is the QoE for the GPU, t_{frame} is the time taken to render the most recent GPU frame, m is the midpoint of the sigmoid shown in Figure 3.6 (B) and s is a scaling factor used to control the slope of the sigmoid. The value of s is calculated using the derived equation:

$$s = \frac{FPS_{target} - m}{ln\left(\frac{1-QoE_{target}}{QoE_{target}}\right)}$$

where FPS_{target} is the target frame rate and QoE_{target} is the QoE at the target frame rate. The parameters used to calculate the GPU QoE throughout this work are presented in Table 3.4 (B), and are based on the work by Zinner et al. (2010).

As was the case with the exponential QoE relationship described in Section 3.4.1, this QoE utility function can be applied to different workloads, and does not only apply to the GPU. Specifically, it can be applied to *throughput* workloads where work must be

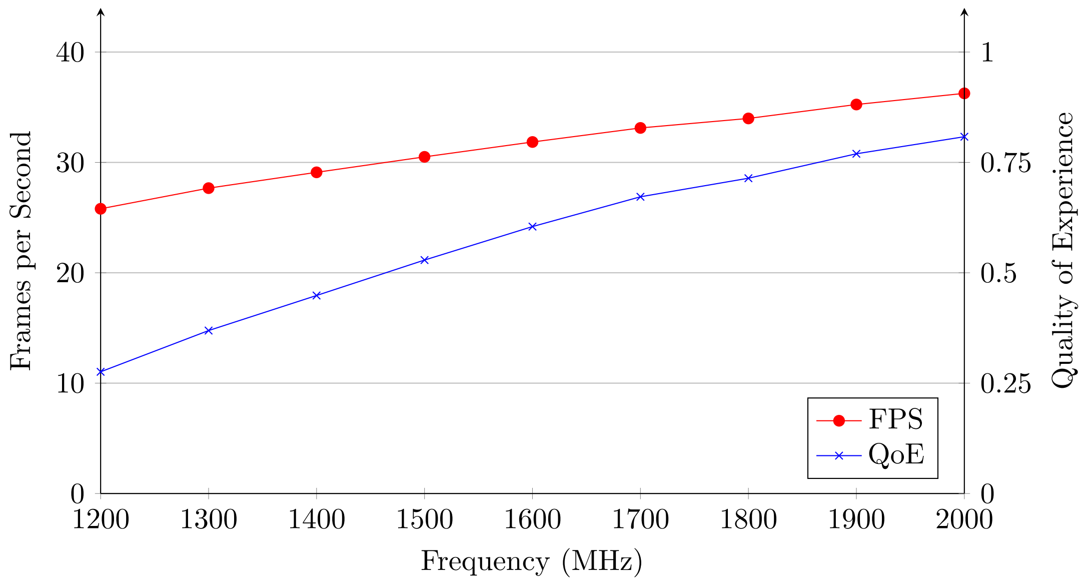


Figure 3.7: The number of frames rendered per second and corresponding QoE when running The Chase on an Odroid XU3 as the frequency is varied.

completed at a minimum rate. For example, this type of model could be applied to video decoding where each frame of the video must be decoded before a frame-rate-imposed deadline.

3.5 Concluding Remarks

In this chapter we provided an overview of approaches to modelling and optimising for user experience found in the literature. We began by defining the term Quality of Experience (QoE), an objective measure of user experience, and relating this to Quality of Service, a measure and set of schemes used to divide up system resources. The important realisation is that QoE is a measure of how acceptable the performance of a device or service is to the end user, whilst QoS is focused on low-level device operation, and serves to ensure the components receive their required service from the rest of the system.

As one of the key contributions of this thesis, we defined two different types of user experience: latency-sensitive and throughput-sensitive user experience. The former relates to workloads where the user must wait for an action to complete, and hence too long delays result in a degraded experience. In the latter case, the user must be provided with updates at a minimum rate to remain satisfied, and this applies to workloads such as video playback and 3D game rendering. For each of these types of user experience we provided a parameterised model which can be used to translate delay, in the case of latency-sensitive user experience, or rate, in the case of throughput-sensitive user experience, into a QoE value in the range of 0 to 1. We use these utility functions throughout the rest of this thesis. We also briefly considered compute workloads, which are workloads which must complete as fast as possible. These workloads usually do not contribute to user experience, except by causing the user to wait for their completion, and hence we aim to complete as-fast-as-possible for these workloads. We also briefly considered how to calculate the QoE for the complete system, and concluded that one must take the product of all component and service level QoE contributions as a single poorly performing component will greatly degrade the overall experience.

Finally, we derived specific parameters for two types of user experience used extensively in this thesis. The first is QoE for the web browser, which we base on the latency-sensitive user experience utility function, using results from the literature to tune the model. Secondly, we created a model for GPU QoE, based on the throughput-sensitive user experience utility function. This model is also tuned based on relevant research found in the literature.

Chapter 4

Characterising CPU Efficiency

At the heart of any modern mobile device lies the CPU which is responsible for running the operating system, coordinating the other components and executing the bulk of a large number of applications. As the CPU performs such a vital role, it is imperative that it is able to operate efficiently such that it is able to deliver high levels of performance whilst remaining as energy efficient as possible. High energy efficiency is extremely important for all devices, as it affects the cost of operation and thermal characteristics, but this property is especially important for mobile, battery-powered devices.

In the previous chapter, we discussed user experience models and how the raw, low-level performance of a device can be translated into Quality of Experience (QoE). These models allow us to understand how the end-user of a device perceives the operation, and how satisfied they are as a result. In this chapter, we look into the properties of CPU traffic, how the CPU responds to memory system perturbations and how typical CPU workloads respond to DVFS scaling on the CPU.

We begin by looking at the characteristics of traffic generated by the CPU to the memory system in Section 4.1. Next, in Section 4.2 we run typical mobile workloads on real hardware and observe how the performance of user experience delivered by these workloads scales with CPU frequency. This is followed by a simulation-based CPU workload analysis in Section 4.3. This analysis allows us to observe how typical workloads scale with CPU DVFS, and gives insight into why these workloads may or may not scale well as the CPU frequency is adjusted. Next, we look at the design space for CPU DVFS adjustment by visualising the Pareto frontier for a number of CPU workloads in Section 4.4. Finally, we conclude with a summary of the chapter.

4.1 Profiling CPU Traffic

The CPU is arguably the most important single component in a modern smartphone. First of all, it is the component that is used to execute the Operating System, which provides an abstraction layer between applications and the hardware they are running on. The CPU is responsible for executing most applications, and for coordinating the operation of the other IP blocks contained in the SoC. It is therefore vital to understand how CPUs perform when resource constrained, especially as these insights can help to avoid a large impact on the user experience. Additionally, this also helps us to answer our second research question for a CPU: *From a low-level perspective, which level of service - in terms of latency and bandwidth - do typical components such as CPUs and GPUs require from the shared memory system?*

A CPU is an inherently complex device which produces complex traffic patterns due to interaction with caches and the rest of the memory system. It has both data and instruction dependencies, which may cause it to stall when the required data must be fetched from main memory due to it not residing in the low-latency caches. Due to this stalling, the latency as seen by the CPU should be kept as low as possible in order to ensure high and efficient CPU performance. As the frequency of the CPU increases, the number of stall cycles per memory access increases, resulting in wasted energy and reduced performance improvement. Therefore, for high-frequency operation, it is vital that the memory latency remains as low as possible if high energy efficiency is required.

In order to demonstrate the type of traffic generated by the CPU we show an Inter-Transaction Time (ITT) histogram for the CPU when running a SPEC2000 (Henning, 2000) benchmark, *eon*, in Figure 4.1. These results are generated using a fixed-latency memory model where each DRAM access takes exactly 150ns. This allows us to visualise the access patterns of the CPU more clearly without the interference from the detailed memory model.

There are multiple significant peaks shown in Figure 4.1. This first and most significant peak is at the access latency of the L1 cache. This is caused by CPU memory accesses which hit in the L1 caches. There is a second and much smaller peak caused by memory accesses hitting in the L2 cache. These are still serviced with a low latency and allow the CPU to operate relatively efficiently. Finally, there is a third peak for transactions which access the main memory. These have a very high latency and therefore there is a high inter-transaction time. The accesses to main memory are those that cause the greatest CPU slowdown as it stalls.

In addition to the *eon* results, we run BBench, and present the ITT for CPU transactions in Figures 4.2. As before, there is a set of peaks at low ITT values, which are due to data being cached in the L1 and L2 caches. However, as BBench accesses a large amount of data, only a limited proportion of the data is cached, and therefore the majority of

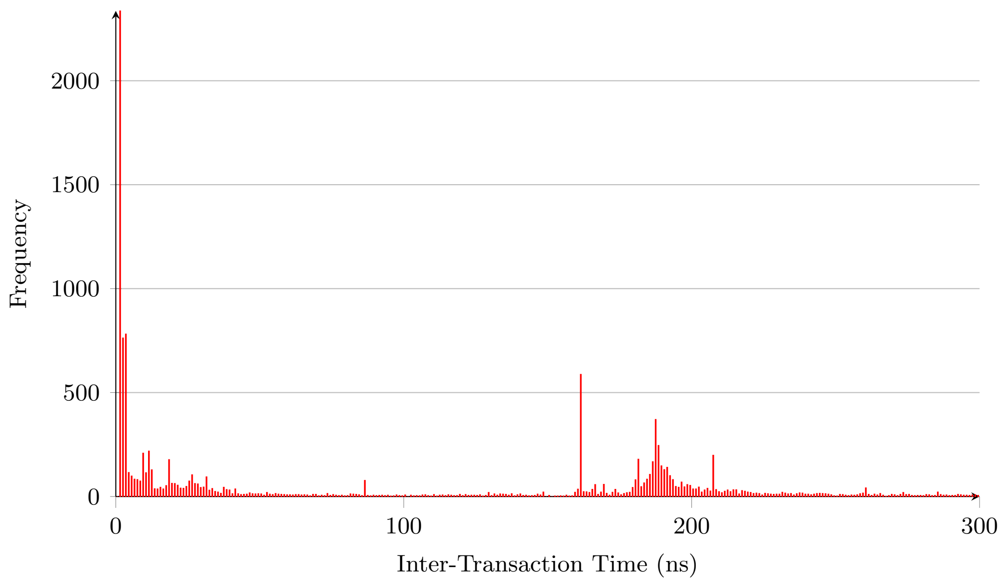


Figure 4.1: Inter-Transaction Time (ITT) histogram for eon from SPEC2000 with 150 ns memory latency, 1 ns bins

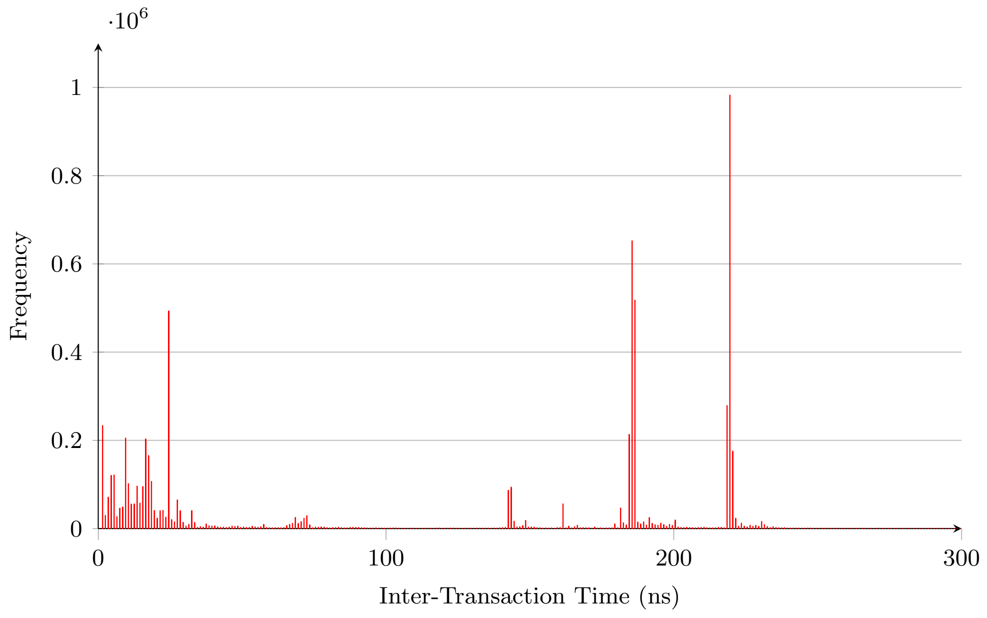


Figure 4.2: ITT histogram for BBench, 1 ns bins, 1 second run-time

accesses are handled by the main memory, which causes the two main peaks on the graph.

Figure 4.3 shows the Instructions-Per-Cycle (IPC) for a set of short BBench runs as the memory latency is varied. BBench is run from a checkpoint and runs for a total of 0.5 seconds, as opposed to running the full BBench benchmark. This allows us to look at the impact on parameters such as IPC without running the complete benchmark, but does not allow correlation with the final BBench score. The results show that as the latency increases, the IPC decreases. This confirms that the CPU is a latency sensitive

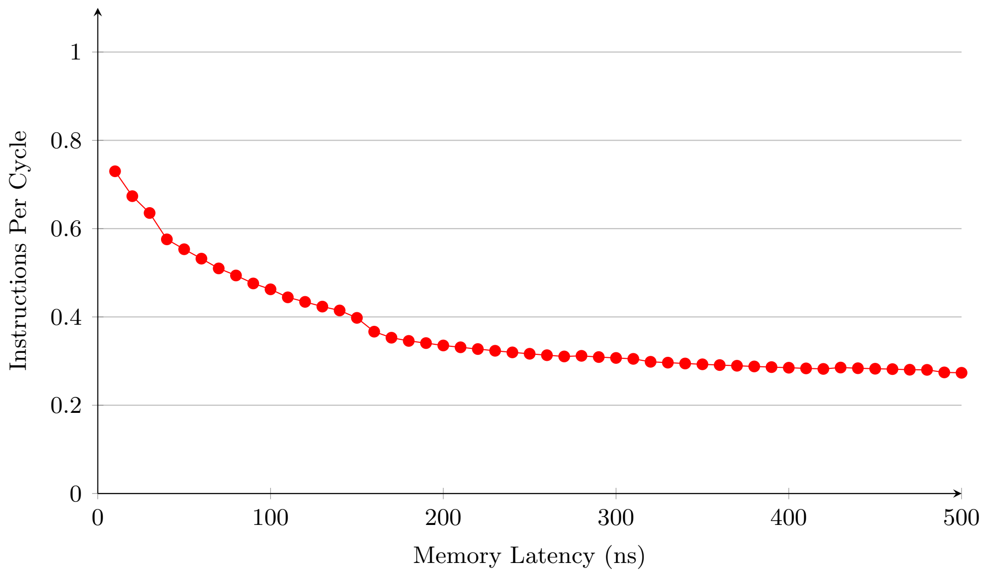


Figure 4.3: IPC as memory latency varies for a 0.5s BBench run

device, and that it therefore needs to be provided with a low latency to ensure proper operation, else the experienced performance suffers. Later results show that the latency increases by more than 300 ns when a GPU is introduced into the system (we show an increase of 328 ns for one frame). This motivates the need for limiting the impact the GPU can have on the memory system, and provides part of the mechanic for trading off GPU performance for CPU performance.

In this first section we focus on BBench as our CPU workload as it provides a realistic user-focused use case representative of a task an end user would perform, and a set of SimPoints (Sherwood et al., 2002) is created for the workload. The set of BBench SimPoints is executed for a number of different voltage and frequency operating points. For each operating point the change in performance and the energy consumption is measured.

In Figure 4.4 we present the trade-off between benchmark run time and the energy consumed by the CPU, shown relative to a CPU frequency of 1.7 GHz — the maximum operating frequency for the Exynos 5 Dual (Samsung, 2012). The energy is estimated by multiplying V^2f for each of the DVFS operating points by the run time, assuming a roughly constant load for the duration of the benchmark. This gives a high level estimate for the dynamic energy consumption, and simply serves to illustrate the energy saving potential for workloads such as BBench.

There is a non-linear relationship between the benchmark slowdown and the energy consumed, and the relationship is split into two key parts. For small reductions in CPU frequency there is a significant decrease in CPU energy consumption for a small increase in overall benchmark run time. For larger CPU frequency reductions, there are diminishing returns as the run time increases more rapidly than the energy savings.

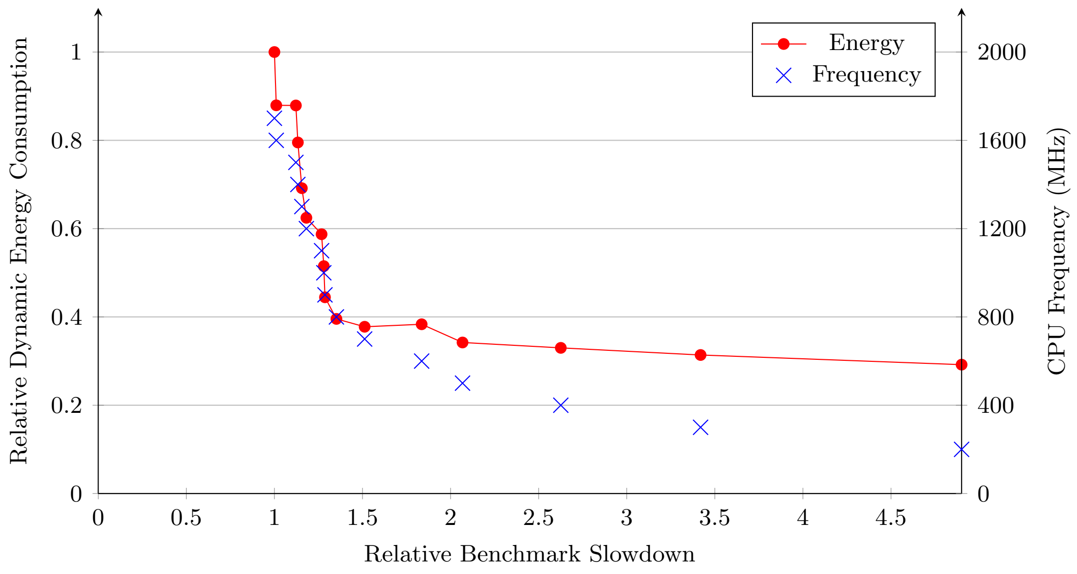


Figure 4.4: Energy consumption and CPU frequency as a function of system slowdown.

Small reductions in frequency allow large energy savings, with minimal performance loss, and therefore it is possible to trade-off the performance of the CPU for energy consumption efficiently within this range of frequencies. Please note that this range of frequencies is dependent on the particular workload, and is based on how memory bound the workload is.

The non-linear relationship between the performance and energy consumption of the workload occurs because the average memory access latency does not vary significantly as the CPU frequency is adjusted. Therefore, the benchmark spends a large proportion of time waiting for data from main memory, and the CPU idles, wasting energy. Modest reductions in the CPU frequency result in fewer wasted cycles whilst still performing a similar amount of work in a given time period. Once the CPU frequency has been reduced significantly, the CPU is unable to process the data at the rate that the memory can supply it and the run time increases significantly, resulting in poor energy-efficiency due to the high run time.

Taking the energy-delay product, we determine that the ideal DVFS point in terms of total energy consumption is at a CPU frequency of 800 MHz. Operating at 800 MHz results in an increase in run time of 35% and a decrease in energy consumed of over 60% relative to 1.7 GHz. In order to relate this reduction in energy and performance to QoE, Figure 4.5 shows the geometric mean for web page render times in BBench as the memory latency seen by the CPU is varied, and shows the relationship between CPI and web page render time. As the memory latency increases, the time taken to render each web page increases, as does the CPI of the CPU for a given frequency. This not only demonstrates that the CPU is able to operate more efficiently at lower frequencies due to the relative decrease in memory latency, but also provides us with a

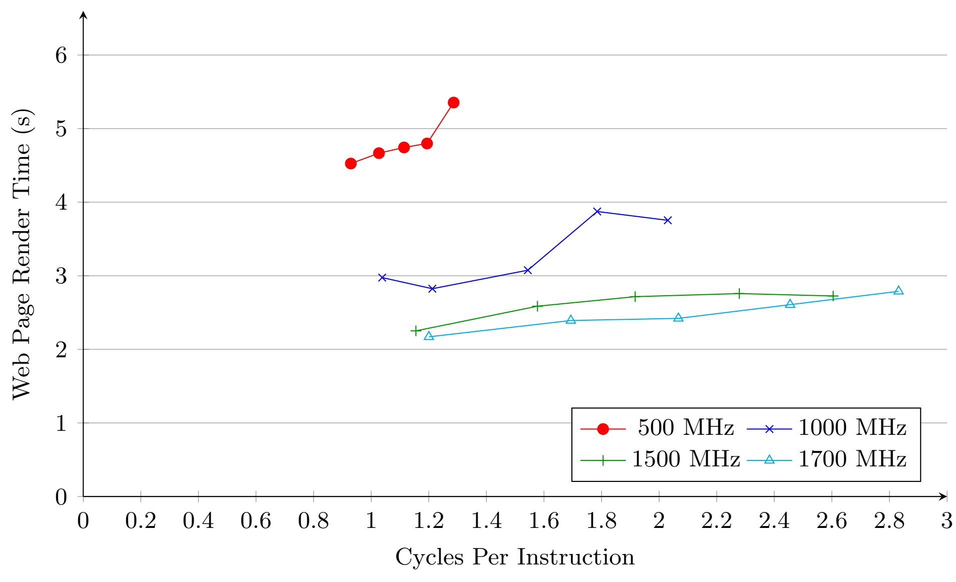


Figure 4.5: CPI vs. geometric mean of web page render times for BBench as CPU frequency varies.

link between low-level system metrics and higher-level user experience. It thus provides a QoE-specific reference for the results presented in this paper (similar to the link shown between MOS and QoS disturbance shown by Fiedler et al. (2010)) and allows QoE to be used as illustrated in Figure 3.1.

4.2 CPU Workload Frequency Sensitivity

In this section we observe how different workloads respond to scaling the CPU frequency. This helps us to understand which workloads respond to an increase in CPU frequency, and which do not, as well as providing insights into why this may be the case. The results in this section and the following two sections serve to answer our third research question: *How does CPU frequency affect performance and user experience for a spectrum of workload scenarios, ranging from compute-bound to memory-bound?*

In order to investigate the frequency sensitivity for various workloads, we run experiments on the Odroid XU3 development board from Hardkernel (Hardkernel, 2015b). This board utilises a Samsung Exynos 5422 which contains a total of eight CPU cores: four ARM Cortex A15 cores and four ARM Cortex A7 cores. For our experiments we focus solely on the A15 cores, and in fact only observe the single core operation in order to ensure that our results are not affected by multi-threading and coherency issues. Therefore, all other cores in the system are disabled at the kernel level. We adjust the frequency of the A15 core from 1.2 GHz to 2 GHz in 100MHz steps.

For these experiments, we run the entire workload at a fixed frequency and hence observe how the workload as a whole responds to changing the frequency. This allows us to

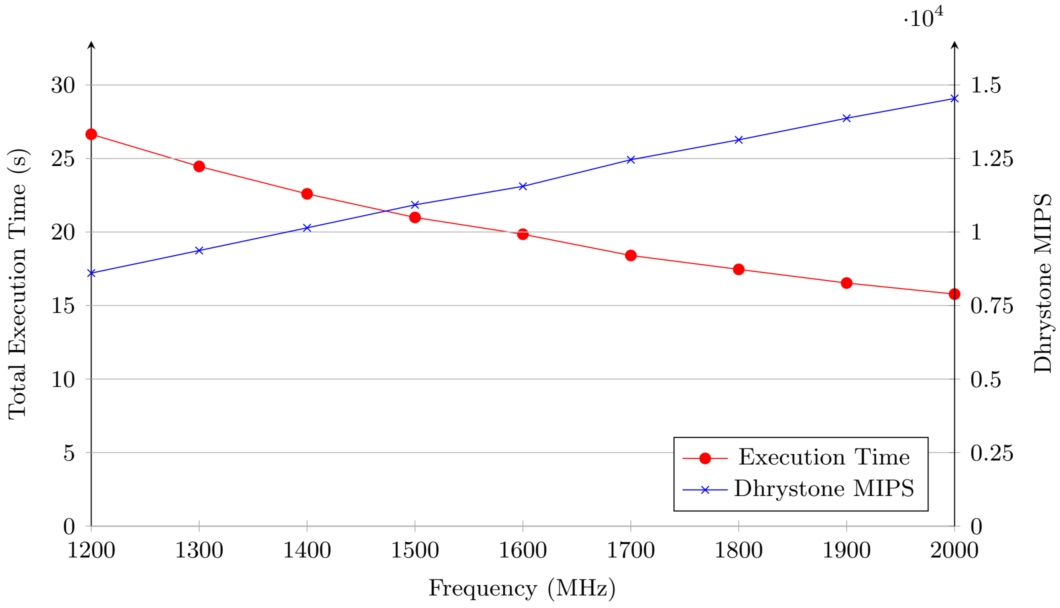


Figure 4.6: Execution time and DMIPS for Dhrystone benchmark running on the Odroid XU3 as the CPU frequency is swept.

identify the single best frequency for a particular workload, but therefore does not take into account any workload phases. However, the information gained in this section is enough to determine which workloads have a sensitivity to CPU frequency, and which do not.

We perform CPU frequency sweeps for the following workloads: Dhrystone, Memcpy, RLBench, AndEBench, AnTuTu. An overview of these workloads is presented in the literature review in Section 1.3.3. Each workload is run a total of 10 times for each frequency, and we present the results averaged across 9 iterations, discarding the first. The first iteration is discarded due to measurement issues on the development board due to which the first iteration of the workload for each frequency often contains large measurement errors.

4.2.1 Compute-Bound Workloads

We begin by investigating how the CPU frequency affects the performance of compute bound workloads. We consider Dhrystone and AndEBench which are both typically compute-bound workloads. Both Dhrystone and AndEBench are designed to measure the performance of the CPU, and therefore aim to fit within the caches of the CPU, and have minimal interaction with the memory. For example, Dhrystone will only access the main system memory during start up when it is initially launched. AndEBench has a little bit more iteration with the memory as it runs as a graphical application within Android, and therefore must deal with screen updates. However, the main benchmark code for AndEBench remains compute bound.

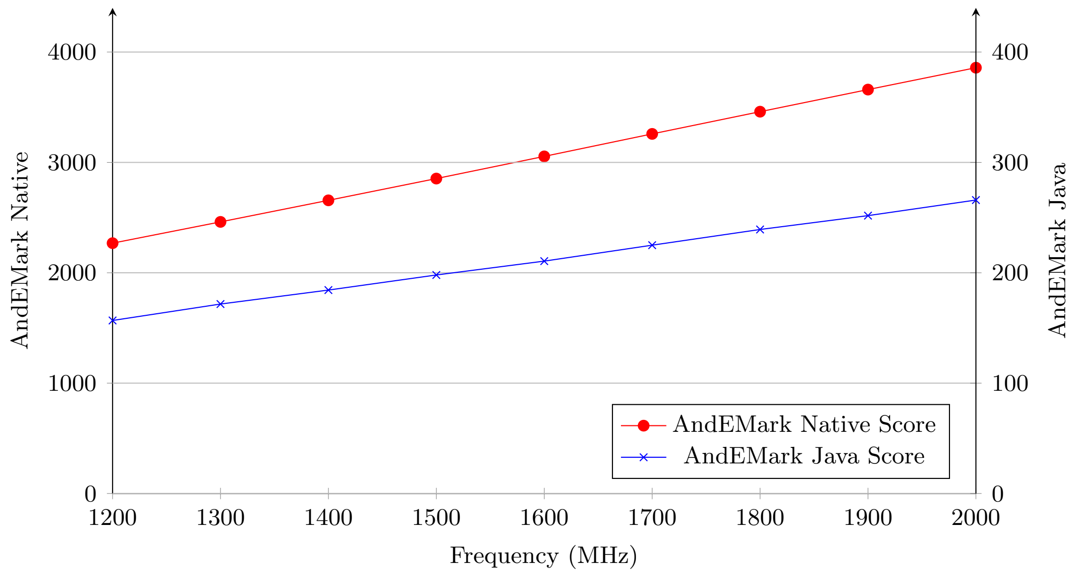


Figure 4.7: AndEMark score for AndEBench benchmark running on the Odroid XU3 as the CPU frequency is swept.

Dhrystone performs a fixed amount of work, and measures the time taken and the Dhrystone MIPS achieved. We present the time taken for the workload to execute, as well as the total DMIPS as calculated by the benchmark in Figure 4.6. As the frequency is increased, the run time decreases non-linearly—a doubling in frequency results in a halving the run time, or doubling the performance. If we observe the DMIPS for the benchmark, we notice that there is a linear increase with respect to the CPU frequency. The linear scaling of DMIPS with frequency demonstrates that the workload is compute-bound, and has minimal interactions with the rest of the memory system. On average, the benchmark achieves 7.26 Dhrystone MIPS per MHz (DMIPS/MHz), with a standard deviation of 0.05 DMIPS/MHz. This small standard deviation clearly demonstrates the linear scaling of this compute-bound workload with CPU frequency.

Next, we look at AndEBench which measures the performance of native and Just-In-Time (JIT) code on Android devices. The benchmark presents two scores, AndEMark Native and AndEMark Java, which measure the native and Java performance respectively. We present the results of a CPU frequency sweep for AndEBench in Figure 4.7. As was the case with Dhrystone, the performance of the workload scales linearly with the CPU frequency, and again demonstrates that compute-bound workloads scale well with the CPU frequency. When we observe the native AndEMark scores, we get an average of 1.91 AndEMark/MHz with a standard deviation of 0.015 AndEMark/MHz, clearly demonstrating the linear relationship between the score and the CPU frequency. A similar linear relationship is observed for the Java scores which have an average of 0.132 AndEMark/MHz with a standard deviation of 0.00072 AndEMark/MHz. Please note, we intentionally do not present run time numbers for AndEBench as the workload scales the amount of work done in order to run within approximately one minute.

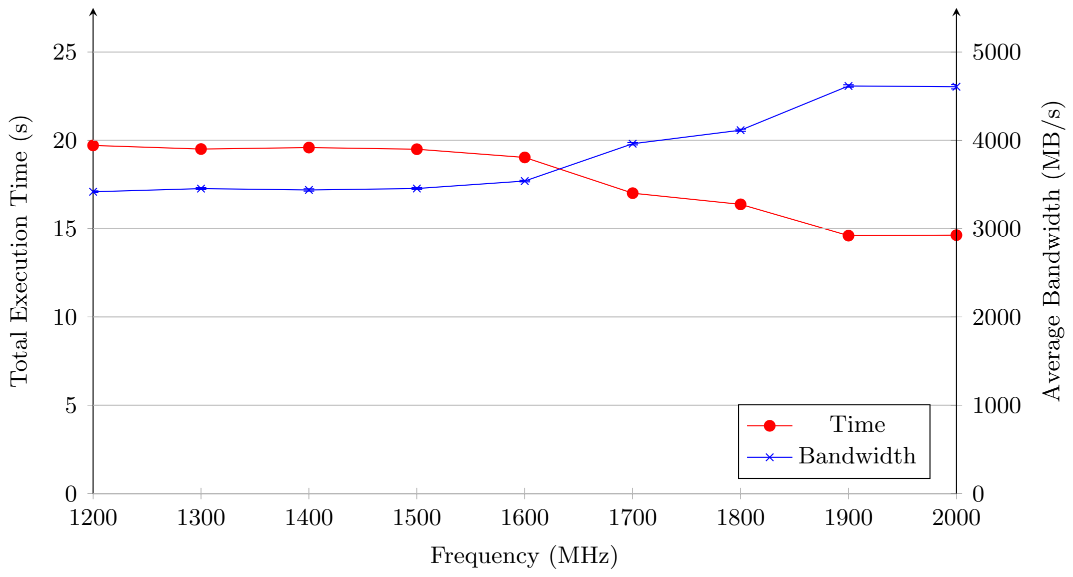


Figure 4.8: Execution time for Memcpy benchmark running on the Odroid XU3 as the CPU frequency is swept.

Therefore, all runs of the workload at different frequencies take roughly the same length of time to execute.

We have demonstrated that the performance of compute-bound workloads scales linearly with the CPU frequency. Therefore in order to obtain the highest performance for a particular workload, it is important to run it at the highest available frequency. Should the energy consumption be an issue, or simply if the highest level of performance is not required, the CPU frequency can be adjusted to give workload performance within the desired range, assuming a priori knowledge of the workload.

4.2.2 Memory-Bound Workloads

In the previous section, we observed how compute-bound workload scale with the CPU frequency, and demonstrated a linear relationship between the performance of the workload and the frequency of the CPU. In this section, we look at memcpy which is a mostly memory bound workload—for very low CPU frequencies this workload can be compute-bound as can be seen in upcoming Figure 4.10. This workload copies data from one memory location to another whilst measuring the bandwidth achieved, and therefore spends the majority of the execution reading from or writing to memory. We configure Memcpy to copy 64 MB of data a total of 1000 times, and measure the overall time taken and average transfer bandwidth achieved. It is important to note that the bandwidth measured is based on the data transferred not the bandwidth to and from memory, which is roughly double the data transfer bandwidth.

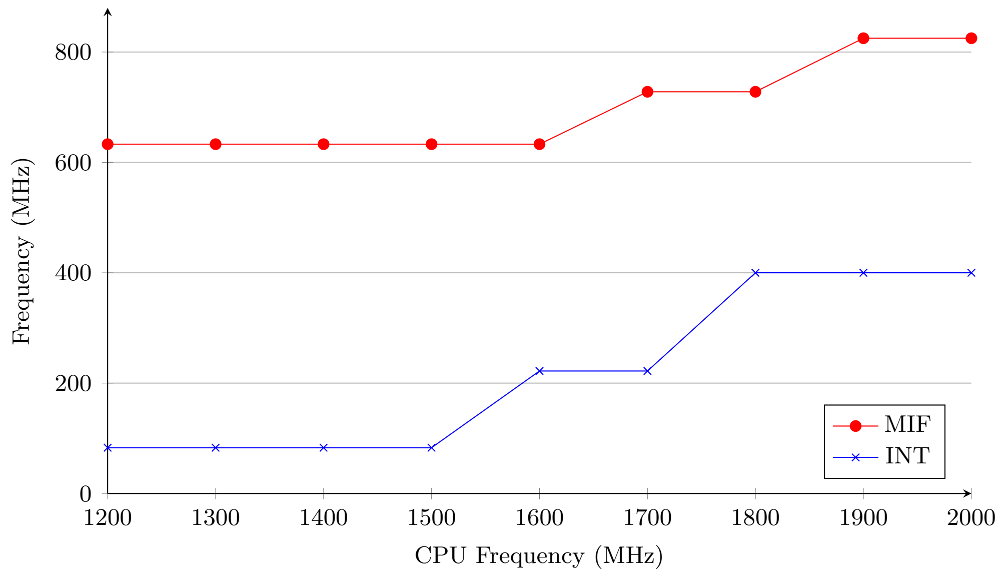


Figure 4.9: Relationship between A15 CPU frequency and other clock sources for the Odroid XU3

We sweep the CPU frequency and present the run time and the average bandwidth, for Memcpy in Figure 4.8. Given that Memcpy is a memory bound workload for sufficiently high CPU frequencies, we would expect to see little to no change in run time or bandwidth as the CPU frequency is varied. However, we observe that for frequencies greater than 1.5GHz the performance of the workload on the Odroid XU3 improves. This is due to the memory clock sources being tied to the CPU clock source, and therefore the usable memory bandwidth increases as the CPU frequency is increased. Figure 4.9 shows the relationship between the CPU frequency and the memory clock sources. When these frequency increases are compared with the run time and bandwidth shown in Figure 4.8, it is clear that the improvements in workload performance correlate with these frequencies. However, it is apparent that the performance of the workload does not improve for the 1.2 GHz to 1.5 GHz range of frequencies, during which the memory clocks are not adjusted.

In order to demonstrate how the performance of Memcpy varies with CPU frequency, we present results from a gem5 simulation. The gem5 simulation is configured such that the memory frequency remains constant and that only the CPU frequency is adjusted. We run the gem5 simulation for a larger range of frequencies, 100 MHz to 2000 MHz in 100 MHz steps, to show how Memcpy transitions from a compute-bound state to a memory-bound state. As the gem5 simulator is significantly slower than real hardware, we configure the benchmark to transfer the same 64MB of data, but only perform 10 iterations, as opposed to the 1000 on the real hardware. The results are presented in Figure 4.10. First of all, it is important to note that the workload is compute-bound for frequencies around 100 to 300 MHz, and then transitions from being compute-bound to memory-bound as the frequency is increased. The workload is compute bound for

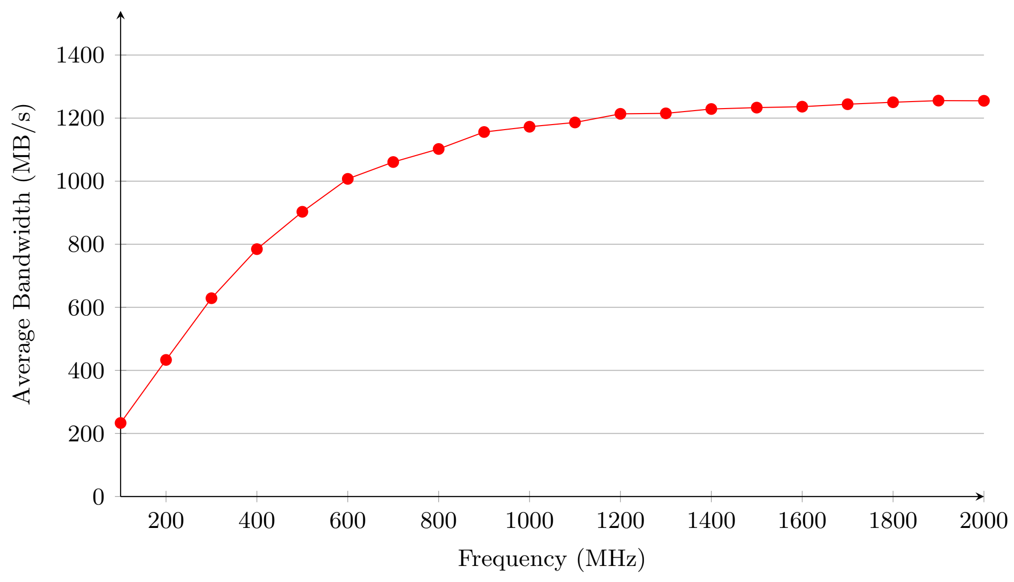


Figure 4.10: Average memory bandwidth for Memcpy benchmark running on the gem5 simulator as the CPU frequency is swept.

low frequencies as the ratio of CPU frequency to memory frequency and latency is a lot smaller, and therefore the CPU becomes the bottleneck in the transfer. For the range of frequencies available for the A15 core on the Odroid XU3, the workload is clearly almost completely memory-bound as increasing the CPU frequency has little to no effect. This further reinforces that the performance improvements seen on the Odroid are due to the increased memory frequency.

We have shown that a memory bound workload has minimal sensitivity to the CPU frequency. For this reason, a memory-bound workload should be run at a low CPU frequency, as increasing the frequency only results in increased energy consumption with close to no performance improvement. Naturally, the frequency should not be reduced too significantly, as the workload will become compute-bound.

4.2.3 Real Use Cases

In the previous two sections we covered compute- and memory-bound workloads. It is uncommon for an entire mobile workload to be compute-bound or memory-bound, and therefore we need to consider more typical mobile workloads. These workloads are neither compute- or memory-bound for the entire execution.

The first workload we consider is RedLicense Labs SQLite Benchmark (RedLicense Labs, 2012), which measures the SQLite performance of an Android system. This workload is comprised of many phases, ranging from compute-bound to memory-bound, and each phase covers a different type of SQL transaction. The benchmark measures the time for each phase, as well as the overall time to complete all phases of the workload. Due to

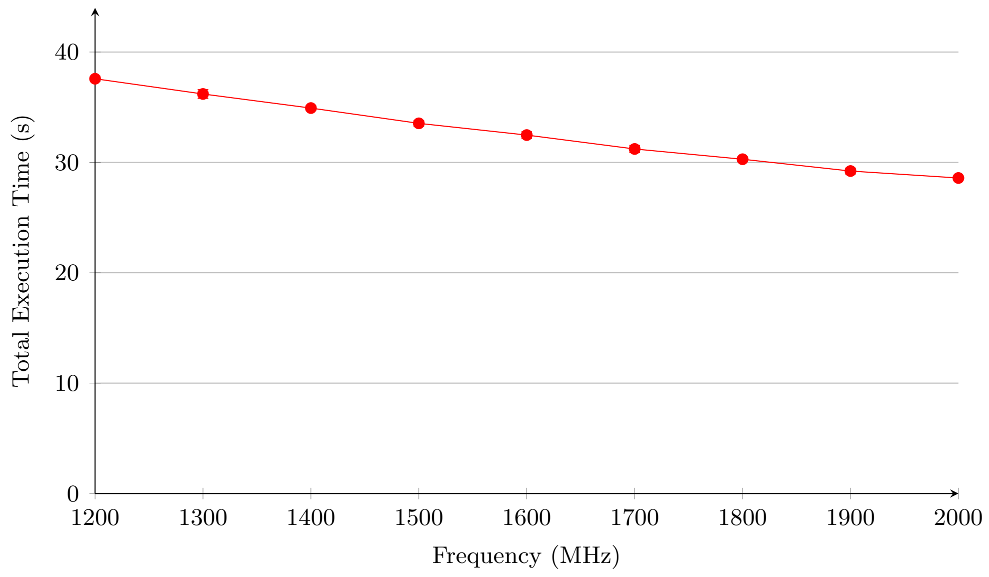


Figure 4.11: Execution time for RedLicense Labs SQLite benchmark running on the Odroid XU3 as the CPU frequency is swept.

the different phases of the workload, we expect the workload to show some sensitivity to CPU frequency due to the Compute-bound phases. However, the response should be limited by the memory-bound phases of the workload.

We present the overall benchmark run time for the CPU frequency sweep in figure 4.11. It is apparent that the benchmark is not memory bound, as the time taken to run the benchmark decreases as the frequency of the CPU is increased. However, the increase in benchmark performance is not a linear function of the CPU frequency and therefore the benchmark as a whole is not compute-bound either. Relative to 1200 MHz, we observe a 24 % reduction in benchmark run time at 2000 MHz, which is a 67 % increase in CPU frequency. This demonstrates that the performance of the benchmark is a non-linear linear function of the CPU frequency, and is hence not compute-bound.

Secondly, we look at the AnTuTu 4 benchmark (Antutu, 2015) which measures the performance of different in-system components. This workload not only stresses the CPU, but also profiles the performance of the GPU for 2D and 3D applications, as well as performance of the DRAM. Therefore, the workload has many different phases, each of which place differing strains on the system. For example, when benchmarking the performance of the GPU, there is less strain placed on the CPU as the GPU is doing the majority of the work. However, the CPU is still required to coordinate all of the system components and must pre-process the individual frames before these are passed to the GPU for rendering. We present the overall score obtained from the benchmark in Figure 4.12, which is calculated by summing all individual test scores. The figure also includes the score per MHz, which is an indication of how well the benchmark scales with frequency—a flat line indicates that the workload scales linearly with frequency.

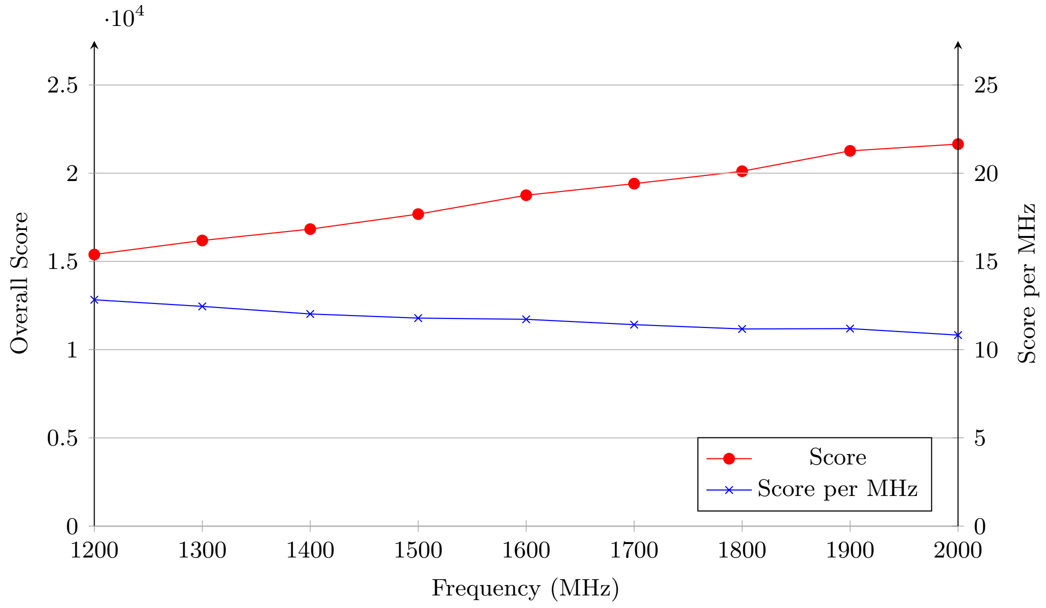


Figure 4.12: Overall benchmark score and score per MHz for AnTuTu 4 benchmark running on the Odroid XU3 as the CPU frequency is swept.

The overall score scales non-linearly with CPU frequency, as can be seen by the negative gradient of the score per MHz line, and exhibits characteristics which are neither memory-bound nor compute-bound. However, as we shall subsequently show, the individual components of the benchmark can be either compute-bound or memory-bound.

We present the scores for the CPU-specific sub-benchmarks of AnTuTu in Figure 4.13. These phases of the application focus purely on the CPU performance, and therefore have minimal interaction with the rest of the system, and rarely access the DRAM. For this reason, these phases of the application scale linearly with CPU frequency and are compute-bound across the range of frequencies tested. Therefore, these phases of the workload will scale well with the CPU frequency.

We show how the 2D and 3D graphics benchmark performance varies with CPU frequency in Figure 4.14. The 2D graphics score has minimal sensitivity to CPU frequency, but does show a minor increase in performance as the frequency is increased. This occurs as the CPU is not required to do much pre-processing for 2D graphics workloads, and therefore the majority of the performance is due to the GPU and memory system. This part of the benchmark appears to be memory-bound, but we assume that it is in fact either GPU-bound or rate limited, and hence the measured score saturates. If the 2D graphics phase were memory-bound, we would expect to see a similar characteristic to that seen for Memcpy in Section 4.2.2 due to the memory frequency being increased in tandem with the CPU frequency.

The 3D graphics performance, on the other hand, relies much more heavily on the CPU to prepare the data for the GPU. Therefore, it shows a greater sensitivity to the CPU frequency. However, it does not respond linearly to to the CPU frequency and therefore

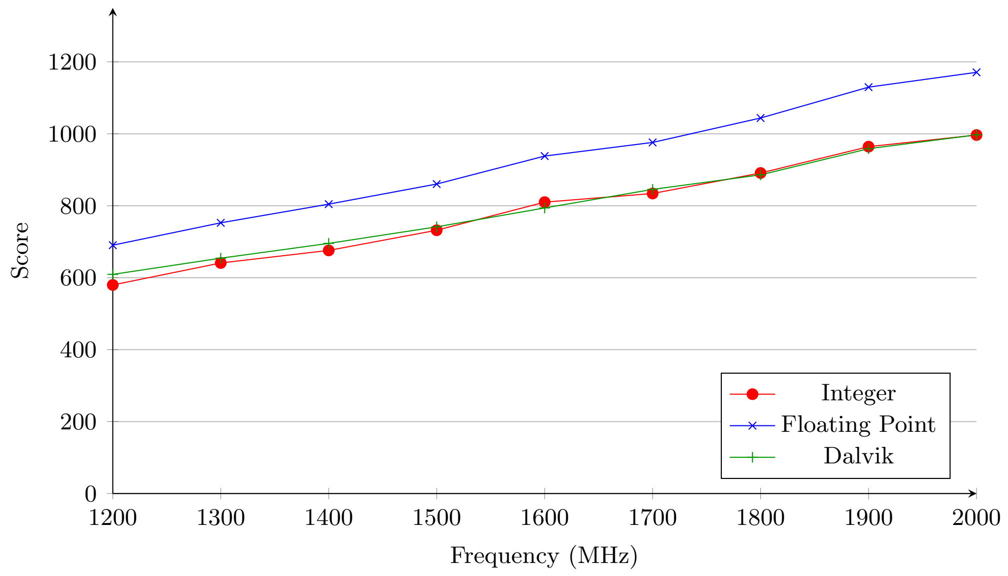


Figure 4.13: CPU scores for AnTuTu 4 benchmark running on the Odroid XU3 as the CPU frequency is swept.

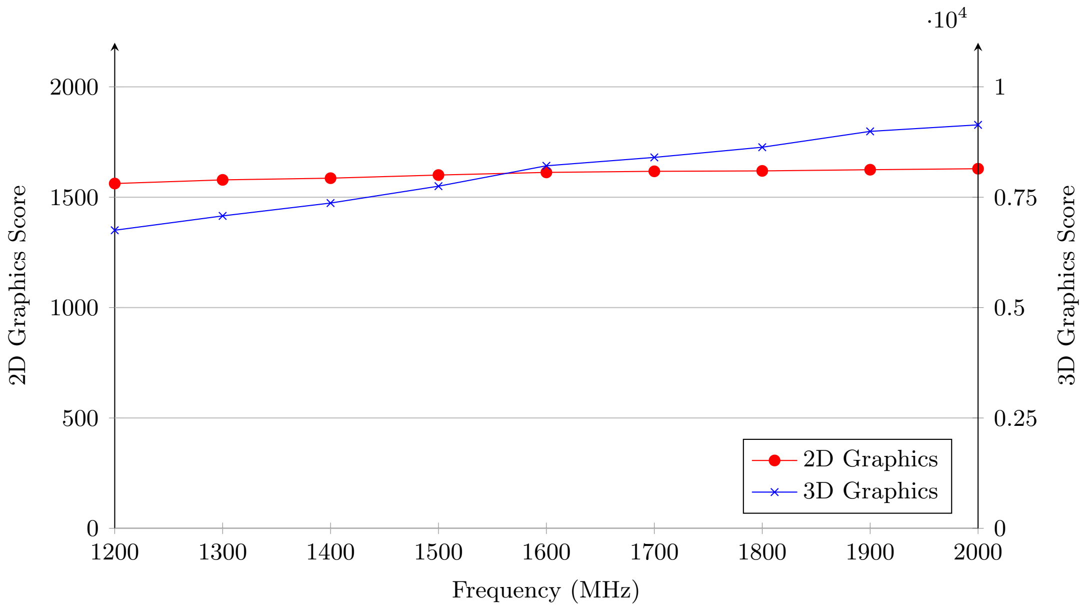


Figure 4.14: GPU scores for AnTuTu 4 benchmark running on the Odroid XU3 as the CPU frequency is swept.

is not compute-bound. The CPU is required to prepare the frames for the GPU, and then passes these to the GPU to be rendered. Typically, the CPU will process one or more frames ahead of what is being rendered on the GPU, but it not able proceed beyond this. Therefore, no matter how fast the CPU is able to prepare the frames for the GPU, it will be limited by the GPU performance. Therefore, the 3D graphics performance is a function of both CPU and GPU processing ability. It is worth noting that due to the communication between CPU and GPU, the memory will also come into play for a typical mobile system.

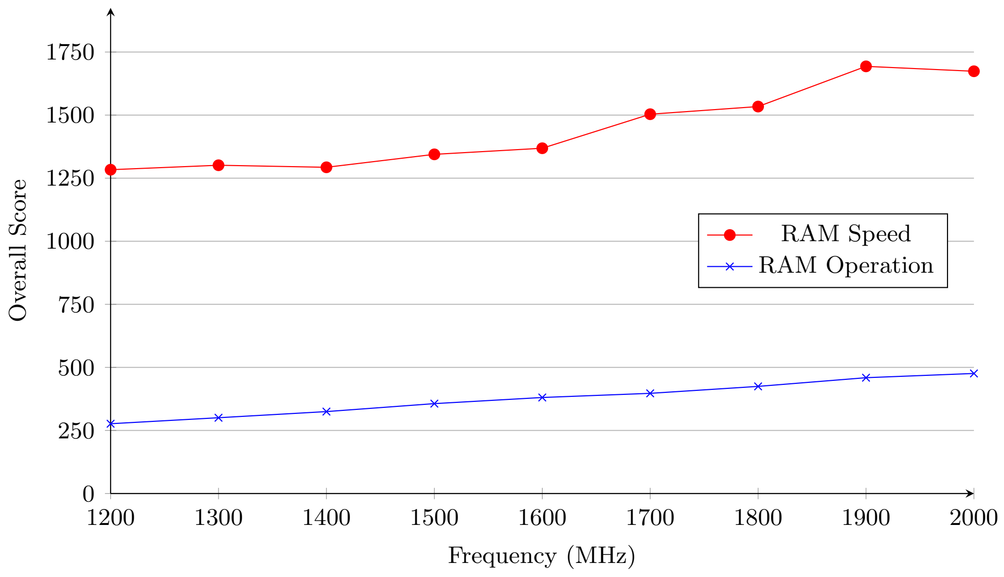


Figure 4.15: RAM scores for AnTuTu 4 benchmark running on the Odroid XU3 as the CPU frequency is swept.

Finally, we show the RAM scores from AnTuTu in Figure 4.15. The RAM Speed score shows the amount of bandwidth that the CPU was able to achieve when accessing the RAM, whilst the RAM Operation score shows how effectively the CPU was able to use the data from the RAM. Please note that the units of both of these measurements are unclear and are not presented by the benchmark. The RAM operation score is purely dependent on the CPU frequency, and scales linearly with frequency as it is compute bound. The RAM speed score exhibits a similar trend to the Memcpy bandwidth results presented in Figure 4.8, and is memory-bound. This is expected as both this phase of AnTuTu and Memcpy measure the bandwidth they are able to achieve when accessing the DRAM.

In this section we demonstrated that workloads often have phases in which they can be compute- or memory-bound, as well as phases that sit between these two extremes. It is important to understand the type of workload in order to determine the most optimum frequency at which to run the CPU.

4.2.4 Workload Analysis Summary

We have shown that there are different types of CPU workloads, each of which exhibit varying responses to the CPU frequency. Compute-bound workloads scale linearly with the CPU frequency, as they have minimal reliance with other components in the system. For these workloads, in order to achieve the highest level of performance, it is vital to run them as fast as possible, at the expense of higher energy consumption. We have also considered memory-bound workloads, which do not scale significantly, if at all, when the CPU frequency is adjusted. When possible, these workloads should be run at the lowest frequency that can sustain the required workload throughput. This, in general, results in the lowest energy consumption whilst satisfying performance requirements. Finally, we considered more complex workloads, which have phases which are both memory-bound and compute-bound, as well as phases which fall into neither of these categories. For the compute-bound phases, the CPU frequency should be increased, whilst in the memory-bound phases it should be reduced. It is more complex choosing the ideal frequency for the other phases of such workloads, as an intermediate frequency may offer a similar level of performance to a higher frequency, and therefore should be favoured over higher frequencies.

The results from this workload analysis help us to understand how different CPU workloads perform under various constraints, and hence we are able to understand when increasing the frequency of the CPU will improve the performance of the workload. This is important to understand when optimising the system for performance and power, but it is even more vital when optimising for user experience. For example, if it is clear that it is not possible to further improve the performance of a particular workload by increasing the CPU frequency, then the same applies to the user experience. Conversely, if the user experience is satisfied and we know that a reduction in frequency will significantly reduce the experience delivered by the workload, then we can avoid this undesired user experience reduction.

4.3 Fine-Grained Workload Analysis

In the previous section, we observed how workloads responded to CPU frequency, but only considered workloads as a whole, i.e., running at a fixed frequency for the entire execution of the workload. In modern devices DVFS is used to adjust component frequency on the fly based on demand. This is most commonly used for the CPU which can consume a large amount of energy. Typically, a software DVFS governor is used which is able to adjust the CPU frequency based on some pre-determined rules. It is, however, important that both the correct governor is chosen and that the governor is able to operate at a sensible rate to save the most energy.

In this section we look at how well workloads respond to DVFS scaling using fine-grained forking of the gem5 simulator. We begin by presenting and discussing our gem5-based analysis framework, highlighting the key advantages and disadvantages of the approach, and then present the results of the analysis for four mobile workloads. The workloads presented have been chosen to cover a large spread of typical applications and workload characteristics, as well as demonstrating that the analysis framework is working correctly.

4.3.1 Forking the gem5 Simulator

Typical workloads are comprised of multiple phases of execution, each of which places differing demands on the CPU and memory system. Some of these phases can be compute-bound, as they are comprised of small loops which have minimal interaction with main memory, and only require data local to the CPU of closely coupled caches, whilst others can be heavily memory-bound. Therefore, there is often no single ideal DVFS operating point for a given workload, and the CPU frequency must be adjusted according to the particular workload phase.

In this section, we present and use a gem5-based analysis framework which can be used to determine which phases of a workload are compute-bound, memory-bound or neither, and assist in the determination of an ideal operating frequency for each workload phase. This analysis framework is able to operate at different workload sampling granularities, and is hence able to simulate the operation of a DVFS governor working at different governing intervals.

The gem5 simulator supports checkpointing, i.e., capturing the entire state of the system such that it can be resumed at a later point in time. These checkpoints allow gem5 users to start experiments at a known, fixed point in the execution, and thereby have comparable results without the overhead of booting the guest system multiple times. These checkpoints can be captured at any point during the execution, and contain the whole simulation state, except for the state of the caches. In order for the simulator to create a checkpoint, all in-flight messages must be drained, that is, they must reach their end-points, before the checkpoint is captured. The act of draining the caches and stopping all in-flight transactions creates a perturbation in the system, and can have a large impact on the simulation results. For example, as the cache state is lost during checkpointing, the subsequent simulation will start with cold caches which must be warmed before the simulation results are reliable. This is a major issue when trying to perform a set of very short time-span experiments. Hence, we opt not to use checkpointing for fine-grained analysis of workloads.

The analysis framework we present here does not rely on checkpointing, but instead relies on forking. Forking is the process of copying the entire state of an application, to create an identical copy of the application. This can be performed without any loss of state, and can occur at any time. Therefore, it is possible to run a simulation and create an exact copy of the simulation without perturbing the state of the simulated system by forking the entire simulation. It also avoids the draining issues associated with checkpointing. This allows the simulated system to be copied and parts of the simulation to be changed without impacting the accuracy of the simulation.

We provide an example of our forking-based analysis framework in Figure 4.16, and use it to explain how we utilise forking to analyse workloads at a fine granularity. We run

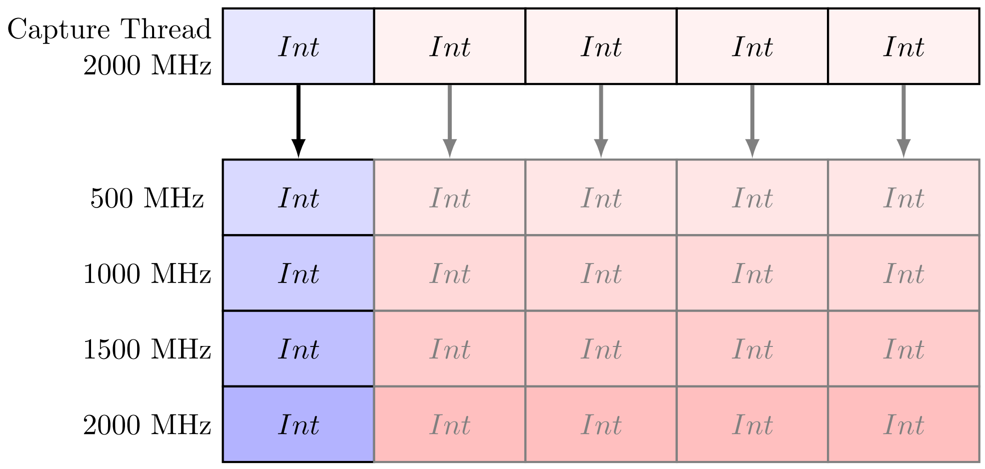


Figure 4.16: Overview of the forking infrastructure.

a simulation in gem5 using the same configuration as we wish to analyse, and run it at the highest frequency we wish to analyse. This is shown as the Capture Thread in Figure 4.16. The Capture Thread is forked every time interval, shown with *Int*, which is measured in simulation time. The thread is forked once for each frequency we wish to investigate. In the figure, we wish to evaluate the workload at four different frequencies, and hence fork the Capture Thread four times. Each of these forked instances is modified to adjust the CPU frequency to the desired value, and then runs for one interval before the simulation is ended. When the child simulation ends, it writes out the simulation statistics, and hence we are able to capture the statistics for each interval and frequency. It is important to note that the Capture Thread is not halted, and keeps running continuously.

Our simulation framework has advantages over other simulation based approaches, and even over real hardware. First of all, and most importantly, we are able to finely slice workloads without affecting their execution. Each of these slices in then evaluated using a different system configuration and can be directly compared each set of slices is synchronised to the Capture Thread. If we were to run the simulations independently and just write out the simulation statistics periodically, as is possible with the gem5 simulator, the simulated systems would gradually diverge. This makes it near impossible to synchronise and directly compare phases of the workload. The same applies to performing these experiments on real hardware. In our approach, the only limits are the granularity of the simulator forking, and disk space.

Secondly, we are able to investigate any number of frequencies or settings for a workload with this framework. If the simulation allows a parameter to be changed, then it can be adjusted as part of the forking framework. Therefore, this framework is able to be extended to assess the impact of more than just CPU frequency.

In the above example, we stated that the forking interval is specified in terms of simulation time. However, the forking interval can be specified in simulation ticks, akin to time, or number of executed instructions. Each method has inherent advantages and disadvantages. Specifically, when performing time-based forking of the simulator, it allows the idle periods to be captured, which is not the case when performing instruction-based forking, as the CPU will execute no instructions during an idle period. This also applies to periods of low CPU activity. Therefore, if idle periods are to be properly considered, time-based sampling is to be used. However, the gem5 simulator is event based, and some of the CPU statistics are only incremented when the CPU leaves a particular state. Therefore, when using time-based sampling, some statistics may be incorrectly attributed to a particular fork, and need to be backfilled in order for them to be valid.

For our experiments, we choose time-based sampling, and backfill some of the CPU statistics in order to obtain the correct measurements. As we are looking at CPU DVFS states, it is important that we consider the effects of idle periods. These are times that the CPU can either be powered off altogether, or can run at the lowest DVFS state in order to save energy. Time-based sampling of workloads also allows us to extend the forking infrastructure to simulate different CPU governors offline. We evaluate the energy consumption of the different DVFS governing strategies using in-house, IPC-based regression models to predict the energy consumed by the CPU for each forking interval. Our models are gathered using a methodology similar to that presented by Walker et al. (2015).

In later chapters, we use the results obtained by forking the gem5 simulator to run a DVFS governor offline. We we have profiled the workload, we are able to switch between the different DVFS states in order to simulate the effects of a DVFS governor running on the system. In fact, as we have effectively run the workload at multiple different frequencies concurrently, we have oracle knowledge regarding the best frequency to run a particular workload sample at. Therefore, we are able to choose the most energy efficient operating points, or those that give us the desired performance.

Only the parent gem5 simulation is allowed to fork, and therefore this serves as a synchroniser for all of the other gem5 simulations. Each forked instance of the simulator runs from the point of forking for a fixed interval, and therefore all instances forked at the same time, start with the exact same state. This avoids divergence issues which can arise from running the workload separately in multiple gem5 simulators or on real hardware. For example, an operating system has a scheduler which can make different process scheduling decisions at run time, and small perturbations in the system can cause the scheduler to make different scheduling decisions. Additionally, operating systems have periodic events, which run every time interval. By reducing the CPU frequency, the ratio of these processes relative to the workload being profiled increases, and therefore the workload runs would diverge. Both of these issues are addressed by the forking infrastructure presented here.

Each forked instance writes a full set of gem5 statistics to disk, and therefore a large number of statistics are available over time and for each CPU frequency. We post-process these statistics to extract information about the workload, as well as to implement offline DVFS governors.

4.3.2 Detailed Forking-Based Workload Analysis

In this section we use the aforementioned forking infrastructure to analyse typical workloads to determine why they scale they way the do, and to determine how much potential exists for DVFS. We use the forking infrastructure to profile four different workloads, which are Dhrystone, Memcpy, BBench and The Chase. We run each workload at four different frequencies: 500 MHz, 1000 MHz, 1500 MHz and 2000 MHz. The forking is performed every 100 us for Dhrystone, Memcpy and The Chase, and every 10 us for BBench. We run for a total of 2.5 seconds for the former workloads, and 10 seconds for BBench. The forking is synchronised to the 2000 MHz run, and therefore the lower frequency runs may skip small sections of the workload. However, given the large number of samples, the error in the overall result should be minimal.

In Table 4.2 we present the time scaling of the workloads as the frequency of the CPU is changed for all intervals for the workload. This demonstrates that the forking infrastructure is working as intended as the workloads scale as anticipated based on how compute-bound or memory-bound they are. Additionally, it gives an impression for the best frequency to run a particular workload at. For example, in the case of Dhrystone, the ideal frequency is clearly 2000 MHz as there is a linear scale in the run time for the workload. For Memcpy, the opposite holds true as there is minimal performance improvement as the frequency is increased. Both of these results are in line with the results presented in Sections 4.2.1 and 4.2.2. For a workload such as BBench we can see that there is a large performance improvement when moving from 500 MHz to 1000 MHz, but much smaller improvements when further increasing the frequency. This indicates that, on average, BBench does not require a particularly high CPU frequency, but also that the lowest frequency incurs a large performance penalty. Hence, at a high level, it appears that BBench should be run at around 1 GHz.

Dhrystone

In Section 4.2.1 we presented the Dhrystone benchmark, and demonstrated that it scales linearly with CPU frequency on real hardware. Specifically, we swept the CPU frequency from 1200 MHz to 2000 Mhz, and demonstrated that the Dhrystone performance in terms of DMIPS was a linear function of CPU frequency. We use Dhrystone to demonstrate that our forking infrastructure is able to reproduce the properties of the workload. We present MIPS values for Dhrystone at all four different frequencies in Figure 4.19. The

Workload	500 MHz	1000 MHz	1500 MHz	2000 MHz
Dhrystone	10.0	5.0	3.3	2.5
Memcpy	3.2	2.7	2.6	2.5
The Chase	5.5	3.5	2.8	2.5
BBench	19.1	12.9	11.0	10.0

Table 4.2: Run time in seconds for each workload for each frequency as estimated by the forking infrastructure.

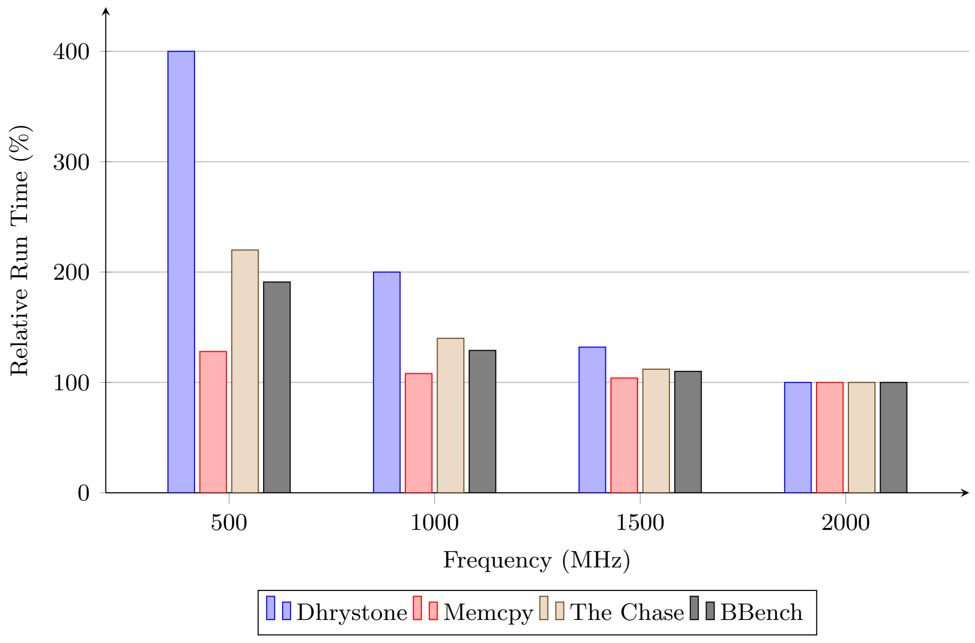


Figure 4.17: Run time, relative to 2000 MHz, for all four workloads evaluated using the gem5 forking infrastructure

data for the figure is generated by choosing a single frequency, e.g., 500 MHz, for each interval and determining the MIPS for each interval.

It is well documented that Dhrystone is an extremely compute-bound workload as the working set is small enough to fit in the caches. This is not only visible from the scaling in MIPS as is shown in Figure 4.19, but also from the run time scaling shown in Figure 4.17. Both of these figures show that the workload is able to scale linearly for the entirety of the capture execution, and that the workload is likely comprised of two phases, both of which are able to scale linearly with CPU frequency. Additionally, we verified that Dhrystone does not access the main memory and only accesses data from the caches. Figure 4.18 clearly shows the same trend as Figure 4.19 as the workload has very consistent operation and is comprised of a small number of phases.

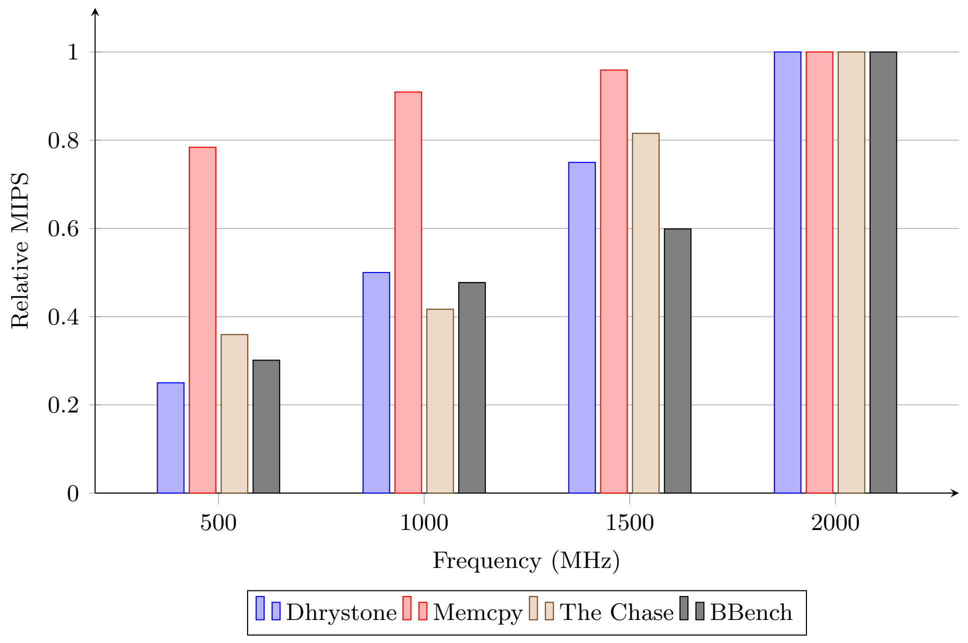


Figure 4.18: Average MIPS relative to 2000 MHz for all four workloads evaluated using the gem5 forking infrastructure

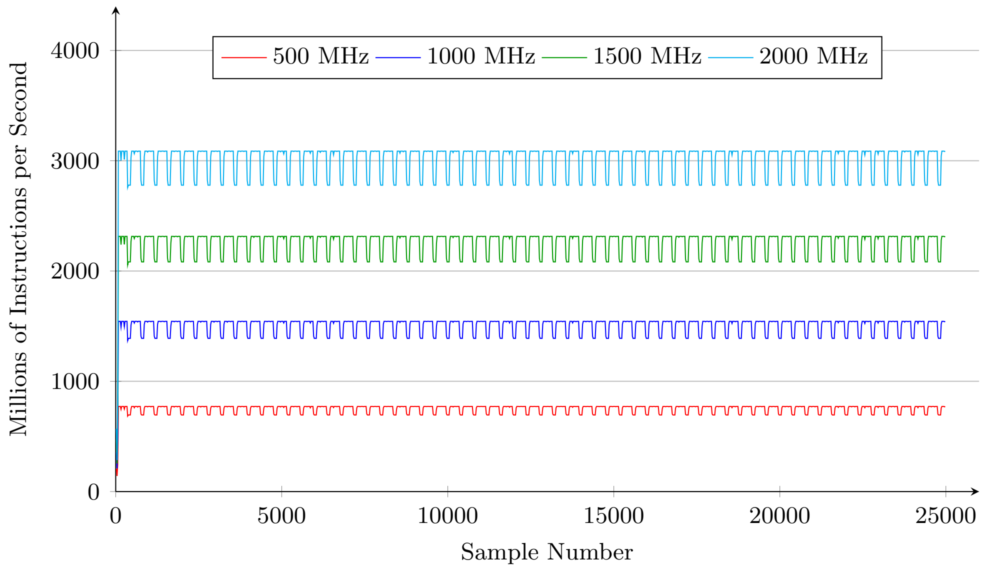


Figure 4.19: MIPS for Dhrystone for the four frequencies explored using the gem5 forking infrastructure

Our results for Dhrystone match those from the real hardware presented in Section 4.2.1. These results were captured by forking the gem5 simulator for a total of 25000 intervals, each of which was simulated at four different frequencies. As we have the same scaling results as we would expect from the real hardware, we can conclude that the forking framework is able to accurately reproduce a compute-bound workload.

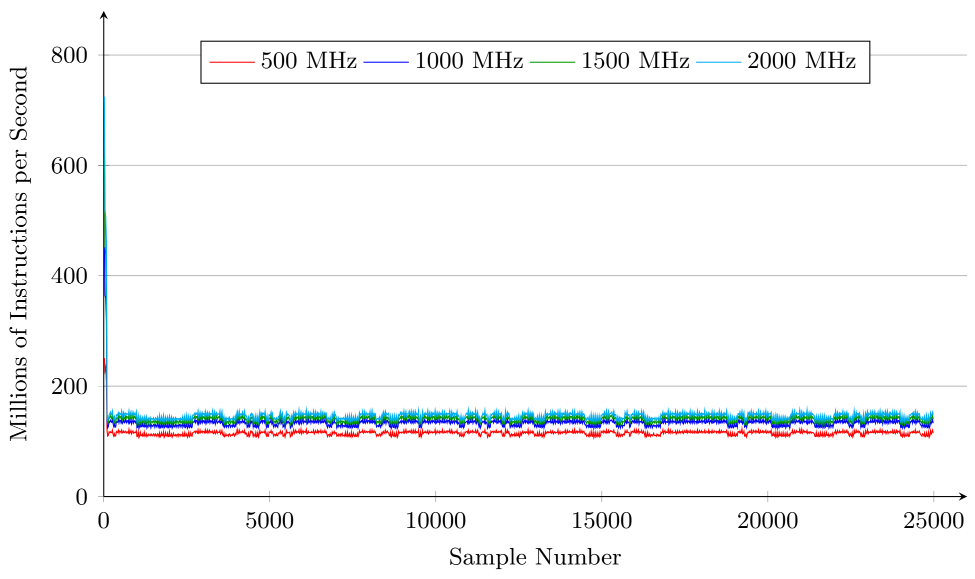


Figure 4.20: MIPS for Memcpy for the four frequencies explored using the gem5 forking infrastructure

Memcpy

Next, we observe the operation of Memcpy, which was originally presented in Section 4.2.2. This is a memory-bound workload, and therefore we expect close to no performance scaling as the frequency of the CPU is increased. We present the MIPS over time for Memcpy in Figure 4.20. Aside from a peak at the start of the benchmark caused by loading the benchmark and setting it up, it is clear that there is little improvement at different frequencies. Higher frequencies perform marginally better than lower frequencies, but the effect is extremely minor. We are able to see the same effect in the relative overall MIPS presented in Figure 4.2, in which the 500 MHz run takes slightly longer than the higher frequency runs, but there is little change between the higher frequency runs.

Referring back to Figure 4.18 we observe that changing the CPU frequency from 2000 MHz to 500 MHz results in only a 27% decrease in average relative MIPS across the workload. Therefore, for a workload such as Memcpy, it does not make sense to increase the frequency above 500 MHz. This finding concurs with the results from Section 4.2.2.

The Chase

The Chase is a demo for the Unity graphics engine, and has been briefly discussed in Section 1.3.3.3. The Chase is a graphics workload which relies on both the CPU and the GPU. The CPU must pre-calculate each frame before passing it off to the GPU for rendering. For this reason, the performance of the workload is not purely limited by the GPU, but is also heavily reliant on the CPU, and great care must be taken to ensure

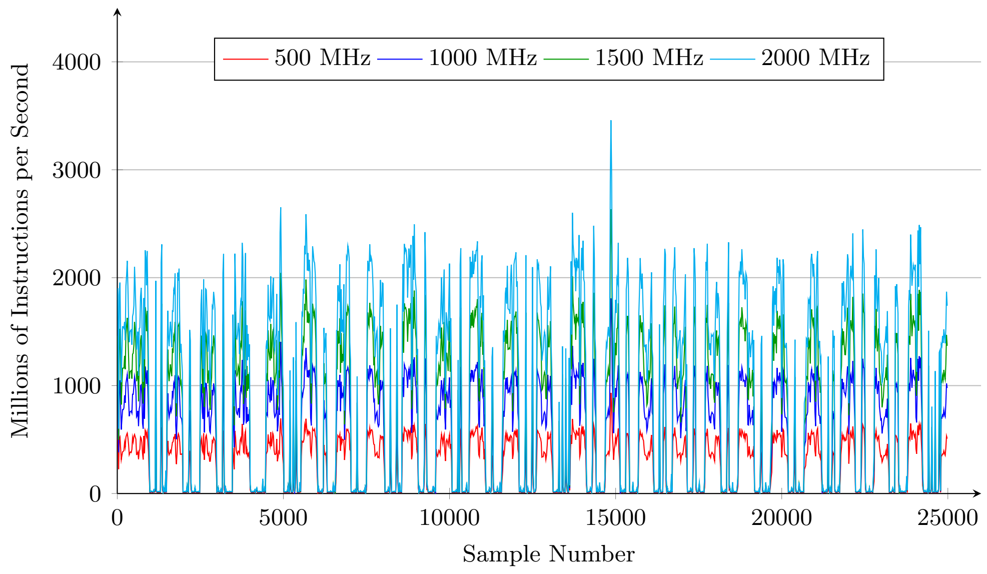


Figure 4.21: MIPS for The Chase for the four frequencies explored using the gem5 forking infrastructure

that both components are operating efficiently. We use a modified GPU driver which allows us to use the full GPU driver stack without having a GPU present in the system. This ensures that the CPU is still doing the same work as it would on a real device, without requiring the GPU to be present.

We present the results of our forking analysis in Figure 4.21, which shows how the MIPS vary for the different CPU frequencies tested. The peaks in the figure are the parts of the workload in which the CPU is preparing the data for the GPU, and the troughs represent the periods of time when the CPU is waiting for the GPU to render the frame. The shorter the time for a peak and a trough, the higher the frame rate of the GPU.

The figure demonstrates that it is possible to improve the performance of the CPU-specific portions of the workload by increasing the CPU frequency. This can be seen for each of the peaks within the workload, during which the MIPS roughly scales with frequency. However, the performance of the workload does not improve during the periods it is waiting for the GPU, and thus the overall scaling of the workload is limited by the portion of time that it must wait for the GPU. Therefore, for workloads which rely on both CPU and GPU performance, the overall performance of the workload is limited by the performance of the slowest of the two components. Specifically, if the GPU is too slow, it will limit the performance, whilst the CPU becomes the bottleneck when the GPU is sufficiently fast. The importance of the two components varies based on the components themselves, but also on the workload as more intensive workloads can place more strain on the components themselves.

The Chase is neither memory-bound or compute-bound, and instead has phases of high CPU activity, which are close to being compute-bound, and phases of GPU activity,

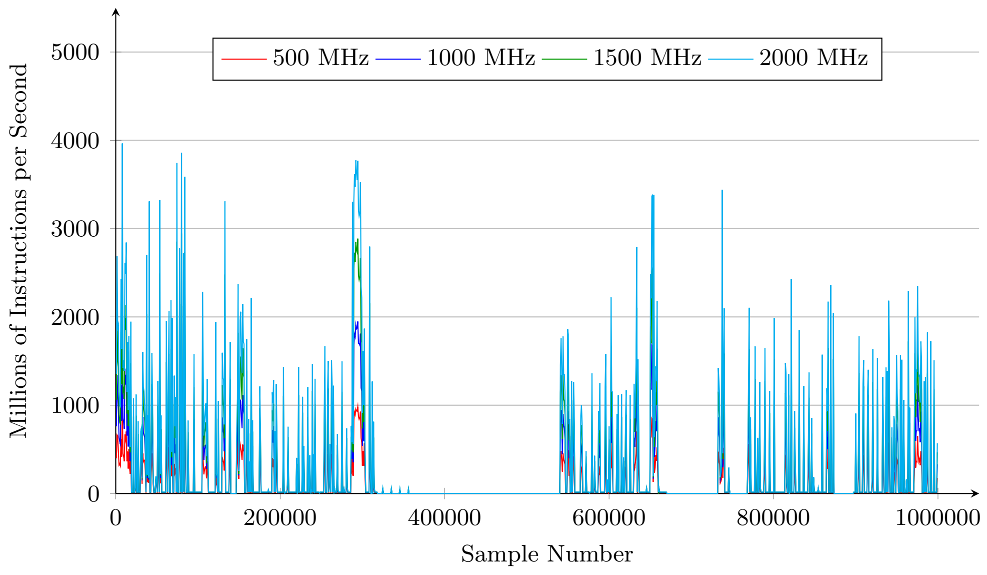


Figure 4.22: MIPS for BBench for the four frequencies explored using the gem5 forking infrastructure

during which the workload will be limited by the GPU and the memory system. This is apparent in both the time scaling presented in Figure 4.17. When the frequency of the CPU is increased from 500 MHz to 1000 MHz the run time of the workload decreases from 5.5 seconds to 3.5 seconds, a 37% reduction in run time. However, this decreases when doubling the frequency again to 2000 MHz, which results in only a 29% reduction in the run time. Overall, when moving from 500 MHz to 2000 MHz, a quadrupling in CPU frequency, the performance of the workload only improves by 55%. This demonstrates that it is not scaling with CPU frequency, and is in fact help back by the GPU-bound portions of the workload.

BBench

The final workload that we analyse with the forking infrastructure is BBench. There are key differences for the BBench runs, as we fork the simulation every 10 us, rather than every 100 us, and also run for a total of 10 seconds at 2000 MHz, as opposed to 2.5 seconds. Therefore, we are able to observe the operation of BBench at a much finer granularity than for the other workloads. This is important as BBench is not as regular as the other workloads which only had a limited number of different phases. BBench, as is apparent from Figure 4.22, does not have well defined phases of execution.

We profile the start of a BBench run, in which the web browser is loaded, and the BBench workload itself starts. The workload begins by showing a start page, which is displayed for a couple of seconds before the actual workload starts. This causes a long idle period, which can be seen in Figure 4.22. There are also idle periods for each web page shown during the execution of the workload as there is a brief pause between each scrolling action. These idle periods do not change when the CPU frequency is scaled as the CPU is idle for a fixed period of time, not a number of instructions. Hence, as was the case for the chase, the workload performance does not scale linearly with CPU frequency as the workload performance is limited by the length of these idle periods.

In general, BBench performance does improve as the frequency is increased, but these improvements are limited by the length of the idle periods within the workload. When the run time scaling presented in Figure 4.17 is considered, it is apparent that BBench does not respond significantly to CPU frequency once the frequency has reached around 1000 MHz. When the frequency is increased from 500 MHz to 1000 MHz, the workload run time reduces by 32%. However, when the frequency is again doubled to 2000 MHz, there is only a 22% run time improvement. Across the range of frequencies investigated, we only see a 48% performance improvement for a four-fold increase in frequency. This demonstrates that the run time for BBench is very limited by the length and quantity of the idle periods.

4.4 Visualising the Pareto Frontier

It is important to understand the design space when making both design-time and run-time decisions. This is imperative when optimisation is concerned, as the wrong decisions can result in a truly non-optimal decision, wasting time and energy, as well as jeopardising all attempts to provide a high QoE. CPU DVFS is used to trade the energy consumption of the CPU for the performance. When DVFS is used wisely, it is possible to achieve a large reduction in the energy consumption, with minimal impact on the performance. For this reason, it is important to understand the different types of workload, and when particular workloads are memory-bound or compute-bound. We have provided high-level overviews of how four different workloads scale in the previous section, but have only considered static frequencies, and have not considered dynamic DVFS adjustment.

In this section we use the forking framework described in Section 4.3 to demonstrate how the use of DVFS can affect the performance and efficiency of a CPU workload. Specifically, as we concurrently execute the workload at multiple frequencies we are able to switch the frequency for each sample of the workload. This allows us to emulate the operation of DVFS offline, and with oracle knowledge—we know the impact each frequency will have for each sample. By generating random frequency walks, we are able to determine the size and shape of the design space for each of the workloads discussed in the previous section. However, as each workload is comprised of a minimum of 25,000 samples, it is impossible to evaluate the whole design space due to the large number of potential permutations.

In order to explore the design space, we employ a genetic algorithm which is able to find the Pareto frontier. A genetic algorithm takes a set of potential solutions in the form of a genome, and evaluates each solution to determine how well it performs. This is called the fitness of the individual. A set of genomes is generated, and each is evaluated to determine the fitness for each individual prior to generating the next generation. To create offspring, two individuals are chosen, and these are combined through crossover in which parts of each parent's genome are combined to create a child. Alternatively, a single parent may mutate in order to create offspring, hence increasing the diversity in the gene pool. This helps to ensure that the genome remains varied and that the algorithm does not get stuck in a local optima. This new generation of individuals is then evaluated, and the process repeats.

We implement our genetic algorithm in Python using the DEAP module (Fortin et al., 2012). We seed our genetic algorithm by generating a pool of random frequency walks where each sampling interval is assigned one of the four profiled frequencies. Specifically, we generate a 25,000 entry frequency walk for each individual. These frequency walks may be comprised of one or two of the profiled frequencies, or may be comprised of a all four frequencies. We ensure that there are many different frequency walks generated as we want the gene pool to be as large as possible. We calculate fitness in terms of run time and energy consumption as these are the two properties we can trade. We used the NSGA-II selection algorithm by Deb et al. (2002) which is a multi-objective selection algorithm and is designed for finding the Pareto frontier by selecting individuals close to or on the Pareto frontier. The genetic algorithm is run for a total of 100 generations, 101 including the initial seed generation. Each generation consists of 100 individuals, 50 of which are used to generate the subsequent generation. We use two point crossover, in which the genome is crossed over at two points, the first and last coming from parent A and the middle coming from parent B. We use crossover to produce 70% of the children, mutation is used 20% of the time and 10% of the time a parent is cloned. A mutation rate of 5% is used which means that each gene has a 5% chance of being mutated randomly to one of the four frequencies.

A typical DVFS implementation is not able to instantaneously switch the operating frequency or voltage, and instead there is a small delay for each adjustment. This is most significant when increasing the voltage, due to the capacitance of the circuit. Frequency adjustments must be postponed until the operating voltage has reached the required level, as otherwise errors can occur. We take into account the switching penalties incurred when changing DVFS states and punish both the run time and the energy consumption. We incur a 1 us penalty for every DVFS state transition, and consume 1 μJ of energy. This allows us to model the effects from the voltage regulators and the clock generators found in real hardware. The results presented in this section assume that there is no idle sleep state, and therefore the best choice for the governor to make is to choose the lowest available frequency for idle periods.

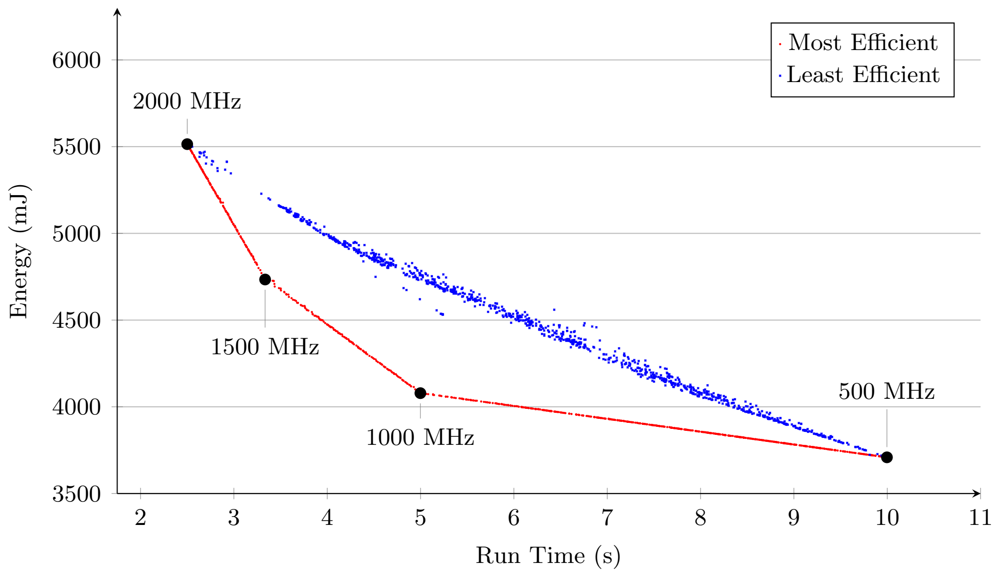


Figure 4.23: Pareto frontier for Dhrystone with a sampling interval of 100us

4.4.1 Dhrystone

We begin by looking at the Pareto frontier for Dhrystone. As this is a compute bound workload, the performance of the workload scales with the CPU frequency, and hence there is a wide range in both performance and energy. We present the Pareto frontier for Dhrystone in Figure 4.23, as well as showing some of the least efficient operating points. In addition to the Pareto frontier, we also show the four frequencies used for the analysis. These points were calculated by running at a fixed frequency for the entire execution. It is immediately apparent that the shape of the Pareto-optimal frontier is governed by the available DVFS points. In order to move along the frontier between two of the DVFS points, one must interpolate between them, and hence different proportions of the two frequencies are used. If too much switching occurs, then there is both an energy and a performance penalty as the CPU is often not able to operate for a brief period of time during a DVFS transition.

By observing that the run time scales linearly with CPU frequency, it is again clear that Dhrystone is a compute-bound workload which scales well. However, the energy scaling is not linear with the CPU frequency, and instead increases much more rapidly as the frequency is increased. Therefore, although Dhrystone performance scales with CPU frequency, it is important to ensure that the energy consumption remains acceptable when running at high frequencies.

When the least-efficient operating points are considered, it is clear that these use a lot more energy for the same level of performance offered by the more efficient points. Dhrystone is an always-active workload, with no idle periods, and only has one or two phases of operation. Therefore, the least efficient operation occurs when the frequency is changed between the highest and lowest frequencies rapidly.

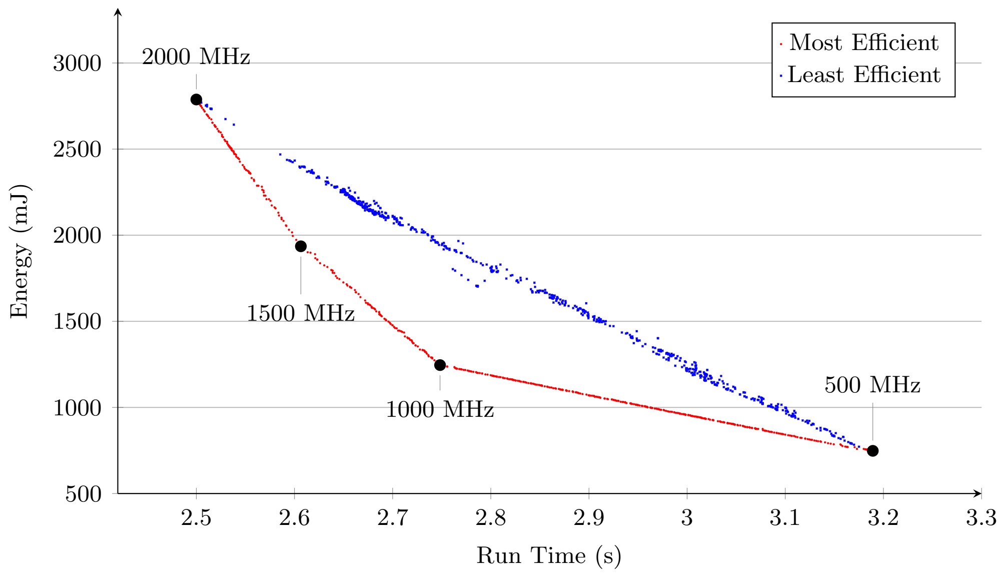


Figure 4.24: Pareto frontier for Memcpy with a sampling interval of 100us

4.4.2 Memcpy

We follow the Dhrystone analysis with Memcpy. As previously stated, this is a memory-bound workload, and therefore it has very little sensitivity to CPU frequency. We present the Pareto frontier for the workload in Figure 4.24. It is clear that there is a very small range of performance points that can be selected for the workload due to the fact that it does not scale. Specifically, there is only a very small range of different performance points for the frequencies used in our analysis. There is a clear increase in the energy consumption as the CPU frequency is increased, although there is no significant corresponding performance increase. This is extremely unfavourable as the increased energy consumption is not reflected in the performance of the workload, and hence the CPU is operating inefficiently for higher frequencies.

4.4.3 The Chase

In this section, we look at the Pareto frontier for The Chase. As discussed previously, this is a much more realistic workload when compared to Dhrystone and Memcpy, and therefore it is much more interesting to analyse. We consider two different sampling intervals, which we generate by combining the smaller sampling intervals from the original analysis, which were 100 us. The intervals are combined by either summing or averaging the results from the previous N intervals to generate a new interval. We consider 100 us and 100 ms intervals. We consider 100 us as the fine granularity gives an impression of what can be achieved if current DVFS techniques could operate on a more fine granularity. 100 ms is in the typical range for modern DVFS governors, and therefore demonstrates what these can achieve.

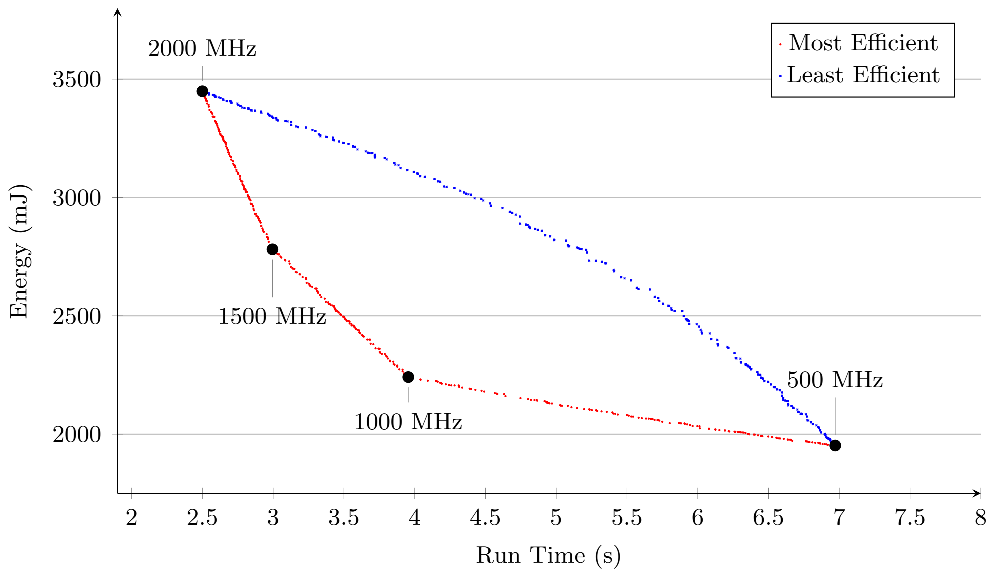


Figure 4.25: Pareto frontier for The Chase with a sampling interval of 100 us

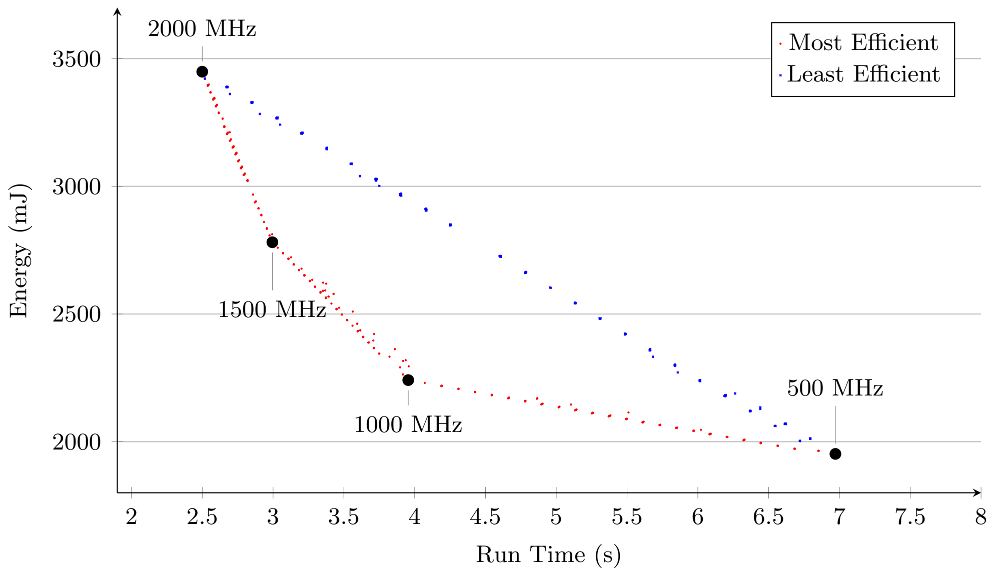


Figure 4.26: Pareto frontier for The Chase with a sampling interval of 100 ms

Figure 4.25 shows the Pareto frontier for The Chase when 100 us sampling intervals are used. As was the case with the previous results, the four available frequencies set the shape of the Pareto frontier. There is some distinct bowing for the least efficient points, which is caused by two different factors. The first is due to the rapid switching throughout the workload, where each frequency transition causes a time and energy penalty to be incurred. Secondly, the workload has many idle periods. If a high frequency is chosen for one of the idle periods, then there is no performance improvement, but there is an energy penalty. This is very inefficient and causes a large number of inefficient operating points.

We show the 100 ms Pareto frontier in Figure 4.26. As was the case with the 100 us

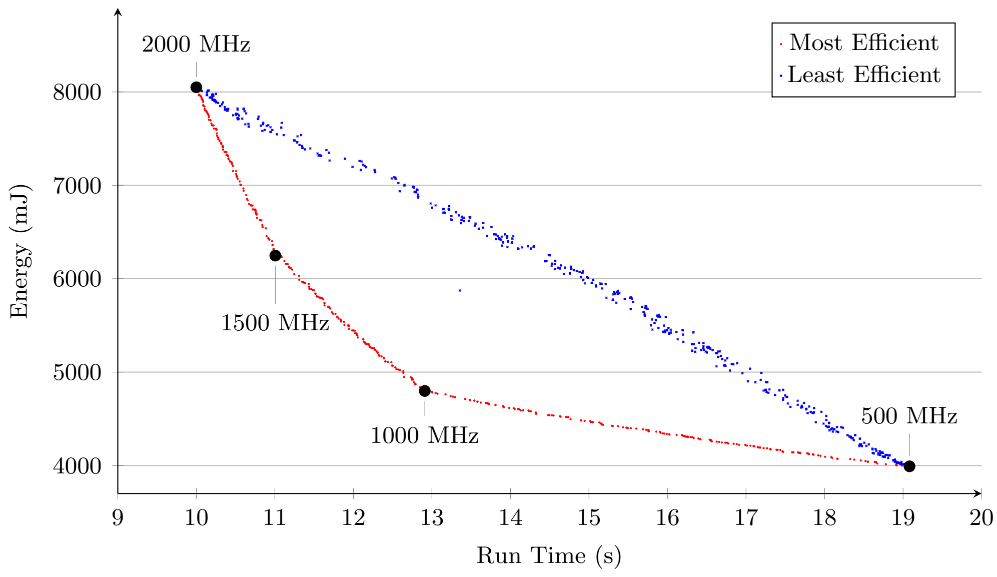


Figure 4.27: Pareto frontier for BBench with a sampling interval of 100 us

results, the shape of the Pareto frontier is governed by the four available frequencies. However, one of the key differences lies in the least efficient operating points. First of all, it is worth noting that these do not bow out in the same was as the 100 us results. This is due to the less frequent switching, and hence there are fewer performance and energy penalties for the larger interval size. Secondly, the points of operation are very discrete. This is due to aliasing with the period of the workload. Specifically, for a larger interval size, the interval will include both a peak and trough of the workload (see Figure 4.19). Therefore, almost every interval includes both a compute-bound and an idle portion of the workload. The compute-bound portion will scale with the frequency, whilst the idle portion will not scale at all. The overall results is that there are only very discrete points at which the workload can operate when trying to find the worst possible choices.

4.4.4 BBench

Finally, we consider BBench. This workload is the most realistic CPU workload analysed with the forking framework. Whilst this workload has been sampled every 10 us, we only consider 100 us samples, as was the case for The Chase, as these are representative of both a future governing strategy and a current governing strategy. We present the 100 us Pareto frontier for BBench in Figure 4.27, and present the 100 ms Pareto frontier in Figure 4.28. As was the case for The Chase, there is a wide range of trade offs that can be made for BBench. This workload scales well from 500 MHz to 1000 MHz, and therefore that part of the Pareto frontier covers a wider range of performance points than for other sections of the frontier.

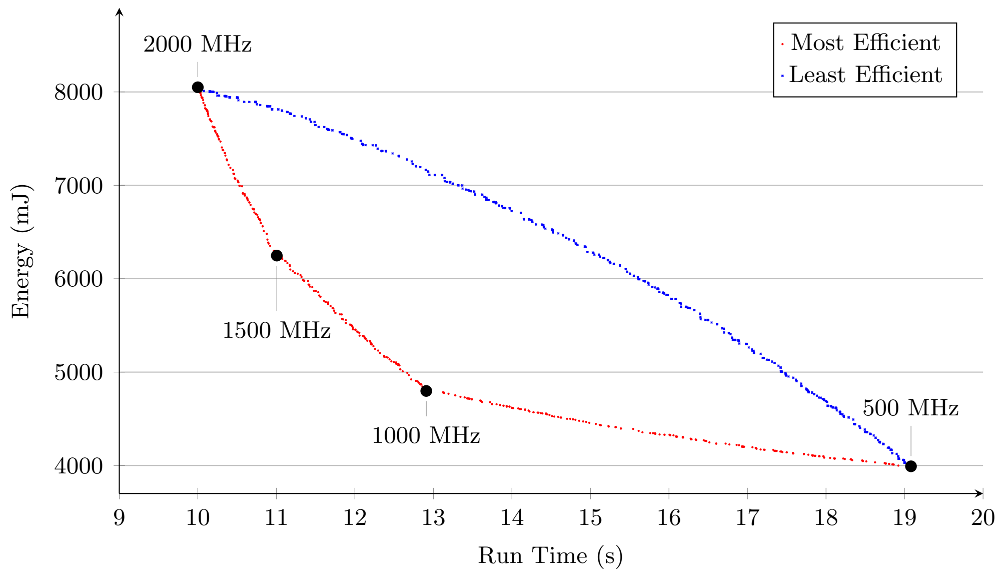


Figure 4.28: Pareto frontier for BBench with a sampling interval of 100ms

We begin by considering the 100 us interval size as is shown in Figure 4.24. BBench is the first workload which has a stronger curve to the Pareto front. This is caused by the short activity intervals found in BBench, and therefore it is possible to choose the ideal frequency for each of these intervals. This can save a large amount of energy relative to running at a fixed frequency for the entire execution.

Finally, we consider the 100 ms intervals for BBench, which are shown in Figure 4.28. The pareto frontier for the 100 ms results is similar to that for the 100 us results. However, the worst case results are significantly worse. This is most likely due to the fact that BBench has periods of activity which are much shorter than 100 ms and therefore a governing interval of 100 ms is too long. Therefore, there are periods of time when either the frequency is too high or too low, either wasting energy or sacrificing performance. These same situations can occur for the 100 us results, be the genetic algorithm is much less likely to find them, and would need to run for many more generations.

4.5 Concluding Remarks

In this chapter we look at the CPU and CPU workloads to understand when and why a CPU may behave inefficiently. We have demonstrated that the traffic generated by the CPU to the memory system are should have a low latency to ensure high CPU performance, as otherwise the CPU can become starved resulting in a performance drop. We have also briefly touched upon how the caches in the system improve the CPU memory latency, and help to ensure high CPU performance. We have also considered the energy consumption of the CPU relative to the CPU frequency and workload performance, and have provided evidence to show that it is possible to save a large amount of energy whilst incurring a minimal workload performance slowdown.

We then considered different types of CPU workloads and observed how of these responded to changes in the CPU frequency on real hardware. Specifically, we began by considering compute-bound workloads which should scale linearly with increasing CPU frequency, and demonstrated this to be the case for two workloads: Dhrystone and AndEBench. We then looked at how Memcpy performance scales as the CPU frequency is varied, and demonstrated that CPU frequency as a minimal impact on the performance of memory-bound workloads. Finally, we considered more typical workloads, which are neither memory-bound nor compute-bound. These workloads may have phases which can be memory- or compute-bound, but the overall execution of the workload cannot be categorised into either of these categories. These workloads, such as RLBench and AnTuTu, respond to changes in the CPU frequency but have a limited response when the frequency is increased as performance is limited by the memory-bound regions of execution. Hence, for such workloads it often is illogical to run the CPU at the lowest frequency, or highest frequency, as an intermediate frequency offers more efficient execution.

Next, we presented a gem5-based methodology which uses process forking to investigate workload frequency sensitivity without perturbing the execution of the workload. This framework allows us to concurrently execute the same workload at multiple CPU frequencies and directly compared the results on a slice-by-slice basis. We demonstrated that workloads such as Dhrystone scale linearly across the entire execution of the workload, and therefore the higher CPU frequency always results in improved performance. Workloads such as Memcpy, on the other hand, show close to no performance improvement at any stage of execution, with the exception of a very brief interval at the start of the workload when it initially loads. We then considered The Chase, which is a workload which utilises both the CPU and the GPU. This workload showed a high sensitivity to CPU frequency during the CPU-bound portions of the workload, but showed close to no improvement for the GPU-bound sections of the workload. Therefore, whilst this workload showed sensitivity to CPU frequency, the performance was limited by the GPU-bound regions of execution. Finally, we considered the CPU workload called BBench, which measures the web browser performance. This workload loads a series of web pages and scrolls through them with a fixed delay between each scrolling action during which the CPU is idle. As was the case for The Chase, the scalability of BBench is mostly limited by the idle periods.

Finally, we determined the Pareto frontier for four CPU workloads, and demonstrated that the shape of the frontier is governed by the available DVFS operating points. We showed that the design space for compute-bound workloads allows a large range of performance levels and energy consumptions to be traded, whilst a memory-bound workload only scales in the energy axis, and has little scope for performance improvement. Workloads such as The Chase and BBench allow for a range of trade off opportunities, but these are limited by the scalability of the different phases of the workload.

In this section we have demonstrated when and why the CPU may operate inefficiently, and have also investigated how different CPU workloads respond to changes in the CPU frequency. In the next section we act upon these findings, and demonstrate that it is possible to optimise the performance of the system by adjusting the CPU frequency to ensure that the CPU is operating efficiently, and that the workload performance is high enough to satisfy the end-user.

Chapter 5

Optimising for CPU User Experience

Traditionally, mobile systems are expected to deliver high performance on demand, whilst remaining as power efficient as possible. This is typically accomplished through a combination of both hardware and software optimisations which work in tandem to dynamically adjust performance and power consumption based on requirements. The hardware must be able to enter a low power state when the system is idle, thereby allowing long sleep periods between relatively infrequent periods of use. The software must ensure that the number of wake ups are minimised, and that no unnecessary work is done. When the system is in use, however, the components must strive to remain as power and energy efficient as possible whilst meeting performance requirements. A noteworthy example of a commonly found runtime energy saving mechanism is DVFS which is used to save energy on components such as CPUs and GPUs (Pallipadi and Starikovskiy, 2006; Spiliopoulos et al., 2011; Schmitz et al., 2005; Hsu and Feng, 2005).

DVFS has been employed for many years to reduce the energy consumption of computer systems when high performance is not required for an application or workload. It initially was used for large scale server farms, and allowed for a large amount of energy to be saved during less busy periods, hence reducing running cost significantly. From there it migrated to single machines, and allowed for more efficient home computing. In the modern era, this mechanism has made it to mobile phones and tablets, and is used to increase the time between charges, as well as thermal management. However, there is scope for further reducing the energy consumption by tweaking the DVFS management strategies.

Mobile systems have adopted the governors used for DVFS adjustment from the higher power server and home computer market, such as the *Ondemand* governor (Pallipadi and Starikovskiy, 2006). Whilst these governors are designed to reduce the energy consumption and thermal dissipation when the workload is not placing a high strain on the CPU, they do not always work well for mobile workloads. Additionally, when it comes to servers and high performance computing, it is often desirable to achieve as high performance as possible, and therefore even small performance losses are often not tolerated. For this reason, the governors designed for these systems can often perform sub-optimally for mobile systems, wasting large amounts of energy even when there is no discernible performance improvement. This results in reduced device run time, and increased heat production, which results in a reduced level of user experience.

The typical DVFS governors found on mobile devices monitor the CPU load to make the majority of their decisions. This applies especially to the *Ondemand* and *Conservative* governors, both of which rely on very similar algorithms. They both increase the CPU frequency when the CPU load is sufficiently high, and reduce it again when the load drops below a set threshold. Whilst CPU load is a sensible metric on which to make such decisions for typical server and home computer workloads, it does not make as much sense on mobile platforms. Mobile platforms are required to deliver a sufficiently high level of user experience, and often this does not mean the highest performance. Measuring CPU load causes the CPU frequency to be increased when the workload has a lot to do, not when the user demands higher levels of performance.

We have presented models which are able to take measurable low-level metrics, and use these to determine the level of user experience delivered by a mobile device in Chapter 3. These models provide a hook which can be used to optimise a running system for the desires of the end user, rather than focusing on delivering high performance. Using these models it becomes possible to trade off the performance and energy consumption of a system without impacting the user experience. Additionally, the use of these models allows for a greater range of energy savings when compared with traditional performance-based system optimisation techniques.

In Chapter 4, we investigated when the CPU is able to operate efficiently. The CPU is the component which coordinates almost all other in-system components, and is responsible for running the operating system and the bulk of most applications. It is therefore vital that the CPU is able to operate efficiently and to understand when it is unable to do so. In modern systems, a CPU is able to operate at a number of different frequencies and corresponding voltages which provide different levels of performance. The frequency is adjusted at run time by a DVFS governor. There are many different types of DVFS governors, but a typical governor makes DVFS decisions based on the CPU load and other low-level system metrics. Whilst these provide the opportunity to save energy when the system is not under full load, or by sacrificing performance, they are not able to take into account the requirements and desires of the end user.

In this chapter we begin to answer our final research question: *Making various assumptions about user preferences - which we will term user experience models - how can CPUs, GPUs as well as complete systems be run-time optimised in such a way that the end-user* *remains satisfied?* We combine the QoE modes presented and discussed in Chapter 3 with the CPU efficiency investigations presented in Chapter 4. We begin in Section 5.1 by observing the operation of the *Ondemand* and *Conservative* governors for a set of four workloads using gem5 simulations. We compare and contrast the operation of the typical governors to a pair of *Oracle* governors which adjust the frequency based on IPC. We then present an algorithm for DVFS governing based on the IPC of the workload, and link this back to QoE in Sections 5.2 and 5.3. As part of this, in Section 5.3.2, we profile a set of the applications in software to determine the relationship between the measurable low-level metrics and application performance. We then use this relationship in a QoE-Aware DVFS governor which is able to adjust the CPU frequency based on the profiling information. Following our simulation-based results, we present results from real hardware in Section 5.4. We demonstrate that it is possible to create a simple governor based on this information that is able to reduce the energy consumption of the workload whilst minimally impacting the performance and user experience delivered by the workload.

5.1 CPUFreq Governor Analysis and Oracle Governing

We begin by comparing the operation of the CPUFreq governors found on the majority of mobile devices to some *Oracle* governing strategies to determine how well standard governors operate for different types of application. We consider both the resulting workload performance as well as the energy consumption as part of this analysis. In this section we use the results from the DVFS forking analysis presented in Section 4.3 to determine which decisions an *Oracle* governor would make for four different workloads. The use of an *Oracle* governor allows us to determine how much energy we are able to save relative to the standard DVFS governors, and how much performance we must sacrifice to achieve this. The results from the forking analysis provide all of the information required to create an *Oracle* governor as we know the impact that each DVFS decision will have on the performance and energy consumption of the workload.

We create a configurable *Oracle* governor which uses MIPS/MHz to determine the frequency that should be chosen for each governing interval. MIPS/MHz is equivalent to the number of instructions per cycle executed on the CPU. For a given application, as the frequency of the CPU is increased, the IPC will reduce in most cases. The *Oracle* governor uses this information and will only allow a limited reduction in IPC for any governing interval relative to the lowest operating frequency. For each governing interval, the governor calculates the IPC for each frequency, and then compares the IPC for each frequency to the IPC at the lowest frequency. The governor then chooses the highest frequency that has an IPC which is within a specified range of the IPC at the lowest frequency. Specifically, it allows for a specified reduction in IPC relative to the lowest operating frequency for each governing interval.

```
$frequencies \leftarrow$ set of available frequencies in ascending order
$f_{min} \leftarrow frequencies[0]$
$freq_0 \leftarrow frequencies[0]$
**for** each governing interval, $i$ **do**
  **if** $IPC_i[f] <$ absolute IPC threshold **then**
    $freq_i \leftarrow f_{min}$
  **else**
    **for** each frequency, $f$, in $frequencies$ **do**
      **if** $IPC_i[f]$ / $IPC_i[f_{max}] >$ relative IPC threshold **then**
        $freq_i \leftarrow f$
      **end if**
    **end for**
  **end if**
**end for**
```

Figure 5.1: Algorithm for the *Oracle* DVFS governor

To clarify the operation of the *Oracle* governor, we present the following example. Let us assume we have a system with three possible CPU frequencies which are 500 MHz, 1000 MHz and 1500 MHz which achieves 500 MIPS, 900 MIPS and 1000 MIPS respectively. If we set the governor to tolerate up to a 20% MIPS/MHz reduction, then the governor will set the CPU frequency to 1000 MHz as this is 90% as efficient as the lowest frequency. The highest frequency, 1500 MHz, only achieves 67% of the MIPS/MHz at 500 MHz and therefore is not chosen by the governor. In the event that the MIPS/MHz at 1000 MHz had been less than 80% of the 500 MHz value, then the 500 MHz frequency would have been chosen instead.

By varying the amount of allowable IPC drop we are able to configure the *Oracle* governor to favour higher or lower frequencies. As this is an oracle governor, the DVFS decisions only consider the current interval and do not look at history for the workload. We consider two different *Oracle* governor configurations. The first allows up to a 50% reduction in IPC, whilst the second only allows for a 20% reduction in IPC relative to the lowest frequency. Therefore, the second governor configuration will only increase the frequency in the cases where the workload scales well with frequency, at the expense of performance.

We provide a reference for our *Oracle* governing strategies by implementing the operation of the *Ondemand* and *Conservative* CPUFreq governors offline based on the same data set. These offline versions of the governors have been written to behave as closely as possible to the implementations found in the Linux kernel. We monitor the CPU load statistic from the gem5 simulator in order to mimic the operation of the CPUFreq governors. This is the closest available proxy for the CPU load as measured by the kernel for a traditional DVFS governor implementation. For reference, we have provided pseudo code representations of our implementations in Appendix B.1.1. For both the *Ondemand*

and *Conservative* governors we start them at the lowest frequency, 500 MHz, at the start of the workload.

We evaluate the performance and energy consumption of the two different *Oracle* governors, as well as the *Ondemand* and *Conservative* governors in terms of both performance and energy consumption. We use the same energy models as presented in Section 4.3. As we do not run complete workloads in the forking infrastructure, we are only able to compare the run time for the different workloads to determine the performance variation, and hence do not present user experience numbers for this section. When evaluating the governors, we run them every 10 ms. This is towards the higher end for the typical DVFS governors, but can be found in real systems. This threshold applies to all governors, including the *Oracle*.

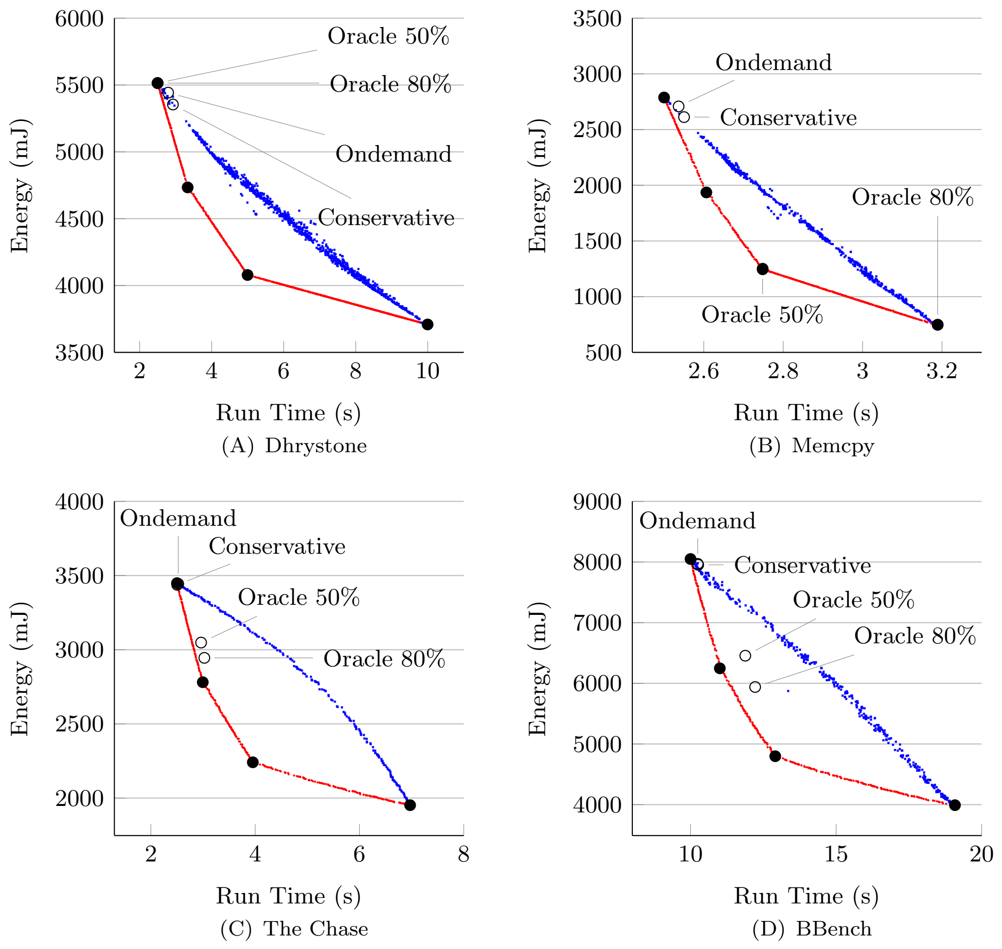


Figure 5.2: Pareto frontiers showing run time and energy trade-offs for the CPUFreq governors and the *Oracle* governor.

Before considering the operation of the governors on each of the workloads, we present overall results for both run time and energy consumption in Figure 5.2. First of all, it is worth noting that for a compute-bound workload all of the governors are able to achieve roughly the same run time and energy consumption, with the *Ondemand* and

Conservative governors losing a small amount of performance, but also saving energy. This is due to the fact that these governors are configured to start at the lowest frequency for each of the runs, and hence will spend at least the first governing interval at the lowest frequency. *Ondemand* is quicker to raise the frequency and therefore has a shorter run time and higher energy consumption.

Next, we consider the Memcpy results, which show that it is possible to sacrifice some of the performance for very large energy savings. This is expected due to the workload being memory bound. Again, as was the case for Dhrystone, the governors start the first governing interval at 500 MHz before increasing the frequency for subsequent intervals. Therefore, there is a mild performance and energy reduction relative to the 2 GHz run. It is worth reiterating that the *Ondemand* and *Conservative* governors adjust the frequency based on the CPU load. The CPU load is high during a Memcpy run, and therefore the frequency is increased even if this does not significantly improve performance. The *Oracle* governors do not observe the CPU load, and therefore choose lower frequencies.

The Chase is a workload comprised from phases with high CPU usage and phases with low CPU usage. The traditional governors take some time to switch between the frequencies and therefore are not able to exploit the changes in the workload as readily as might be desired. However, the *Oracle* governors are able to exploit these workload features to save energy with a minimal reduction in workload performance. The *Oracle* governor with the 50% threshold is able to save some energy relative to the standard governors with a minimal performance loss, whilst the 80% *Oracle* governor is able to save significantly more energy, albeit with a marked performance decrease.

Finally, BBench is a workload which places a constantly changing load on the CPU. Many of the periods of load are very short, and therefore the traditional governors are not able to respond before the opportunity has passed. Therefore, the standard governors are unable to save a large amount of energy for the workload and do sacrifice some performance. The *Oracle* governors are able to save energy relative to the standard governors, with the 50% governor achieving a higher level of performance.

5.1.1 Dhrystone

As in previous sections, we begin our analysis by looking at the compute-bound Dhrystone workload. This workload scales close to linearly with performance and therefore all of the governors in question, choose the highest operating frequency. We opt not to show a figure for the Dhrystone workload, as this provides no additional insight into the operation of either the *Oracle* governors, or the standard CPUFreq governors.

Both of the *Oracle* governor configurations opt for the highest frequency of 2 GHz for every governing interval due to the linear workload scaling. The *Ondemand* and *Conservative* governors both adjust the CPU frequency based on the load. As the

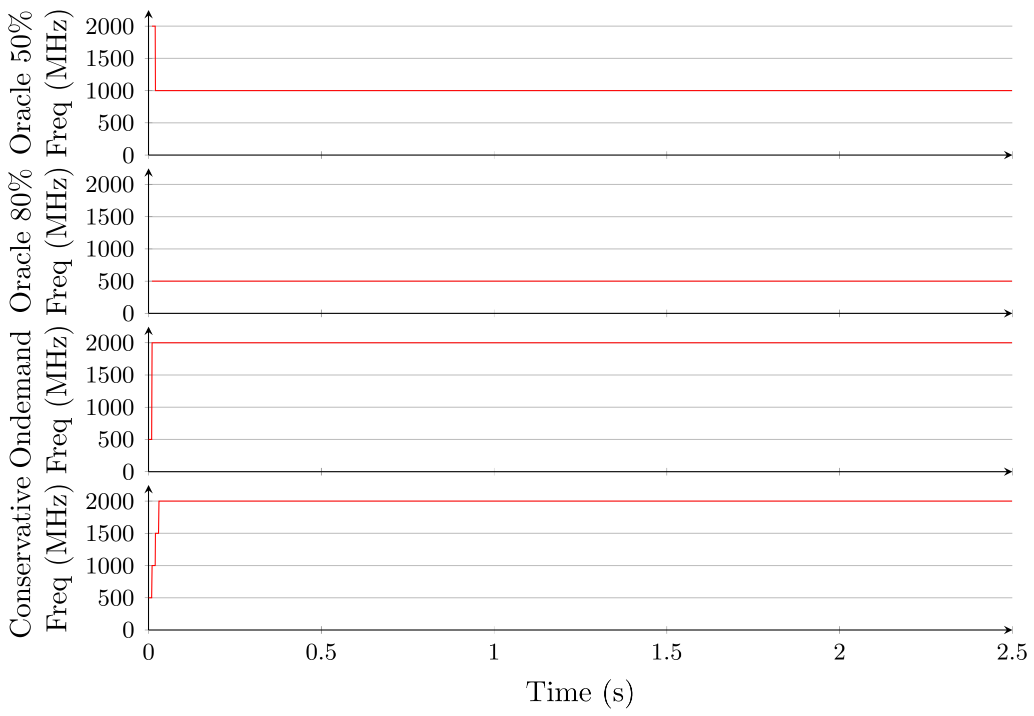


Figure 5.3: DVFS frequency adjustments for various governors for the Memcpy benchmark

workload always places a high load on the CPU, both the *Ondemand* and *Conservative* governors choose the highest frequencies for the duration of the workload. The exception occurs during the first few governing intervals during which the frequency is gradually increased one step at a time from 500 MHz for the *Conservative* governor as the governor will only adjust the frequency one step at a time. Also, the *Ondemand* governor remains at the lowest frequency for the first governing interval before increasing the frequency to the maximum.

5.1.2 Memcpy

As we have demonstrated previously, Memcpy is a memory-bound workload which does not scale significantly with CPU frequency once the CPU frequency is sufficiently high. Specifically, for low-enough CPU frequencies, the workload can be compute-, or partially-compute-bound, but once the frequency is increased sufficiently, the workload becomes almost purely memory-bound and does not scale with CPU frequency. Therefore, an ideal governor should not raise the frequency once the workload has stopped scaling due to the diminished returns on investment. Our gem5 experiments have shown that the workload transitions to being memory-bound between 500 MHz and 1000 MHz for the system simulated. Therefore, there is some, if minimal, performance improvement within the first two frequencies.

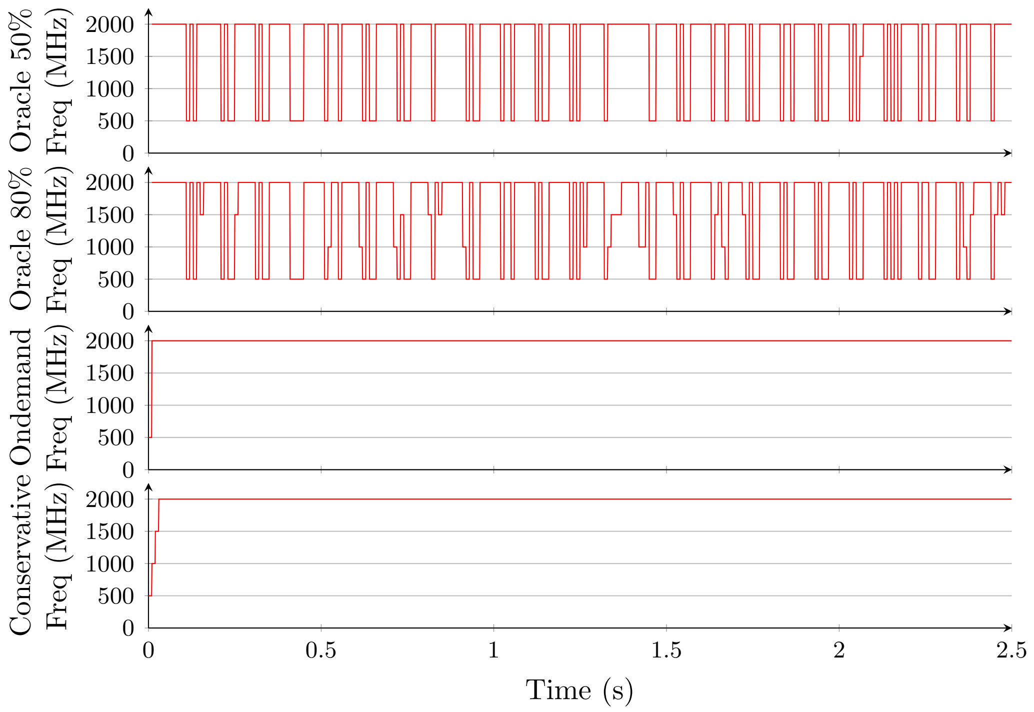


Figure 5.4: DVFS frequency adjustments for various governors for The Chase

Figure 5.3 shows the frequency choices made over time for each of the four governors simulated. We begin by considering the *Oracle* governor with the 50% threshold. This governor is more likely to choose a higher CPU frequency as it is configured to tolerate a larger reduction in CPU efficiency. Aside from choosing the highest CPU frequency for the first two governing intervals—this is the part where the workload is initially loaded and is more compute-bound than for the rest of the execution—the governor chooses the 1 GHz frequency. This indicates that there is a greater than 50% efficiency reduction when moving beyond the 1 GHz frequency. The 80% *Oracle* governor, on the other hand, chooses the 500 MHz frequency for the entire workload.

For this workload, both of the CPUFreq governors do a very poor job, and both switch to the highest CPU frequency as they did for Dhrystone. This is due to the fact that whilst Memcpy is very memory-bound, it places a high load on the CPU as far as the load measured by the kernel is concerned. Therefore, these choose the highest frequency even if it does not offer increased performance. Again, as was the case for Dhrystone, these governors started at the lowest frequency and then increased the frequency until 2 GHz was reached.

5.1.3 The Chase

The Chase is a workload which transitions between phases which are reliant on the operation of the CPU and phases in which the CPU must wait for the GPU to render

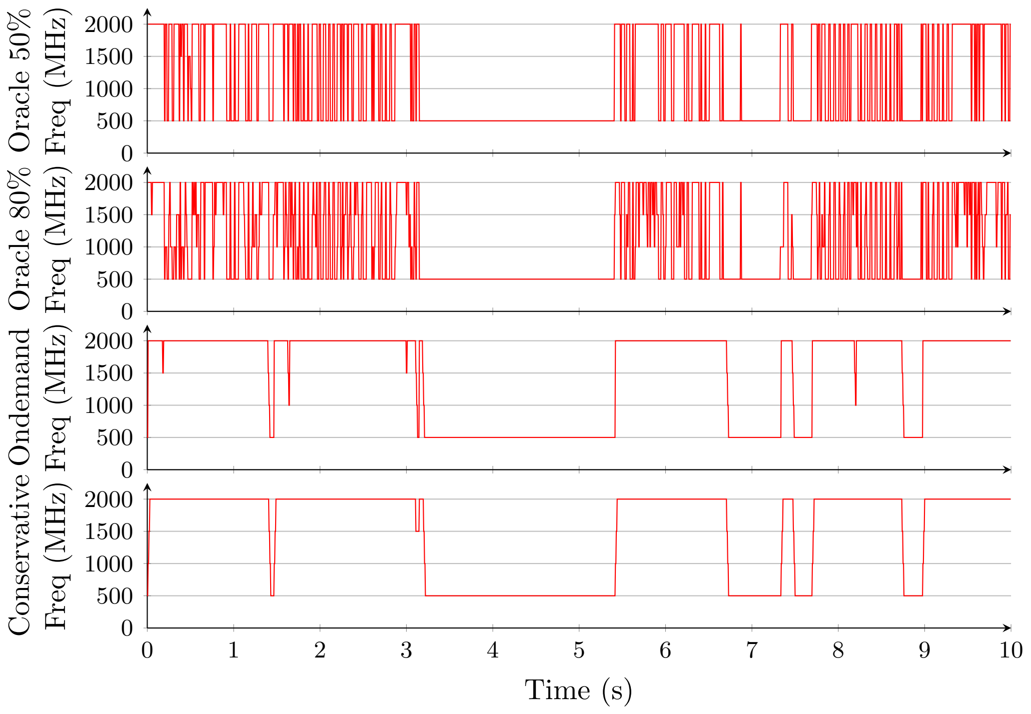


Figure 5.5: DVFS frequency adjustments for various governors for BBench

the frame. There is some scope for performance scaling for the CPU-specific portions of the workload, but the portions which rely on the GPU do not scale with CPU frequency. Additionally, the portions of the workload during which the execution is mostly reliant on the GPU still place a high load on the CPU. Therefore, it is difficult for the standard CPUFreq governors to make the right decisions.

Figure 5.4 shows the operation of the governors for a 2.5 second window of The Chase. Again, we see similar behaviour from the *Ondemand* and *Conservative* governors, which both opt for the highest CPU frequency as soon as they are able to switch to it. This is due to the aforementioned high CPU load and is unavoidable if these governors are used.

The *Oracle* governors are able to choose lower frequencies for the portions where the workload performance scales less well. The 80% *Oracle* governors is more aggressive when choosing the lower frequencies and therefore chooses these more often than the 50% *Oracle* governor which is more tolerant of less efficient performance. Both governors are able to reduce the frequency relative to the typical governors, hence saving energy.

5.1.4 BBench

Finally, we consider our most complex workload, BBench. This workload does not have a handful of different phases, but is comprised of significantly more. Therefore, this is the most challenging workload to govern, as the performance at any given frequency is continuously changing. In this workload, there is a lot more score for the *Oracle* governors as they are able to make instant decisions regarding CPU frequency and do not need to operate on past history. We consider a larger window of BBench, and observe the operation of the governors over a 10 second, as opposed to 2.5 second, window.

This is the first workload where we see the *Ondemand* and *Conservative* governors choosing non-maximum CPU frequencies. Both the *Ondemand* and *Conservative* governors make similar decisions for BBench, and favour periods of roughly constant frequency. In general, these governors will use either the highest or lowest frequencies for these sorts of workloads as the workloads want to compute as quickly as possible or are idle. Specifically, when the CPU load is high, it is usually high for all frequencies and therefore the governors will try and choose the highest frequency. When the workload has an idle period, the governors will switch down to the lowest frequency. There are not many occurrences where the CPU load falls off significantly when the frequency is increased, resulting in high use of the minimum and maximum frequencies, and little use of intermediate frequencies.

The *Oracle* governors are able to choose lower frequencies as they are not monitoring the CPU load itself. The 50% *Oracle* governor mostly chooses the 1.5 GHz and 2 GHz frequencies, and only chooses lower frequencies for very short periods of time. There is no period of time for the workload during which the CPU is totally idle, and therefore there is no extended period of time where the 500 MHz frequency is chosen. The 80% *Oracle* governor, on the other hand, chooses lower frequencies much more often than the 50% *Oracle* as it will switch to lower frequencies more easily as the relative performance reduces.

There is a notable difference between the frequency profiles seen for the *Oracle* governors and the CPUFreq governors. The CPUFreq governors choose the lowest frequency for a long period of time—over 2 seconds—whilst no such pattern is visible for the *Oracle* governors. The CPUFreq governors observe the CPU load, and therefore will not increase the frequency of the CPU if the CPU is not at full load. On the other hand, the *Oracle* governors look at the relative performance scaling for the interval, and will choose a higher frequency if it offers higher relative performance, even if the CPU is not at full, or close to full, load. This can result in an increase in energy consumption as the CPU frequency is increased for a very small number of instructions if this is found to improve performance for that window. However, neither approach is ideal. The CPUFreq governors waste energy by increasing the frequency based on load, and therefore can do poorly for memory-bound phases of the workload, whilst the *Oracle* governors can waste energy during periods of relative inactivity.

5.2 Adjusting CPU DVFS based on QoE

Mobile devices including, but not limited to, smartphones and tablets run mobile operating systems such as Android, which is based on Linux. Android makes use of the Linux kernel CPU governors which trade off the performance of the CPU for energy savings. In general, these governors control the voltage and frequency of the CPU based on demand, and will try to operate at the lowest energy required to run the workload. This requires them to monitor the demand on the system, and then use this information to determine the required frequency in order to sustain it. A range of governors exist, and each will adjust the frequency based on slightly different metrics or will aim to save as much energy as possible.

The Linux CPU DVFS governors were originally designed for non-mobile systems, such as desktop machines, laptops and servers. These governors do not have a concept of user experience, and simply optimise based on the performance requirements placed on the CPU by the workload or workloads. Therefore, based on the governor chosen and corresponding settings used, these governors can either waste energy or restrict performance too severely. This results in a degraded user experience, and is clearly undesirable in the resource-constrained world of mobile systems.

In order to address this issue, we demonstrate QoE-aware CPU governing, and implement an example QoE-aware governor. This governor uses knowledge about the workloads and the delivered user experience to determine the correct DVFS configuration for the CPU. This ensures that the performance of the system components is only reduced when the user experience allows it. Therefore, this governor tries to ensure that the performance reduction does not impact the level of QoE delivered. We apply the QoE models described in Sections 3.4.1 and 3.4.2 as part of our QoE-aware system governor.

5.2.1 Optimising for QoE

We demonstrate the benefit of QoE-aware system optimisation by creating a QoE-aware system governor in simulation. This governor adjusts the DVFS operating points of the CPU based on the QoE delivered, and the target QoE for the system as a whole. Our DVFS algorithm is presented in Figure 5.6 and is designed to be integrated in software as part of the normal Linux governor framework. We first demonstrate the operation of this algorithm in simulation using gem5, and them present results for real hardware using the Odroid XU3. We provide an overview of the XU3 development board in Section 1.3.2. For simulation, the algorithm is run in a system controller which sits alongside the rest of the system, monitors components unobtrusively and adjusts the CPU configuration. However, for the experiments using the XU3 we integrate the governor into the existing CPUFreq framework found in the Linux kernel.

```
if CPU idle then
  Decrease CPU operating level to minimum
else
  if Predicted QoE higher than target then
    Decrease CPU operating level
  else if Predicted QoE lower than target then
    Increase CPU operating level
  else
    Do nothing
  end if
end if
```

Figure 5.6: Algorithm for QoE-Aware CPU governing

As the governor is unable to directly measure the delivered QoE, it is required to make use of the QoE utility functions discussed earlier, and low-level proxy metrics. These low-level metrics can be measured during runtime in order to determine if the level of experience delivered by the application are acceptable. For the CPU workloads, we look at the IPC which gives a measure of CPU efficiency which additional is tired to the frequency of the CPU. By profiling the CPU workload, we are able to determine which IPC is required on average in order to provide the desired level of user experience.

The CPU governor uses the IPC of the CPU and the knowledge of the application running to estimate the delivered QoE based on the relationship presented in Section 3.4.1. As we shall demonstrate in Section 5.3, it is possible to use a low-level metric such as IPC, which can be measured from the CPU performance counters, to measure the CPU performance and predict the delivered QoE. If the predicted QoE is higher than the target, then the DVFS operating point of the CPU is reduced to the next lowest level. On the other hand, if the predicted QoE is insufficient to meet the target, then the operating point of the CPU is raised. Finally, in the case where the QoE prediction matches the target, no adjustments are made. In addition to detecting when the CPU DVFS configuration needs to be adjusted based on the load, the algorithm detects when the system is idle (less than 1% CPU utilisation) and will drop the operating point of the CPU to the lowest level. This reduces the energy consumption significantly, at the cost of a small performance penalty when coming out of the idle state.

5.2.2 Experimental Methodology

In this section, we describe the experimental methodology used in this chapter. We simulate a system with both a multi-core CPU and a multi-core GPU in gem5. As before, the memory system and peripherals are modelled in detail. In addition to the standard system components, we model a system controller which monitors the state and performance of the system. The system controller then uses this knowledge and

CPU Clock	200 MHz-1.7 GHz, 100 MHz steps
CPU Voltage	0.925V-1.3V
CPU Caches	L1: 32+32 kB, L2: 1 MB
GPU Cores	4-16, multiples of 4
Volatile Memory	OS: 1024 MB
Operating System	Android 4.1.1 (Unmodified)
Kernel Version	3.9.0-rc7 (Unmodified)

Table 5.1: System Configuration

derived user experience metrics to make informed decisions regarding the DVFS state of the CPU.

The system configuration is summarised in Table 5.1 and a system diagram is shown in Figure 5.7. Our simulated system can be configured with either a single-core or a dual-core CPU. All CPU cores and associated level 1 and level 2 caches sit in the same voltage and frequency domain, as is shown in the diagram. Hence, all DVFS adjustments affect both cores (in the dual-core configuration) and caches. The voltage and frequency domains are implemented as detailed by Spiliopoulos et al. (2013). As part of the memory system we model a dual channel LPDDR3 memory and associated controller. We use First-Ready First-Come-First-Served (FR-FCFS) scheduling with an open row policy and prioritise the CPU traffic over other system traffic, as is the case in many commercial SoCs.

In order to assess the performance of the CPU governor we run BBench (Gutierrez et al., 2011), RLBench (RedLicense Labs, 2012) and AndEbench (Levy, 2012). BBench measures the time it takes to load a series of popular web pages from disk, RLBench benchmarks the SQLite performance of the system and AndEBench evaluates the CPU performance using pre-compiled code and interpreted Java.

We estimate the dynamic energy consumption of the CPU and DRAM. We match the DVFS operating points of the Samsung Exynos 5250 (Samsung, 2012) on the CPU, and use these to calculate the energy consumption. Specifically, we calculate the dynamic power consumption as per $\alpha CV^2 f$, but ignore the α (activity percentage) and C (capacitance) terms as we focus on the relative energy consumption. We estimate the energy usage of the DRAM by assuming each bit accessed consumes a constant amount of energy. These are coarse-grained models, but they are sufficient to demonstrate the merits of QoE-aware system optimisation.

We ensure that we use realistic time-scales that match those used in real consumer devices. Specifically, for the CPU governor we use a period of 10 ms, which is comparable to that of the standard Linux and Android governors. In addition, this matches the distinct phases of changes in behaviour seen in real workloads which ensures that optimisation takes place on the correct time-scale (Sunwoo et al., 2013).

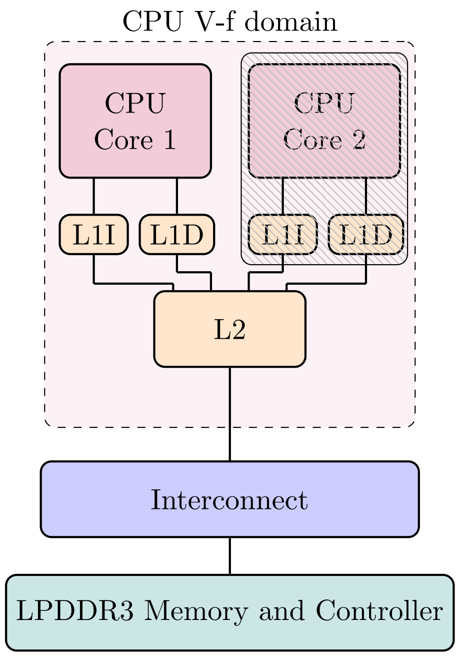


Figure 5.7: System diagram showing the CPU configuration, as well as the memory system.

5.3 Linking High- and Low-level Performance

The CPU governor detailed in Section 5.2.1 requires a low-level metric which can be used as a proxy for the delivered user experience. This is required as the actual workload performance is not available to the governor without serious software-stack modifications, and by the time the workload completes it will be too late to affect the delivered QoE. We therefore require a link between a measurable low-level metric and user experience. For this reason, we demonstrate a link between IPC for the CPU and the delivered QoE in this section.

5.3.1 Linking QoE to IPC

Prior to linking the user experience to a IPC, we demonstrate why IPC can, and should be linked to the user experience. We run a frequency sweep for BBench and plot the number of instructions executed on average across the benchmark in Figure 5.8 (A). As the frequency increases, the number of instructions executed also increases, but it does so non-linearly. This relationship is non-linear as the CPU is required to stall for other system components, such as the memory, and more cycles are wasted when the CPU stalls at higher frequencies. IPC is a measure of the wasted cycles, and is able to abstract from the time taken to give a pure measure of CPU efficiency.

In Figure 5.8 (B) we plot the instructions-per-second versus the attained QoE. Again, there is a non-linear relationship between the two metrics. The relationship flattens

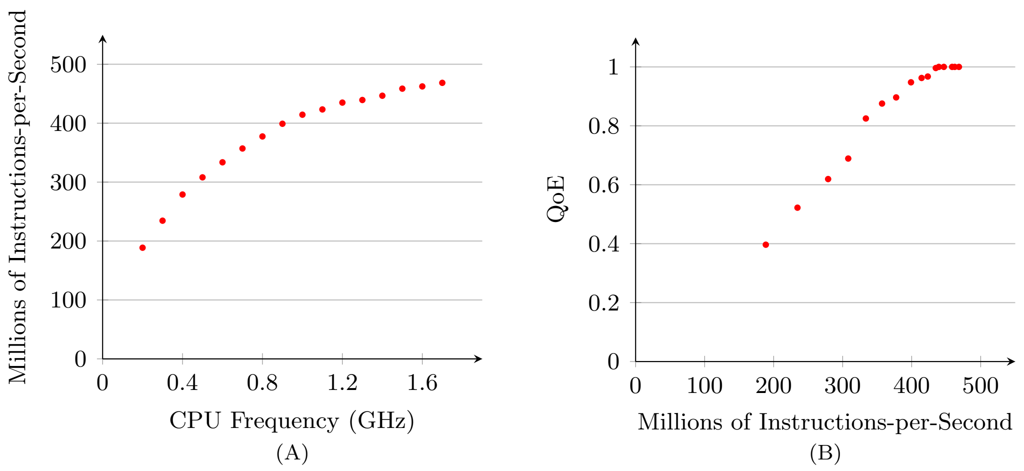


Figure 5.8: Relationship between CPU Frequency, MIPS and QoE for a single-core BBench run

off because the QoE saturates once the user is satisfied, and therefore increases in the number of instructions executed within a time period do not increase the delivered user experience. The QoE saturation can be exploited in order to reduce the energy consumption without affecting the delivered user experience.

5.3.2 Workload Profiling

In order to explore the link between IPC and QoE, we need to profile the workload. This provides us with the average IPC for the complete workload run, as well as the delivered user experience. For future runs of the workload, the IPC can be measured periodically at run time by the OS governor, and the delivered QoE can be predicted. By using the QoE information, the governor is able to determine if the CPU frequency should be increased, decreased or left at the current level.

We begin by profiling BBench — the web browser performance benchmark. We profile BBench by performing a frequency and voltage sweep which covers all voltage-frequency pairs of the Exynos 5250. For each operating point, an entire BBench run is completed and the geometric mean web page load time and relative energy consumption is recorded. In addition, the average IPC of the benchmark is measured. This allows us to relate the operating frequency, relative energy consumption, IPC and workload performance in terms of web page load time and QoE. The results are presented in Figure 5.9 (A).

As is to be expected, when the frequency CPU is decreased the web page render time increases. However, due to interactions with the memory system and devices, the relationship between operating frequency and workload performance is non-linear. This occurs because the memory system has a minimum latency, and can cause the CPU to stall as it waits for data. As the frequency of the CPU increases, the number of cycles wasted per stall increases, and therefore the operation of the CPU becomes less efficient. This results in diminishing returns with regards to the operating frequency, and can also waste significant amounts of energy. At low operating frequencies workloads are often CPU bound, as opposed to memory bound, and therefore are able to operate much more efficiently.

The CPU energy usage when running BBench is also presented in Figure 5.9 (A), and is shown relative to the energy consumption at 1.7 GHz. As expected, as the energy consumption is dependent on the square of the voltage, higher frequencies, and hence voltages, consume more energy to run the benchmark. Due to the inefficiency of the CPU at higher operating frequencies and voltages, it is worth noting that the whilst the overall benchmark run time decreases, the energy consumption increases. This demonstrates that it is not always advantageous to run as-fast-as-possible, before moving to a low-energy state. Naturally, this is dependent on the workload, as well as the performance and user experience requirements placed on the system.

The change in CPU efficiency can be seen in the IPC of the benchmark. Low frequencies result in a much higher IPC than lower frequencies. This demonstrates that fewer CPU cycles are wasted waiting for data from the memory system, or stalling on peripheral interactions. There is a non-linear trade off in the IPC as the frequency changes, which is again due to the effect of the memory latency on the CPU performance. At very low frequencies, the CPU is able to operate more efficiently than at higher frequencies.

Finally, the QoE for BBench is shown in the bottom plot of Figure 5.9 (A). The QoE is calculated using the equations presented in Section 3.4.1. The delivered QoE increases as the frequency increases, but starts to saturate once the memory latency becomes the limiting factor and therefore the CPU begins to stall more frequently as it waits for the data from memory.

The non-linear nature of the QoE-frequency relationship can be exploited in order to save energy as small changes in impact have little-to-no effect on the delivered user experience. In addition, for certain web pages, the QoE saturates as the web page render time drops below that which is deemed acceptable by the majority of the population.

The mapping between QoE and the IPC of the CPU can be used during run time to determine if the CPU frequency needs to be adjusted in order to achieve a minimum target QoE. Specifically, the desired QoE level can be translated into an IPC target which must be met on average. In the case where the IPC at run time is lower than the target then the frequency of the CPU can be reduced as either the current portion of execution is placing a low demand relative to operating frequency of the CPU, or the frequency is too high and will achieve a higher QoE than targeted. The inverse is also true, and higher-than-profiled IPC can be used to indicate that the workload is not running fast enough, and will deliver a comparatively poor user experience.

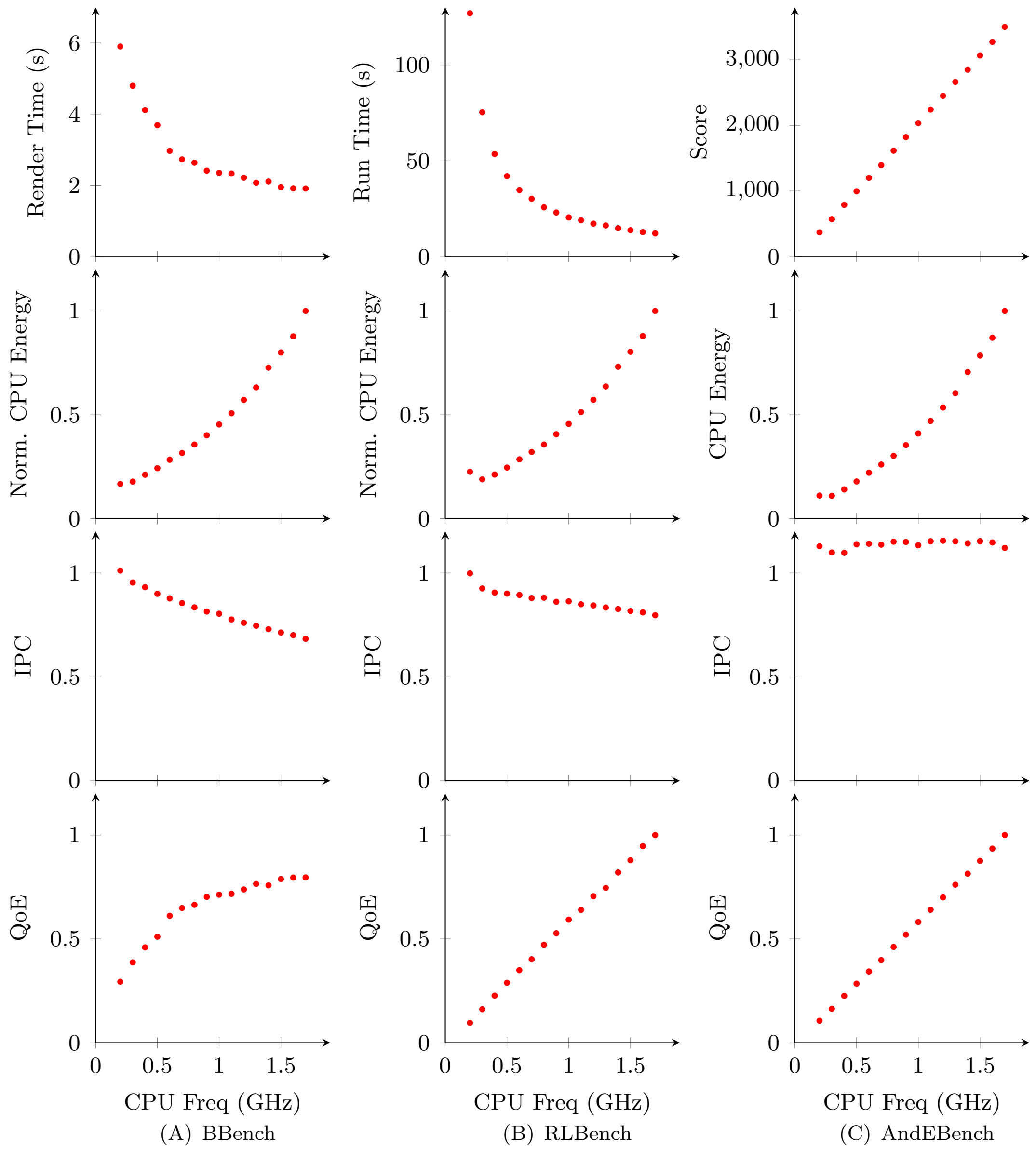


Figure 5.9: Frequency sweep for BBench, RLBench and AndEBench

We also perform a frequency sweep for RLBench, and present the results in Figure 5.9 (B). As before we present the workload performance, the relative energy consumption, the IPC and the QoE for the workload. It is important to note that RLBench is not a user orientated workload. The goal of RLBench is to determine how long a system takes to complete a set of SQL transactions, and rank mobile devices based on SQL performance. Due to this, RLBench presents an absolute time taken for all of the transactions to complete and, in general, the lower this time the better the performance of the system. It is therefore desirable to achieve as low a time as possible, which makes it difficult to assign a QoE metric to the RLBench score. Therefore, we calculate QoE relative to the RLBench score at the highest frequency, which converts the RLBench score into a linear relationship.

The RLBench results display the same non-linear relationship between the CPU frequency and the run time for the benchmark as BBench. Again, the same relationship between the energy consumption and the frequency holds, and whilst the run time decreases, the overall energy consumption increases. The IPC for the workload is non-constant, and reduces as the frequency is increased, which allows us to derive a useful relationship between the IPC and the user experience.

AndEBench is also profiled in the same manner as BBench and RLBench. AndEBench measures the CPU performance using both native, pre-compiled code and interpreted Java code. In order to accomplish this it runs tight loops in both modes, and counts the number of iterations of the loop completed within a fixed time period. Higher iteration counts signify higher CPU performance. As was the case with RLBench, AndEBench is not designed to deliver a service to the user, and simply serves to measure device performance. For this reason, we again choose to set the highest QoE point at the highest attainable score, and normalise lower scores accordingly to obtain the relative QoE.

The AndEBench score demonstrates a linear relationship between the CPU frequency and the score attained. This linear relationship signals that AndEBench does a lot of computation using both small amounts of data, that are most likely able to fit in the cache, and that it uses tight loops. Therefore, there is minimal dependence on the memory latency, and the workload is able to scale with the CPU frequency. This conclusion is also re-affirmed by the near-constant IPC across all frequencies tested. As we will show in later sections, this roughly-constant IPC makes it difficult to determine the delivered QoE at run time and therefore makes it hard to choose the appropriate frequency for the workload.

The following section takes the results of the profiling presented here, and uses the IPC-QoE relationship during run time to adjust the DVFS configuration of the CPU. This allows a minimum QoE to be targeted, and then the governor is able to use the corresponding IPC information to adjust the CPU frequency accordingly.

5.3.3 CPU Governing Results

Following the workload profiling detailed in the previous section, we demonstrate the operation of the QoE-aware CPU governor described in Section 5.2.1.

We consider the three workloads used in the previous section: BBench, RLBench and AndEBench, and initially focus on BBench as it is the most realistic benchmark from a user experience and interaction point of view. Using the profiling results, we specify a target QoE for BBench, translate this into an IPC threshold which is then used with the governor. As detailed in the methodology, we run the governor in a system controller which sits alongside the rest of the system and is able to monitor the performance of various in-system components unobtrusively. This ensures that our monitoring does not affect the performance of the system, as well as accelerating the development of the governor.

We validate the operation of the QoE-aware CPU governor in a single-core and a dual-core system. For the dual-core system, our algorithm scales the DVFS configuration based on the operation of the most efficient core, i.e., the core with the highest IPC. This is done to ensure that the system operates as efficiently as possible and to avoid one idle core from impacting the performance of the system.

We compare the delivered QoE and energy consumption of our QoE-aware governor with the standard, built-in Linux DVFS governors. The Linux CPU governors are designed to trade off performance and power, and are included as part of the Linux kernel. Many of the published DVFS algorithms require specific hardware or software which is not widely available (Choi et al., 2004; Shen et al., 2012; Jung and Pedram, 2010). Therefore, we use the Linux CPU governors as a starting point to evaluate the QoE-aware CPU governor presented herein as they act as a reference for the techniques used in real systems. Please note that energy has been normalised to the *Performance* governor, which represents the worst case energy consumption, but is not a normally used governor in mobile systems.

5.3.4 BBench

We begin by looking at the QoE-aware governor for BBench. Initially, we look at a single core system, and then move on to a dual core system.

5.3.4.1 Single-Core

We run BBench using the *Powersave*, *Interactive*, *OnDemand*, *Conservative* and *Performance* governors built into Linux, and present the results in Figure 5.10 (A). As expected, the *Performance* governor represents the fastest that the workload can run, hence achieving a high QoE, but also the maximum energy consumption, as the CPU is constantly at its highest frequency. The *Powersave* governor represents the opposite case as the CPU runs at its lowest frequency, and the workload takes the longest time to run. Therefore, it has the lowest QoE and also represents the lowest dynamic energy consumption. The results from these two governors represent the best and worst case performance and energy consumption.

The *OnDemand* governor adjusts the CPU DVFS based on a work queue and runs at a low frequency until demand increases. At this point it increases to the maximum frequency and then gradually reduces the frequency until a sensible level is reached. There is a clear slowdown when using the *OnDemand* governor as is reflected in the QoE. The *Interactive* governor behaves in a similar way to the *OnDemand* governor, except that

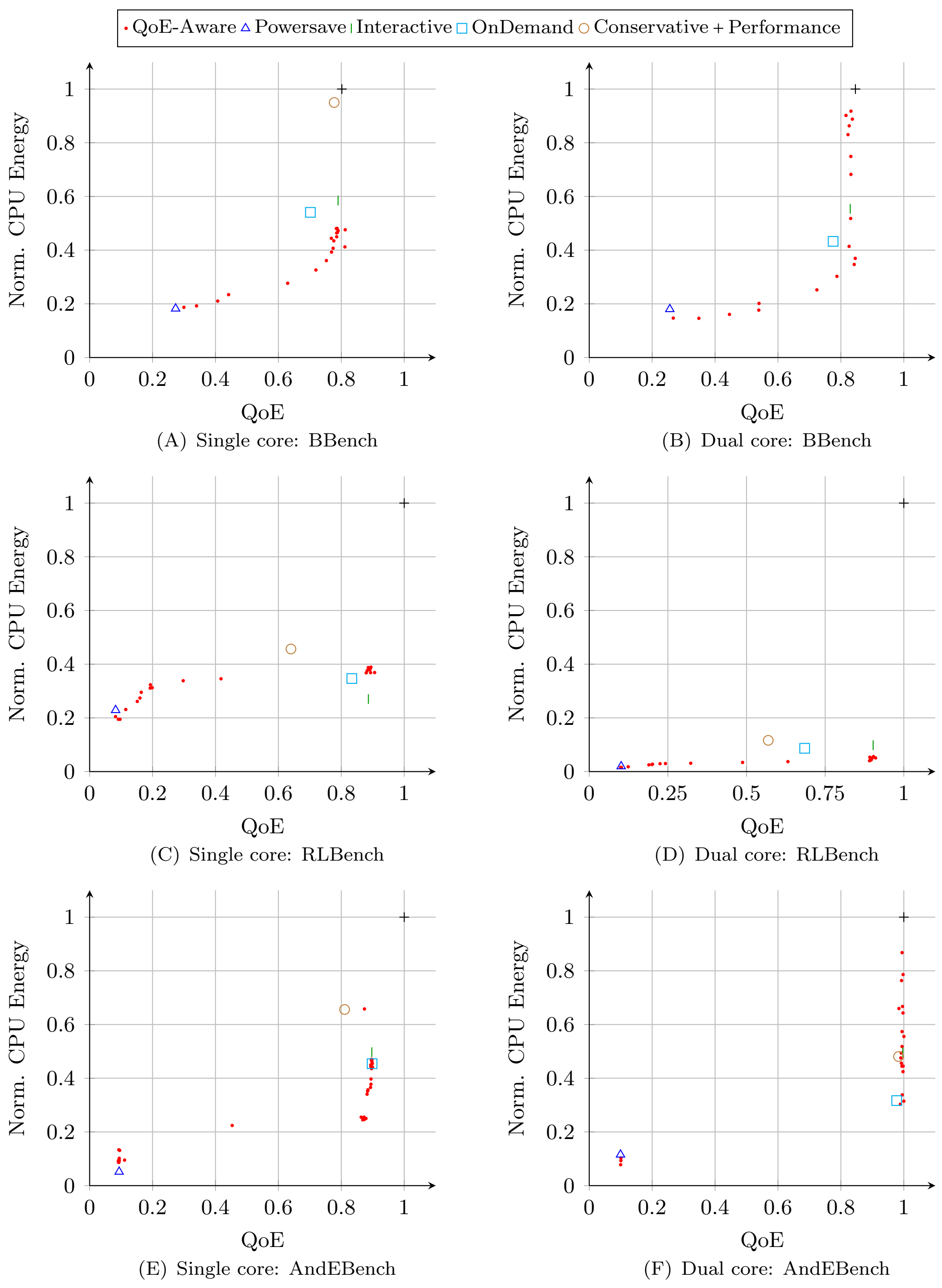


Figure 5.10: Trading off CPU QoE and energy usage in single and dual core systems for BBench, RLBench and AndEBench

rather than a single application triggering an increase in frequency, the governor observes the overall CPU load over a timer period. Figure 5.10 (A) shows that the *Interactive* governor does not cause a significant slowdown relative to the *Performance* governor. The *Conservative* governor behaves in a similar method to the *OnDemand* governor but switches frequency much more slowly. Therefore, if there are sudden changes in the workload, the *Conservative* governor can be slow to respond and the CPU frequency can be either too high or too low for extended periods of time. The increased energy consumption can be seen in Figure 5.10 (A).

Figure 5.10 (A) also shows the performance of the IPC based QoE-aware governor presented in this work. We sweep the QoE target in order to determine the impact it has on the CPU performance and energy consumption. Our proposed governor is able to remain above a specified QoE target by monitoring the IPC of the CPU and adjusting the performance accordingly. This demonstrates that it is possible to measure a simple metric which can be found in the performance monitoring unit of many mainstream CPUs in order to successfully track an optimised operating point, as described in Section 5.2.1.

Our QoE-aware CPU governor is able to outperform the *OnDemand*, *Conservative* and *Interactive* CPU governors in terms of QoE and energy consumption. As the governor is able to select operating points which have little-to-no effect on the benchmark performance, there is additionally minimal impact on the perceived experience, QoE. By adjusting the QoE target we are able to adjust the trade off we make and therefore are able to tune the performance-energy trade off.

We present the frequency distribution for the QoE-aware governor in Figure 5.11. We run the governor in three different configurations: low QoE, good enough and highest QoE. In the low QoE configuration, the governor favours low frequencies, and spends over 77 % of the execution time at the lowest frequency, 200 MHz. Negligible time is spent at frequencies above 700 MHz in the low QoE configurations.

The inverse is true in the highest QoE configuration, and the governor spends the majority of the time choosing frequencies at the high end. 1700 MHz is chosen for almost 62 % of the execution time. Some time is also spent at lower frequencies, including 200 MHz as the governor will drop to the lowest frequency automatically if the CPU is idle, and will then gradually increase the frequency in response to the estimated QoE when the CPU resumes execution.

Finally, in the case where the governor is configured to deliver good-enough QoE for the workload, a large proportion of the time is spent at frequencies between the two extremes. This demonstrates that the governor attempts to run at the target IPC, determined from the desired QoE, and will adjust the frequency in order to achieve this goal. In this configuration, there is still a significant proportion of time spend at the lowest and highest frequencies. This is due to the fact that the governor will drop to the lowest frequency when idle, and will saturate at the highest frequency.

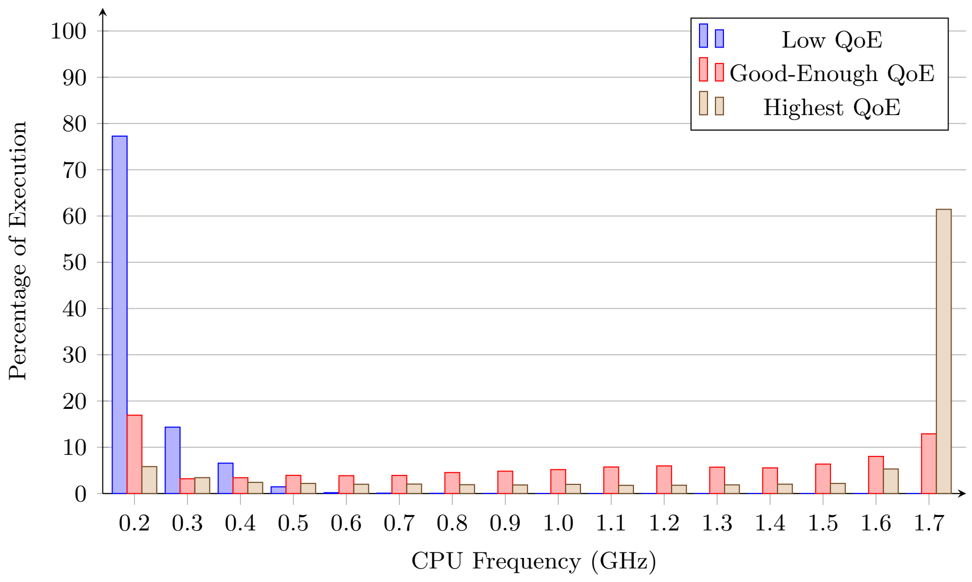


Figure 5.11: Frequency distribution for low, good-enough and highest QoE governor configurations

5.3.4.2 Dual-Core

We run BBench in a dual-core system, and present the results in Figure 5.10 (B). As both cores are in the same voltage and frequency domains, the overall energy consumption increases. However, the maximum attained QoE increases as there are more computational resources available to process the web pages.

As was the case in the single-core setup, the QoE governor is able to deliver a higher QoE than the *OnDemand* governor whilst reducing the energy consumption. It matches the QoE delivered by the *Interactive* governor, but again is able to do so while consuming less energy.

5.3.5 RLBench

In this section, we look at the QoE-aware governor operation for RLBench in single- and dual-core systems. RLbench is a workload that runs a set of smaller SQLite tests to benchmark the SQLite performance for an Android system.

5.3.5.1 Single-Core

Figure 5.10 (C) presents the results for RLBench using the same set of governors. Figure 5.10 (C) shows that it is possible to greatly reduce the energy consumption of the CPU at minimal QoE reduction. Due to the bursty nature of the workload, the *OnDemand* and *Interactive* governors are able to do well as they can immediately switch up to the maximum frequency. Our QoE-aware governor, on the other hand, gradually increases and reduces the frequency causing a small performance and energy consumption penalty for the benchmark. The *Conservative* governor is slower to switch and therefore has both poor QoE and increased energy usage.

5.3.5.2 Dual-Core

Figure 5.10 (D) shows the RLBench results for a dual-core system. In the case of RLBench, the workload is mostly single threaded and therefore there is little improvement in the workload performance when the operating frequency of the CPU is increased. The power consumption of the CPU increases for the *Performance* governor, but is reduced for other governors. This is due to the fact that RLBench is single-threaded and therefore it will only run on one of the two cores, leaving the other free for background OS processes. Therefore, these background processes are able to run without interfering with the operation of the benchmark.

In all cases the QoE-aware governor is able to reduce the power consumption to below the level of the standard Linux governors. However, as RLBench is a bursty workload, there is a small impact on the QoE for any target chosen. This is due to the time taken for the governors to increase the frequency in response to load from the workloads. However, this applies to all governors tested and is not specific to the QoE-aware governor.

5.3.6 AndEBench

Finally, we observe the QoE-aware governor when running AndEBench, which has a constant IPC-frequency relationship. This makes it difficult to determine an appropriate IPC target for the CPU. This makes it harder to govern the CPU SVFS operating points using the IPC.

5.3.6.1 Single-Core

We present results for AndEBench in Figure 5.10 (E), and demonstrate the ability to match the Linux DVFS governors in terms of QoE. AndEBench measures the performance of pre-compiled native code and the Dalvik virtual machine for the CPU. The AndEBench native benchmark is designed to be very computationally intensive, as well as efficient. In addition most of the data required for the benchmark can be locally cached, and therefore there are few accesses to main memory once the benchmark is running. Therefore, the whole benchmark operates with a high IPC. Due to this, there is a very sharp trade-off between the QoE target set for the QoE-aware governor and the benchmark performance. With the QoE-aware governor we are able to select a target such that the workload performance, and delivered QoE, is not impacted, but there are significant savings in consumed energy. As Figure 5.10 (E) shows, we are able to perform at similar levels to the *Conservative*, *Interactive* and *OnDemand* governors whilst using significantly less energy. However, it is difficult to choose a sensible threshold due to the IPC-frequency relationship.

5.3.6.2 Dual-Core

AndEBench is a multi-threaded workload, and therefore places strong demands on both cores in the system. Therefore, it has similar characteristics to the single-core system as the amount of work done by the workload increases rather than being distributed across the two cores. However, due to the constant IPC when the frequency is adjusted, it is difficult to choose an IPC level which is able to reduce the energy whilst still maintaining the required levels of performance. The QoE-aware governor is able to target a performance level which matches that of any of the Linux governors, and it is able to match the OnDemand governor in terms of energy savings.

5.4 Uniting CPU Efficiency, Application Performance and User Experience on Real Hardware

In order to optimise the CPU for the user experience, we need a mapping between low-level metrics, the performance of the CPU itself and the user experience delivered. We create this mapping on a per-application basis as a proof-of-concept. The low-level metrics must be easily available on real hardware, as it must be possible to query the system at run time in order to optimise dynamically. Therefore, we are limited to standard metrics which can be found in the Performance Monitoring Unit (PMU) of a typical CPU. The PMU contains a range of different counters which are incremented on specific system events, such as on a CPU clock cycle or when an instruction is committed. In general, there are counters for the CPU and the caches, but some systems may also have counters for the memory controller, GPU and other in-system devices. However, as we are focusing on optimising for the CPU in this chapter, we only consider CPU metrics.

When optimising for user experience, it is also important to consider how efficiently the CPU is operating, and to understand when increasing the CPU frequency will have close to no effect on the delivered performance and user experience. A typical measure of CPU efficiency is the number of instructions committed per clock cycle (IPC). A higher IPC indicates that the CPU is operating more efficiently as it is able to commit a greater number of instructions for the same number of clock cycles as at lower IPC values. In general, as the frequency of the CPU increases, the IPC will drop as the ratio of the memory latency and the CPU clock period increases. Therefore, a cache miss results in a greater number of wasted cycles at higher frequencies than at lower frequencies. It is worth noting that the IPC is not only affected by the memory latency and CPU frequency, but also by other device accesses as well as the workload itself.

As was covered in Chapter 4, the efficiency of the CPU is largely a function of the workload running on it. Therefore, we are required to look at the IPC characteristics for each workload we wish to analyse. Extremely compute-bound workloads, such as Dhrystone, will have a constant IPC irrespective of the frequency as they are only reliant on data in the caches. Memory-bound workloads, such as Memcpy can have a vastly different IPC depending on the CPU frequency as they are limited by the bandwidth to and from memory. Hence, increasing the CPU frequency when the workload is already memory-bound results in a large decrease in IPC with close to no performance improvement.

5.4.1 Profiling the Workloads

We begin by linking the efficiency of the CPU to the raw application performance and to the level of user experience delivered. We choose to measure the CPU efficiency by observing the IPC when running an application. This gives a clear idea of the proportion of CPU cycles that are being wasted, and allows us to determine how efficiently the CPU is operating. Additionally, as the CPU frequency is increased, the IPC will drop for non-compute-bound applications giving a clear idea of how compute-bound a particular application is.

We profile applications on the Odroid XU3 development board using the same setup as in Section 4.2. Specifically, we run applications on a single A15 core, and sweep the CPU frequency from 1.2 GHz to 2 GHz in 100 MHz increments. For each increment we measure the performance of the workload, as well as the number of instructions and cycles which occurred during the application execution. These are measured using the Performance Monitoring Unit (PMU) which contains a set of hardware counters which can be used to count occurrences of specific events such as CPU clock cycles or instructions committed.

5.4.1.1 Dhrystone

We begin by looking at the relationship between IPC and performance for a compute-bound workload, Dhrystone. We show the relationship between IPC, total execution time and Dhrystone MIPS in Figure 5.12. As compute-bound workloads are not significantly affected by the performance or connection to other in-system components, they are limited by the performance of the CPU. Therefore, higher CPU frequencies should

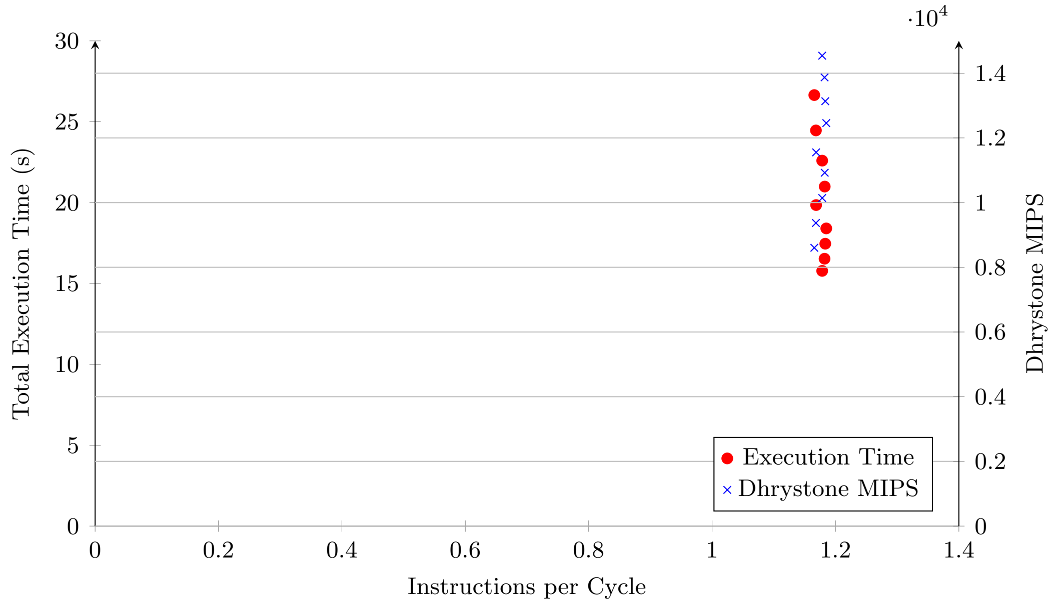


Figure 5.12: Instructions per Cycle versus Execution time and DMIPS for Dhrystone benchmark running on the Odroid XU3.

result in higher performance in the majority of cases. When running a frequency sweep for Dhrystone, we notice that the IPC does not change significantly as the operating frequency is adjusted. This is the expected behaviour as the workload is limited by the capabilities of the CPU itself, and each iteration takes a fixed number of instructions and cycles. For a workload such as Dhrystone, the majority of variation comes from background processes and the operating system itself. The workload has an approximately constant IPC for all frequencies profiled on the development board.

5.4.1.2 Memcpy

Memcpy, as previously discussed, is a very memory-bound workload. This is especially the case for the range of frequencies tested on the Odroid XU3. Figure 5.13 shows the relationship between IPC, total execution time and memory bandwidth. Please note that the Memcpy IPC-performance relationship is affected by the memory frequency changes discussed in Section 4.2.2. Hence, when increasing from low frequencies the IPC initially reduces and the performance of the workload improves insignificantly, before a large jump in performance is seen due to the increased memory bandwidth available.

5.4.1.3 The Chase

As previously discussed, The Chase is a graphics-heavy workload. However, this workload is reliant of the operation of the CPU to pre-compute the frames before they are passed off to the GPU for rendering. Therefore, it is vital that the CPU is clocked fast

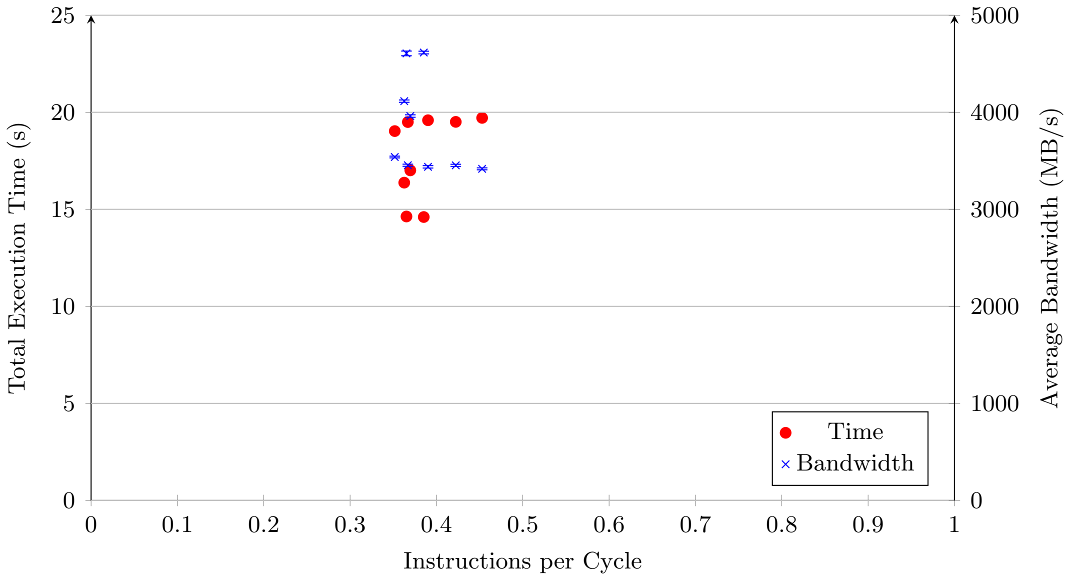


Figure 5.13: Instructions per Cycle versus Execution time and Bandwidth for Memcpy benchmark running on the Odroid XU3.

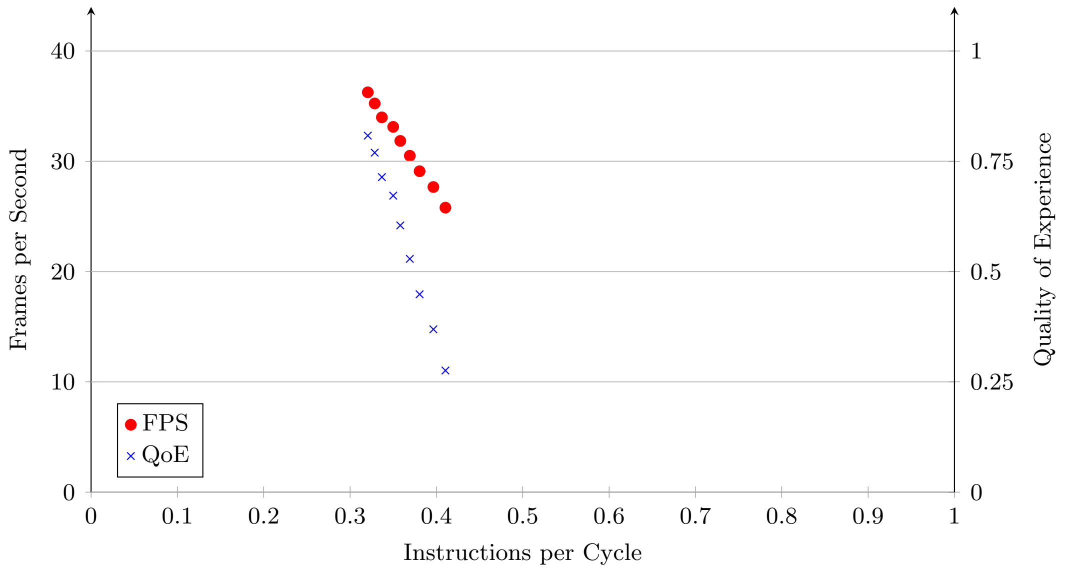


Figure 5.14: Instructions per Cycle versus Frames per Second & QoE for The Chase on the Odroid XU3

enough to ensure that the delivered user experience is high. We present the relationship between IPC, frames rendered per second and QoE for The Chase in Figure 5.14. The Chase has a low IPC, in general, which decreases as the frequency of the CPU is increased. There is a close to linear relationship between the IPC of the workload, and the delivered level of performance and user experience. The QoE is calculated assuming a target frame rate of 60 Hz. The near-linear relationship between the IPC and the QoE provides a large range of frequencies over which it is possible to trade off user experience for energy consumption.

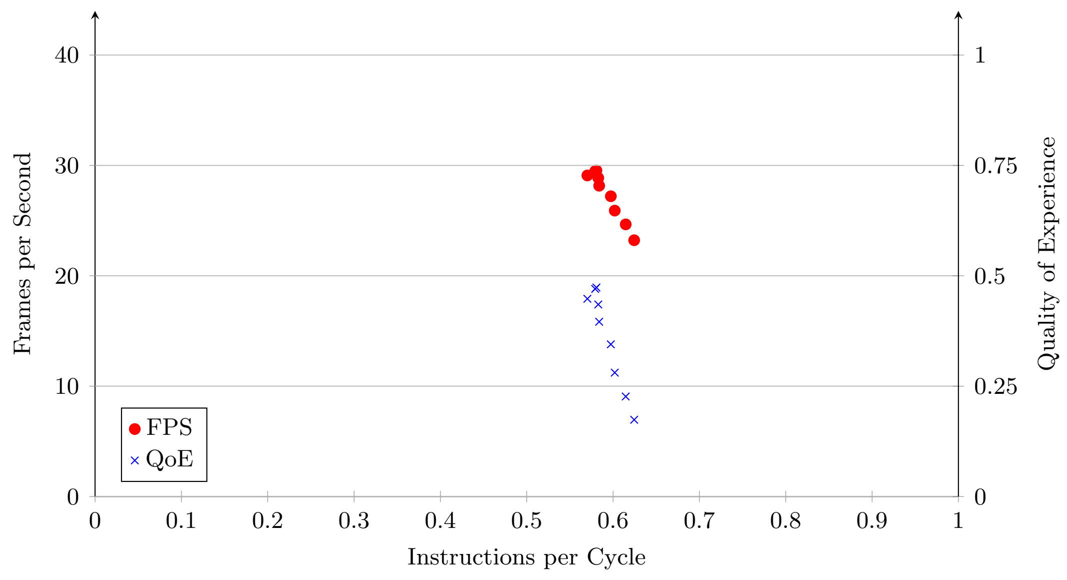


Figure 5.15: Instructions per Cycle versus Frames per Second & QoE for Transporter on the Odroid XU3

5.4.1.4 Transporter

Finally, we consider Transporter which is another GPU heavy workload. However, this workload places a greater strain on the CPU as the CPU is used to calculate the lighting for the scene, and therefore places a limit on the QoE that can be delivered by the workload. Figure 5.15 presents the relationship between the IPC for the workload, the frames rendered per seconds and the level of user experience delivered. As was the case for The Chase, there is a large performance and QoE variation for the range of frequencies tested. Due to the heavier reliance of the CPU to compute the lighting for the scene, Transporter has a higher IPC than The Chase, and the frame rate delivered is also lower. As the frequency of the CPU is increased, the frame rate and QoE increase, whilst the IPC drops due to the increased number of wasted CPU cycles.

5.4.2 QoE-Aware Governing on Real Hardware

In the previous section we presented the relationship between the IPC and the performance for a set of workloads. In this section, we run our IPC-based DVFS governor on real hardware, specifically the Odroid XU3, and sweep the target IPC. We observe how the operation of the workload changes as the IPC target is swept.

5.4.3 Dhrystone

Dhrystone has a constant IPC irrespective of the operating frequency on the Odroid XU3. We present the results of an IPC target sweep in Figure 5.16, and show how the

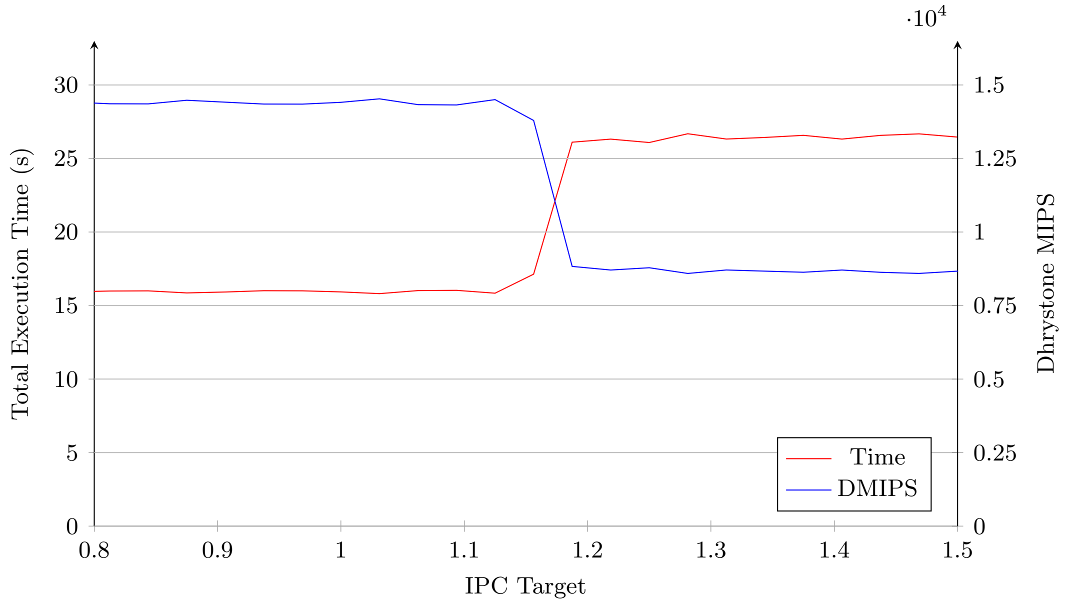


Figure 5.16: Dhrystone Odroid total execution time & DMIPS as IPC target is swept

run time and performance in terms of DMIPS changes as the IPC target is changed. When the IPC target for the governor is swept there is no change in the performance of the workload until the IPC target reaches a value of 1.2. This is the value we saw when profiling the workload in Section 5.4.1.1. Once this IPC target is reached the performance of the workload rapidly declines before plateauing. Due to the constant IPC nature of the workload there is little scope for adjusting the performance of the governor by changing the IPC target.

5.4.4 Memcpy

The second workload we consider is the memory-bound Memcpy. When profiling the workload in Section 5.4.1.2 we observed that Memcpy had a much lower IPC than Dhrystone due to the workload being mostly memory bound. Therefore, the CPU spends a large portion of time waiting for the memory system, resulting in a low IPC. We present the results for an IPC target sweep in Figure 5.17, which relates the IPC target to the time taken to run the workload, as well as the average transfer bandwidth measured by the workload. During profiling we saw that Memcpy had an IPC of around 0.4. When the IPC target reaches the value we see that the performance of the workload is rapidly reduced.

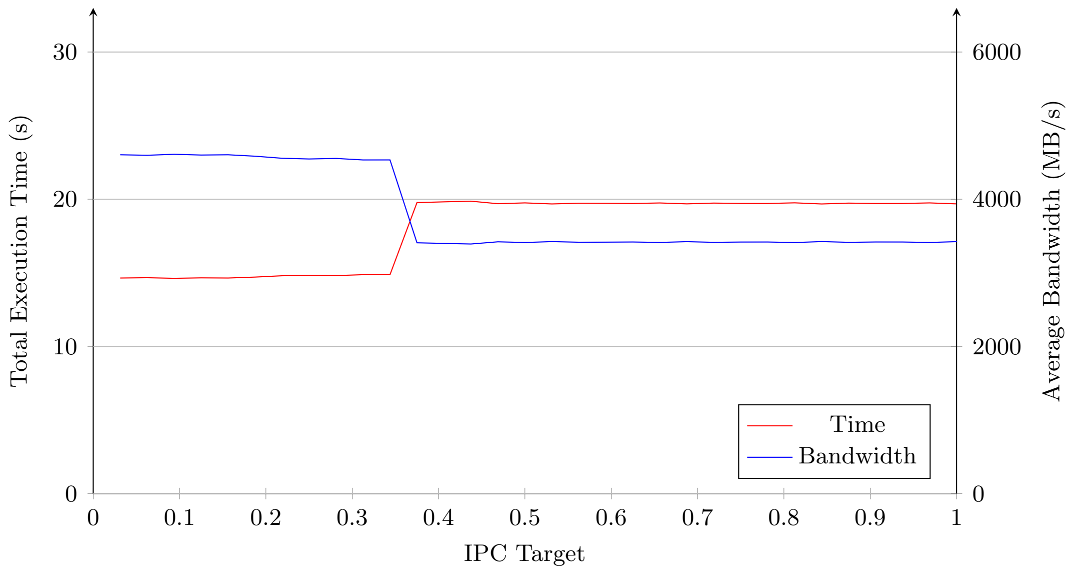


Figure 5.17: Memcpy Odroid total execution time & DMIPS as IPC target is swept

5.4.5 The Chase

The Chase is the first graphically intensive workload that we perform an IPC target sweep on. When profiling the workload in Section 5.4.1.3 we observed that there was a near linear relationship between the IPC for the workload, and the number of frames rendered per second, as well as the QoE. Figure 5.18 shows the results of the IPC target sweep, and shows the relationship between the IPC target, the number of frames rendered per second and the QoE delivered by the workload. The Chase is the first workload where we are able to make meaningful trade-offs by varying the IPC target. For Dhrystone and Memcpy we were able to force the CPU to a low or a high frequency by adjusting the IPC target, but we were not able to easily pick an intermediate level of performance. For those workloads, this is the expected behaviour when using IPC to adjust the CPU frequency. However, for The Chase we are able to choose an IPC target which gives a well defined intermediate level of performance, and hence we are able to use the IPC target to tune the performance and QoE delivered by the workload to our needs. By varying the IPC target between 0.25 and 0.375 we are able to adjust the QoE delivered by the workload from 0.76 down to 0.3.

5.4.6 Transporter

Transporter is the second workload that we run on the Odroid XU3 that has heavy reliance on the GPU. However, as previously discussed, this workload places a high load on the CPU due to the lighting being computed on the CPU. When profiling this workload in Section 5.4.1.4 we saw that it had a higher IPC than The Chase. We present the relationship between the IPC target, the frames rendered per second and QoE in

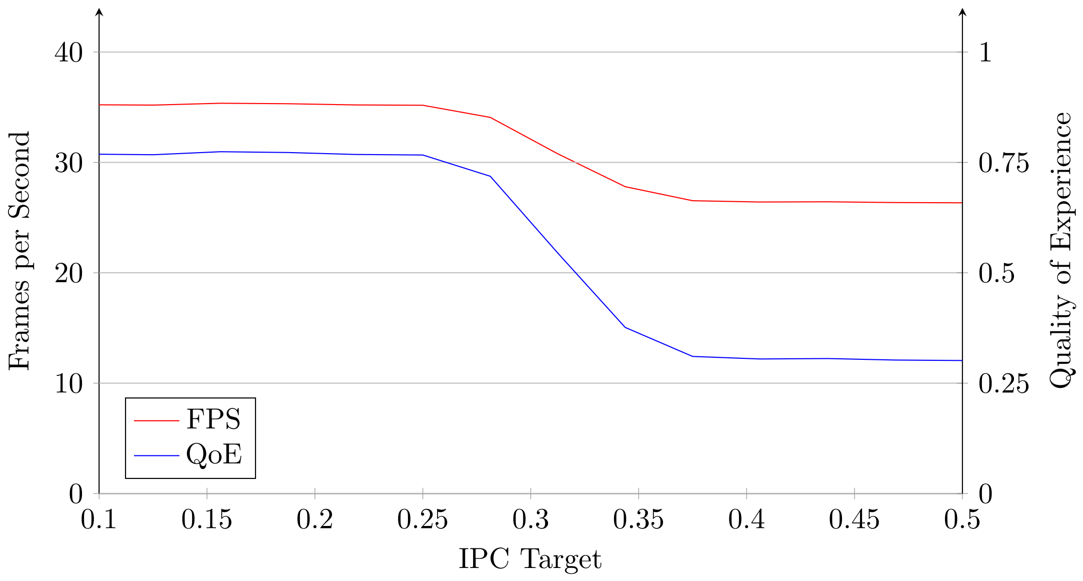


Figure 5.18: The Chase Odroid FPS & QoE as IPC target is swept

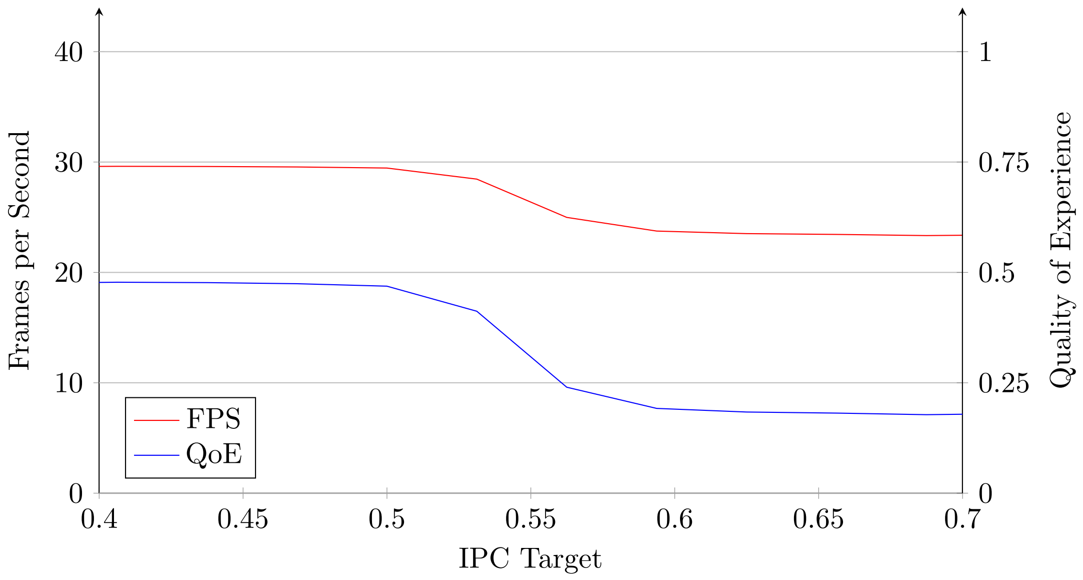


Figure 5.19: Transporter Odroid FPS & QoE as IPC target is swept

Figure 5.19. As was the case for The Chase, we are able to control the performance of Transporter well by adjusting the IPC target. In general, Transporter offers a lower level of performance and user experience than The Chase. However, by adjusting the IPC target from 0.5 to 0.6 we are able to reduce the QoE from 0.48 down to 0.19.

5.5 Concluding Remarks

In this chapter we have built upon the CPU profiling presented in Chapter 4, and have presented both an *Oracle* CPU governor, and an IPC-based CPU governor. These governors use information about the workload to determine when the frequency should be increased or decreased. We have demonstrated that we are able to save energy relative to the DVFS governors used in modern devices by implementing a DVFS governing strategy based on system performance counters and workload profiling.

We have demonstrated that a simple *Oracle* CPU governor is able to save energy relative to the standard *Ondemand* and *Conservative* CPUFreq governors used on modern devices. The standard governors both monitor the CPU load as measured by the kernel, and increase or decrease the frequency based on the measured load. Both of these governors employ hysteresis to avoid changing the frequency too rapidly, and therefore can be both slow to react to workload changes, but can also waste a large amount of energy. We demonstrated that our *Oracle* governors, which measure how well the workload responds to frequency scaling, is able to select more energy efficient operating points than the standard governors. Thus, our *Oracle* governors are able to save energy, whilst maintaining similar levels of performance.

Our *Oracle* governors measured the IPC of the workload for each operating frequency, and compared this value to determine the most appropriate frequency for a governing interval. Due to the successful operation of the *Oracle* governors, we decided to base our CPU governor implementation on IPC. Our DVFS governor utilises the performance counters found in modern CPUs to measure the IPC of the component without impacting the performance or energy consumption in a measurable way. We combined workload profiling with the IPC-based governor to determine an IPC threshold for each workload. Therefore, for each workload, we established a link between the IPC of the workload, the performance of the workload and the level of user experience delivered. This allowed us to choose an appropriate IPC threshold for the DVFS governor. We verified the operation of the DVFS governor in both simulation and in real hardware, and have compared the operation of our DVFS governor to those found on real hardware.

Chapter 6

Optimising Complete System User Experience

Mobile systems are comprised of a number of different components which must work together in order to provide a high level of user experience. When a single component does not operate as intended it can impact the user experience for the system as a whole, and therefore great care must be taken when optimising the system. In Chapter 3 we discussed QoE models for different workload types, and how the QoE for a complete system can be calculated. In the subsequent chapters, we focused heavily on optimising the CPU, which is one of the most important single components as it runs the operating systems and the majority of the code for many applications. Therefore, we followed the user experience modelling chapter by an analysis for CPU efficiency in Chapter 4, which considered how the CPU and different CPU workload types responded to the memory system and the frequency of the CPU. Finally, in Chapter 5 we acted upon the knowledge gathered in the previous chapters, and presented an IPC-based CPU DVFS governor which was able to adjust the frequency of the CPU for a range of workloads through a specified IPC target.

In this chapter we consider the operation of the system as a whole. We being by providing a performance analysis for another important system component: the GPU. This answers our third research question: *Across a range of typical graphics workloads, what effect does the scaling of GPU core count and available memory bandwidth have in terms of performance and user experience?* The GPU is responsible for rendering the frames before they are displayed by the display controller. Therefore, if the GPU is not able to render the frames in time, the frame rate reduces and the QoE delivered by the system is reduced as the end user starts to notice some stuttering. Hence, it is vital that the GPU is able to operate at a sufficiently high level to ensure that the QoE remains high enough to satisfy the end user.

Introducing a GPU into the system results in additional memory traffic, which in turn increases the latency to memory for the other system components, such as the CPU. As we discussed in Section 4.1, the CPU is a latency-sensitive component, and higher memory latency can result in significantly decreased CPU performance. This means that it is vital to understand the trade-off between the CPU performance and the GPU performance to avoid one component significantly impacting the operation of the other.

In Section 6.1 we begin by profiling the operation of the GPU, and investigate how the performance of the GPU is impacted by the number of active GPU cores and by the available memory bandwidth. We look at the operation of a set of typical GPU workloads as the available bandwidth is artificially varied in a simulation. We then observe how the introduction of GPU traffic impacts the performance of a CPU workload, BBench, in Section 6.2. We then use this information to determine how to best trade off the performance and requirements of the CPU for those of the GPU in Section 6.3. We introduce a GPU governor which adjusts the number of active GPU cores based on the user experience delivered by the GPU in Section 6.4. Next, in Section 6.5 we combined the CPU and GPU governors into a combined system governor, which adjusts the operation of the CPU and GPU in tandem and attempts to maintain an overall high QoE. Finally, in Section 6.6, we present an extension to the IPC-based DVFS governor presented in Chapter 5 which also takes into account the memory accesses made by the workload to determine which CPU frequency should be chosen.

6.1 Analysing and understanding the operation of the GPU

In this section, we observe how the GPU responds to core scaling and bandwidth limitation and observe GPUs transitioning from being memory bound to compute bound. This information allows us to predict how the GPU will respond to different QoS mechanisms which may affect the bandwidth available to the GPU.

6.1.1 Workloads

The GPU frames used for this chapter are summarised in Figures 6.1 and 6.2. Egypt, and Pro are from GLBenchmark 2.1 (GLBenchmark, 2012), whereas Taiji is from RightWare's Basemark ES 2.0 (Rightware, 2012) and the Navigation benchmark, henceforth referred to as Navi, is from 3DMark Mobile 2.0 (Futuremark, 2012). All frames are rendered at a resolution of 1920x1080 pixels (Full HD) and represent typical resolutions used for current high end devices. These frames have been selected as they represent a set of real use cases, which cover both gaming and navigation, as well as providing a varied set of bandwidth requirements.

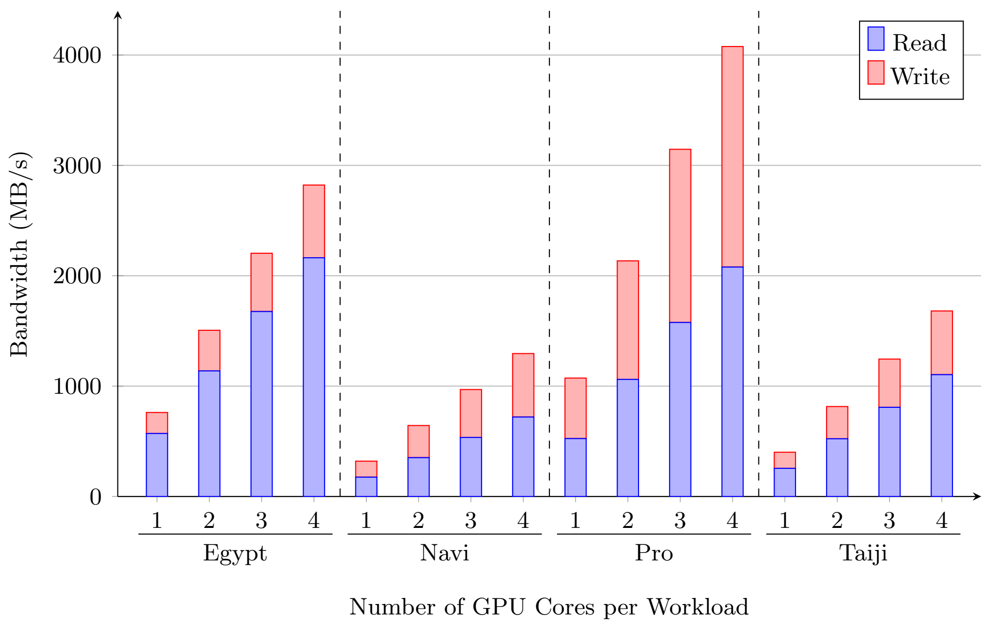


Figure 6.1: GPU Frames - Read, write and total bandwidth at maximum FPS as number of cores scales

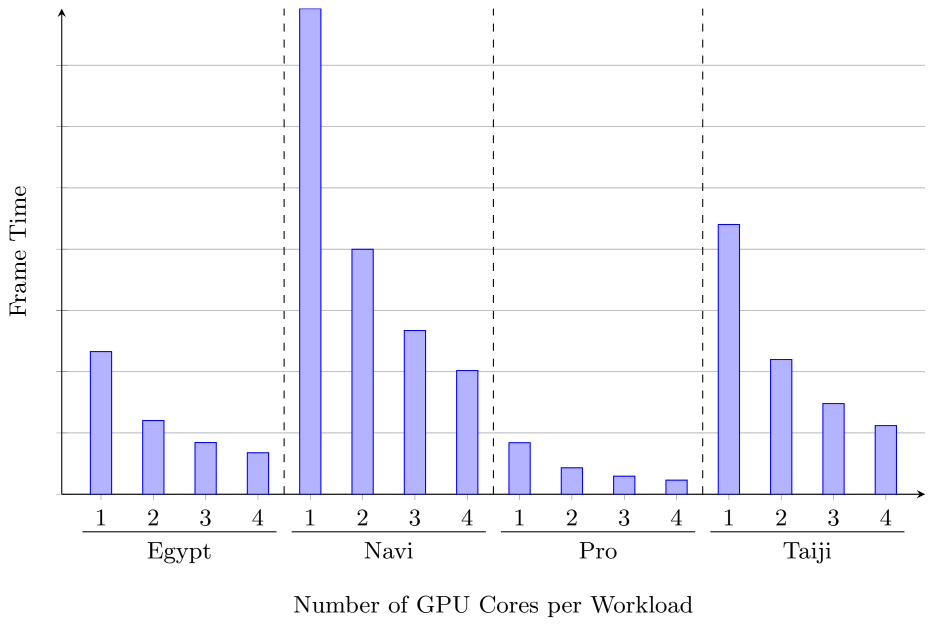


Figure 6.2: GPU Frames - Relative frame time as number of cores scales

6.1.2 Core Count Scaling

We begin by observing how the GPU bandwidth requirements and performance vary as the number of available cores are varied for a set of benchmarks. The GPU is standalone, i.e. there is no interaction between the CPU and the GPU. The frames are rendered independently of the workload running on the CPU, and are not controlled by the CPU. The frames rendered on the GPU are supplied as a memory dump which includes all of the information required by the GPU to render the frame. The GPU model is responsible

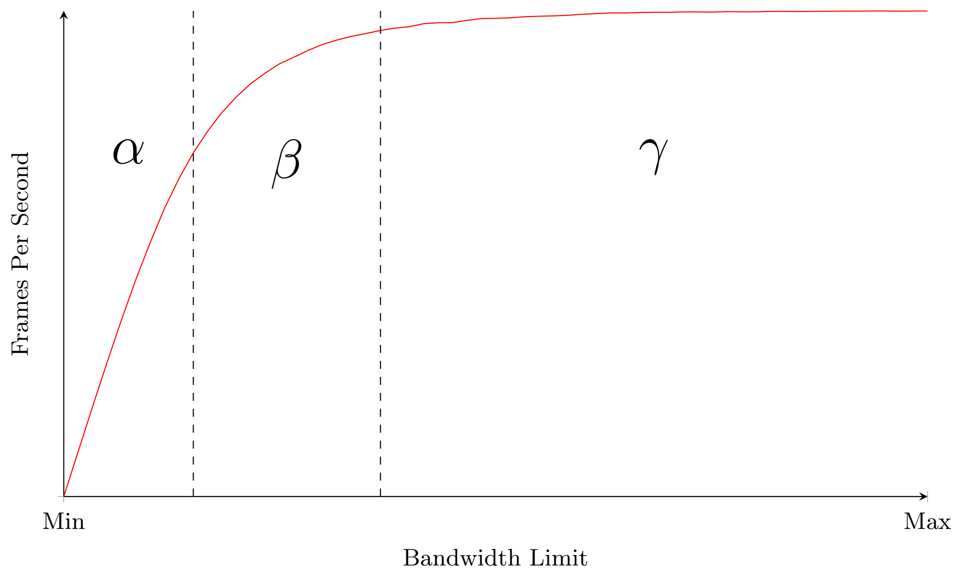


Figure 6.3: Response of GPU to bandwidth restriction. α is bandwidth limited, γ is compute limited and β is the transition from bandwidth to compute limited.

for loading the data it processes into the memory of the simulated system, rather than it being calculated and placed there by the CPU. At the start of a GPU frame, the relevant memory locations are filled by loading a GPU memory dump into the memory. This happens instantaneously as far as the simulated system is concerned, and is therefore not included in our investigation.

Next, we look at the traffic generated by a GPU, as well as investigating at how a GPU responds to being bandwidth limited.

6.1.3 Bandwidth Limitation

GPUs place noticeably different demands on the memory system when compared to a CPU. This section looks at the performance of a GPU when it is under bandwidth constraints, as well as how to improve the CPU performance by reducing the amount of available GPU bandwidth.

Unlike a CPU, a GPU is a bandwidth sensitive device; provided a GPU gets a sufficiently high average bandwidth to memory, it is tolerant of periods of high latency (Jeong et al., 2012; Wang, 2011). One caveat is that a GPU needs to finish rendering the frame before it is due to be displayed on the screen; failure to do this results in poor user experience as the frame rate suffers.

A GPU can operate in one of two states: bandwidth limited or compute limited. When the GPU is bandwidth limited (Figure 6.3α) it is unable to render at its maximum rate as it is unable to get enough bandwidth to memory. As the GPU starts to become compute saturated the bandwidth it requests from the memory begins to level off (Figure 6.3β)

as the GPU is not able to process it at the arrival rate. If the bandwidth available to the GPU is increased slightly, then the GPU becomes permanently compute limited, and renders the frame at the fastest possible rate (Figure 6.3γ).

When the GPU is bandwidth limited, the GPU spends a significant proportion of time waiting for the data it requires to arrive from memory. Conversely, when the GPU is compute limited the GPU is constantly processing the data. Although this results in the fastest possible render time and the best possible user experience, this creates unnecessary contention. Supplying a compute limited GPU with extra memory bandwidth has little to no effect on the experienced performance.

When the GPU is between the bandwidth and compute limited states, it is able to render at an almost maximum frame rate. By varying the bandwidth available to the GPU in this region, it is possible to efficiently trade off GPU bandwidth for performance, which in turn affects the performance of the CPU.

As the GPU moves from being bandwidth limited to compute limited, we see diminishing returns as small quantities of additional bandwidth have little effect on the frame rendering time as the GPU becomes more compute bound (a real example is shown in Figure 6.4). It is therefore not beneficial to provide the GPU with too large bandwidths as this does not result in efficient GPU operation, and simply increases costs as the system becomes over dimensioned.

When the GPU is moving between the bandwidth and compute limited states, there is a non linear trade off between maximum bandwidth usable by the GPU, and the GPU frame rate. By limiting the available GPU bandwidth the bursty parts of the traffic profile are smoothed out, reducing the performance slightly, but resulting in a more constant, lower bandwidth traffic profile. This has less impact on the memory system, and therefore the rest of the system.

The GPU model renders each frame as a set of tiles, which allows different GPU cores to process sections of the final frame in parallel. The demand on the GPU varies with each tile, and therefore the bandwidth to the GPU varies significantly. This introduces a high degree of unpredictability into the GPU traffic, and makes it more complicated to model as a synthetic traffic source. Sample traffic for a single core Taiji run is shown in Figure 6.5.

A GPU requires a minimum amount of data in order to render the frame; this bandwidth increases once a GPU has multiple cores and performs tile based rendering due to the increased sharing and the limited caching available for an integrated GPU. Stevens (2010) provides us with some typical bandwidths for a SoC with GPU, video decoder and LCD controller which are running at a resolution of 1920x1080. In total, these are said to require around 2.5 GB/s of bandwidth alone. Experiments show that the real resource requirements are even higher still, as the GPU is able to consume over

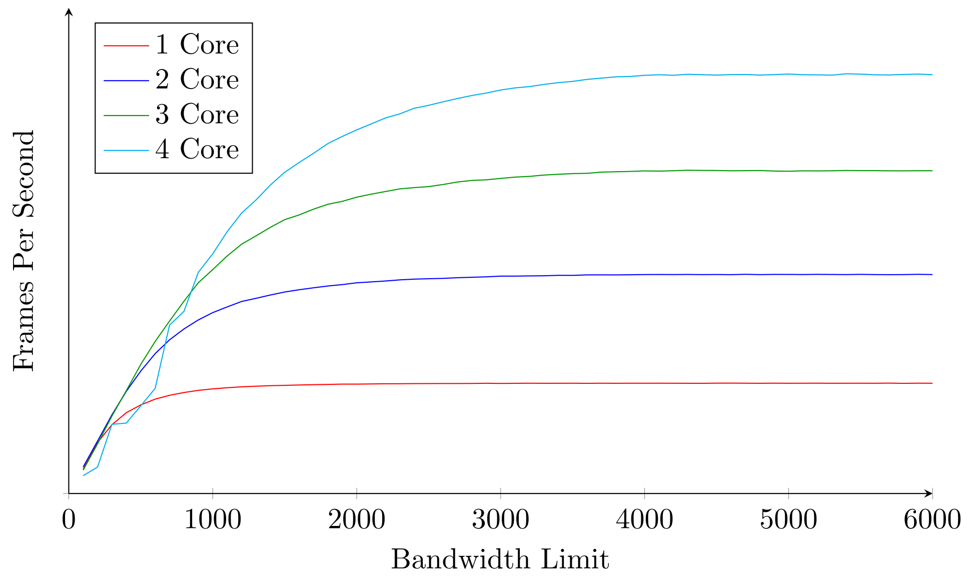


Figure 6.4: The affect of limiting bandwidth on Taiji as the number of cores is varied.

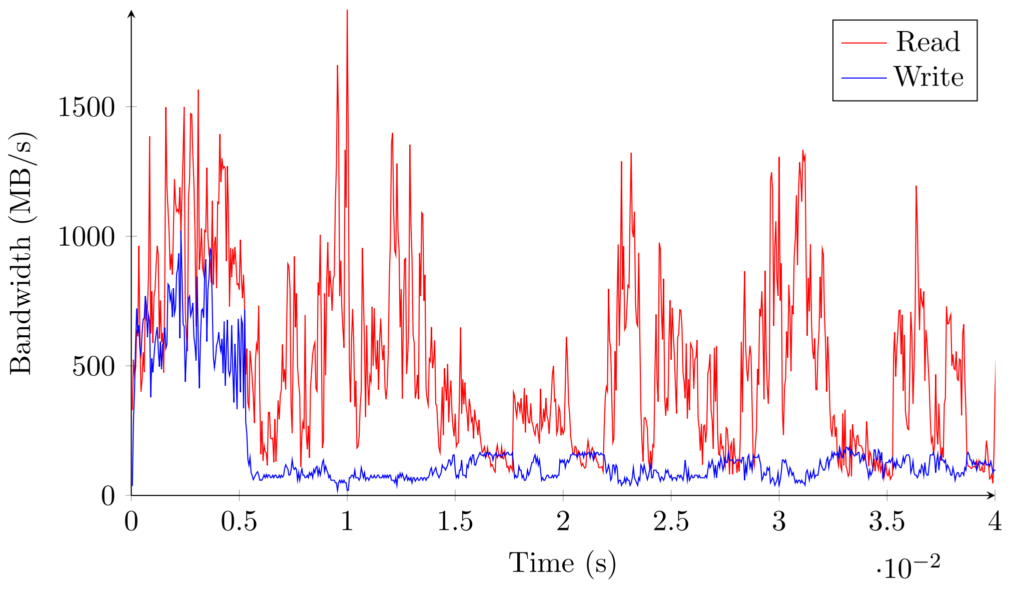


Figure 6.5: GPU bandwidth requirements over time for Taiji rendered on 1 core.

4 GB/s by itself when rendering some frames. Note that this concerns the average memory bandwidth over one frame, rather than more fine-grained bursty behaviour, during which the GPU requests yet more bandwidth from the memory system.

The GPU can handle a number of concurrent threads, which cause it to request significant bandwidth from the memory. Due to the high resource requirements the GPU issues transactions in quick succession, resulting in the ITT shown in Figure 6.6. This also demonstrates that the GPU has few dependencies between the transactions, which is why it is latency tolerant.

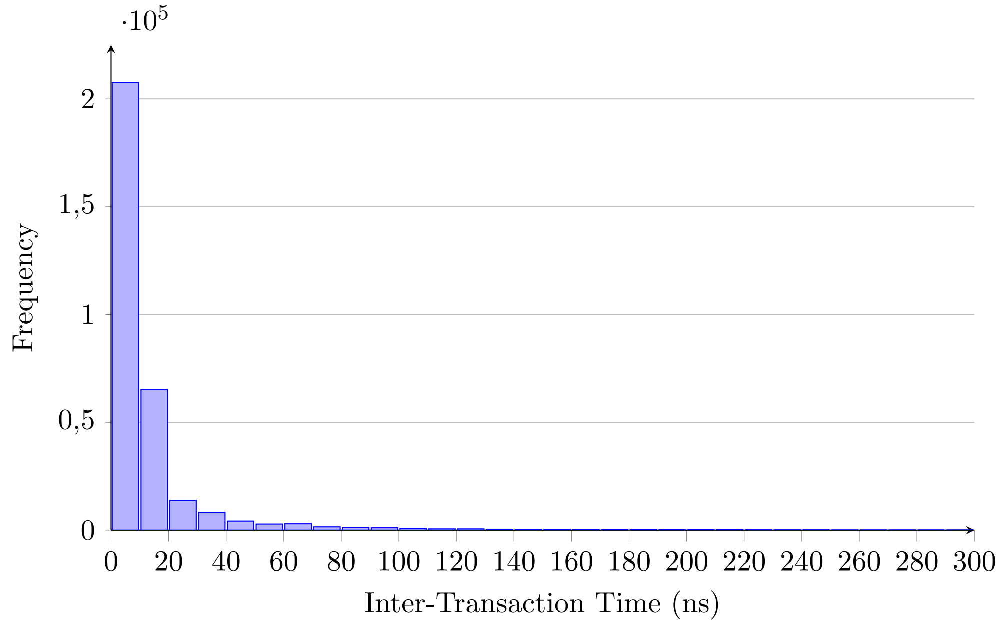


Figure 6.6: ITT for Pro frame with bins of 10ns

6.1.4 Choosing the number of active GPU cores

In this section, we observe the performance and energy characteristics of a GPU by adjusting the number of available GPU cores. The simulated GPU is dual-ported, and hence the number of GPU cores is varied from 2 through 16 in increments of 2, thereby ensuring that the load on each port is equal. Please note that there is no other traffic present in the system so these results apply specifically to the GPU in isolation. The GPU should render the frame before the deadline imposed by the screen refresh rate is reached. If the GPU fails to render the frame in time, then the frame will be skipped, thereby resulting in a decreased overall frame rate and a decreased user experience. On the other hand, if the GPU completes frame rendering significantly before the deadline, then it can be power-gated, saving energy (Wang et al., 2011; Allen et al., 2009). Therefore, in order to determine the ideal GPU configuration it is important to understand the relationship between the number of cores and the time taken to render different GPU frames.

Figure 6.7 shows, for varying core counts, the time for which the GPU can be powered off as it has completed rendering before the deadline. Please note that the frame times have been obfuscated. The results shown assume a target frame rate of 60 FPS, which is shown as a horizontal dashed line in order to demonstrate the maximum amount of time the GPU can be powered off. Values above the x-axis indicate that GPU can be powered off whilst values below the x-axis show GPU configurations where performance is insufficient to achieve 60 FPS. It can be seen that Navi requires at least 8 cores to render at 60 FPS, and that Taiji requires 6 as, otherwise, the frame takes longer to render than the allowed time per frame.

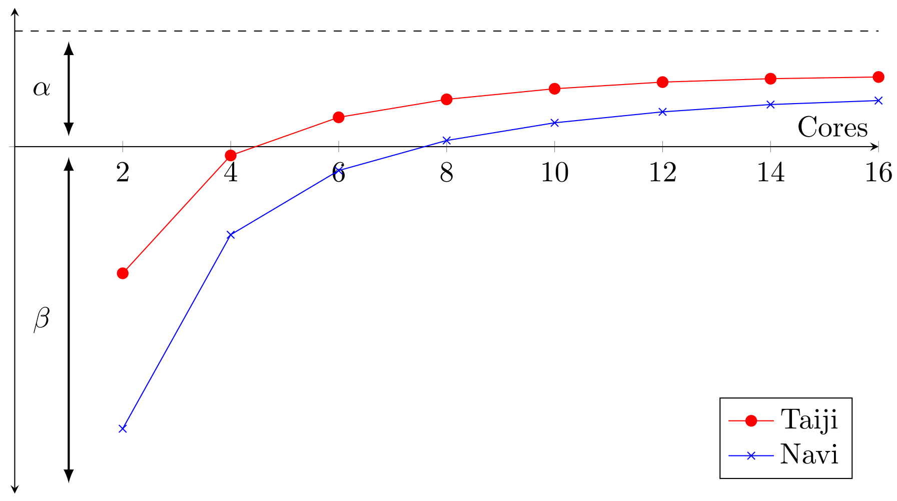


Figure 6.7: Amount of time where GPU can be powered off for Navi and Taiji.

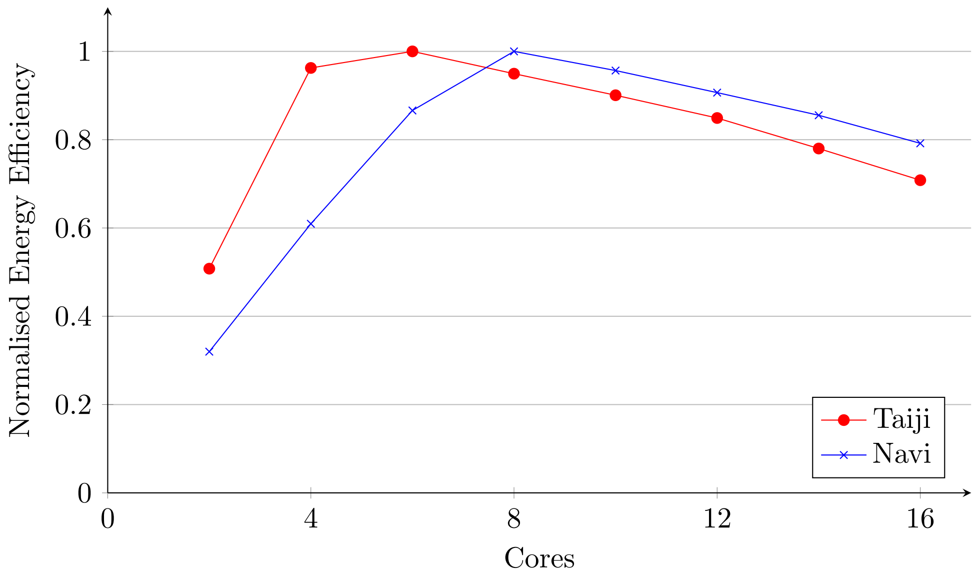


Figure 6.8: Performance per Watt for Navi and Taiji as the number of cores is varied.

The run time has been used to calculate the energy consumption for the GPU for each of the core configurations when rendering each of the two frames. These results, which again assume a screen refresh rate of 60 Hz, are summarised in Figure 6.8, and are presented in terms of GPU efficiency, i.e. the performance in frames per second per Watt. These results demonstrate that it is important to minimise the number of active GPU cores in order to keep energy usage low.

As the GPU generally requires a large amount of data to render a frame and is able to process vast quantities in parallel, it has a large impact on the dynamic power consumption of the DRAM. However, once the GPU has reached its maximum allowed frame rate, such as 60 Hz, it will request roughly the same amount of data (this can be seen in

Figure 6.14). Therefore, whilst the dynamic DRAM power consumption increases, the total energy consumed in the DRAM remains roughly constant per frame rendered.

Next, we use this insight to investigate how effectively the bandwidth of the GPU can be traded for the performance of the CPU.

6.2 Performance Trade-offs: Trading GPU for CPU

When a GPU is rendering a frame it requests a significant amount of data from the main memory of the system; the extra traffic increases the latencies to memory for all devices in the system. However, CPUs are latency-sensitive devices whose performance degrades with increasing latency, and therefore it is possible for CPU performance to suffer under heavy graphics loads. This, in turn, can lower the performance of the GPU as the CPU is unable to calculate the next frame to send to the GPU. To combat this, the CPU is usually given a priority which is higher than the GPU in the memory system. The result is that the CPU traffic is serviced before the traffic originating from the GPU. However, this can result in the GPU not receiving its bandwidth requirements, and failing to render the frame in time. The CPU traffic is less regular than that of the GPU, and therefore prioritising it over the GPU traffic can result in lower memory efficiency, and thus lower overall bandwidth to the memory.

To ensure that the GPU achieves a sufficiently high frame rate, the GPU must receive a frame specific quantity of data at a sufficiently high rate, i.e. a minimum average bandwidth. If the target time to render the frame is short, the average bandwidth increases until the GPU is permanently compute bound. The amount of bandwidth required is dependent on the frame being rendered; more complex frames usually require more bandwidth. As each frame rendered has different bandwidth requirements, the optimal operating point varies for each frame. Some frames require large amounts of bandwidth to render, and are mostly bandwidth bound (for example Pro). Others spend most of their time in a compute limited state, which is therefore limited by the rate at which the GPU can process them (for example Navi).

The bandwidth required to render a frame is not known beforehand. At best one can assume that the bandwidth required to calculate the next frame will be similar to the previous frame rendered. This assumption holds provided that the scene rendered does not significantly change. Some studies show that it is possible to determine the progress through a frame by looking at the percentage of tiles that have been completed by the GPU (Jeong et al. (2012)). While this does not give an indication of bandwidth, it does give an indication of progress which can be used to derive the QoS priority for the GPU. This could potentially be combined with the assumption that the frame bandwidth does not change significantly to determine the approximate bandwidth required such that the GPU renders the frame on time.

All	1.0 GB/s
	2.0 GB/s
	3.0 GB/s
	4.0 GB/s
	Unlimited
Egypt 90%	3.2 GB/s
Egypt 95%	3.6 GB/s
Navi 90%	2.0 GB/s
Navi 95%	2.6 GB/s
Pro 90%	4.5 GB/s
Pro 95%	5.2 GB/s
Taiji 90%	2.2 GB/s
Taiji 95%	2.8 GB/s

Table 6.1: Bandwidth limits for each frame

In order to evaluate the relative performance of the CPU and GPU in different configurations we run a benchmark on the CPU at the same time as the GPU is rendering a frame. As previously mentioned, the CPU is running Android, which is in turn executing BBench. While running BBench on the CPU at the same time as running a benchmark on a GPU is not a typical use case, each by itself is a representative workload.

The GPU model is complex as it simulates the full architecture of a GPU, including the ALUs and threads. Therefore, the simulator performance drops significantly when the GPU model is used, reducing the amount of time the simulation can feasibly simulate. As a consequence, we limit each BBench run to a period of 50 ms, and set the GPU to render a particular frame as rapidly as possible, i.e. there is no delay between successive frames. Each frame is rendered in two different configurations: with and without a limit on the maximum bandwidth it can consume.

In order to provide a reference for these results, BBench was run for the same period of 50 ms without the GPU producing any traffic. This gives baseline statistics for the CPU latencies, as well as the IPC for the CPU when having no contention in the memory system. The lower the IPC, the worse the performance of the CPU, thus providing an easily comparable measure of performance and user experience. When running in conjunction with the GPU, BBench was started from the same checkpoint in order to ensure that the results were comparable.

The limits which have been chosen are summarised in Table 6.1. All frames have been simulated with the limits shown in the ‘All’ category. Each frame has also been simulated using frame-specific limits which aim to limit the frame rate to 90% or 95% of the maximum frame rate achieved without bandwidth limitation. Each frame has been simulated with the GPU running 4 cores as this is the configuration in which the GPU generates the most bandwidth, and is representative of current mobile GPUs found in consumer SoCs.

Benchmark	IPC	Read Latency
Baseline (no GPU)	0.870	58.4 ns
Egypt	0.766	288.1 ns
Navi	0.816	170.2 ns
Pro	0.718	387.3 ns
Taiji	0.810	182.7 ns

Table 6.2: BBench and GPU - No bandwidth limitation

Benchmark	IPC	Read Latency Reduction
Egypt 90%	0.824	45.1%
Egypt 95%	0.810	32.0%
Navi 90%	0.857	39.3%
Navi 95%	0.846	35.1%
Pro 90%	0.758	16.3%
Pro 95%	0.753	7.8%
Taiji 90%	0.835	24.7%
Taiji 95%	0.836	26.9%

Table 6.3: BBench and GPU - Results with per-frame bandwidth limits

6.2.1 Measuring the effect of GPU traffic on memory latency

The results show that as the latency of the CPU increases, the IPC decreases (see Table 6.2). When the GPU is run without limiting the bandwidth it can consume significant resources, and therefore CPU latency increases are seen. The latency for the Pro and Egypt frames, which each require very large amounts of bandwidth, increases significantly (up to 6x the latency without the GPU), causing large penalties to the IPC. Pro shows the highest IPC decrease of the frames tested as it requires large amounts of both read and write bandwidth, and shows a decrease increase of 21%. Egypt causes the second highest IPC decrease at 13%.

When the GPU bandwidth is limited such that it is only able to use 90% or 95% of the bandwidth consumed when the GPU renders at its maximum frame rate, then the latency of the CPU decreases, along with corresponding IPC increases. The results are summarised in Table 6.3, and are shown graphically in Figures 6.9 and 6.10.

With very low limits for the GPU such as the 1 GB/s limit, there is minimal impact on the CPU performance and latency. However, this has an large impact on the performance of the GPU as all frames use over 1 GB/s when rendering with 4 cores without a limit on the GPU bandwidth. Therefore, this causes a significant frame rate decrease. Navi has the least impact on the CPU performance overall as it is a fairly low-bandwidth frame that is mostly compute limited.

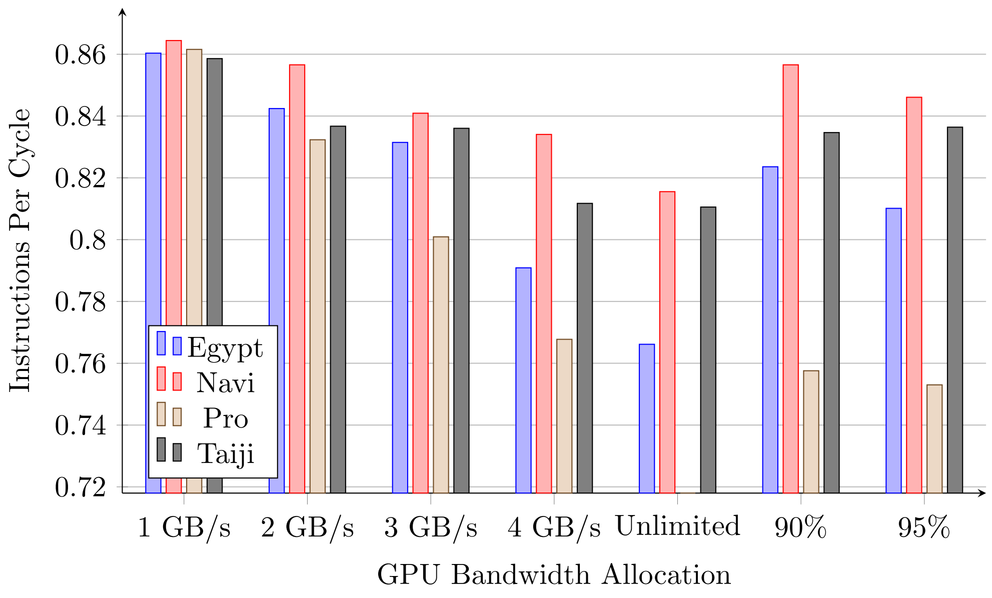


Figure 6.9: CPU IPC for chosen GPU bandwidth limits

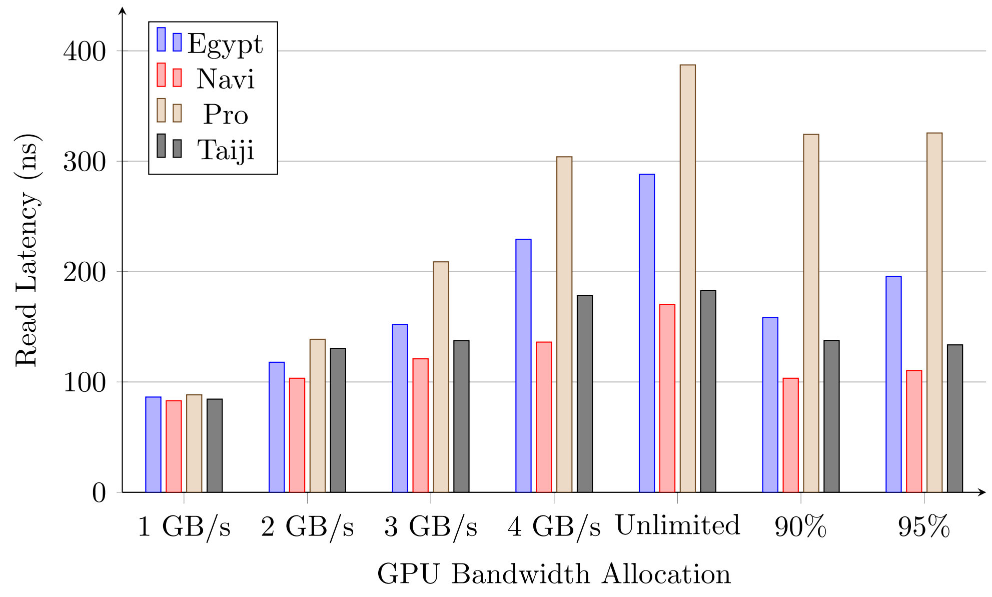


Figure 6.10: CPU read latency for chosen GPU bandwidth limits

When the bandwidth provided to the GPU for Egypt, Navi and Pro is increased, the CPU latency is seen to increase, and performance decreases. However, the Taiji benchmark does not follow this trend. As the bandwidth to the GPU is increased, the CPU's performance is seen to increase. However, this can be attributed to the temporal nature of the CPU and GPU as they can both exhibit bursty behaviour. If the GPU and CPU request a large amount of bandwidth at the same time they will both lower each others performance. Therefore, the increase in CPU performance indicates that the GPU is rendering the frame more rapidly, thereby moving the bandwidth intensive phases to be out of phase with those of the CPU. This demonstrates the importance of simulating with real workloads.

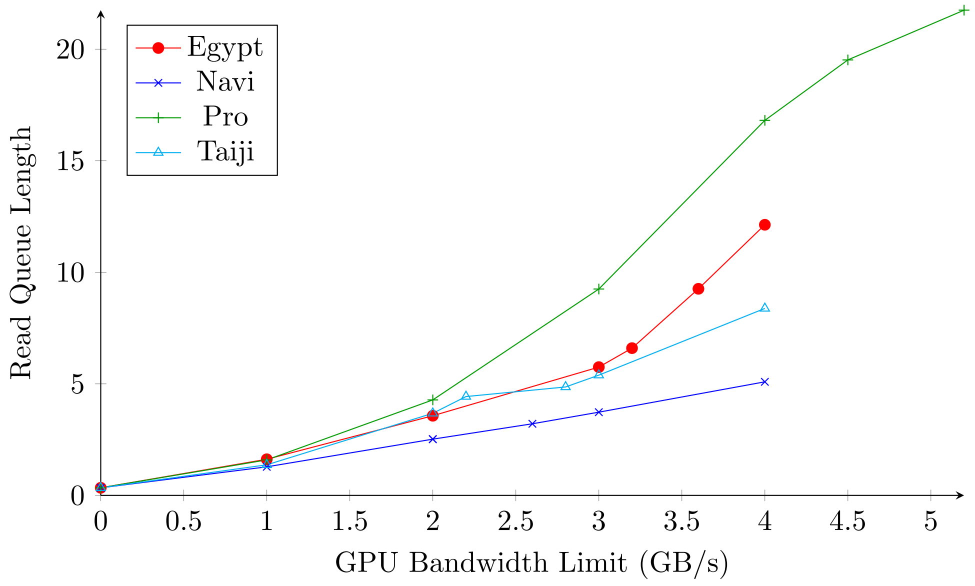
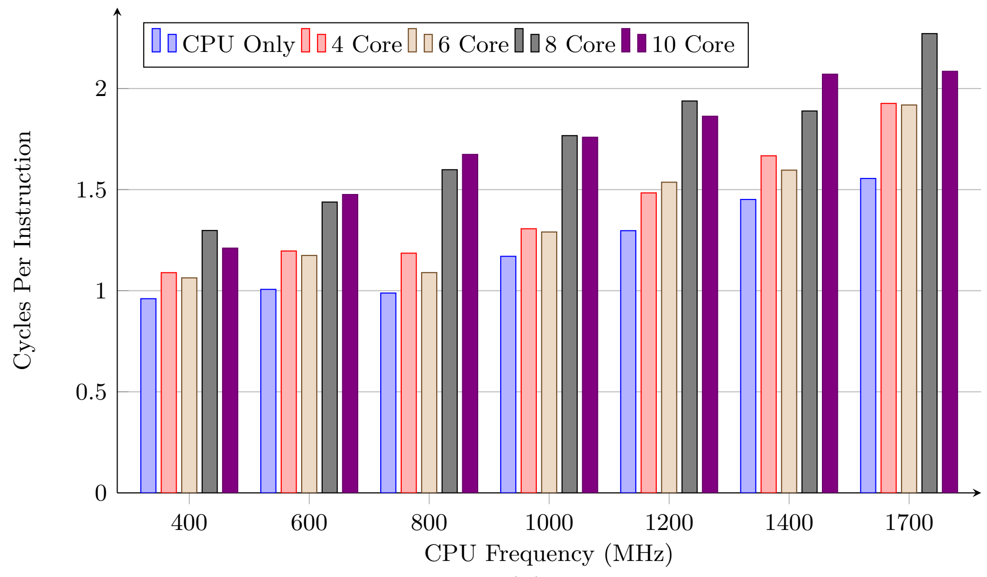


Figure 6.11: Read queue length as available GPU bandwidth changes

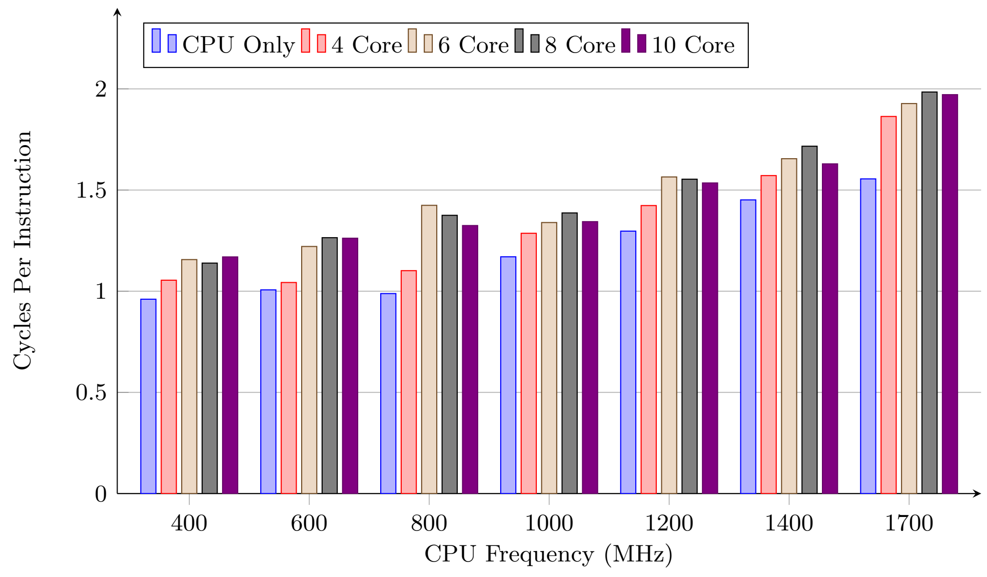
If we observe the DRAM statistics for the GPU frames, and compare the in-row hit rate of the run without the GPU to any of the runs with GPU, it is immediately clear that the addition of GPU traffic has a large impact on the DRAM performance as it drops down to below 40%. The GPU uses a separate address range to the CPU and therefore the accesses from one are in a different row to the accesses from the other. However, when the amount of GPU traffic increases, the hit rate increases slightly again due to the GPU accesses dominating the traffic to the memory controller.

When observing the DRAM controller's average read queue utilisation, we can see that it increases rapidly as the volume of GPU traffic increases (see Figure 6.11). Without the GPU, the average read queue length is less than a single request. However, when the GPU is rendering Pro, the average queue length increases to beyond 20 requests when the GPU is unlimited. This is the main contributor to the latencies seen in Table 6.2.

Our results clearly show that the introduction of a GPU has a significant impact on the latency seen by the CPU, and hence reduces the performance and user experience. Through the addition of a bandwidth limiter, we are able to reduce the volume of traffic injected by the GPU, and thus load on the memory system. This, in turn, reduces the contention seen by the CPU, resulting in a lower latency, higher IPC and thus improved performance and experience, provided that satisfactory GPU performance is maintained.



(A) Navi



(B) Taiji

Figure 6.12: CPI for BBench with varying GPU core count.

6.3 Picking the correct combination of CPU and GPU configurations

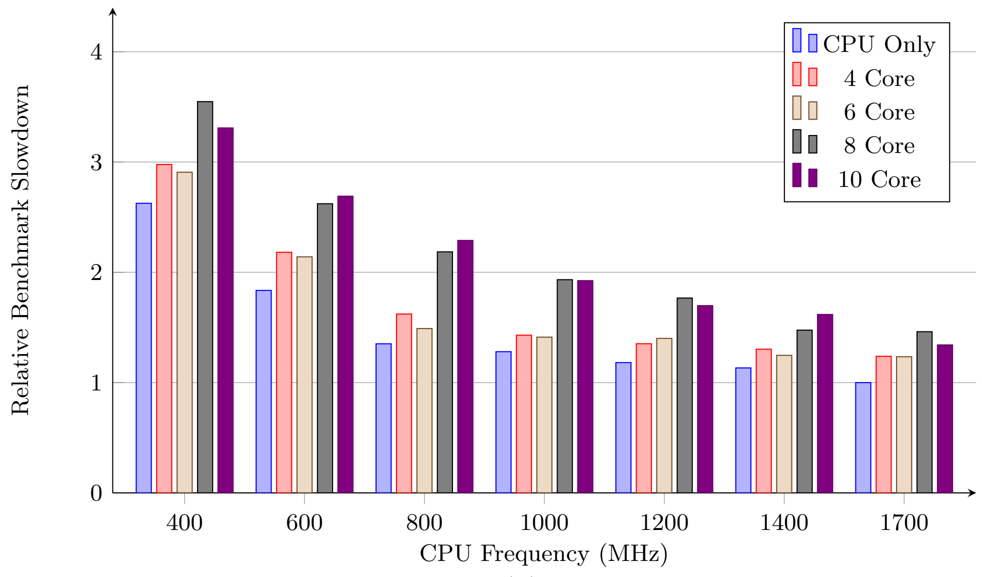
In this section, we investigate how the additional GPU traffic affects the performance of the CPU. We then use this to determine what the ideal CPU frequency and GPU core count should be in order to minimise the overall system power consumption, whilst maintaining a sufficiently high QoE. Without the GPU rendering in the system, the most energy-efficient operating point for the CPU is 800 MHz. The GPU frames Navi and Taiji required 8 and 6 cores, respectively, in order to render at 60 Hz or more. In this section, we change the frequency of the CPU in 200 MHz steps from 400 MHz to 1400 MHz, as well as running at the maximum frequency of 1700 MHz. Simultaneously, we run the GPU in 6 and 8 core configurations for Taiji or the 8 and 10 core configurations for Navi. This allows us to observe which configuration is best for the complete system. Please note that the DRAM controller uses FR-FCFS scheduling, and no attempt is made to prioritise the CPU traffic over the GPU traffic. All trade-offs are investigated statically and no adjustments occur at run-time.

Figure 6.12 shows the change in CPI for the CPU as the load placed on the shared memory changes with the GPU core count. The results clearly demonstrate that the addition of the GPU has significantly increased the latency to main memory, thus reducing the performance of the CPU. It can also be seen that Navi requests more bandwidth from the memory than Taiji, as it has a larger impact on the CPI of the CPU. If we compare the CPI measured here to the CPI shown in Figure 4.5, we see that we should expect roughly 4.2 seconds on average for a page to load and scroll for 8 core Navi, and 3.7 seconds for 6 core Taiji. For reference, using the CPI of the CPU-only run at 800 MHz - which is 0.99 - we estimate the mean time for a web page to be approximately 3.1 seconds. When running at the maximum clock frequency of 1.7 GHz without a GPU the average web page time is 2.4 seconds, and therefore reducing the CPU frequency to 800 MHz leads to a 30% increase in the average render time.

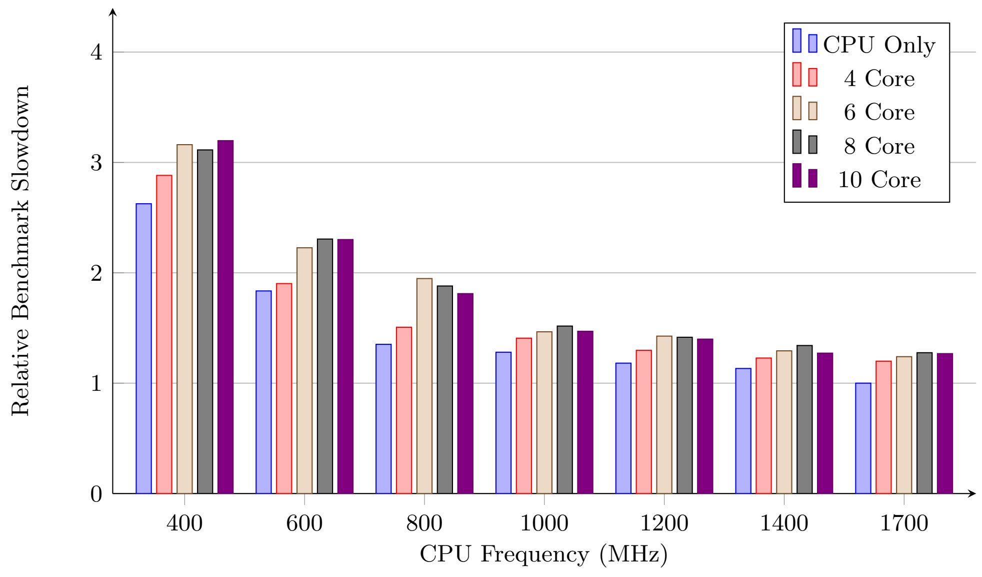
The increase in the CPI of the CPU manifests itself in an increased run time for the benchmark. The overall impact on user-perceived CPU performance is shown in Figure 6.13. It is worth noting that the impact of additional GPU cores has little influence on the run time for the CPU once the maximum frame rate has been reached. Therefore, there are similar run times for 8 and 10 core configurations for Navi, and 6 and 8 core configurations for Taiji.

The addition of a GPU has a significant impact on the overall memory bandwidth consumed, and hence increases the power consumption of the memory. This is illustrated in Figure 6.14. The memory power consumption is similar to the change seen in CPI, and this is to be expected as the contention between the CPU and the GPU is in the shared memory. However, the power consumption is averaged across the run time, and therefore the overall energy usage increases significantly as the run time of the benchmark rises due to resource contention.

Figure 6.15 shows the relationship between the slowdown of the benchmark (shown relative to 1.7 GHz) and the energy consumed in the CPU. The frames rendered on the GPU, the number of GPU cores, and the DVFS points of the CPU are changed. For reference, the CPU-only runs from Figure 4.4 are shown using a fine dotted line at the bottom of the diagram. First of all, it is apparent that for any scenario which includes a GPU, the CPU energy usage increases as the overall run time for the benchmark increases due to the intensified resource contention. The increased load has a secondary effect on the CPU performance as the CPU spends a larger proportion of time waiting for



(A) Navi

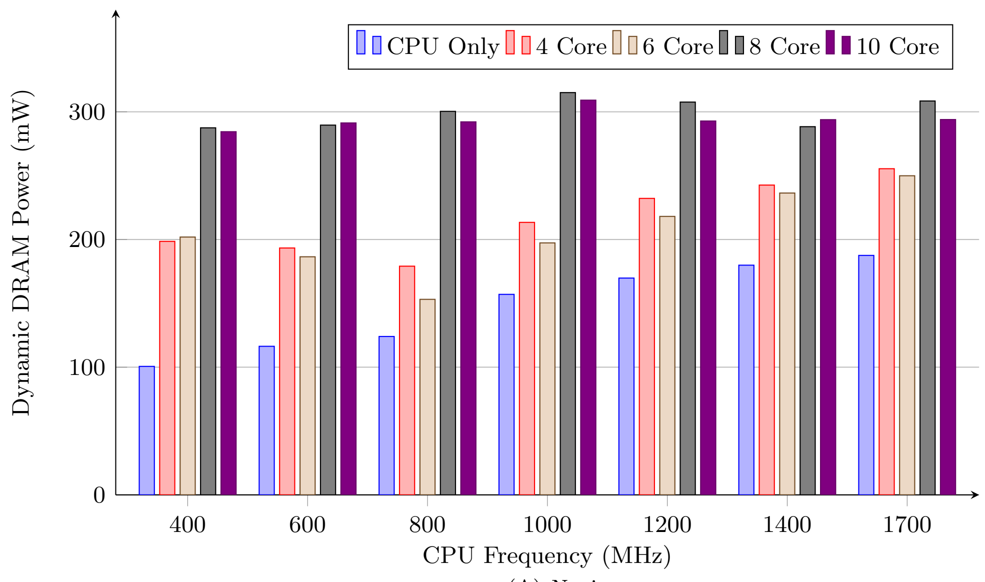


(B) Taiji

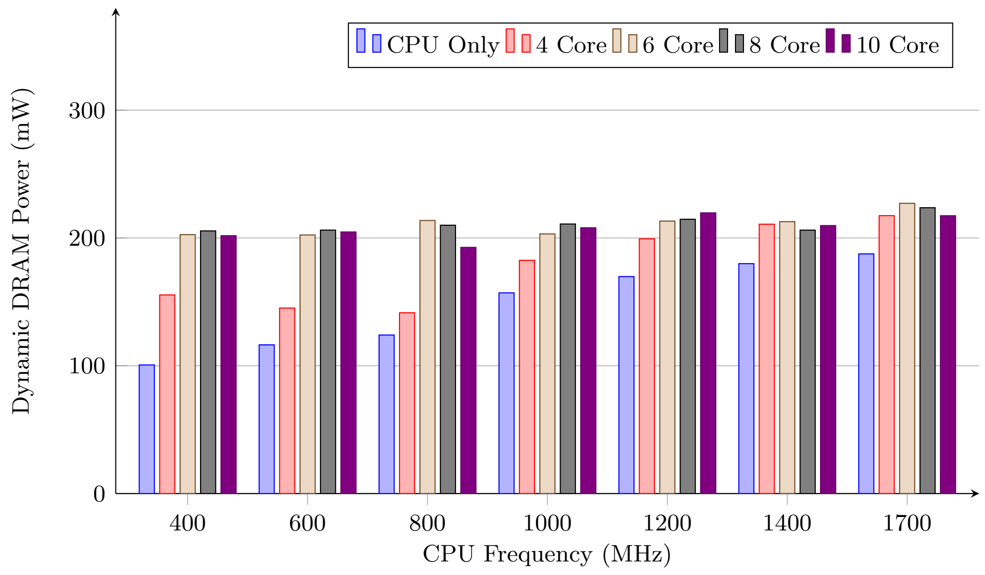
Figure 6.13: Run time for BBench with varying GPU core count.

data from memory. Therefore, the higher the GPU-induced load on the shared memory, the smoother the trade-off between run time and energy usage. There is also a tight grouping between the 8 and 10 core GPU configurations for Navi, and the 6 and 8 core configurations for Taiji, which is to be expected given the previous figures.

The addition of the GPU shifts the ideal point for certain GPU core configurations and frames. Specifically, for Taiji in the 6 and 8 core configurations the most energy-efficient CPU frequency (determined by minimising the energy-delay product) shifts from 800 MHz to 1 GHz, as does the 10 core configuration for Navi. The 8 core configuration for Navi remains most efficient at 800 MHz. Therefore, as the load induced by the GPU increases, the ideal operating point for the CPU increases. This motivates the need



(A) Navi



(B) Taiji

Figure 6.14: Dynamic DRAM power for BBench with varying GPU core count.

for dynamic run time adjustment of system configuration based on both QoS and QoE measurements (see Figure 3.1). Run-time adjustment based on QoE requires measurable metrics, such as the CPI of the CPU and the frame rate of the GPU. As previously stated, the metrics of interest change as the applications running on the device change, and therefore run time adjustment also is needed to keep track of the ever-changing requirements and demands placed on the system. Finally, a set of knobs which can be adjusted in order to tweak the system performance is required.

We compare standard Linux CPU governors to our results. The *Performance* (The Linux Kernel Archives, 2014) governor will choose a CPU frequency near the top left of Figure 6.15, whilst the *Powersave* governor (Pallipadi and Starikovskiy, 2006) will

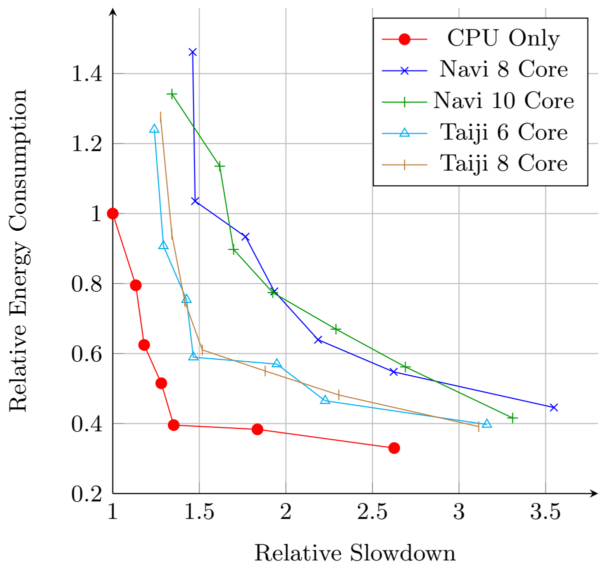


Figure 6.15: Relative slowdown vs. relative energy consumed by the CPU as the load placed on the shared memory varies due to the GPU traffic. The points on the lines show the different DVFS points for the CPU.

choose the bottom right. The *OnDemand* governor will try to use the lowest DVFS point possible given the load on the CPU but does not take into account the user experience when switching the frequency. QoE provides a user-centric measure of performance which would allow a governor to choose an experience-optimised DVFS point.

6.4 GPU Governing

Following the evaluation of the CPU governor in Chapter 5, we evaluate the performance of a GPU governor. We present our GPU governing algorithm in Figure 6.17. The algorithm considers the QoE delivered by the GPU by utilising the QoE models presented in Section 3.4.2. Every governing interval the governor calculates the frames per second for the last governing interval, and uses this value to compute the delivered QoE. This calculated QoE value is them compared to the target QoE. If the QoE is lower than the target, then the number of GPU cores is increased and if the QoE is above target then the number of active GPU cores is decreased. Should the QoE be close to the target QoE, then no changes are made. When increasing or decreasing the number of active cores, the governor predicts number of cores to reach the target QoE. Using these predictions allows faster switching, as well as helping to ensure that the target QoE is met.

To evaluate the GPU governor we run the GPU in isolation, i.e., with only a GPU and associated memory system, and observe the QoE delivered by the GPU for a set of GPU workloads. We present a system diagram in Figure 6.16. For each workload and core combination we calculate the QoE using the equation presented in Section 3.4.2 based

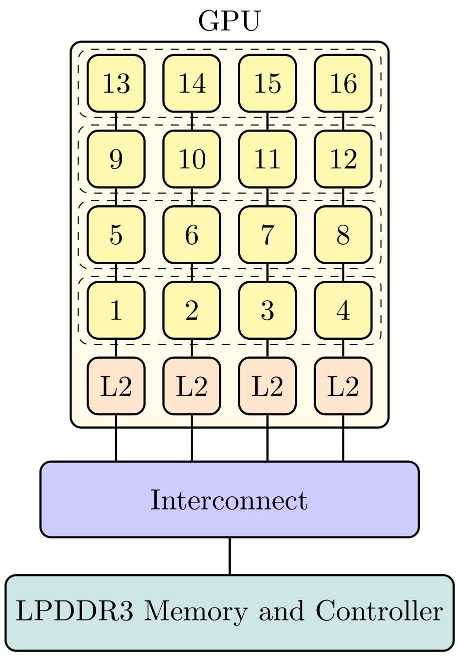


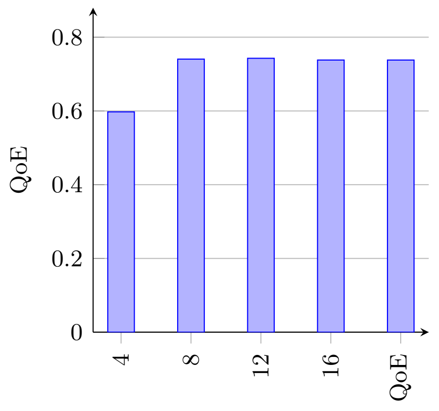
Figure 6.16: System diagram showing the GPU configuration, as well as the memory system.

```
if GPU QoE higher than target then
    if Predicted QoE sufficient then
        Decrease number of active cores by $N$
    end if
else
    if GPU QoE lower than target then
        Increase number of active cores by $N$
    else
        Do nothing
    end if
end if
```

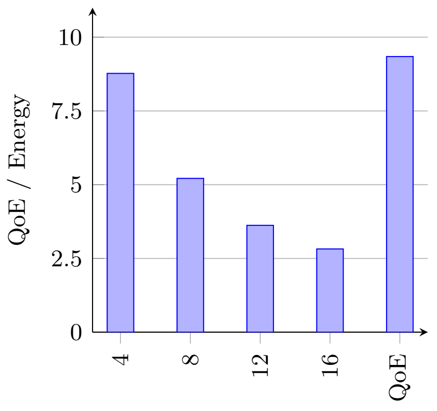
Figure 6.17: Algorithm for QoE-Aware GPU governing

on the frame time and assuming a target render time of 16.7 ms. In addition to the frame time and QoE we calculate the energy consumed in the GPU as described in Section 5.2.2.

The detailed execution-driven GPU model is slow, and therefore we utilise a trace-driven approach to validate the GPU governing algorithm. We capture memory traces for the set of GPU workloads detailed in Section 6.1.1, for varying GPU core counts. Specifically, we consider the number of GPU cores in increments of 4, ranging from a minimum of four cores up to 16. We use the captured traces to create a GPU trace player which responds to back-pressure from the memory system, and therefore will slow the execution of the traces in response to memory system contention. The operation of the GPU trace player has been verified against the GPU model itself, and the memory traffic generated has been validated.



(A) QoE



(B) QoE / Energy

Figure 6.18: Average GPU QoE and QoE-Energy ratio for varying GPU core counts and the QoE-aware GPU governor

The average QoE across all GPU benchmarks is shown in Figure 6.18. When observing the QoE, it is worth noting that not all frames can achieve the highest QoE with the GPU configurations we test. This is due to the fact that the frames have been taken from popular mobile graphics benchmark suites (Rightware, 2012; Futuremark, 2012; Kishonti, 2015) and therefore are designed to place very high demand on the GPU. For this reason not even the highest-end mobile GPUs are capable of delivering 60 FPS for the mode demanding benchmarks, resulting in a decreased QoE on average.

The QoE delivered by the QoE-aware GPU governor is able to match the QoE delivered when running with 16 cores. This is because the GPU governor tries to maximise the QoE for the GPU, and will therefore try and operate at or above the GPU FPS target, 60 FPS. However, it is worth noting that the QoE-aware governor is able to match the performance whilst consuming less energy. This is clearly visible when the QoE per energy consumed is presented. The QoE/Energy results are calculated by dividing the QoE by the energy on a frame by frame basis, then taking the mean. For the benchmarks tested, the QoE-aware GPU governor provides the same QoE as running with 16 cores constantly enabled, whilst reducing the energy consumption by 12% on average.

6.5 CPU and GPU Governing

We follow the CPU- and GPU-only evaluation with a set of experiments making use of both the CPU and the GPU. These experiments serve to demonstrate that the two parts of our QoE-aware governor can work in tandem to combine CPU and GPU governing. It is especially important that the CPU governor is able to cope with the additional strain placed on the system by the high-bandwidth GPU. Figure 6.19 shows the topology of the system we use to evaluate the combined governing strategies.

A GPU is a high bandwidth device, and therefore can introduce a large amount of contention in the memory system, as was shown in Section 6.1. This, in turn, increases the L2 miss latency, and therefore increases the average latency from the CPU to the memory system. The result is that the IPC decreases as the CPU stalls more frequently, and operates less efficiently. However, as this is visible in the IPC of the CPU, the governor is able to detect the change and is able to adjust the CPU frequency accordingly. We initially present results for a single-core system, and follow these with the corresponding dual-core results.

A number of different GPU benchmarks are used as each frame places different demands on the memory system. We present the results with our QoE-Aware governors in the system, and the results are normalised to the *Performance* governor energy consumption. We compare our QoE-aware governor to the *Performance* governor and to the *OnDemand* governor. The *Performance* governor shows the worst-case scenario for the CPU. In addition, we do not adjust the number of GPU cores and therefore always run with 16 cores enabled making the *Performance* governor the worst case scenario. However, for the *OnDemand* governor, we run the GPU with our GPU-scaling algorithm which adjusts the number of active cores based on performance. This ensures that the CPU workload performance is run under the same conditions as for our QoE-aware governor, and thus is comparable.

6.5.1 Single Core CPU

Figure 6.20 (A) shows the performance of BBench under different governors and Figure 6.20 (B) shows the average power consumed in the CPU, GPU and DRAM. The GPU benchmarks used are listed in Table 6.4. As expected, the use of the QoE-aware governor results in a QoE decrease relative to the *Performance* governor. However, the mean increase of web page render time is just 9%, which is significantly less than the decrease in energy consumption of 60%. On average, the QoE of the *OnDemand* governor is reduced by 18% compared to the *Performance* governor. The QoE-aware governor decreases by 6% and the *OnDemand* governor decreases by 9% relative to the *Performance* governor.

When we introduce our QoE-aware CPU and GPU governors, we reduce the power consumed in the CPU significantly — in some cases by as much as 70%. On average we reduce the CPU energy consumption by 60% and the power consumption by 64% compared to the *Performance* governor. The mean GPU power consumption is reduced by 3% relative to the *Performance* governor for our QoE-aware governor, whilst the GPU power consumption increases by almost 5% for the *OnDemand* governor. This increase is due to the fact that workload takes longer to run, and therefore more frames get executed. Finally, the DRAM power consumption is decreased by almost 17% for the *OnDemand* governor whilst it is decreased by 6% for the QoE-aware governor. This

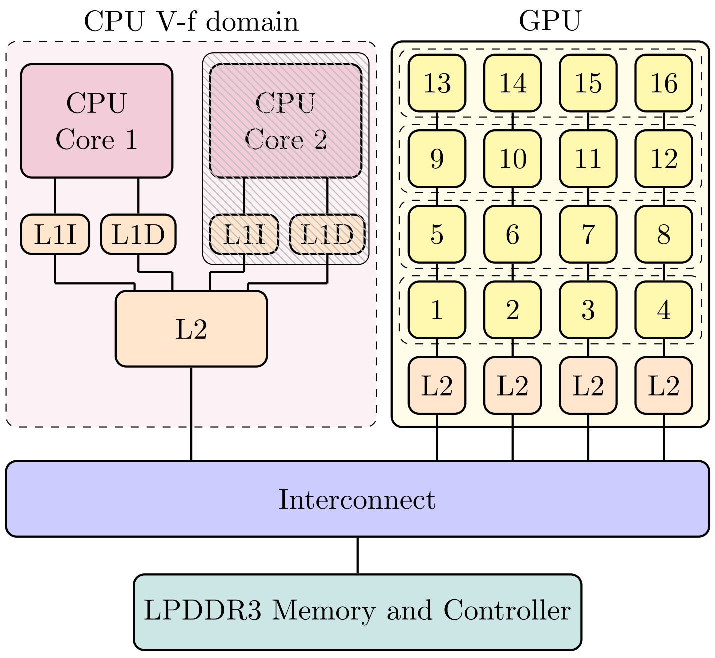


Figure 6.19: System diagram showing the CPU and GPU configuration, as well as the memory system.

GPU Benchmark Index	Benchmark Suite
1-2	Low-level compute benchmark, no suite
3-7	GFXBench (Kishonti, 2015)
8-12	3DMarkMobile (Futuremark, 2012)
12	Basemark ES (Rightware, 2012)

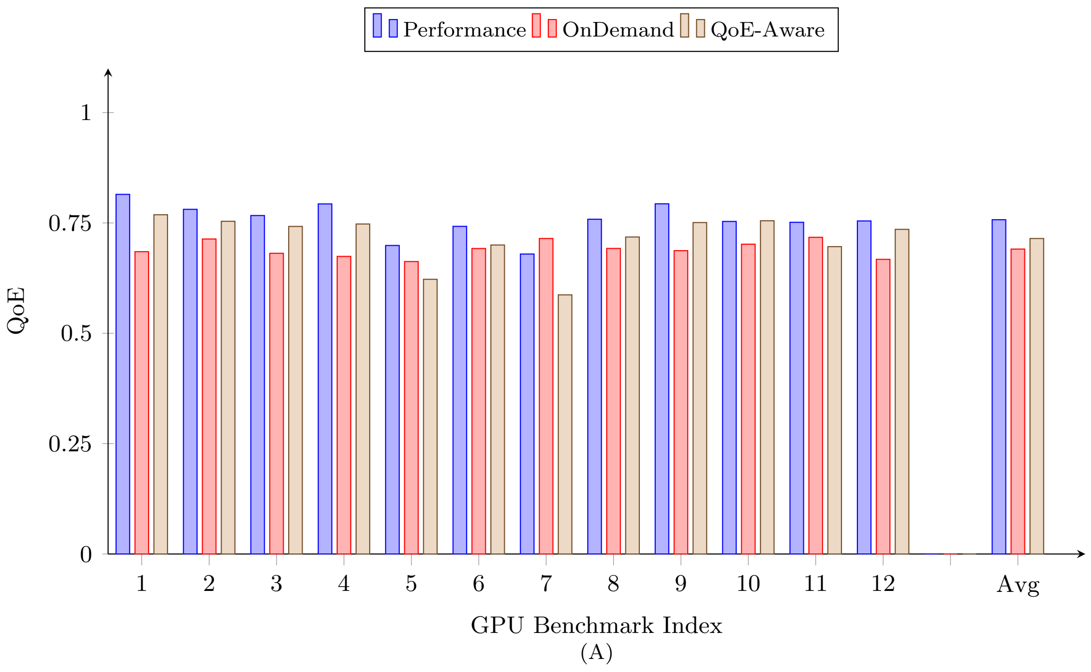
Table 6.4: GPU Benchmark suites used

is due to the fact that the QoE aware governor performs better than the *OnDemand* governor and therefore will request the same amount of data from the DRAM over a shorter period of time, resulting in increased power usage.

6.5.2 Dual Core CPU

In this section we present the results for a dual-core system. As was the case in Section 6.5.1, we present the results for BBench running alongside GPU traffic whilst running both the QoE-aware CPU and GPU governors. We compare the power consumption and performance of the governors to the *Performance* and *OnDemand* governors.

The QoE is shown in Figure 6.21 (A). It is worth highlighting that the QoE aware governor is able to outperform the *Performance* governor whilst certain GPU benchmarks are running. The cause of this is that the GPU traffic is less bursty when using the QoE-aware GPU governor, and therefore this is less interaction with the CPU. Therefore, the CPU is not stalled waiting for data from main memory and is able to offer higher, and more satisfying, performance. In general, the QoE-aware governor is able



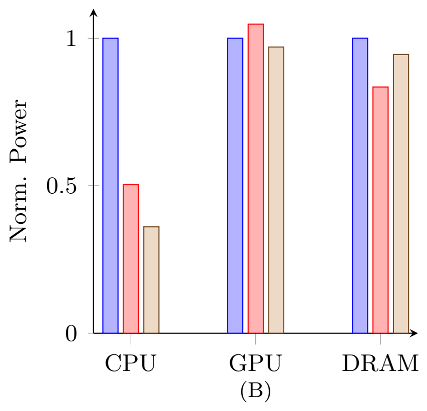
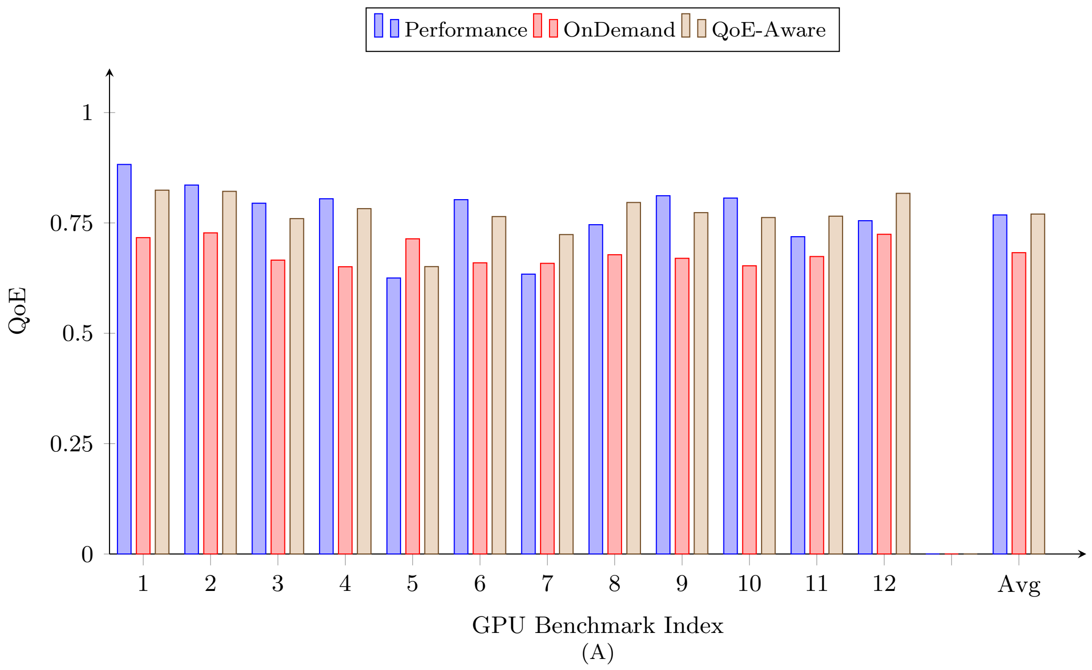


Figure 6.20: CPU, DRAM and GPU performance and power consumption for a single-core system

to outperform the *OnDemand* governor. The average QoE delivered by the *OnDemand* governor is 11% lower than when the *Performance* governor is used. As the QoE-aware governor is able to reduce the impact the GPU traffic has on the CPU performance, it is able to meet the level of QoE delivered by the *Performance* governor. Therefore, the QoE-aware governor offers higher performance than traditional governors, whilst also providing greater energy savings.

Figure 6.21 (B) shows the average power consumption for the CPU, GPU and DRAM for a dual-core system. As was the case with the single-core results, both the *OnDemand* and QoE-Aware governors are able to reduce the energy and power consumption of the CPU dramatically. In the dual-core case, both the *OnDemand* and QoE-aware governors perform similarly across the board for power reduction. The *OnDemand* governor reduces the CPU power consumption by 59% relative to the *Performance* governor, whilst our QoE-aware governor is able to reduce the power usage by 60%. In



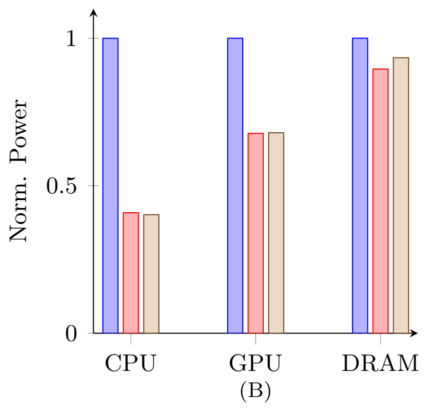


Figure 6.21: CPU, DRAM and GPU performance and power consumption for a dual-core system

the case of the GPU, the *OnDemand* and QoE-aware governors both reduce the power consumption by 33% relative to running with all 16 cores enabled continuously. However, it is worth noting that both the *OnDemand* results and the QoE-aware governor results use our GPU governor to control the number of active GPU cores. Therefore, these can only be directly compared to the case where all 16 GPU cores were enabled for the entire simulations, as is the case with the *Performance* governor results. Finally, the *OnDemand* governor is able to reduce the DRAM power consumption by 10%, whilst the QoE-aware governor reduces this by 6%.

6.6 System Governing

In this section we build upon the IPC-based governor presented in Chapter 5, and extend it so that it is able to act upon accesses to memory, as well as IPC. This gives the governor additional knowledge, and ensures that the frequency is not raised when it will not improve performance and that it is not lowered when the performance will suffer. As was the case for the IPC-based governor, we only consider statistics which can accessed in the hardware PMU counters as we wish to minimise the overhead of measuring these system characteristics. There are a large number of different PMU counters, and therefore it is difficult to determine which counters make the most sense to measure. In this section, we first choose an additional PMU counter to monitor from the DVFS governor based on regression analysis, and then demonstrate that we are able to set a target for each performance counter measured in order to adjust the performance of various workloads to meet the demand.

6.6.1 Selecting the PMU Counters

As we extend the IPC governor to consider an additional performance counter, we need to determine which performance counter to choose. We choose the performance counter that gives us the information we require to make the correct decisions by performing a linear regression analysis to relate the gem5 statistics from the forking infrastructure results presented in Section 5.1 to the decisions made by the *Oracle* governor. However, as gem5 provides a large number of statistics, and there are many PMU counters found in real devices, it is difficult to determine which PMU counter to monitor to make the desired decision. A large number of PMU counters can contain the same or similar information, which we need to filter out prior to running a least-squares regression analysis.

To filter out the statistics which contain statistically similar information, we calculate the Pearson correlation coefficient between each of the statistics. We then cluster the statistics based on their correlation coefficients, which allows us to determine which statistics contain the same information about the state of the system. This allows us to determine which sets of statistics can be reduced to a single statistic prior to running the regression analysis. We use the choices made by the 80% *Oracle* governor for the frequency decisions correlated in this analysis. We present a post-clustering Pearson correlation coefficient heatmap for The Chase in Figure 6.22. A high value, approaching 1 (shown in red) indicates a strong positive correlation between statistics, whilst a low value approaching -1 (shown in blue) shows a strong negative correlation. Values around 0 (shown in green), indicate that there is no overall correlation between two statistics.

First of all, it is worth highlighting that all statistics demonstrate a strong self-correlation, as is shown by the diagonal on the heatmap. Secondly, there are two main groups with

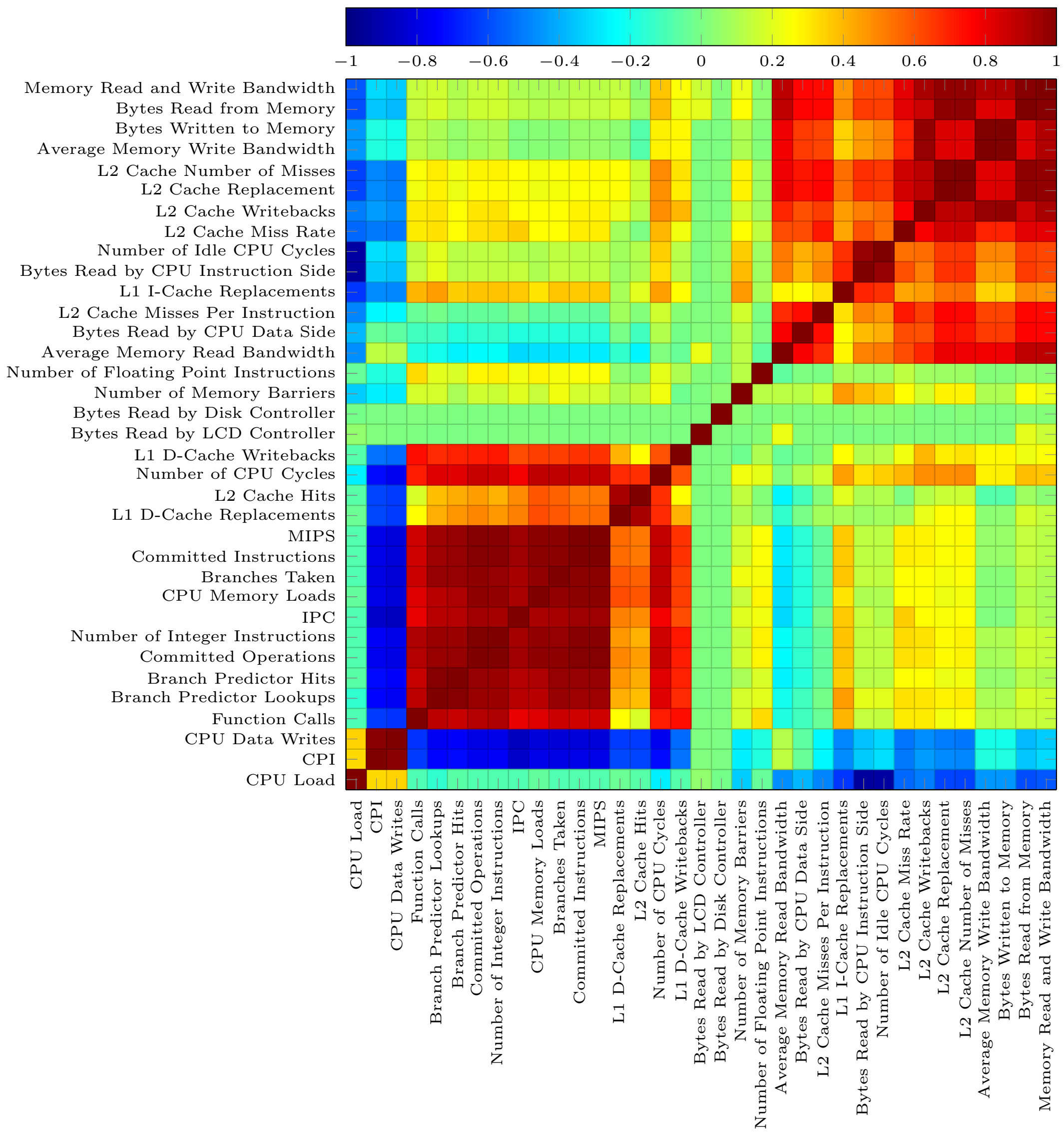


Figure 6.22: Heatmap showing the Pearson correlation coefficient for statistics captured for The Chase.

in the correlation. The first of these groups, shown in the bottom left, is related to CPU statistics, which the second, shown in the top right, is comprised mainly of memory system metrics. Additionally, there are some statistics which appear to have minimal correlation with other statistics, which are shown between the two main clusters. Finally, CPU Load demonstrates mild correlation with statistics such as CPU and an inverse correlation with the memory system statistics.

We begin by considering the CPU statistics. For these it is apparent that a lot of the statistics contain similar information when the operation of the system is considered. For example, MIPS is strongly correlated with the number of committed instructions and operations, as these are included in the calculation of the MIPS statistic. Similarly, CPI and IPC are strongly and inversely correlated as one metric is the inverse of the other. However, both are again correlated with MIPS as a higher IPC results in a higher instruction throughput for a given frequency. There are some metrics related closely to the CPU which have a less significant correlation. These statistics are those for the L1 and L2 caches. For example, L2 Cache Hits are strongly correlated with the L1 D-Cache Replacements as a hit in the L2 cache will result in the value being cached in the L1 cache, resulting in a L1 cache replacement. There is also a correlation between the cache statistics and the other CPU metrics. This is expected as a higher level of CPU performance in general requires more cache hits.

The memory system statistics are grouped together based on the accesses to main memory. For example, the overall read and write memory bandwidth is strongly correlated with the number of bytes written to and read from memory. There is a strong correlation between the bytes read from memory and the overall bandwidth, which indicates that the majority of the memory bandwidth is due to reads, and that a smaller percentage is due to writes. The number of bytes written to memory shows identical correlation to the write bandwidth, as would be expected, which indicates that one of these can certainly be removed. The average read bandwidth, the bytes read by the CPU data side and the MPKI for the L2 cache all show similar characteristics. This shows that there is a large number of misses caused by accesses emitted by the CPU data side which are then serviced by the main memory. This again is a grouping of statistics which can be reduced to a single statistic.

Finally, there are four statistics which show fairly weak correlation with the majority of other metrics. These include the bytes read by the disk and LCD controllers, which indicate that either these consume constant bandwidth, or have little activity. In the case of the LCD controller it will read data from main memory for every displayed frame. This is a periodic event which consumes a constant bandwidth on average. The periodicity causes the metric to have poor correlation with the other metrics considered. Once The Chase has loaded and is running, there are minimal accesses to the disk, and therefore there is poor correlation for disk accesses. The number of floating point instructions does not correlate well with the majority of statistics, but shows some weak correlation with the CPU statistics. It appears that The Chase does not contain a large number of floating point instructions on the CPU, and hence there is poor correlation. Finally, we consider the number of memory barriers, which are used to ensure memory access ordering. These show weak correlation with the CPU metrics, and a slightly stronger correlation with the memory system characteristics.

Now that the statistics have been clustered, and the clusters identified, it is possible to reduce the number of statistics prior to running a regression analysis. We filter down the statistics from gem5 to the following subset of statistics, prior to performing the regression analysis:

L2 Cache Hits	L1 D-Cache Writebacks
Bytes Read by LCD Controller	Bytes Read by Disk Controller
Number of Memory Barriers	Number of Floating Point Instructions
L2 Cache Misses Per Instruction	L1 I-Cache Replacements
CPU Idle Cycles	L2 Cache Miss Rate
L2 Cache Misses	Bytes Written to Memory
Bytes Read from Memory	CPU Load

Using the above statistics we run a linear least squares regression analysis to determine which of these statistics are the best predictors for the decisions made by the *Oracle* governor. We aim to determine which additional performance counter should be monitored in order to improve the operation of the QoE-aware governor. Therefore, we wish to use the performance counter in addition to the IPC already used as part of the governor. Hence, we run the linear regression analysis assuming that the IPC metric has already been selected. We try all combinations of metrics which fit the vector form:

$$y = \beta X + \epsilon$$

where y is the value we are trying to predict, β is a vector regression coefficient, X is vector of metrics and ϵ is an additional error term which is used as an offset. In our case, we wish to predict the frequency which is chosen by the governor using the metrics measured from the system. Hence, y represents the operating frequency of the CPU. We expand the previous equation, and insert the IPC metric to obtain:

$$y = \beta_0 IPC + \beta_1 x_1 + \epsilon$$

where β_0 and β_1 are the regression coefficients for the IPC and the metric being tested, respectively. We do not allow a frequency of 0 to be selected by the governor, and use 500MHz as the lowest frequency. This requires that we run the regression analysis using the ϵ error term which can be used to offset from zero.

We run the regression analysis and present the results in Table 6.5. The table shows the value of R^2 which is a measure of model fitness. Specifically, a value of 0 indicates that the model is not able to predict the value of y at all, whilst a value of 1 indicates that the model is able to perfectly predict y. Hence, the value of R^2 should be maximised. Please note that we run the regression analysis with the IPC metric and with a constant offset, but do not show these in the table. Instead, we only show the name of the secondary statistic. It is worth noting that there is a very small spread in terms of the R^2 value. When running with just IPC, we see an R^2 value of 0.7002, which indicates that these additional metrics don't add a lot more useful information. Nevertheless, it appears that L2 cache statistics are the most suitable predictors for the decisions made by the *Oracle* governor.

R^2	Secondary Statistic
0.7349	L2 Cache Hits
0.7102	L2 Cache Misses
0.7082	Bytes Read from Memory
0.7077	L2 Cache Misses Per Instruction
0.7055	L1 D-Cache Writebacks
0.7052	L2 Cache Miss Rate
0.7051	L1 I-Cache Replacements
0.7045	CPU Load
0.7043	Bytes Written to Memory
0.7041	Number of Memory Barriers

Table 6.5: Overall R^2 value for different secondary metrics resulting from linear least squares regression analysis

From the results of the regression analysis and by consulting the ARM Cortex A15 data sheet, we determine that the extra performance counter to measure is the number of level 2 data cache read refills. This performance counter is incremented whenever the L2 cache refills a cache line due to a read from main memory. Therefore, this statistic is correlated with the number of reads to main memory, but excludes data which is already present in the L2 cache. Hence, this statistic only measures reads from main memory itself, which ensures that the CPU frequency is not decreased due to reads from the low-latency L2 cache.

6.6.2 Advanced Governing Algorithm

In this section we take the IPC-based governor, and extend it to monitor the L2 read refill performance counter in addition to the IPC. The addition of this performance counter allows us to determine when the CPU is making a large number of memory accesses, and hence we can reduce the frequency of the CPU when it becomes more memory bound. This saves energy and should result in a minimal performance impact when the correct thresholds have been chosen. We do not act upon the value of the performance counter directly, and instead divide the value of the performance counter by the number of cycles. This ensures that if the governing interval is change, the same ratio can be used and hence the governor does not need to be reconfigured. We call this term Memory Reads Per Cycle, or MRPC.

Figure 6.23 shows the CPU frequency governing algorithm which has been implemented as part of the CPUFreq framework. The governor monitors three performance counters: the number of cycles, the number of committed instructions and the number of read refills for the L2 cache. These performance counters are used to calculate the number of Instructions Per Cycle, IPC, and the number of Memory Reads Per Cycle, MRPC. For each governing interval, the governor first reads the values of the performance counters,

```
if CPU idle then
    new_freq = min_freq;
else
    if IPC greater than IPC_Threshold + 0.125 then
        delta_freq = -100000;
    else if IPC less than IPC_Threshold - 0.25 then
        delta_freq = 300000;
    else if IPC less than IPC_Threshold - 0.125 then
        delta_freq = 200000;
    else
        delta_freq = 100000;
    end if
    delta_freq -= MRPC * MPRC_weighting;
    new_freq = cur_freq + delta_freq
    if new_freq greater max_freq then
        new_freq = max_freq;
    else if new_freq less min_freq then
        new_freq = min_freq;
    end if
end if
```

Figure 6.23: Algorithm for IPC- and MRPC-based QoE-Aware CPU governing. Frequency is measured in kHz.

and calculates the aforementioned metrics. Then, the governor checks if the CPU was idle or close to idle since the governor last ran, and in this case reduces the frequency to the minimum level. The governor then compares the IPC during the last interval to the interval target, and uses this to set a frequency delta. If the measured IPC is significantly higher than the target, then the frequency delta is set to negative 0.1 GHz to reduce the frequency. If the measured IPC is lower than the target, then the delta is set to 0.3 GHz, 0.2 GHz or 0.1 GHz based on the difference. Following the IPC measurement, the governor measures MRPC. In this case, the frequency delta is reduced based on the number of memory reads per cycle multiplied by a weighting. The higher MRPC, then further the frequency delta is reduced. Finally, the CPU frequency is updated based on the current frequency and the frequency delta.

As with the governing algorithm presented in Chapter 5, the IPC threshold should be set based on the particular workload running. The addition of the MRPC measurement allows the governor to detect when the workload in question is performing a large number of memory accesses and therefore is becoming more memory bound. This causes the governor to reduce the operating frequency of the CPU even if the IPC is sufficiently high, and therefore the energy wastage is reduced for periods of high memory activity.

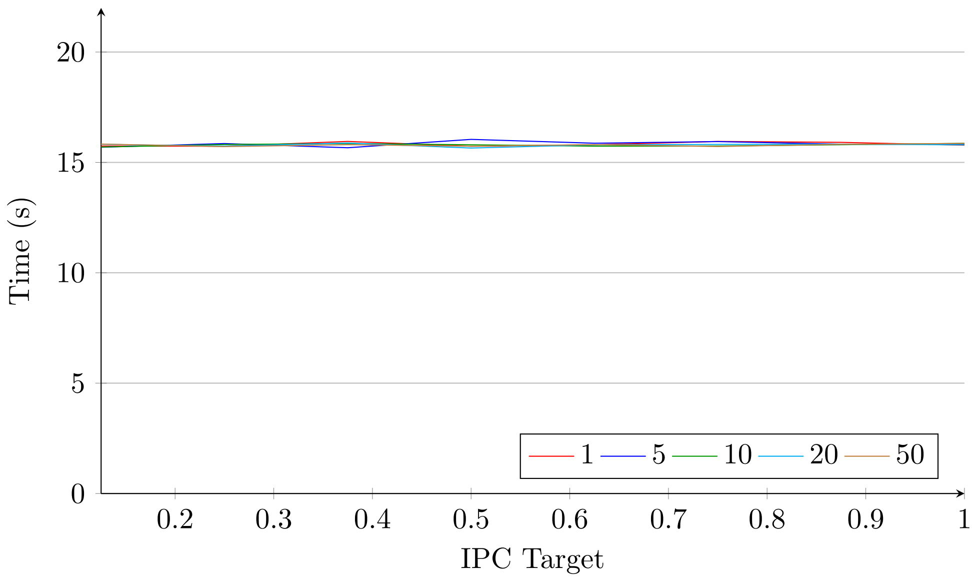


Figure 6.24: Dhrystone total execution time as IPC and MRPC targets are swept

6.6.3 Advanced System Governing Results

We evaluate the improved DVFS governor by performing a two-dimensional parameter sweep, in which we adjust both the IPC and the MRPC targets. This allows us to determine the ideal IPC and MRPC setting for each application, as well as understanding the impact of each parameter on the other. We consider IPC thresholds of 0.125 to 1.0 in 0.125 steps, and MRPC weightings of 1, 5, 10, 20 and 50, where each weighting is multiplied by 4194304, or 2^{22}. These values allow us to compare the performance of the governor at a large range of frequencies. We begin by looking at the operation of both Dhrystone and Memcpy, as these represent the two extremes for the CPU governor. Following these results we present results for both The Chase and Transporter as these are two system-level workloads which make use of both the CPU and GPU.

6.6.3.1 Dhrystone

The results for Dhrystone are presented in Figure 6.24, which shows how the run time for the workload changes as both the IPC and the MRPC target are changed. First, it is worth remembering that the IPC for Dhrystone is approximately 1.2, which is outside of the range of IPC thresholds tested for this governor. We choose not to evaluate the governor for IPC values greater than 1, as any application that is able to operate with such a high IPC will be compute bound, and should hence be run at a high frequency. However, our governing results demonstrate that the addition of the MRPC metric does not affect the operation of the governor of a typical range of IPC values when applied to a compute-bound workload. For all governor settings tested the results for Dhrystone

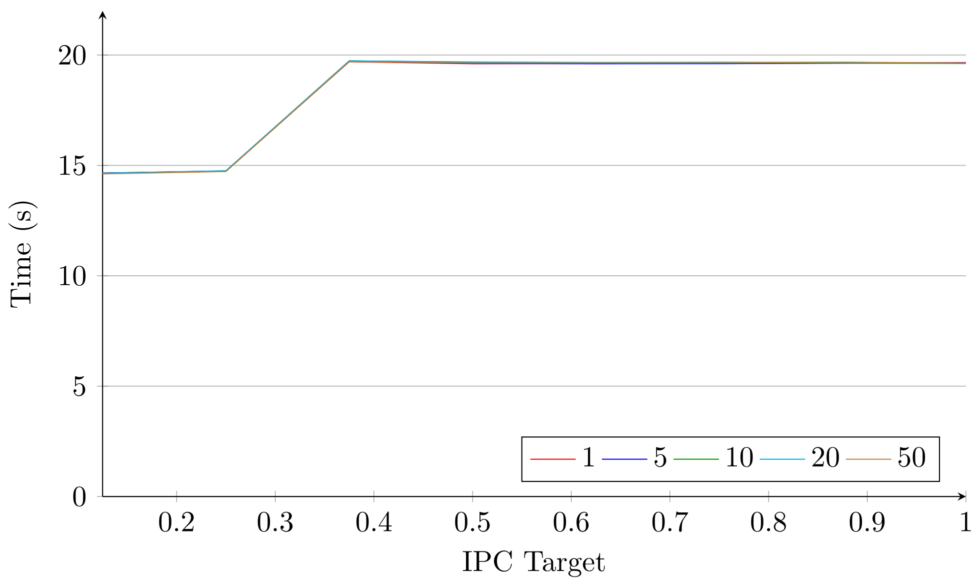


Figure 6.25: Memcpy total execution time as IPC and MRPC targets are swept

match those presented in Section 5.4.3. Therefore, we conclude that whilst the addition of the additional performance counter does not improve the operation of the governor for a compute bound workload, the resulting workload performance does not decrease either.

6.6.3.2 Memcpy

The results of the IPC and MRPC sweep for Memcpy are shown in Figure 6.25, and show how the run time of the benchmark varies as the two governor parameters are swept. The workload transfers a fixed amount of data, and therefore the transfer bandwidth is inversely correlated with the run time of the benchmark, and is hence now shown. As was the case for Dhrystone, this workload performs identically to the results presented in Section 5.4.4. Initially, this result appears counter-intuitive as Memcpy preforms a large number of reads from memory. However, as the workload is copying a large amount of memory, it disables caching for the memory accesses to avoid trashing the data already present in the cache. Therefore, the value of the MPRC metric always remains low and the frequency is not reduced by the large number of memory accesses. However, the IPC-based governing still reduces the frequency as was previously demonstrated.

6.6.3.3 The Chase

The Chase is a graphical workload which we use to analyse the operation of the improved DVFS governor. As this workload has phases where it is reliant on the CPU, and phases where the CPU is waiting for the GPU to render the next frame, we expect to see that the governor operates differently to the earlier presented IPC-only governor. This

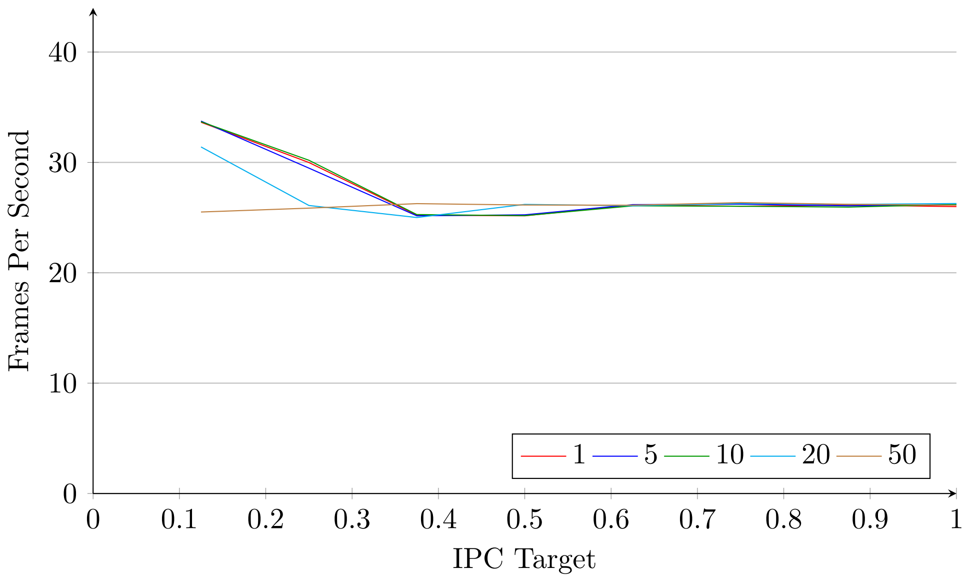


Figure 6.26: The Chase Odroid FPS as IPC and MRPC targets are swept. The MRPC weighting is shown in the legend.

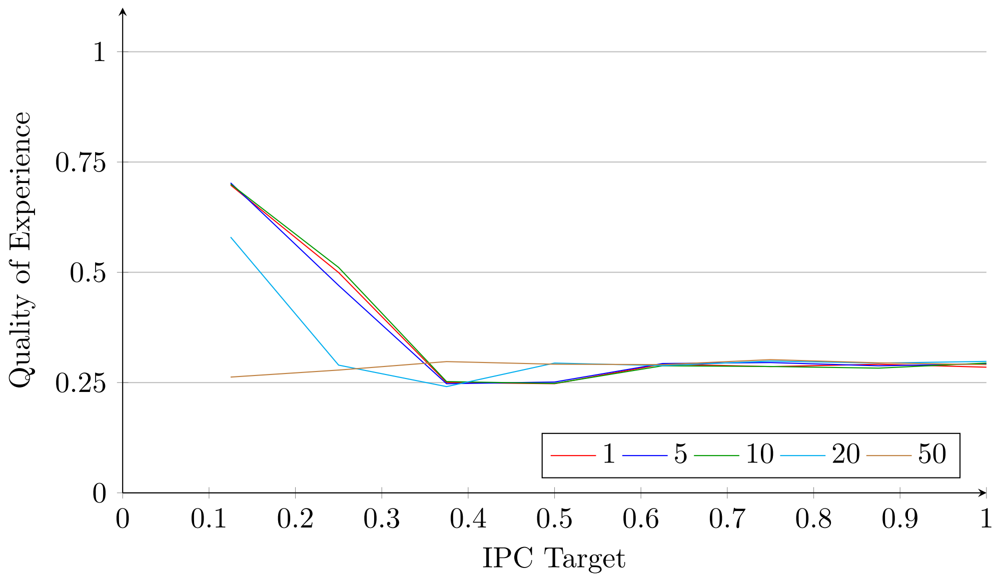


Figure 6.27: The Chase Odroid QoE as IPC and MRPC targets are swept. The MRPC weighting is shown in the legend.

workload utilises both the CPU and GPU and hence performs a large number of accesses to memory in order to pass the data between the two components. Therefore, we expect the addition of MRPC monitoring to affect the performance of the workload. Figure 6.26 shows how the frame rate of the GPU varies as the parameters are swept for The Chase, whilst Figure 6.27 shows the corresponding QoE. We also show the average frequency used by the CPU when running the workload in Figure 6.28.

As we demonstrated when profiling The Chase in Section 5.4.1.3, the workload has an

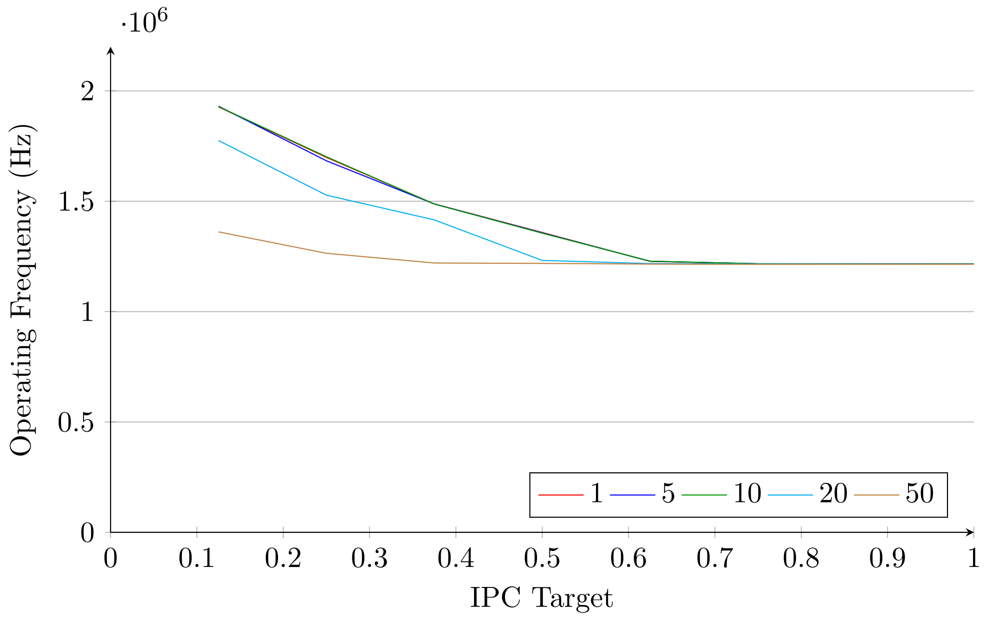


Figure 6.28: Average CPU frequency for The Chase as IPC and MRPC targets are swept. The MRPC weighting is shown in the legend.

IPC of 0.3 at the highest operating frequency of 2GHz on the Odroid XU3 to approximately 0.4 when running at the lowest frequency of 1.2GHz. From our results it is obvious that there is no change in the operation of the workload once the IPC target is set above 0.4, which is as expected. Below this range, both the IPC and the MRPC parameters have an effect on the operation of the governor. In general, the lower the IPC target, the higher the performance of the workload as the governor chooses higher operating frequencies, as can be observed in Figure 6.28. We also observe that higher settings for the MRPC weighting cause the governor to reduce the frequency. There is no discernible effect for lower MRPC weightings in the range of 1 to 10 as these are not significant enough to cause the frequency to be reduced. For the weightings of 20 and 50, the performance of the workload is lower, as can be seen in the resulting FPS and QoE, which is caused by the governor selecting a lower operating frequency on average. As is expected, a higher MRPC weighting results in a lower operating frequency as the governor is more likely to reduce the operating frequency when memory reads from the L2 cache are detected. For The Chase, the addition of MRPC results in lower CPU energy consumption, but does result in a noticeable reduction in user experience relative to the pure IPC-based DVFS governor.

6.6.3.4 RLBench

The RLBench application runs through a set of SQL transactions, each of which place a different load on the CPU and on the memory system. We present the overall run time for RLBench when the IPC and MRPC targets are swept in Figure 6.29. Additionally, the average frequency is presented in Figure 6.30. As was the case for The Chase,

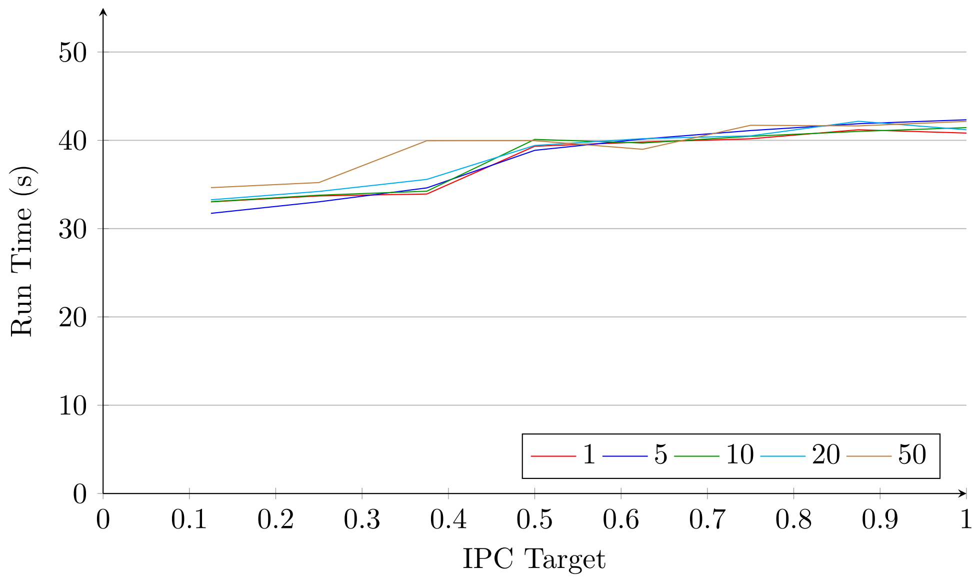


Figure 6.29: Overall run time for RLBench as IPC and MRPC targets are swept. The MRPC weighting is shown in the legend.

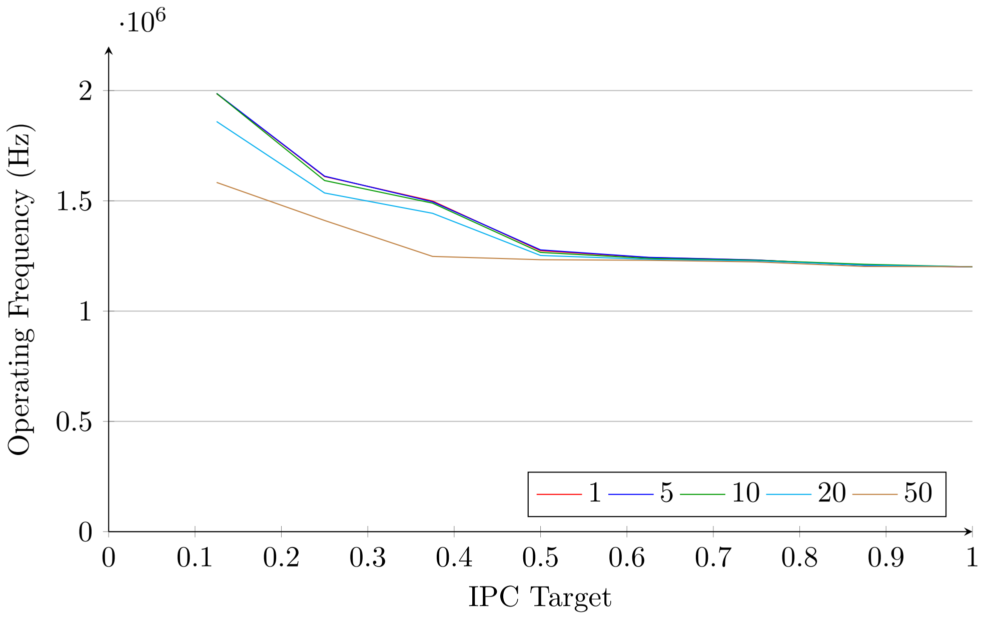


Figure 6.30: Average CPU frequency for RLBench as IPC and MRPC targets are swept. The MRPC weighting is shown in the legend.

the addition of MRPC results in the governor selecting a lower range of frequencies on average, which results in a degradation in workload performance. However, for an IPC target of 0.16 and an MRPC weighting of 50, the governor chooses an average frequency of 1.58GHz, whilst the governor with an MRPC weighting of 1 chooses 1.98GHz on average. This is a 20% reduction in operating frequency. For the same configuration parameters, the workload run time only increases by 1.61 seconds, from 33.03 seconds to 34.64, which is a sub-5% increase in run time. This demonstrates that whilst the low IPC target causes the governor to favour high operating frequencies, the introduction of the MRPC measurement forces the governor to choose a reduced operating frequency when there are a large number of accesses to main memory. The results for an IPC target of 0.32 and an MRPC weighting of 50 demonstrate the same characteristic. Therefore, these results demonstrate that the frequency of the CPU is automatically reduced when the workload becomes significantly memory-bound.

6.7 Concluding Remarks

In this chapter we combined the knowledge from the previous chapters with analysis of GPU behaviour and workloads to allow us to optimise for complete system performance, and user experience. We began by looking at the memory requirements for a set of GPU workloads in terms of bandwidth and the service required from the memory system. We established that a GPU is much more memory latency tolerant when compared to a CPU due to the fact that a GPU will operate on many threads at once, and is still able to complete work on one thread even if other threads are starved of data. Therefore, it is possible to sacrifice a small amount of GPU bandwidth in order to ensure that the CPU meets the latency targets required for efficient operation.

We moved on to look at how the GPU responds to both the available bandwidth, and the number of active GPU cores. We demonstrated that a minor reduction in available GPU bandwidth, relative to the maximum amount of bandwidth required by the workload when unconstrained, has a minimal effect on the GPU performance, and the GPU remains compute bound. However, for larger memory bandwidth reductions, we demonstrated that the GPU becomes memory-bound, and the performance suffers significantly. As the number of GPU cores increases, the amount of bandwidth required to sustain the operation of the cores increases close to linearly with the number of cores. The same applies to the performance of the workload. We demonstrated that the addition of GPU traffic can significantly affect the performance of a CPU workload. However, by placing a small limit on the amount of available GPU bandwidth we were able to reduce the impact on the CPU workload significantly. The effect on the GPU rendering rate was minimal.

Based on the GPU profiling, we created a GPU governor which adjusts the number of active GPU cores at run time based on the performance of the workload, and the delivered user experience. We demonstrated that we were able to reduce the energy consumption of the GPU significantly whilst maintaining a sufficiently high GPU frame rate and user experience. We then demonstrated the operation of the GPU governor in conjunction with the IPC-based CPU governor presented in Chapter 5. This combination of governors aimed to reduce the energy of both the CPU and the GPU when the level of user experience delivered by each was sufficiently high. We considered two different systems for this analysis: a single-core CPU and a dual-core CPU, both with

GPU. We demonstrated that we were able to reduce the energy consumption relative to the *Ondemand* and *Performance* CPU governors, whilst offering a higher QoE than the *Ondemand* governor, on average for a single-core system. For the dual-core system, we were able to match the QoE delivered by the *Performance* governor, whilst consuming less energy. We consumed a similar amount of energy to the *Ondemand* governor, whilst delivering significantly higher performance.

We presented an extension to our IPC-based, QoE-aware governor, which also took into account the interaction with the memory system. Specifically, our IPC based governor only considered the operation of the CPU when making decisions, but did not take into account the accesses to the memory system directly. We performed correlation analysis to determine which system performance counters contained the same information, and filtered down the set of performance counters accordingly. We then performed a linear regression analysis using the reduced set of performance counters to determine which performance counter was best able to predict the decisions made by the *Oracle* governor for The Chase. This led us to monitor the number of L2 cache refills per cycle to determine how memory bound a particular application was. This not only includes accesses to main memory, but also includes the traffic to, for example, the GPU. Hence, we are able to detect when the workload is dependent on different components in the system, and is hence operating inefficiently. We ran this governor on real hardware, and demonstrated that we were able to detect when workloads such as The Chase and Transporter were not CPU bound, and reduce the CPU frequency accordingly.

Chapter 7

Conclusions

Mobile phones and tablets have become ubiquitous in today's society. They are used to communicate with others via text message and calls, but also to retrieve information, take photos and videos, track the user's location and as entertainment devices. The mobile phone has evolved from a basic, bulky device whose primary function is to make calls, to a fully fledged mobile computer which is expected to meet all demands of the end user. Therefore, a modern mobile device contains significantly more hardware than just a CPU and associated memory system, as was found in the original mobile phones.

A modern smartphone or tablet contains a number of different components which include CPUs, GPUs, ISPs, memory controllers, storage devices as well as bespoke accelerators for various common tasks. Each of these components is required to inter-operate in order to provide the end user with the level of performance they desire. When one or more of these components does not achieve the required level of performance, the user experience is reduced, and the user becomes dissatisfied. Hence, it is vital that all components operate together properly and that one component does not impact the performance of another. A typical system will have QoS measures in place which provide, for example, prioritisation mechanisms which ensure that less vital components suffer first in the event that the system is unable to satisfy all of them. However, QoS mechanisms only consider the low-level operation of the device, and in general do not ensure a high level of performance or user experience.

Delivering a good user experience is imperative for a mobile device. Typically, developers attempt to ensure that the delivered user experience is high by running many of the components as fast as possible when the particular component is in use. However, this is often not required, and a significantly lower level of performance is sufficient to provide the level of experience demanded by the end user. Additionally, running as fast as possible results in high power and energy consumption, which reduces the run time of the battery powered devices. Therefore, each component should operate fast enough to deliver the minimum level of performance required to deliver a high quality of experience, whilst running slow enough to minimise the energy consumption. Additionally, running components at higher frequencies results in greater power dissipation, which, in turn, causes the device to heat up. This can also result in decreased user experience when the body temperature of the device reaches extremes.

Typically, modern smartphones and tablets will run Android, which is built on the Linux kernel, Apple's iOS or Microsoft's Windows Phone operating systems. The ones based on the Linux kernel make use of the CPUFreq framework, which adjusts the CPU frequency based on demand in order to save power and energy, as well as keep the device as cool as possible. The CPUFreq framework was originally designed for server systems and for home computers, and was never directly intended for use in mobile devices. Therefore, it optimises for saving energy when the system is idle, but can perform sub-optimally when the system is under medium load. The most commonly used CPUFreq governors monitor the CPU load to make decisions, and will increase the CPU frequency when there is work to do, and will reduce it otherwise. This makes sense for many home and server applications as the devices themselves are mains powered and have adequate cooling. However, mobile devices must conserve as much energy as possible, and therefore increasing the CPU frequency in response to work is often not the most sensible choice.

In this work we have built upon the Linux CPUFreq framework and have extended it to allow system optimisation based on the delivered user experience, as opposed to simple metrics such as CPU load. As it is not possible to directly measure the quality of experience delivered by a device we have presented user experience models which allow the user experience to be estimated based on low-level metrics. Using such models, it becomes possible to determine the level of experience being delivered to the end user, and then adjust component-level performance in order to either increase or decrease the level of experience delivered to the end user based on a user experience target.

We have presented models for two different types of user experience. This first is latency-based, whilst the second is throughput-based. Latency-sensitive user experience is when the user must wait for a single action to complete. Such an action, such as loading an application, should complete quickly in order to ensure that the end user does not become dissatisfied when waiting for the action to complete. Studies have shown that the end user is willing to tolerate a small delay, and often will not notice the delay provided that it is sufficiently small. However, once the delay has become sufficiently long, and the user becomes aware of the fact that they are waiting for the device, the level of experience quickly reduces. Hence, it is vital that as many user-triggered events complete sufficiently quickly as to not disrupt the experience of the end user.

Throughput-sensitive user experience refers to events which occur regularly and must complete at a minimum rate to provide a high user experience. For example, when playing a game on a mobile device, the GPU must render the frames. If the GPU takes too long to render a frame, then the frame rate drops and the user starts to see stuttering, the displayed frames do not appear as continuous motion, and instead appear as a sequence of discrete images. This results in a greatly reduced user experience. For throughput-based user experience the rate must remain sufficiently high and both continuously low rates and varying rates will impact the level of experience. Hence, the device must ensure that the rate is always sufficiently high. Conversely, should the rate be significantly higher than the level required for a good user experience, then the rate should be reduced to save energy and reduce the generated heat.

Our user experience models link low-level measurable system metrics to the delivered user experience. This allows the run-time estimation of user experience, and therefore provides a hook which allows the system to be optimised for the user experience, as opposed to performance or power. This requires multiple different components. First of all, workloads themselves must be classified to determine how they contribute to the user experience. Secondly, the user experience delivered by different system components must be combined to determine the system-level user experience. Finally, system metrics which can be related to the user experience must be chosen. This provide the facility to measure the user experience delivered by the device at run time.

It is vital to understand the operation of the different system components when adjusting their performance. Firstly, one must understand how the operation of the component itself changes as, for example, the frequency is adjusted. This ensures that the performance of the component is not reduced too significantly when making frequency adjustments, and thus avoids breaches of user experience guarantees, and also helps to understand the impact that the adjustments will have. Secondly, as most components in a SoC share the memory system, the traffic generated by one component affects the performance of the other components due to the memory system interference. Some components, such as the CPU require low memory latency to operate at high levels of performance for many applications, whilst GPUs require a large amount of bandwidth to memory, but are relatively latency tolerant. Therefore, a system-level understanding of the component interactions is required to make the correct run time decisions.

In Chapter 4 we began by considering the traffic generated by the CPU and demonstrated that the performance of typical CPU workload, BBench, is limited by the performance of the memory system by artificially adjusting the memory latency in simulation. We also show that the inter-transaction time is governed by the memory latency and performance of the caches, as well as the workload in question. We then followed this by an analysis of workload types. We demonstrated the performance of compute bound workloads scales with the CPU frequency as these workloads have minimal reliance of the memory system performance. Memory bound applications were shown to have close to no reliance on the CPU frequency once the frequency was sufficiently high. For low frequencies, the ratio of CPU cycle period to memory latency is small, and therefore the workload is frequency dependent. However, beyond that the workload has a very limited response to the CPU frequency. We also considered different types of workloads which have a mixture of compute-bound and memory-bound phases and showed these to have a limited response to CPU frequency due to the memory bound phases.

Following the workload analysis, we analysed the DVFS scalability for a set of four workloads. We used a simulation based approach which allowed us to decompose the workload into intervals which could be directly compared. Each interval was run concurrently at four different frequencies, and a vast number of statistics were gathered for each interval and frequency. These were used to determine the scaling potential of the workload offline. Specifically, we calculated a Pareto frontier for each workload to determine how well the different workloads respond to run time DVFS adjustment. We also demonstrated that it is possible to achieve the same level of performance at many different levels of energy consumption for all tested workloads. It is therefore imperative that the DVFS governors make the correct decisions to ensure that the desired level of performance is achieved with the minimum energy consumption possible.

We followed the workload analysis of Chapter 4 by comparing the performance of the standard Ondemand and Conservative CPU governors to an oracle governor. The oracle governor compared the instruction throughput of the lowest frequency to the instruction throughput at the frequency above it. It then chose the frequency that was within a specified reduction of the throughput at the lowest frequency. We demonstrated that both the Ondemand and Conservative governors will choose the highest frequency the majority of the time for the workloads tested. However, the oracle governor could be configured to pick lower frequencies, and therefore saved energy, albeit at a reduction of performance relative to the CPUFreq governors.

The presented oracle governors used the IPC to determine which frequency should be chosen, and therefore we decide to create a CPU governor which uses IPC to adjust the CPU frequency. However, unlike in the oracle governor tests, a real governor is not able to measure the performance of the workload at multiple different frequencies and then choose the best of those frequencies. Therefore, we opted to profile different workloads to determine the relationship between the IPC of the workload, and the resulting performance and user experience. Using this knowledge we then demonstrated that an IPC-based DVFS governor is able to make intelligent decisions regarding the CPU operating frequency. We profiled the workloads using the Odroid XU3 development board running Android by performing a frequency sweep for each workload, and recording the QoE, performance and IPC for the workload. These relationships then gave us the required information to create a CPU DVFS governor which is able to optimise for user experience.

We initially demonstrated the operation of our IPC-based governor using simulation, this allows us to try many different combinations of IPC targets in parallel by running many instances of the simulator, and hence explore a large part of the design space.

Additionally, it gives significantly more observability than real hardware. We were able to demonstrate that we were able to save energy relative to the standard CPUFreq governors found in Android by choosing an appropriate IPC target for a particular workload. Our approach proved successful in simulation for BBench, RLBench and AndEBench, and was demonstrated to work for both single core and dual core systems. Following the success of our governor in simulation, we proceeded to demonstrate the operation of the governor on the Odroid XU3.

We demonstrated that a fully compute-bound workload such as Dhrystone does not allow the performance to be adjusted significantly when using IPC to make decisions. This is due to the workload having a constant IPC irrespective of frequency as it has minimal reliance on the rest of the system. However, for other workloads we were able to adjust the IPC target to get a variation in workload performance. We run Memcpy, The Chase and Transporter using the IPC-based governor on the XU3. For each of these workloads there was a range of IPC targets which allows the user experience delivered by the workload to be traded for the energy consumption of the workload. Therefore, it is possible to use the IPC and profiling information to adjust the level of user experience of the workload at run time to meet demand.

Following the success of our governing strategies for the CPU, we moved on to consider system-level user experience. One of the other major components in a typical mobile device is the GPU which is responsible for rendering the frames to display before they are passed off to the display controller. Typically, the CPU prepares a frame before passing it off to the CPU via the memory system. The GPU then takes the prepared data, and renders the final frame to be displayed and places this in memory for the display controller to send to the screen. Therefore, any workload which involves the display will place strain on the CPU, GPU and the memory system, resulting in a truly system-level workload.

As with the CPU, we observed typical GPU traffic, and looked at how the GPU responds to perturbations in the memory system, as well as looking at how the number of active GPU cores affects the performance of the GPU. To look at these effects, we used an in-house GPU model which was integrated into the gem5 simulator. Whilst this GPU model did not interact with the CPU directly, and instead rendered frames from memory dumps, this model allowed us to observe GPU traffic and the impact it has on the rest of the system. We began by investigating how the bandwidth available to the GPU affects the rendering performance of the device. We demonstrated that the GPU can have compute- and memory-bound operation, as was the case for the CPU. When there is little bandwidth available for the GPU, the GPU is memory bound and spends the majority of the time waiting for data from memory, resulting in poor performance. However, as the available bandwidth increases the GPU transitions from being memory-bound to compute-bound. When the GPU is compute bound, providing it with additional memory bandwidth does not improve the performance of the device. We demonstrated this behaviour with one through four GPU core configurations. The compute bound performance was found to scale with the number of cores, whilst the memory-bound region of operation increased with the number of cores. This is due to the fact that more memory bandwidth is required to provide the data for the additional GPU cores.

We followed this analysis of GPU behaviour with a study of the ideal number of GPU cores for a graphics workload. We observed that it does not always make sense to run the GPU with the maximum number of cores, especially when user experience is concerned. Once the frame rate becomes sufficiently high to satisfy the end user, further increased performance does not result in increased user experience, and therefore all additional performance results in wasted energy. We demonstrated that only 6 and 8 cores out of 16 in total were required for two GPU workloads to achieve the desired performance.

Next, we investigated the impact of GPU traffic on the performance of the CPU. First of all, we measured the impact of GPU memory traffic on the memory latency as seen by the CPU. The addition of the GPU traffic resulting in a significant increase in the memory latency, in some cases being as much as a 6.6x increase. We then demonstrated that placing a small limit on the memory bandwidth for the GPU resulted in a significant memory read latency improvement. This small bandwidth limit reduces the burstiness of the GPU traffic, and therefore reduces the peak impact on the memory controller. This results in lower latency throughout the memory system, at the expense of a small reduction in GPU performance. However, the reduction in memory performance is insignificant, especially when compared to the overall latency reduction. We demonstrated that we were able to noticeably improve the IPC of the CPU due to the small reduction in GPU bandwidth.

We then investigated how the CPU and GPU interoperate for a larger workload run. Specifically, we observed the impact of GPU traffic for various core configurations of CPU DVFS performance. We ran BBench at a range of frequencies, whilst simultaneously rendering frames on the GPU. As expected, the introduction of the GPU traffic reduced the performance of the CPU, as could be seen in the IPC of the CPU. We demonstrated that the introduction of the GPU traffic results in a significant decrease in CPU performance, at the expense of noticeably greater CPU energy consumption.

As we now understood the impact that GPU traffic has on the CPU performance, as well as the energy consumption, we created a simple GPU governor. This governor monitors the frame rate of the GPU and determines if GPU cores should be switched on or off in order to achieve the desired level of user experience. We demonstrated that we were able to achieve a level of user experience which matches that of running at the highest number of cores, whilst also reducing the energy consumption of the GPU significantly. The QoE per Energy number obtained was higher than just running with the lowest number of cores demonstrating that we were able to operate significantly better than when running without the GPU governor in place.

We then ran the GPU governor alongside the CPU governor presented in Chapter 5, and demonstrated the ability to adjust both the CPU and the GPU performance at the same time to achieve the desired level of user experience. For a single core system the *QoE-aware* governor was able to reduce the energy consumption of both the CPU and GPU relative to both the *Performance* and *Ondemand* governors, whilst delivering higher GPU QoE than the Ondemand governor. For the dual core system configuration, the *QoE-Aware* governor was able to match the performance of the *Performance* governor, whilst matching the energy consumption of the *Ondemand* governor.

Finally, we extended the IPC based governor presented in Chapter 5 to also include information about the memory system. We used the results from our oracle governing study to determine which CPU PMU counters provided us with the required information, and extended our IPC governor to include L2 cache read refills. When there are a large number of reads from memory, this indicates that the CPU is more reliant on the memory system, and is hence more memory-bound. In this case, the frequency of the CPU should be reduced. Our improved governor compares the IPC of the workload to a target IPC, and sets a frequency delta accordingly to either increase, decrease or leave the CPU frequency as is. The governor then uses information about the memory accesses to further reduce the CPU frequency in the event that there are a large number of memory accesses. The improved CPU DVFS governor was able to successfully adjust the CPU frequency in response to memory accesses, resulting in reduced energy consumption and improved system efficiency.

Whilst we have focused on the ARM architecture throughout this work, our results are architecture agnostic, and our methodologies and findings can be applied to a range of systems, irrespective of the architecture. Our user experience models have been chosen such that the can translate simple low-level metrics into user experience. Specifically, we use time, which is the same across all systems. Our governors rely on PMU counters to make decisions, and this is where the biggest hurdle lies if the methodologies are to be applied to another system as different PMU counters must be chosen to replace those we have used for the ARM architecture. However, the majority of modern systems will include a comprehensive set of PMU counters, and therefore these can be exchanged for the ones used herein.

In this thesis we have considered Quality-of-Experience-Aware system optimisation, which is posed as an alternative to simply optimising for performance and power. This type of optimisation allows for greater energy savings by understanding the context of operation, and hence ensuring that the user's expectations are met. We established and defined a relationship between low-level performance and high-level user experience in Chapter 3. Using this knowledge we developed QoE utility functions which translated temporal performance measures into user experience. In order to optimise for user experience using these models, we first needed to understand the requirements of CPU and GPU workloads, and hence performed detailed workload analysis in Chapters 4 and 6.

Additionally, we conducted a detailed fine-grained DVFS analysis in Chapter 4. We proposed and demonstrated a QoE-Aware CPU governor in Chapter 5, and demonstrated it using both simulation and real hardware. A QoE-aware GPU governor was presented and analysed in Chapter 6. Finally, we combined the CPU and GPU QoE-aware governors in Chapter 6, and showed whole system energy savings whilst maintaining a sufficiently high QoE. Overall, we have presented encouraging, architecture-agnostic results and methodologies which we hope will form a vital and significant first step towards future QoE-aware system optimisation strategies.

7.1 Research Questions and Answers

As part of this research, we asked a set of questions which we aimed to answer throughout the course of the work. We list both the questions, as well as the answers below.

1. *Can a quantitative relationship between low-level metrics and user experience be defined which permits realistic estimation of user experience at run time?*

   We have successfully established a relationship between the user experience delivered by a device or service and measurable low-level metrics. We presented QoE utility functions in Chapter 3 which translated time delays and rate into QoE for latency and throughput workloads, respectively. These QoE models are based on relevant research from the literature. We have used these metrics as part of our QoE-aware governors for both the CPU and the GPU presented in Chapters 5 and 6. These governors measured proxies for the user experience at run time, and used this information to determine how the system must be adjusted in order to maintain a sufficiently high level of user experience. Hence, we were able to use low-level metrics to estimate the user experience at run-time, and used these to optimise the system.

2. *From a low-level perspective, which level of service - in terms of latency and bandwidth - do typical components such as CPUs and GPUs require from the shared memory system?*

   We analysed the traffic generated by CPUs and GPUs in Chapters 4 and 6, respectively. We demonstrated that a CPU requires much less bandwidth from the memory system relative to a GPU, but also that the CPU requires a low latency service from the memory system, whilst the GPU is able to tolerate mild increases in latency with minimal impact on the rendering performance. We investigated how a GPU responds to mild bandwidth limitation, and demonstrated that there is a minor decrease in GPU performance. However, at the same time, as there was less pressure on the memory controller, the performance of the CPU increased. This serves to demonstrated that whilst the CPU and GPU require significantly different levels of service from the memory system, they must be considered at the same time in order to ensure that the resulting operation is that which is desired. We also demonstrated that the introduction of large volumes of GPU traffic shift the ideal operating point for a CPU as it must operate at higher frequencies to achieve the same level of performance and user experience compared to when the GPU is not present in the system.

3. *How does CPU frequency affect performance and user experience for a spectrum of workload scenarios, ranging from compute-bound to memory-bound?*

   In Chapter 4 we presented a detailed investigation of how the CPU frequency affects the operation of a range of workloads. These workloads ranged from compute-bound to memory-bound in order to explore both the extremes as well as providing a realistic set of use cases. We demonstrated that compute-bound workloads scale linearly with CPU frequency, whist memory-bound workloads show close to no scaling as the frequency is increased. We also demonstrated that typical CPU workloads lie in between these two extremes.

   In the same chapter we also explored the shape of the DVFS Pareto frontier for four workloads. Again, we included the extremes with one compute- and one memory-bound workload, but also included a graphical workload and a web browser benchmark. We demonstrated that it is possible to make a wide range of different DVFS decisions, with varying energy and performance trade offs. Therefore, it is vital to choose the minimum energy point for a given performance level.

4. *Across a range of typical graphics workloads, what effect does the scaling of GPU core count and available memory bandwidth have in terms of performance and user experience?*

   As with the CPU, we analysed the operation of the GPU in various configurations. Specifically, in Chapter 6, we adjusted the available GPU bandwidth to determine how the GPU responds to receiving insufficient bandwidth from the memory system. We demonstrated that the GPU, like the CPU, can be memory-bound or compute-bound. When there is little bandwidth available to memory, or a particularly memory intensive graphics workload is being run, then the GPU is memory-bound. However, as the available memory is increased, the GPU will transition to becoming compute-bound, and further increases in memory bandwidth will not increase the performance.

   Secondly, in the same chapter, we investigated the effects of the number of GPU cores on the performance of various workloads. When fewer GPU cores are active, the GPU consumes less memory bandwidth as it is not able to process the information as quickly, and is therefore more often compute-bound. This has the side effect of placing less strain on the shared memory system, and therefore there is less of an impact on the CPU performance. Additionally, when the energy consumption is taken into account, it is most optimal to use the fewest number of GPU cores required to meet the frame rendering deadline.

5. *Making various assumptions about user preferences - which we will term user experience models - how can CPUs, GPUs as well as complete systems be run-time optimised in such a way that the end-user remains satisfied?*

   In this work we have created both CPU and GPU governors which act upon proxies for user experience to ensure that the user experience meets the desired level. We have presented these governors in Chapters 5 and 6, and have evaluated them on both real hardware and in simulation. Our governors are based on the QoE utility functions presented in Chapter 3. These governors operate at run time to adjust the performance of either the CPU or the GPU to the desired level of user experience, whilst choosing the lowest frequency or fewest number of active cores required to do so. Therefore, these governors not only attempt to ensure that the user experience is adequate, but also attempt to ensure that the minimum energy is consumed whilst doing so.

7.2 Future Work

In this work we have presented user experience models, analysed the operation of CPUs and GPUs, including their workloads, and have devised governing strategies which are able to adjust overall system energy and performance. However, there remains a large amount of work in this field, which we have been unable to address due to time constraints. This section provides a summary of closely related future work.

7.2.1 Long-Term User Experience Optimisation

In this work we have focused on short-term user experience, but have not considered the impact of our decisions on long-term user satisfaction. Our work provides the initial steps required to create a framework for user experience optimisation that can be applied to a large range of mobile systems. Future work should consider the impact of decisions on the long-term user experience, such as battery lifetime. For example, if the device consumes too much energy ensuring that the user is satisfied in the short-term, then the battery lifetime will be reduced, and that will lower the long-term user experience. Therefore, there is a trade off between short-term and long-term user satisfaction, which lies outside the scope of this work. This is, however, vital to consider when deploying such a strategy to real systems.

7.2.2 Include More System Components

In this work we have considered how to optimise mobile systems for user experience, and have specifically focused on the CPU, GPU and memory system. However, there are a vast number of other components to consider. For example, it is important to consider Image Signal Processors (ISPs), which will process the data from the device camera. These devices consume a vast amount of memory bandwidth, especially when a user records a video. This places a large strain on the memory system, but also on the CPU and GPU which will be used to show the end user what is being recorded and to store the data itself to the device's non-volatile memory. It is therefore important that such a device gets sufficient bandwidth, and that this bandwidth requirement can be satisfied without impacting the performance of the other in-system components. In the event that the bandwidth requirements cannot be met, then the video or photo quality suffers, resulting in a potentially useless recording. However, if the CPU or GPU become starved due to the additional memory bandwidth, then the user is not able to interact properly with the device, resulting in reduced user experience, or even the inability to record what they intended to record.

Other system hardware to consider are components such as video and audio decoders. In modern devices, one will often find both hardware video and audio decoders which can take a large amount of the strain off the CPU. In the event that these are not performing as well as required a decoded video can either suffer in quality, or can stop altogether, whilst audio on the device may start to suffer. Again, this results in a significantly reduced user experience. As was the case for the ISP, it is vital to understand both the requirements of these components in terms of memory bandwidth, but also to determine the impact these have on the other devices in the system. Finally, if one wishes to adjust the operation of these at run time to save energy, one must understand the user experience impact of these decisions.

7.2.3 Investigate a Larger Set of Workloads

Throughout this work we have considered a range of workloads, ranging from compute- and memory-bound benchmarks to full-system graphics and web-browser workloads. However, we have considered a limited subset of all workloads, and have only considered those which can be easily automated. Other workloads, such as word processing or spreadsheet entry have not been considered, although these are becoming more prevalent in the tablet and mobile market as users move from traditional PCs and laptops to more tablet-class devices. These devices are rapidly becoming the dominant interface used to enter and retrieve information, and therefore it is important to evaluate the performance of this class of application.

7.2.4 Conduct a User Study

Throughout this work we have provided an extensive study of low-level device operation, and have presented models which link the low-level performance of the device to the delivered user experience. We have additionally demonstrated that the predicted user experience is high for our approaches. However, we do not present user studies, in which end users are asked to evaluate the subjective performance of the device.

We envision that a user strategy could be used to improve the work presented in this dissertation in a variety of ways. Firstly, it could be used to verify the QoS models presented in Sections 3.2.1 and 3.2.2 by obtaining user feedback for a variety of different scenarios. These scenarios could include, for example, the loading time for an application, or delay between the user requesting an action and the action taking place. These could be used to affirm that the latency-sensitive workload model is correct. Additionally, some rate based experiments, such as those focusing on GPU frame rates, could be used to determine the accuracy of the throughput-sensitive workload model.

A user study could also be used to evaluate the performance of the different governing strategies. As this work stands, the evaluation of the governing strategies focuses more on the device, as opposed to the end user. A user study would allow real-life evaluation of the different governing methodologies presented in this work.

7.2.5 Advanced QoE Governing with Control Theory

Adjusting the frequency of the CPU and monitoring the resulting performance to make the next adjustment is a complex task. This is further compounded by the workload itself which can change significantly between governing intervals. In the best case, a governor can act upon the historic information gathered from previous governing cycles. A typical governor observes metrics such as the CPU load to make decisions. More often than not, the CPU load will be either high or low, and it is less common to see intermediate CPU load levels. This ensures that the governor has a clear, well-defined governing algorithm. However, in the case of a QoE-based governor, it has to make decisions based on either the level of user experience, or the actual performance of the workload. This is much less clear, and poses many issues.

A QoE governor measures the past user experience, and uses this information to adjust system configuration to reach a target user experience for the current governing interval. This can be represented in terms of a control theory feedback loop. In Figure 7.1 we present the QoE-aware governing feedback loop. The governor is provided with an input in the form of a target QoE, QoE_{target}, which is compared to the measured QoE from the last governing interval, $QoE_{measured}$. This results in an error term, err, which can be used to determine if the level of user experience is sufficient, or not. This error term

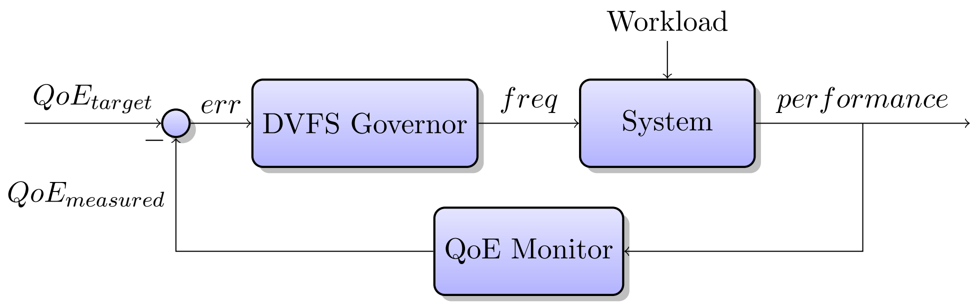


Figure 7.1: Control theory based QoE governing

is then used by the *DVFS Governor* to determine the frequency, $freq$, which should be applied to various parts of the *System*. Based on the capabilities of the system, the *Workload* running and the operating frequencies of the system components, a level of performance is reached. In order to provide a QoE for the last governing interval, this performance in monitored and is converted to a representative QoE score by the *QoE Monitor*. This allows the governor to compare the QoE from the last governing interval to the target QoE.

We propose that the *DVFS Governor* is implemented as a proportional-integral-derivative (PID) controller. Such a controller uses the error term, err in our case, to determine which frequency should be chosen for the next interval. This is done in three parallel stages: proportional feedback, integral feedback and derivative feedback. The proportional feedback term applies a constant multiplier to the most recent error measurement. The proportional feedback is the main correcting force in a PID controller. The integral term tracks the total error, i.e., the sum of all measured errors, and multiplies this by a constant to remove steady state error. Finally, the derivative term acts upon the rate of change in the error, effectively comparing the most recent two errors to determine a rate of change. This term acts as a predictor for changes in the system, as a large error derivative indicates that the system has been perturbed significantly. Tuning a PID controller requires very accurate setting of the component weightings to ensure that the response of the controller is stable, and does not oscillate, whilst still reacting rapidly enough to make quick adjustments.

Such a controller, when coupled with detailed QoE modelling would be able to automatically adjust the performance of system components such as the CPU and GPU based on a QoE target. By adjusting the QoE target it would be possible to adjust the level of QoE delivered by the device. Potential hardships lie in the tuning of the PID controller configurations, and determining which aspects of a QoE measurement are contributing to the overall system performance. For example, it is important to exclude the QoE of the GPU if it is not involved in providing a high user experience.

7.2.6 Hardware-Based QoE Optimisation

The approaches we have presented are all software implementations of QoE governing algorithms, most of which rely on the CPUFreq framework used to the standard Linux DVFS governors. However, aside from the ability to change governing strategy on the fly, there is no reason that the governor cannot be implemented in hardware. By implementing the governor in hardware, there is no overhead for making the DVFS decisions which improves performance, reduces energy consumption and helps to ensure that there is a smooth user experience. Hardware-based QoE-aware governing also allows for the governor to directly integrate with the PMU counters, and hence closely monitor for system events. This would result in faster and more accurate governor responses to changes in the system state.

7.2.6.1 Unifying QoE and QoS Control

By moving the QoE governor to hardware, it can also be more tightly coupled with the QoS mechanisms present which ensure that no components become starved in the memory system, for example. This tight coupling between QoE and QoS allows the QoE governor to inform the QoS mechanisms regarding the relative importance and requirements of the various system components. This in turn allows the QoS mechanisms to provide specific parts of the system with a greater quota of the available bandwidth. This also could provide the QoE governor with knowledge as to where bottlenecks in the system lie, and hence determine if increasing the frequency of a specific component will indeed result in improved QoE.

Appendix A

Forking Infrastructure

In this appendix we detail the implementation of the analysis framework based on forking the gem5 simulator. This framework has been used in Chapters 4 and 5 to give insights into workload scalability, DVFS governing limits and how to design a performance counter based DVFS governor. This appendix provides more detailed information about the forking framework than presented in the chapters themselves.

The gem5-based forking infrastructure is designed to allow a workload to be explored with different system configurations whilst ensuring that the different runs remain synchronised. To this effect, a single primary gem5 simulation is executed, and it is only this simulation that is forked. Each forked simulation is then allowed to execute for a short period of time, prior to being terminated. The reasons for only forking a single gem5 simulation are two-fold and are explored in the following two paragraphs.

First of all, if the child simulations, which are produced as a results of forking, are themselves forked, then the number of simulations rises exponentially, which is undesirable for larger workload runs as the compute resources required also increase exponentially. However, it is worth noting that this approach does potentially have advantages. Specifically, by forking the child simulations it allows decisions made by the simulated system within those child instances to propagate further into the simulation. This allows a much larger set of choices to be explored, but the child simulations would rapidly desynchronise from the original run.

Secondly, by only forking from a single gem5 simulation, it is ensured that all simulations are synchronised to this primary simulation run, and hence can be directly compared. For this to remain true, each child simulation must run for a significantly short period of time as the decisions made by the software and operating system in each child simulation can differ from the primary simulation. Hence, if the child simulations run for too long an interval, they can significantly diverge, and they are no longer comparable.

```
for each governing interval, $i$ do
  Fork the gem5 simulator $N$ times
  for each forked simulation do
    Set the CPU frequency to $freq[N]$
    Run for one forking interval
    Gather statistics for the forking interval
    Terminate the child simulation
  end for
end for
```

Figure A.1: Algorithm for the forking infrastructure

A.1 Comparing the Simulated Intervals

In this section we describe how each set of simulated intervals is compared using the forking infrastructure. As was stated in the previous section, all forked instances of the simulator are tied to a primary simulation, and only run for a short period of time. In our runs with the forking infrastructure developed, we investigate the effect of changing the CPU frequency. This has the effect that running for the same period of time at different CPU frequencies results in differing amounts of work done in each child simulation. Therefore, post-processing is required in order to obtain useful results.

First of all, we determine the number of instructions simulated in the highest-frequency child. Therefore, if we have four simulated frequencies of 0.5 GHz through 2 GHz in 0.5 GHz steps, we first determine how many instructions were simulated in the 2 GHz simulation. We then determine how many were executed in the other, slower simulation runs, and calculate the ratio. This ratio is then used to scale the non-mean statistics for the slower simulation runs. Note that we do not scale any mean statistics, as the mean already takes into account the shorter run time.

The scaled statistics allow us to estimate the actual statistics if the same number of instructions had been executed in all simulations. However, as we are interested in performance we also need to scale the run time for each child simulation. Therefore, if a simulation at the highest frequency ran for 1 ms, and a simulation at a lower frequency executed half the number of instructions in the same period of time, we estimate the run time for the slower simulation to be 2 ms.

We determine the energy consumption using a simple IPC-based energy model. For the lower-frequency intervals we use the scaled statistics for energy calculation. The scaled times are used for performance estimation. By using both of these results we are able to estimate the resulting performance and energy consumption for any choices made by a DVFS governor. Additionally, as we are able to choose the most appropriate frequency which maximises or minimises a given metric, we are able to create an oracle governor which operates on this data.

A.2 Forking Algorithm

We present the forking algorithm used for our experiments in Figure A.1. This algorithm is used to investigate the effect of adjusting the CPU frequency, and allows us to directly compare these effects.

Appendix B

Source Code Listings

This appendix lists the source code for various scripts, kernel modifications and pseudo code for algorithms used throughout the thesis.

B.1 DVFS Governors

This section includes the pseudo code for different DVFS governing algorithms used throughout the thesis. It includes the algorithms used to mimic the operation of the CPUFreq Ondemand and Conservative governors when used with the forking infrastructure. Additionally, this section includes the algorithms used for the Oracle governors used to determine how much energy can be saved relative to the CPUFreq governors.

B.1.1 CPUFreq Governors

In this section, we provide the settings use for the *Ondemand* and *Conservative* CPUFreq governors, when implemented offline for the DVFS case study. We also detail the algorithms implemented for the two governors.

The *Ondemand* and *Conservative* governors use thresholds to when the frequency should be increased, decreased or left the same. These are used to implement hysteresis, and hence avoid the governor switching CPU frequency in response to a small change in the input. These thresholds are applied to the measured CPU load, as determined within the kernel on a typical system, or as derived from the gem5 statistics in our implementation.

Both governors use an up-threshold which represents the CPU load required for the governor to increase the frequency. We set the up threshold to 80% for both governors. Therefore, the CPU must be at least at 80% load for the frequency to be increased. It is worth re-emphasising that the *Ondemand* governor will switch to the highest operating frequency when this threshold is reached, whilst the *Conservative* governor will slowly increase the frequency one step at a time. Therefore, the conservative governor is less responsive to increases in CPU load.

The *Ondemand* and *Conservative* governors also use a down threshold, which is used when decreasing the operating frequency. In the case of the *Ondemand* governor, we use a threshold of 80%, whilst we use 20% for the *Conservative* governor. The CPU load must drop below this threshold for the frequency to be decreased. The lower down threshold for the *Conservative* governor means that it is more likely to remain at a higher frequency whilst there is load on the CPU, and hence will offer more stable performance at the expense of higher energy consumption once the frequency has been increased.

B.1.1.1 Ondemand

In this section, we present the algorithm used for the offline implementation of the *Ondemand* governor. The governor acts upon historic data to determine which frequency should be chosen for the next governing interval. The algorithm is shown in Figure B.1.

```
$frequencies \leftarrow$ set of available frequencies; $i_{max}$ in total
$next\_freq \leftarrow frequencies[0]$
$i \leftarrow 0$
for each governing interval do
  if CPU load $>$ up threshold then
    $i \leftarrow i_{max}$
  else if CPU load $<$ down threshold and $i > 0$ then
    $i \leftarrow i - 1$
  end if
  $next\_freq \leftarrow frequencies[i]$
end for
```

Figure B.1: Algorithm for the Ondemand DVFS governor

B.1.1.2 Conservative

In this section, we present the algorithm used for the offline implementation of the *Conservative* governor. As was the case for the *Ondemand* governor, this governor acts upon historic data to determine which frequency should be chosen for the next governing interval. The algorithm is shown in Figure B.2.

B.2 Kernel

In this section, we include the key snippets of code required to implement the IPC-based CPU DVFS governor, as well as the more advanced IPC- and MRPC-based governor

```
$frequencies \leftarrow$ set of available frequencies; $i_{max}$ in total
$next\_freq \leftarrow frequencies[0]$
$i \leftarrow 0$
**for** each governing interval **do**
  **if** CPU load $>$ up threshold **and** $i < i_{max}$ **then**
    $i \leftarrow i + 1$
  **else if** CPU load $<$ down threshold **and** $i > 0$ **then**
    $i \leftarrow i - 1$
  **end if**
  $next\_freq \leftarrow frequencies[i]$
**end for**
```

Figure B.2: Algorithm for the Conservative DVFS governor

used in later experiments. Please note that we do not include the entire source for brevity, and hence the code listing here serve to illustrate the operation of the governor, rather than providing a fully-functional implementation.

B.2.1 IPC-based DVFS Governor

```c
static DEFINE_PER_CPU(int, counters_enabled) = 0;
static DEFINE_PER_CPU(unsigned, ipc_threshold);

// These are used to measure the number of cycles and number of instructions
// across all cores in the domain.
static DEFINE_PER_CPU(u64, measured_cycles);
static DEFINE_PER_CPU(u64, measured_insts);

// We want to keep track of the time that the counters were running. Only do so
// for the instruction counter for now.
static DEFINE_PER_CPU(u64, insts_old_time_enabled);

// Performance counter specific things:
static DEFINE_PER_CPU(struct perf_event *, cycle_ev);
static DEFINE_PER_CPU(struct perf_event *, inst_ev);

static struct perf_event_attr cycle_hw_attr = {
    .type       = PERF_TYPE_HARDWARE,
    // .config    = 0x11,
    .config     = PERF_COUNT_HW_CPU_CYCLES,
    .size       = sizeof(struct perf_event_attr),
    .pinned     = 1,
    .disabled   = 1,
};

static struct perf_event_attr inst_hw_attr = {
    .type       = PERF_TYPE_HARDWARE,
    // .config    = 0x08,
    .config     = PERF_COUNT_HW_INSTRUCTIONS,
    .size       = sizeof(struct perf_event_attr),
    .pinned     = 1,
    .disabled   = 1,
};
```

```
void enable_pmu_counters()
{
    volatile unsigned int v;
    READ_REG(PMCR, v);
    v |= 0x01;
    WRITE_REG(PMCR, v);
}

/* Enable the performance counters for each CPU included in the current policy */
void gov_enable(int cpu)
{
    struct sb_cpu_dbs_info_s *dbs_info = &per_cpu(sb_cpu_dbs_info, cpu);
    struct cpufreq_policy *policy = dbs_info->cdbs.cur_policy;
    struct cpumask cpu_mask;

    for_each_cpu(cpu, &cpu_mask) {
        printk("CPU: %d\n", cpu);
        printk("Enabling the performance counters\n");

        // Enable the performance counters
        enable_pmu_cycles(cpu);
        enable_pmu_inst(cpu);

        printk("Setting initial ipc target\n");
        per_cpu(ipc_threshold, cpu) = 40;

        // Reset the counters
        printk("Resetting internal counters\n");
        per_cpu(measured_cycles, cpu) = 0;
        per_cpu(measured_insts, cpu) = 0;

        per_cpu(insts_old_time_enabled, cpu) = 0;

        per_cpu(counters_enabled, cpu) = 1;
    }
}

/*
 * Every sampling_rate, we look at the performance counters to determine
 * what the next frequency should be.
 */
static void sb_check_cpu(int cpu, unsigned int load)
{
    struct sb_cpu_dbs_info_s *dbs_info = &per_cpu(sb_cpu_dbs_info, cpu);
    struct cpufreq_policy *policy = dbs_info->cdbs.cur_policy;
    u64 cycles, inst;
    u64 enabled = 0, running = 0;
    struct perf_event * cycle_event, * inst_event;
    int cpu_new;
    u64 overall_cycles = 0, overall_insts = 0;
    long long int freq_next;
    unsigned int cur_freq;
    int error, delta;
    u64 time_enabled;
    long long int ipc;
    u64 overall_time = 0;
    unsigned num_cores = 0;

    struct cpumask cpu_mask;
```

```
if (per_cpu(counters_enabled, cpu) == 0) {
    printk("Enabling PMU counters!\n");
    gov_enable(cpu);
    return;
}

if (have_governor_per_policy()) {
    cpumask_copy(&cpu_mask, policy->cpus);
}
else {
    cpumask_copy(&cpu_mask, cpu_online_mask);
}

for_each_cpu(cpu_new, &cpu_mask) {
    num_cores += 1;

    cycle_event = per_cpu(cycle_ev, cpu_new);
    cycles = perf_event_read_value(cycle_event, &enabled, &running);

    enabled = 0;
    running = 0;

    inst_event = per_cpu(inst_ev, cpu_new);
    inst = perf_event_read_value(inst_event, &enabled, &running);

    // Get the time that we have been counting for and update the reference.
    time_enabled = enabled - per_cpu(insts_old_time_enabled, cpu_new);
    overall_time += time_enabled;

    per_cpu(insts_old_time_enabled, cpu_new) = enabled;

    overall_cycles += cycles;
    overall_insts += inst;

    per_cpu(measured_cycles, cpu_new) += cycles;
    per_cpu(measured_insts, cpu_new) += inst;

    // We need to reset the performance counters:
    local64_set(&cycle_event->count, 0);
    local64_set(&inst_event->count, 0);
}

// Calculate the IPC
ipc = overall_insts;
if (overall_cycles >> 7 != 0) {
    do_div(ipc, overall_cycles >> 7); // IPC * 128
} else {
    ipc = 0;
}

// Get the current frequency
cur_freq = policy->cur;

// Look up the current IPC threshold for the CPU we are running on
ipc_threshold = per_cpu(ipc_threshold, cpu);
if (ipc < ipc_threshold) {
    // Decrease the frequency
    delta = -100000;
    } else {
        // Increase the frequency
        delta = 100000;
    }

    freq_next = cur_freq + delta;
    if (freq_next < policy->min) {
        freq_next = policy->min;
    } else if (freq_next > policy->max) {
        freq_next = policy->max;
    }

    if (freq_next <= cur_freq) {
        // Switching down.
        __cpufreq_driver_target(policy, freq_next, CPUFREQ_RELATION_L);
    } else {
        // Switching up.
        __cpufreq_driver_target(policy, freq_next, CPUFREQ_RELATION_H);
    }

}
```

B.2.2 IPC- and MRPC-based DVFS Governor

```
static DEFINE_PER_CPU(int, counters_enabled) = 0;
static DEFINE_PER_CPU(unsigned, ipc_threshold);
static DEFINE_PER_CPU(unsigned, mrpc_mult);

// These are used to measure the number of cycles and number of instructions
// across all cores in the domain.
static DEFINE_PER_CPU(u64, measured_cycles);
static DEFINE_PER_CPU(u64, measured_insts);
static DEFINE_PER_CPU(u64, measured_l2refill);

// We want to keep track of the time that the counters were running. Only do so
// for the instruction counter for now.
static DEFINE_PER_CPU(u64, insts_old_time_enabled);

// Performance counter specific things:
static DEFINE_PER_CPU(struct perf_event *, cycle_ev);
static DEFINE_PER_CPU(struct perf_event *, l2refill_ev);
static DEFINE_PER_CPU(struct perf_event *, inst_ev);

static struct perf_event_attr cycle_hw_attr = {
    .type       = PERF_TYPE_HARDWARE,
    // .config    = 0x11,
    .config     = PERF_COUNT_HW_CPU_CYCLES,
    .size       = sizeof(struct perf_event_attr),
    .pinned     = 1,
    .disabled   = 1,
};

static struct perf_event_attr inst_hw_attr = {
    .type       = PERF_TYPE_HARDWARE,
    // .config    = 0x08,
    .config     = PERF_COUNT_HW_INSTRUCTIONS,
    .size       = sizeof(struct perf_event_attr),
    .pinned     = 1,
    .disabled   = 1,
};

static struct perf_event_attr l2refill_hw_attr = {
    .type       = PERF_TYPE_RAW,
    // .config     = 0x52,
    .config     = ARMV7_A15_PERFCTR_L2_CACHE_REFILL_READ,
    .size       = sizeof(struct perf_event_attr),
    .pinned     = 1,
    .disabled   = 1,
};

void enable_pmu_counters()
{
    volatile unsigned int v;
    READ_REG(PMCR, v);
    v |= 0x01;
    WRITE_REG(PMCR, v);
}

/* Enable the performance counters for each CPU included in the current policy */
void gov_enable(int cpu)
{
    struct sb_cpu_dbs_info_s *dbs_info = &per_cpu(sb_cpu_dbs_info, cpu);
    struct cpufreq_policy *policy = dbs_info->cdbs.cur_policy;
    struct cpumask cpu_mask;

    for_each_cpu(cpu, &cpu_mask) {
        printk("CPU: %d\n", cpu);
        printk("Enabling the performance counters\n");

        // Enable the performance counters
        enable_pmu_cycles(cpu);
        enable_pmu_inst(cpu);
        enable_pmu_l2refill(cpu);

        printk("Setting initial ipc target\n");
        per_cpu(ipc_threshold, cpu) = 40;

        printk("Setting initial MRPC multiplier\n");
        per_cpu(ipc_threshold, cpu) = 5;

        // Reset the counters
        printk("Resetting internal counters\n");
        per_cpu(measured_cycles, cpu) = 0;
        per_cpu(measured_insts, cpu) = 0;
        per_cpu(measured_l2refill, cpu) = 0;

        per_cpu(insts_old_time_enabled, cpu) = 0;

        per_cpu(counters_enabled, cpu) = 1;
    }
}

/*
 * Every sampling_rate, we look at the performance counters to determine
 * what the next frequency should be.
 */
```

```
static void sb_check_cpu(int cpu, unsigned int load)
{
    struct sb_cpu_dbs_info_s *dbs_info = &per_cpu(sb_cpu_dbs_info, cpu);
    struct cpufreq_policy *policy = dbs_info->cdbs.cur_policy;
    u64 cycles, inst;
    u64 enabled = 0, running = 0;
    struct perf_event *cycle_event, *inst_event, *l2refill_event;
    int cpu_new;
    u64 overall_cycles = 0, overall_insts = 0, overall_l2refills = 0;;
    long long int freq_next;
    unsigned int cur_freq;
    int error, delta;
    u64 time_enabled;
    long long int ipc;
    u64 overall_time = 0;
    unsigned num_cores = 0;

    struct cpumask cpu_mask;

    if (per_cpu(counters_enabled, cpu) == 0) {
        printk("Enabling PMU counters!\n");
        gov_enable(cpu);
        return;
    }

    if (have_governor_per_policy()) {
        cpumask_copy(&cpu_mask, policy->cpus);
    }
    else {
        cpumask_copy(&cpu_mask, cpu_online_mask);
    }

    for_each_cpu(cpu_new, &cpu_mask) {
        num_cores += 1;

        cycle_event = per_cpu(cycle_ev, cpu_new);
        cycles = perf_event_read_value(cycle_event, &enabled, &running);

        enabled = 0;
        running = 0;

        inst_event = per_cpu(inst_ev, cpu_new);
        inst = perf_event_read_value(inst_event, &enabled, &running);

        enabled = 0;
        running = 0;

        l2refill_event = per_cpu(l2refill_ev, cpu_new);
        l2refill = perf_event_read_value(l2refill_event, &enabled, &running);

        // Get the time that we have been counting for and update the reference.
        time_enabled = enabled - per_cpu(insts_old_time_enabled, cpu_new);
        overall_time += time_enabled;

        per_cpu(insts_old_time_enabled, cpu_new) = enabled;

        overall_cycles += cycles;
        overall_insts += inst;
        overall_l2refills += l2refill;
        per_cpu(measured_cycles, cpu_new) += cycles;
        per_cpu(measured_insts, cpu_new) += inst;
        per_cpu(measured_l2refill, cpu_new) += l2refill;

        // We need to reset the performance counters:
        local64_set(&cycle_event->count, 0);
        local64_set(&inst_event->count, 0);
        local64_set(&l2refill_event->count, 0);
    }

    // Calculate the IPC
    ipc = overall_insts;
    if (overall_cycles >> 7 != 0) {
        do_div(ipc, overall_cycles >> 7); // IPC * 128
    } else {
        ipc = 0;
    }

    // Calculate the MRPC
    mrpc = overall_l2refills;
    if (overall_cycles >> 12 != 0) {
        do_div(mrpc, overall_cycles >> 12); // MPC * 4096
    } else {
        mrpc = 0;
    }

    // Get the current frequency
    cur_freq = policy->cur;

    // Look up the current IPC threshold for the CPU we are running on
    ipc_threshold = per_cpu(ipc_threshold, cpu);

    // Compare IPC to IPC target, and set delta. Switch more rapidly
    // for larger delta.
    if (ipc < ipc_threshold) {
        delta = -100005;
    } else if (ipc > (ipc_threshold + 32)) {
        delta = 300005;
    } else if (ipc > (ipc_threshold + 16)) {
        delta = 200005;
    } else {
        delta = 100005;
    }

    // Deal with memory accesses
    delta -= mpc * 1000 * per_cpu(mpc_mult, cpu);

    freq_next = cur_freq + delta;
    if (freq_next < policy->min) {
        freq_next = policy->min;
    } else if (freq_next > policy->max) {
        freq_next = policy->max;
    }

    if (freq_next <= cur_freq) {
        // Switching down.
        __cpufreq_driver_target(policy, freq_next, CPUFREQ_RELATION_L);
    } else {
        // Switching up.
        __cpufreq_driver_target(policy, freq_next, CPUFREQ_RELATION_H);
    }

}
```

References

Alia, M., Wold Eide, V. S., Paspallis, N., Eliassen, F., Hallsteinsen, S. O., and Papadopoulos, G. a. (2007). A Utility-Based Adaptivity Model for Mobile Applications. *21st International Conference on Advanced Information Networking and Applications Workshops (AINAW'07)*, pages 556–563.

Allen, G., Nabrzyski, J., Seidel, E., Albada, G. D., Dongarra, J., and Sloot, P. M. A., editors (2009). *Computational Science ICCS 2009*, volume 5544 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg.

Antutu (2015). AnTuTu Benchmark. https://play.google.com/store/apps/details?id=com.antutu.ABenchMark&hl=en.

Apteker, R. T., Fisher, J. A., Kisimov, V. S., and Neishlos, H. (1995). Video acceptability and frame rate. *IEEE Multimedia*, 2(3):32–40.

ARM (2014). Fast Models.

ARM (2015a). Workload Automation. https://github.com/ARM-software/workload-automation.

ARM (2015b). Workload Automation: Memcopy. https://github.com/ARM-software/workload-automation/blob/a747ec7e4c2ea8a25bfc675f80042eb6600c7050/wlauto/workloads/memcpy/src/jni/memcopy.c.

Austin, T., Larson, E., and Ernst, D. (2002). SimpleScalar: an infrastructure for computer system modeling. *Computer*, 35(2):59–67.

Bakhoda, A., Yuan, G. L., Fung, W. W. L., Wong, H., and Aamodt, T. M. (2009). Analyzing CUDA workloads using a detailed GPU simulator. *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 163–174.

Binkert, N., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M. D., Wood, D. A., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., and Krishna, T. (2011). The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1.

Bischoff, S., Hansson, A., and Al-Hashimi, B. M. (2013). Applying of Quality of Experience to System Optimisation. In *PATMOS*, pages 91–98.

Brooks, P. and Hestnes, B. (2010). User measures of quality of experience: why being objective and quantitative is important. *IEEE Network*, 24(2):8–13.

Butko, A., Garibotti, R., Ost, L., and Sassatelli, G. (2012). Accuracy Evaluation of GEM5 Simulator System. In *7th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pages 1–7.

Carroll, A. and Heiser, G. (2010). An Analysis of Power Consumption in a Smartphone. In *USENIX ATC*, pages 271–284.

Choi, K., Dantu, K., Cheng, W.-C., and Pedram, M. (2002). Frame-based dynamic voltage and frequency scaling for a MPEG decoder. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design - ICCAD '02*, pages 732–737, New York, New York, USA. ACM Press.

Choi, K., Soma, R., and Pedram, M. (2004). Fine-grained dynamic voltage and frequency scaling for precise energy and performance trade-off based on the ratio of off-chip access to on-chip computation times. In *DATE*, pages 4–9. IEEE Comput. Soc.

Collange, D., Costeux, J.-l., Labs, O., and Antipolis, S. (2008). Passive Estimation of Quality of Experience. *Journal of Universal Computer Science*, 14(5):625–641.

Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. (2002). A fast and elitist multi-objective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197.

Egilmez, B., Memik, G., Ogrenci-Memik, S., and Ergin, O. (2015). User-specific Skin Temperature-aware DVFS for Smartphones. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, pages 1217–1220, San Jose, CA, USA. EDA Consortium.

ETSI TR 102 274 (2010). Human Factors (HF); Quality of Experience (QoE) requirements for real-time communication services. pages 1–37.

Falaki, H., Mahajan, R., Kandula, S., Lymberopoulos, D., Govindan, R., and Estrin, D. (2010). Diversity in smartphone usage. *MobiSys*, page 179.

Feng, K. (2012). *Quality-of-Service for Network-on-Chip-based Smartphone/Tablet Systems-on-Chip*. PhD thesis, University of Toronto.

Fiedler, M., Hossfeld, T., and Tran-Gia, P. (2010). A Generic Quantitative Relationship between Quality of Experience and Quality of Service. *IEEE Network*, 24(2):36–41.

Fortin, F.-A., De Rainville, F.-M., Gardner, M.-A., Parizeau, M., and Gagné, C. (2012). {DEAP}: Evolutionary Algorithms Made Easy. *Journal of Machine Learning Research*, 13:2171–2175.

Futuremark (2012). Futuremark Releases 3DMarktextregisteredMobile ES 2.0. http://www.futuremark.com/pressreleases/50623/.

Ge, Y. and Qiu, Q. (2011). Dynamic thermal management for multimedia applications using machine learning. In *Proceedings of the 48th Design Automation Conference on - DAC '11*, page 95, New York, New York, USA. ACM Press.

Gem5.org (2015). Publications.

Geomerics (2015a). Elighten — Geomerics. http://www.geomerics.com/enlighten/.

Geomerics (2015b). Transporter — Geomerics. http://www.geomerics.com/portfolio-item/transporter/.

Ghinea, G. and Thomas, J. P. (2005). Quality of perception: user quality of service in multimedia presentations. *IEEE Transactions on Multimedia*, 7(4):786–789.

GLBenchmark (2012). GLBenchmark 2.1.

GPGPU-sim (2012). GPGPU-sim.

Gutierrez, A., Dreslinski, R. G., Wenisch, T. F., Mudge, T., Saidi, A., Emmons, C., and Paver, N. (2011). Full-system analysis and characterization of interactive smartphone applications. In *IISWC*, pages 81–90.

Hansson, A., Agarwal, N., Kolli, A., Udipi, A. N., and Wenisch, T. (2014). Simulating DRAM controllers for future system architecture exploration. In *ISPASS*, pages 1–10.

Hardkernel (2015a). hardkernel/linux. https://github.com/hardkernel/linux/tree/o-droidxu3-3.10.y-android.

Hardkernel (2015b). Odroid XU3. http://www.hardkernel.com/main/product-s/prdt_info.php?g_code=G140448267127.

Henning, J. (2000). SPEC CPU2000: measuring CPU performance in the New Millennium. *Computer*, 33(7):28–35.

Hirsch, D., Lluch-Lafuente, A., and Tuosto, E. (2006). A Logic for Application Level QoS. *Electronic Notes in Theoretical Computer Science*, 153(2):135–159.

Hong, S. and Kim, H. (2010). An integrated GPU power and performance model. *Proceedings of the 37th annual international symposium on Computer architecture - ISCA '10*, 38(3):280.

Hsu, C.-H. and Feng, W.-C. (2005). Effective dynamic voltage scaling through CPU-boundedness detection. *Power-Aware Computer Systems*, 3471(December).

Ibarrola, E., Liberal, F., Taboada, I., and Ortega, R. (2009). Web QoE Evaluation in Multi-agent Networks: Validation of ITU-T G.1030. *AAS*, pages 289–294.

ITU-T (2005). G.1030: Estimating end-to-end performance in IP networks for data applications.

Jagatheesan, A. and Li, Z. (2013). Application Defined Computing in smartphones and consumer electronics. *CCNC*, pages 857–858.

Jeong, M. K., Erez, M., Sudanthi, C., and Paver, N. (2012). A QoS-aware memory controller for dynamically balancing GPU and CPU bandwidth use in an MPSoC. *Proceedings of the 49th Annual Design Automation Conference on - DAC '12*, page 850.

Jung, H. and Pedram, M. (2010). Supervised Learning Based Power Management for Multicore Processors. *IEEE CASICS*, 29(9):1395–1408.

Kim, H., Agrawal, N., and Ungureanu, C. (2012). Revisiting storage for smartphones. *ACM Transactions on Storage*, 8(4):1–25.

Kishonti (2015). GFXBench. http://www.gfxbench.com.

Krause, M., van Hartskamp, M., and Aarts, E. (2008). A Quality Metric for Use with Frame-rate Based Bandwidth Adaptation Algorithms.

Kuipers, F., Kooij, R., Vleeschauwer, D. D., and Brunnström, K. (2010). Techniques for Measuring Quality of Experience. *Wired/Wireless Internet Communications, Lecture Notes in Computer Science*, 6074:216–227.

Levy, M. (2012). EEMBC Brings Order to the Chaos of Android Benchmarking. http://www.eembc.org/andebench/about.php.

Li, X., Yan, G., Han, Y., and Li, X. (2013). SmartCap : User Experience-Oriented Power Adaptation for Smartphone's Application Processor. In *DATE*, number 1.

Linaro (2015). linux-linaro-tracking. https://git.linaro.org/?p=kernel/linux-linaro-tracking.git.

Lopez, D., Gonzalez, F., Bellido, L., and Alonso, A. (2006). Adaptive multimedia streaming over IP based on customer oriented metrics. In *2006 International Symposium on Computer Networks*, pages 1–7. IEEE.

Magnusson, P. S., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Moestedt, A., and Werner, B. (2002). Simics: A full system simulation platform. *Computer*, 35(2):50–58.

Mu, M., Mauthe, A., and Garcia, F. (2008). A Utility-Based QoS Model for Emerging Multimedia Applications. *2008 The Second International Conference on Next Generation Mobile Applications, Services, and Technologies*, pages 521–528.

Nah, F. F.-H. (2004). A study on tolerable waiting time: how long are Web users willing to wait? *Behaviour & Information Technology*, 23(3):153–163.

Nielsen, J. (2009). Powers of 10: Time Scales in User Experience.

Nvidia, C. (2008). Programming guide.

Ofcom (2015). Facts & figures.

Page, T. (2013). Usability of text input interfaces in smartphones. *Journal of Design Research*, 11(1):39–56.

Pallipadi, V. and Starikovskiy, A. (2006). The ondemand governor: Past, present, and future. *Linux Symposium*, pages 215–230.

Pathania, A., Jiao, Q., Prakash, A., and Mitra, T. (2014). Integrated CPU-GPU Power Management for 3D Mobile Games. In *DAC*.

Pollin.de (2015). Odroid XU3. http://www.pollin.de/shop/dt/NzU2OTgxOTk-/Bausaetze_Module/Entwicklerboards/ODROID_XU3_Einplatinen_Computer_SAMSUNG_Exynos_5

RedLicense Labs (2012). RL Benchmark: SQLite. https://play.google. com/store/apps/details?id=com.redlicense.benchmark.sqlite.

Rightware (2012). Basemark ES 2.0. http://www.rightware.com/ benchmarking-software/product-catalog/.

Rix, A., Beerends, J., Hollier, M., and Hekstra, A. (2001a). Perceptual evaluation of speech quality (PESQ) - a new method for speech quality assessment of telephone networks and codecs. *IEEE ICASSP*, 2:749–752.

Rix, A. W., Beerends, J. G., Hollier, M. P., and Hekstra, A. P. (2001b). Perceptual evaluation of speech quality (PESQ) - a new method for speech quality assessment of telephone networks and codecs. *IEEE ICASSP*, 2:749–752.

Romdan, N. (2008). ARM Fast Models Virtual Platforms for Embedded Software Development. *Information Quarterly Magazine*, 7(4):33–37.

Rosenfeld, P., Cooper-Balis, E., and Jacob, B. (2011). DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Computer Architecture Letters*, 10(1):16–19.

Roto, V. (2006). *Web Browsing on Mobile Phones Characteristics of User Experience.*

Samsung (2012). Enjoy the Ultimate WQXGA Solution with Exynos 5 Dual. *White Paper.*

Schatz, R., Egger, S., and Platzer, A. (2011). Poor, Good Enough or Even Better? Bridging the Gap between Acceptability and QoE of Mobile Broadband Data Services. In *ICC*, pages 1–6. IEEE.

Schmitz, M., Al-Hashimi, B., and Eles, P. (2005). Cosynthesis of energy-efficient multimode embedded systems with consideration of mode-execution probabilities. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(2):153–169.

Shafik, R., Yang, S., Das, A., Maeda-Nunez, L., Merrett, G., and Al-Hashimi, B. (2015). Learning Transfer-based Adaptive Energy Minimization in Embedded Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1.

Shen, H., Lu, J., and Qiu, Q. (2012). Learning based DVFS for simultaneous temperature, performance and energy management. *ISQED*, pages 747–754.

Sherwood, T., Perelman, E., Hamerly, G., and Calder, B. (2002). Automatically characterizing large scale program behavior. In *Tenth international conference on architectural support for programming languages and operating systems on Proceedings of the 10th international conference on architectural support for programming languages and operating systems (ASPLOS-X) - ASPLOS '02*, page 45, New York, New York, USA. ACM Press.

Shrestha, S. (2007). Mobile web browsing. In *Proceedings of the 4th international conference on mobile technology, applications, and systems and the 1st international symposium on Computer human interaction in mobile technology - Mobility '07*, volume 07, page 187, New York, New York, USA. ACM Press.

Shye, A., Scholbrock, B., and Memik, G. (2009). Into the Wild: Studying Real User Activity Patterns to Guide Power Optimizations for Mobile Architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture - Micro-42*, page 168, New York, New York, USA. ACM Press.

Soldani, D. (2010). Bridging QoE and QoS for Mobile Broadband Networks.

Spiliopoulos, V., Bagdia, A., Hansson, A., Aldworth, P., and Kaxiras, S. (2013). Introducing DVFS-Management in a Full-System Simulator. In *MASCOTS*.

Spiliopoulos, V., Kaxiras, S., and Keramidas, G. (2011). Green governors: A framework for Continuously Adaptive DVFS. *IGCCW*, pages 1–8.

Stevens, A. (2010). QoS for High-Performance and Power-Efficient HD Multimedia Systems.

Stone, J. E., Gohara, D., and Shi, G. (2010). OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science & Engineering*, 12(3):66–73.

Sunwoo, D., Wang, W., Ghosh, M., Sudanthi, C., Blake, G., Emmons, C. D., and Paver, N. (2013). A Structured Approach to the Simulation, Analysis and Characterization of Smartphone Applications. In *IISWC*.

The Linux Kernel Archives (2014). The Linux Kernel Archives.

Unity (2015). Unity - The Chase. http://unity3d.com/pages/the-chase.

Walker, M. J., Das, A. K., Merrett, G. V., and Al-hashimi, B. M. (2015). Run-time Power Estimation for Mobile and Embedded Asymmetric Multi-Core CPUs. *HIPEAC Workshop on Energy Efficiency with Heterogenous Computing, Amsterdam, NL, 19 - 21 Jan 2015.*

Wang, G. (2011). Memory Access Characterization of Scientific Applications on GPU and Its Implication on Low Power Optimization. *2011 International Conference on Computational and Information Sciences*, pages 47–52.

Wang, P.-H., Yang, C.-L., Chen, Y.-M., and Cheng, Y.-J. (2011). Power gating strategies on GPUs. *ACM Transactions on Architecture and Code Optimization*, 8(3):1–25.

Wang, R., Chen, L., and Pinkston, T. M. (2013). An Analytical Performance Model for Partitioning Off-Chip Memory Bandwidth. *IEEE International Parallel & Distributed Processing Symposium.*

Weicker, R. P. (1989). Dhrystone benchmark (Ada version 2): rationale and measurements rules. *ACM SIGAda Ada Letters*, IX(5):60–62.

Yourst, M. T. (2007). PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In *2007 IEEE International Symposium on Performance Analysis of Systems & Software*, pages 23–34. IEEE.

Yuan, W. and Nahrstedt, K. (2003). Energy-efficient soft real-time CPU scheduling for mobile multimedia systems. In *SOSP*, page 149, New York, New York, USA. ACM Press.

Zakharenko, V. (2012). *FusionSim: Characterizing the Performance Benefits of Fused CPU/GPU Systems.* PhD thesis.

Zhu, Y. and Reddi, V. J. (2013). High-performance and energy-efficient mobile web browsing on big/little systems. *HPCA*, pages 13–24.

Zinner, T., Hohlfeld, O., Abboud, O., and Hossfeld, T. (2010). Impact of frame rate and resolution on objective QoE metrics. In *QoMEX*, number June, pages 29–34. IEEE.