It is too hot in here! A performance, energy and heat aware scheduler for Asymmetric multiprocessing processors in embedded systems.

Willy Michel Henri Wolff, BSc, MSc

School of Computing and Communications

Lancaster University

A thesis submitted for the degree of

Doctor of Philosophy

April, 2023

It is too hot in here!

A performance, energy and heat aware scheduler for Asymmetric multiprocessing processors in embedded systems.

Willy Michel Henri Wolff, BSc, MSc.

School of Computing and Communications, Lancaster University

A thesis submitted for the degree of *Doctor of Philosophy.* April, 2023

Abstract

Modern architecture present in self-power devices such as mobiles or tablet computers proposes the use of asymmetric processors that allow either energy-efficient or performant computation on the same SoC.

For energy efficiency and performance consideration, the asymmetry resides in differences in CPU micro-architecture design and results in diverging raw computing capability. Other components such as the processor memory subsystem also show differences resulting in different memory transaction timing. Moreover, based on a bus-snoop protocol, cache coherency between processors comes with a peculiarity in memory latency depending on the processors operating frequencies.

All these differences come with challenging decisions on both application schedulability and processor operating frequencies. In addition, because of the small form factor of such embedded systems, these devices generally cannot afford active cooling systems. Therefore thermal mitigation relies on dynamic software solutions.

Current operating systems for embedded systems such as Linux or Android do not consider all these particularities. As such, they often fail to satisfy user expectations of a powerful device with long battery life.

To remedy this situation, this thesis proposes a unified approach to deliver high-performance and energy-efficiency computation in each of its flavours, considering the memory subsystem and all computation units available in the system. Performance is maximized even when the device is under heavy thermal constraints. The proposed unified solution is based on accurate models targeting both performance and thermal behaviour and resides at the operating systems kernel level to manage all running applications in a global manner.

Particularly, the performance model considers both the computation part and also the memory subsystem of symmetric or asymmetric processors present in embedded devices. The thermal model relies on the accurate physical thermal properties of the device. Using these models, application schedulability and processor frequency scaling decisions to either maximize performance or energy efficiency within a thermal budget are extensively studied.

To cover a large range of application behaviour, both models are built and designed using a generative workload that considers fine-grain details of the underlying microarchitecture of the SoC. Therefore, this approach can be derived and applied to multiple devices with little effort.

Extended evaluation on real-world benchmarks for high performance and general computing, as well as common applications targeting the mobile and tablet market, show the accuracy and completeness of models used in this unified approach to deliver high performance and energy efficiency under high thermal constraints for embedded devices.

Declaration

I declare that the work presented in this thesis is, to the best of my knowledge and belief, original and my own work. The material has not been submitted, either in whole or in part, for a degree at this, or any other university.

Willy Michel Henri Wolff

April, 2023

Acknowledgements

This PhD would not have been possible without the funding from Lancaster University by an EPSRC studentship and an industrial CASE studentship with ARM.

First, I want to thank my supervisors: Dr. Barry Porter, Dr. Zheng Wang and Geraint North; and the jury: Pr. Joe Finney and Pr. Steve Furber. Many thanks for your crucial feedback.

I would like to thank Pr. Philippe Clauss and Pr. Cédric Bastoul. Without you, I would never have embarked on this path.

Special thanks to Arm folks in Cambridge, Austin and Manchester that helped me to tame the beast I was working on. Especially to David Brooke and Christopher Tory, I will always be grateful to have been able to snoop your knowledge out-of-order. You helped me to put my work in-order. Thank you guys, I will continue to speculate before retiring.

Thanks to the CE-OSS-kernel-power team in Arm for the long discussions about scheduling and power.

Thanks to Pierre David, Alain Ketterlin and people from ICPS at Université de Strasbourg that helped me throughout my studies.

I want to thank all my colleagues in SCC. Particularly Ben and Andrew with whom I had the pleasure of sharing an office. And Helena, Alex and Pierre with whom I shared TA and some other nice discussions about PhD life.

Thank you to Peter Garraghan, Bran Knowles, John Vidler and Andrew Scott for helping me to tackle the issues from a different perspective.

Many thanks to my friends near and far who have supported me through these years, Adilla, Juan, Déborah, Tidiane, Flow, Gilles, Kevin, Lucie, Pauline, Romain, Yann, Fab, Julien, Didier, Adam, Maemi, Fan, Dji, Mat, PK and Josh.

Thank you Seb, it was short but intense. I truly miss you.

Thanks to my family that supported me throughout my studies.

Leaving the most important until last, my wife Aude. Your unwavering love

during these long long nights and all these years is priceless.

List of Publications

Contributing Publications

Wolff, W., Porter, B., "Performance Optimization on big.LITTLE Architectures: A Memory-latency Aware Approach". In: *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems.* New York, NY, USA: ACM, June 2020, pp. 51–61. ISBN: 9781450370943. DOI: `10.1145/3372799.3394370`

Wolff, W., Porter, B., *What am I waiting for? Energy and Performance Optimization on big.LITTLE Architectures: A Memory-latency Aware Approach.* Dec. 2020

Additional Publications

Taylor, B., Marco, V. S., **Wolff, W.**, Elkhatib, Y., Wang, Z., "Adaptive deep learning model selection on embedded systems". In: *Proceedings of the 19th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems.* New York, NY, USA: ACM, June 2018, pp. 31–43. ISBN: 9781450358033. DOI: `10.1145/3211332.3211336`

Ye, G., Tang, Z., Fang, D., Chen, X., **Wolff, W.**, Aviv, A. J., Wang, Z., "A Video-based Attack for Android Pattern Lock". In: *ACM Transactions on Privacy and Security* 21.4 (Nov. 2018), pp. 1–31. ISSN: 2471-2566. DOI: `10.1145/3230740`

Linux Kernel and Other Project Contributions

Szyprowski, M., Kozlowski, K., **Wolff, W.,** *ARM: dts: exynos: Disable frequency scaling for FSYS bus on Odroid XU3 family.* 2020. URL: `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=9ff416cf45a08f28167b75045222c762a0347930`

Luba, L., Choi, C., **Wolff, W.**, Kozlowski, K., *memory: samsung: exynos5422-dmc: Add module param to control IRQ mode.* 2020. URL: `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=4fc9a0470d2dc370289e9d883feb41e5dd2c6303`

Luba, L., Choi, C., **Wolff, W.**, Kozlowski, K., *memory: samsung: exynos5422-dmc: Adjust polling interval and uptreshold.* 2020. URL: `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=74ca9e46107879551e2625bfbfbfd71da1881b82`

Wolff, W., Lezcano, D., Kumar, V., *thermal/drivers/cpufreq_cooling: Fix return of cpufreq_set_cur_state.* 2020. URL: `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=ff44f672d74178b3be19d41a169b98b3e391d4ce`

Mihailescu, M., **Wolff, W.**, Kozlowski, K., Szyprowski, M., *ARM: dts: exynos: Add CPU perf counters to Exynos54xx boards.* 2017. URL: `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=c4f2fc00defc65950dfabce7a4c70cd2a289111d`

Wolff, W., Kozlowski, K., Zolnierkiewicz, B., *ARM: dts: exynos: fix incomplete Odroid-XU3/4 thermal-zones definition.* 2017. URL: `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=e740731dae9470f7fb86efa643ec881a66d4e4c0`

Wolff, W., Rui, Z., *thermal: fix source code documentation for parameters.* 2017. URL: `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=0d76d6e1eede5f2aa13695cb4c9d763bb3555e3e`

Wolff, W. *ARM: dts: exynos: add CCI-400 PMU nodes support to Exynos542x SoCs.* Under review. 2019. URL: `https://lore.kernel.org/patchwork/patch/1061141/`

Wolff, W., Lowe-Power, J., Travaglini, G., Nikoleris, N., kokoro, *gem5: config, arm: memoryMode test.* 2019. URL: `https://gem5.googlesource.com/public/gem5/+/ea088f5150d03d4481555ecbbfa2afba3a87468a`

Szyprowski, M., **Wolff, W.**, Kozlowski, K., *ARM: dts: exynos: Disable frequency scaling for FSYS bus on Odroid XU3 family.* 2020. URL: `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=9ff416cf45a08f28167b75045222c762a0347930`

Contents

List of Figures

List of Tables

List of Abbreviations

ALU	Arithmetic Logic Unit
AMP	Asymmetric MultiProcessing
ARMA	AutoRegressive Moving Average
CAS	Capacity-Aware Scheduler
CCI	Cache Coherent Interconnect
CFS	Completely Fair Scheduler
CISC	Complex Instruction Set Computer
CMOS	Complementary Metal-Oxide-Semiconductor
CPI	Cycle Per Instruction
CPU	Central Processing Unit
DLP	Data-Level Parallelism
DRAM	Dynamic Random-Access Memory
DSM	Distributed Shared Memory
DSP	Digital Signal Processor
DVFS	Dynamic Voltage and Frequency Scaling

EAS	Energy-Aware Scheduler
GPU	Graphics Processing Unit
HMP	Heterogeneous MultiProcessing
ILP	Instruction-Level Parallelism
IoT	Internet of Things
IPA	Intelligent Power Allocation
IPC	Instruction Per Cycle
ISA	Instruction Set Architecture
LLVM	Low Level Virtual Machine
MIMD	Multiple Instruction, Multiple Data
MLP	Memory-Level Parallelism
MOSFET	Metal-Oxide-Semiconductor Field-Effect Transistor
MPKI	Misses Per Kilo Instructions
MRPI	Memory Reads Per Instruction
NUMA	Non-Uniform Memory Access
OPP	Operating Performance Point
PE	Processing Element
PID	Proportional Integral Derivative
PIE	Performance Impact Estimation

PMC	Performance Monitoring Counter
PMU	Performance Monitoring Unit
QoS	Quality of Service
RAM	Random-Access Memory
RISC	Reduced Instruction Set Computer
RMSE	Root Mean Square Error
SF	Speedup Factor
SIMD	Single Instruction, Multiple Data
SMP	Symmetric MultiProcessing
SoC	System-on-Chip
SRAM	Static Random-Access Memory
TBASS	Thermal-Balance Aware System Scheduler
TDP	Thermal Design Power
TLP	Thread-Level Parallelism
UMA	Uniform Memory Access
WSS	Working Set Size

Chapter 1

Introduction

Power and thermal dissipation is the key technology limitation that has led to the introduction and proliferation of multicore and now many-core–processors. However, even many-core processors may not overcome the "power wall" [49, 23]. The introduction of asymmetric multicore multiprocessors brings the opportunity to deliver high performance for demanding applications while satisfying energy efficiency for lightweight applications [6, 71].

Nowadays, hardware to support both high performance and energy efficiency is well spread in the small form factor such as mobile and other small embedded systems. It started spreading to the laptop and desktop markets. However, the software to tame such architecture still lacks of maturity. This thesis explores different aspects of software scheduling and frequency scaling for such devices.

Multiple works have explored thoroughly scheduling strategies considering a precise knowledge of the workload to determine the benefit of using the high performance or energy-efficient processing elements of the device. For instance, in [91, 111], the authors use an offline analysis that discovers the workload phase and the performance and energy profiles. These works tend to find Pareto optimality situations for workload phases. The runtime then replays the workload phases depending on the user goal, performance and/or energy consumption. However, this strategy requires a long offline exploration of any single workload, and as such, does not scale.

In the industry, there exist two tremendous examples of scheduling strategies. Operating systems that equip devices from the Apple brand employ a recommendation from the developer to either use the performant cores for intensive tasks or energy-efficient cores for background tasks. However, this strategy could lead to over-provisioning resources to applications that do not have good use of performant cores due to their code construction. In the Linux and Android equipped device market, the EAS scheduler is an automated scheduling system that uses a performance metric derived offline. However, this performance metric is highly dependent on the workload used and fails to capture the performance of the full system.

On the other hand, when the device is under heavy thermal conditions, the scheduling strategies proposed above could be non-optimal. Generally, the strategy to reduce the temperature is to reduce activity intensity. This is performed by employing frequency scaling and activity migration. Following the scheduling strategies presented above, migration between performant cores and energy-efficient cores becomes critical and challenging to maximize performance under thermal pressure. The current state-of-the-art does not put forward a holistic approach of scheduling and frequency scaling considering application performance by taking into account low-level micro-architectural detail.

1.1 Goals and research questions

The work presented in this thesis attempt to bridge the gap between workload scheduling and frequency scaling strategies by considering the efficient use of the underlying hardware which takes into account low-level micro-architectural features and specificities of processing elements. To achieve this goal, this thesis is articulated around these research questions:

RQ 1 To what extent frequency scaling could improve the energy efficiency of a computing device?

RQ 2 To what extent frequency scaling affects the system in terms of performance on multiprocessors-equipped devices where data-coherency is implemented using a bus-snoop protocol?

RQ 3 How to schedule workloads for performance while minimizing energy consumption on systems that use asymmetric multicore multiprocessors?

RQ 4 How can thermal effects be effectively mitigated using software-based techniques, for small-form factor computing devices where active cooling (e.g. fan, liquid cooling) can not be employed?

Each of these research questions is explored using an empirical approach, in which novel software techniques are developed and then tested against common benchmark suites that represent different aspects of workload activity. The considered workload types encompass web-browsing activities using `BBench` and `Speedometer` benchmark suites and video decoding with `mplayer`. Other benchmarks from `Spec`, `PARSEC` and `Splash-3` benchmark suites were used to show the effectiveness of the approach against a wide range of compute- and memory-intensive workloads, as well as single and multithreaded workloads.

1.2 Contributions

One of the major novelties of this work is the fine detail exploration of the memory communication timing considering the data-coherency mechanism to interconnect the processing elements. This thesis makes the following contributions:

- **A characterization of memory latency of data-coherency mechanism based on a bus-snoop protocol considering frequency scaling of processing elements**:

  - design of a microbenchmark infrastructure and memory access sequencer and analysis system to discover and identify specific conditions when the data communication latency is sensitive to the perspective of an application
  - a simple model based on a decision tree to express the specific condition when snoop latency becomes sensitive to user experience
  - a runtime system that is effective in mitigating snoop latency without disturbing the system
  - the approach enables an increase in application performance of up to 40% and reduces energy consumption by up to 70% compared to the Linux default frequency scaling policy

- **A study on scheduling and frequency scaling strategies to optimize performance and energy efficiency under thermal pressure**:

  - provide a thorough study on the thermal imbalance of multi-core device
  - characterization of effective workload quality regarding instruction-level parallelism at runtime
  - an offline characterization of the memory requirement of the application
  - a runtime system as an application scheduler to guide use of the underlying hardware specificities at a microarchitectural level
  - a scheduling strategy that considers thermal characteristics of the device and workload benefits of using a cluster type
  - the approach enables an increase in application performance by 10% on average and reduces energy consumption by 12% on average compared to the Linux default scheduling strategy using a strict thermal policy

1.3 Thesis Overview

This thesis is organised as follows. Chapter 2 provides background information on topics related to this thesis. It discusses process technology trends and their impact on computer architecture. A constructive description of computer organization is presented including microarchitecture design principles and hardware support for parallel processing. The summary of this chapter presents the machine that is used in subsequent chapters. Chapter 3 presents relevant related work related to the scope of this thesis. Chapter 4 studies data-coherency mechanisms in multiprocessor architectures based on a bus-snoop protocol with frequency scaling capability. It discusses a non-linear memory communication latency relative to frequencies of processing elements present in the system. Chapter 5 presents a scheduling strategy that takes into consideration low-level microarchitectural aspects. In particular, this strategy is applied for thermal management on asymmetric multicore multiprocessor systems in a small form factor packaged computing device. Finally, chapter 6 sums up the contributions of this work and details some future research directions.

Chapter 2

Background

Over the past 70 years, progress in the manufacturing process of semiconductors has resulted in improvement of computing technology. Reduction in the size of semiconductors enables the increase in transistor density per area and, thanks to better architectural design, improves the performance of computing devices. In 1965, Gordon Moore predicted in his seminal paper that the number of transistors in a chip would double every year. This prediction was amended in 1975 to every two years. This trend is recognized as *Moore's law* [100, 101].

Unfortunately, technology scaling started stalling around 2005 due to the inherent physical properties of materials. Robert Dennard and his team observed in 1974 that power density stays constant as transistors get smaller [38]. By reducing transistor size, the supply voltage can also be reduced with a proportional current draw and enables increasing switching frequency of the transistor. This concept is referred to as *Dennard scaling* and, in correlation with Moore's law, enables increasing processor performance by improving the manufacturing process. However, at a very small size, the leakage current (and thus the increase in overall power consumption) becomes problematic and leads to a slowdown of further scaling [37, 22, 64].

With Moore's law still in effect while Dennard scaling reaches an end, it results in processors that may no longer switch every transistor at full frequency without exceeding the power or thermal limitations of the chip. This inability to utilize all of the chip's transistors at full frequency simultaneously is known as *Dark Silicon* [23, 49, 134]. One solution to the inability of scaling frequency of a single processor is to use multiple processors operating in parallel while keeping these processors at a safe frequency plateau. Figure 2.1 shows the trend of the past 50 years of microprocessor computing systems.

In the quest to create performant computers while meeting a sustainable power budget, a new strategy was proposed to design multicore processors utilizing different micro-architectures on the same chip. In the literature, this architecture is referred to as heterogeneous or asymmetric multiprocessing [77, 50, 55, 71, 96]. The rationale motivating this architecture is the consideration that some software may deliver similar performance whether running on a performant complex processor or on a power-efficient simpler processor. However, a new difficulty that arises using this architecture is software scheduling [79, 24]. This thesis explores some aspects regarding data-coherency and application scheduling towards power and thermal management of such architecture.

The remainder of this chapter will first establish the relation between power consumption and temperature in section 2.1. Section 2.2 describes the basics of processor architecture to understand common designs (i.e. in-order/out-of-order pipelining, superscalar) to deliver high computing performance but at high-power cost against a power-efficient simple design. In section 2.3, different aspects to bring multiprocessing capability to a processor are discussed and how it can be achieved considering both memory and data-coherency in the subsequent section 2.4.

Finally, section 2.5 summarizes this chapter and presents a computing device that implements a heterogeneous multi-core processor packaged in a small form factor targeting the embedded, tablet and smartphone market. The device is used throughout the thesis to explore some aspects regarding data-coherency and application scheduling towards power and thermal management.

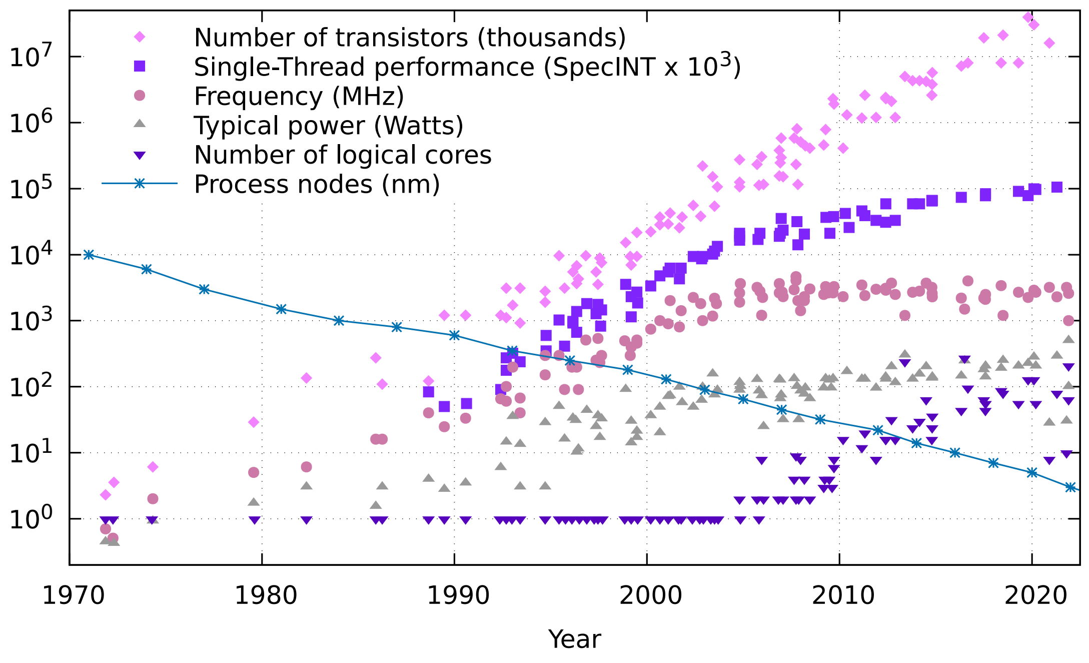


Figure 2.1: Microprocessor trends over the last 50 years. Even though signs of slowdown is appearing, the number of transistors have continued to scale as predicted by Moore's Law for over the past 50 years. Around 2005, power dissipation becomes problematic as Dennard scaling came to an end. This power constraint prevented increase in frequency, which in turn negatively affected CPU core performance. With the difficulty to increase clock frequency, CPU architecture if focusing on exploiting parellism by increasing the number of core on a single chip.

Data collection up to the year 2010 by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten. Data collection from 2010 onward by K. Rupp. Data available at `https://github.com/karlrupp/microprocessor-trend-data`

2.1 Power and temperature basics

This section establishes the relation of power consumption of a device and its translation to heat and temperature.

Power dissipation in a digital CMOS [1] circuit is given by [32, 121]:

$$\text{Power} = \text{Power}_{dynamic} + \text{Power}_{static}$$

This formula consists of two parts, a *dynamic* part that contributes the most to the power and is related to the power consumed by the device during its operation, and a *static* part that relates to *leakage* due to physical properties inherent of the material used to produce a transistor.

The dynamic power is in direct proportion to the transistors that are changing state from on to off (and vice versa) at each clock cycle and is given by:

$$\begin{aligned}\text{Power}_{dynamic} &= \text{Frequency} \times \text{Transistors switched} \times \text{Energy per Transistor}\\ &= \text{Frequency} \times \text{Transistors switched} \times \text{Capacitive load} \times \text{Voltage}^2\end{aligned}$$

The number of transistors switched denotes the activity of the device while executing a workload.

The static power is given by:

$$\text{Power}_{static} = \text{Current}_{leakage} \times \text{Voltage}$$

There are multiple sources of current leakage, with sub-threshold current, gate oxide tunneling current and band-to-band tunneling current being the most prominent. All these sources of leakage increase while transistors size shrink and are also affected by the temperature. This dissertation does not consider directly these notions in the methodology, the interested reader may refer to [95, 70, 25, 112, 1, 26] to get a deeper understanding of these different sources of current leakage.

[1]Complementary Metal-Oxide-Semiconductor (CMOS) is a type of Metal-Oxide-Semiconductor Field-Effect Transistor (MOSFET) which is the basic component in a microprocessor.

Finally, the steady-state temperature of the device can be determined by considering the ambient temperature where the device is operating, the power consumption of the device and the physical and thermal properties of the packaging of the device (as Thermal Resistance). It is given by:

$$\text{Silicon Temperature} = \text{Ambient Temperature} + \text{Thermal Resistance} \times \text{Power}$$

As the power may fluctuate with the workload activity, one may reduce the temperature by using a heatsink or other cooling device to reduce the thermal resistance of the system. The methodology of this thesis to lessen thermal effect considers reducing the power needed to execute a workload by reducing the number of transistors used by using a different processor design on an asymmetric chip, the operating voltage and the frequency of the device.

2.2 Processing Element Architecture

This thesis will study scheduling strategies for processors embedding different micro-architectures in a single chip. This section reviews the evolution of processor architectural design that enables high-performance workload execution.

A program written in a programming language describes what must be done by a machine. The source code of a program is compiled targeting a particular computer architecture, also termed an Instruction Set Architecture (ISA).

The ISA acts as an interface between the software and the hardware, defining available instructions, types and sizes of operands, how the memory is addressed, available registers, etc.

Computational organization or microarchitecture is an implementation of a computer architecture. For example, the Arm Cortex-A7 and Arm Cortex-A15 processors are instances of the ARMv7-A instruction set architecture with distinct organization, favouring different aspects of computing. The A7 favours efficiency and low-energy computing while the A15 targets performance and fast program execution. Both of these microarchitectures can execute the same compiled program as they both implement the same ISA.

For general-purpose processors, two types of ISA are predominant and commonly used, Complex Instruction Set Computer (CISC) where instructions are complex and optimized for particular operations; and Reduced Instruction Set Computer (RISC) where instructions are simple. This statement is an oversimplification of the two concepts, as, over the years, RISC architectures gain fairly complex arithmetic operations such as the *square root* operator. However, there are two main features in a RISC ISA: 1) the instruction format is of a fixed length, with bits of an instruction identifying operation code (opcode) and operand registers, immediate or address; all these fields are positioned at the same place in the instruction encoding. In contrast, CISC encodes instructions in a format of variable length. This helps reduce code density when an instruction uses a variable number of operands. For instance, a `NOP` instruction in ARM A32 RISC ISA is encoded taking 4-bytes, (with opcode `0x00000000`), while on x86-32 CISC ISA, a `NOP` instruction takes only 1-byte (with opcode `0x90`) as this instruction does not have any parameter. 2) RISC architecture employs a *load/store* model where the only instructions that can access the memory are the specific load and store instructions. Data is first loaded to a register before being processed and then stored back to memory. In a CISC ISA, instruction can refer to a memory address to make an operation.

The remainder of this section reviews some important microarchitecture features that are commonly used in processor designs to deliver high computing performance.

Simply put, a program is a sequence of instructions that a processor executes one after another in the order specified by the programmer. The life of an instruction can be dissected into multiple phases or stages, each stage taking 1 clock cycle to finish: 1) the instruction is first fetched from memory (IF), 2) this instruction is then decoded and operands are retrieved (ID), 3) the instruction is executed (EX). 4) If the instruction is a memory operation, the memory access is then performed using the address computed in the last stage (MEM). 5) Finally, the result of the

	Clock number								
Instruction number	1	2	3	4	5	6	7	8	9
Instruction i	IF	ID	EX	MEM	WB				
Instruction $i + 1$		IF	ID	EX	MEM	WB			
Instruction $i + 2$			IF	ID	EX	MEM	WB		
Instruction $i + 3$				IF	ID	EX	MEM	WB	
Instruction $i + 4$					IF	ID	EX	MEM	WB

Figure 2.2: Execution flow on a simple 5-stage pipeline for a RISC processor.

instruction is written back to the register (WB).

One can see that each of these stages is dependent within the instruction but may be independent between consecutive instructions. An essential feature of any modern microarchitecture is to perform these different phases in parallel when possible. When there are no dependencies on data or register naming it is possible to deliver instruction execution at every clock cycle. This technique is called pipelining and Figure 2.2 shows such an execution pattern. Compared to a non-pipelined processor, this processor could execute 5 times faster. On a CISC architecture, complex instructions are generally decoded and dissected into multiple micro-operations at the ID stage. In this case, later stages may take more clock cycles in the pipeline.

An important bottleneck in performance comes from long latency instructions that stop the execution flow. For instance, in the presence of a multi-cycle operation such as a *divide* instruction that may take many cycles to execute and produce a result, no further instruction can be issued to the execution stage with an *in-order* pipeline. Another example is a memory load operation, if a data access misses in the cache, accessing a higher level of the cache hierarchy or off-chip memory may take several hundred cycles until the data is fetched in a register and execution can progress further.

When two instructions are independent, for instance an *addition* that is independent from another *addition* or a *divide* operation, an interesting approach

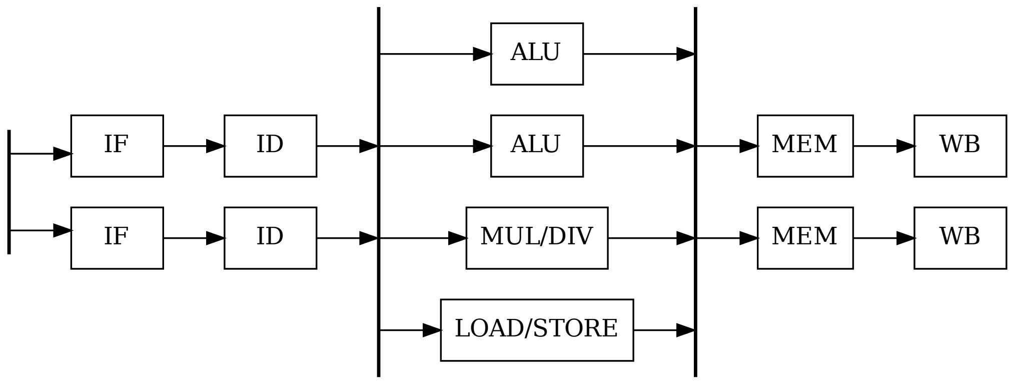


Figure 2.3: An example of a 2-way superscalar processor with 4 execution units.

would be to execute them in parallel. One solution is to multiply the number of *pipeline stream stages*. In the literature, this notion is termed as *superscalar*. Figure 2.3 shows an example of a 2-way superscalar pipeline. In this design, there also exists 2 Arithmetic Logic Unit (ALU), 1 multiply/divide unit and 1 load/store unit. For this last unit, the load/store EXE stage computes the address of the memory location, while the MEM stage performs the actual access. This processor could execute 2 addition instructions during the same cycle (i.e. 2 instructions per cycle or IPC). In this example of superscalar microarchitecture, IF, ID, MEM and WB have been duplicated, though it is not a requirement. In fact, IF and ID could easily sustain a 1 cycle phase, EXE and MEM stages could take multiple cycles to access memory on a load/store instruction, or on a floating-point instruction. Thus, it is often interesting to incorporate only one IF, ID half-pipeline stream and dispatch execution to the right EXE/MEM unit to continue the life of the instruction. Pipelining and superscalar improve execution performances by exploiting Instruction-Level Parallelism (ILP).

Continuing further on the path to exploit ILP, it is often possible to continue instruction execution in parallel with a memory operation. However, in the pipelined superscalar design presented above, though multiple independent instructions can be executed in parallel, the execution of those instructions remains in-order. Thus, when one instruction is taking a long time to execute, the pipeline will stall and performance degrades. One step further to improve performance is to allow *out-of-order execution* of independent instructions, but keeping the instruction retirement of the pipeline in order to maintain the semantics of the program. Over the past years, multiple strategies have been proposed and implemented with great success to allow out-of-order execution such as *scoreboarding* or *Tomasulo algorithm* with the use of *register renaming*, *reservation station* and *reorder-buffer*.

Other performance enhancing features such as *branch-prediction*, *speculative execution*, *data prefetching* exist. The interested reader may refer to any book in Computer architecture for deeper understanding of these concepts [66, 124, 61].

In general, to implement these hardware features to improve the computing performance of the processor, the length of the pipeline is increased as well. In the same manner, each additional feature increases the transistor count and thus the overall power consumption of the processor. Because of the difficulty to dissipate this power due to physical limitations, it becomes harder to improve the performance of a single Processing Element (PE). To cope with that limitation, improvement in hardware design is pushing towards the exploitation of parallelism by adding multiple processing elements that operate simultaneously. The next section introduces different techniques to implement multiprocessing capability.

2.3 Multiprocessing

When multiple instructions do not have dependency between them, a superscalar microarchitecture allows them to execute simultaneously, as long as there are available execution units. When the software contains large portions of code that are independent, a single superscalar processor unit may not be enough to exploit the full performance potential of the software. However, the software can be written in such a way as to expose a *thread* of execution and use processors capable of executing several software threads simultaneously to exploit Thread-Level Parallelism (TLP). This architecture is referred to as Multiple Instruction, Multiple Data (MIMD) in Flynn's taxonomy[53].

There exist various designs to achieve this goal, such as multi-core where the core part within a processor is multiplied, or multiprocessor where multiple full processors are present and interconnected. Another approach is hardware multithreading. In multithreading, the computational part of the pipeline is shared amongst software threads, but the hardware that holds the context of a particular software thread is duplicated (i.e. register files, program counter and other registers to control the logical core).

Another form of multiprocessing is to apply an identical treatment on multiple data at once, such as vector instructions. For instance, applying a filter to an image to transform each pixel of an image independently could be treated by vector instructions. Vector instruction exploits Data-Level Parallelism (DLP) and is referred as Single Instruction, Multiple Data (SIMD). Multiple SIMD instruction set extensions are present in the market, such as NEON and SVE for the Arm ISA[7, 11, 129] or SSE and AVX for the x86 ISA[67] among others.

When the workload is well-known and highly specialized, a specific hardware architecture targeting that workload can be used. For instance, this is the case for graphical computation or highly vectorizable workload with the use of Graphics Processing Unit (GPU), or Digital Signal Processor (DSP) where the memory architecture and pipeline are optimized to treat signal based algorithms for audio, speech processing and telecommunication among others. In this case, these different processing units implement a different ISA than the CPU to fulfill their duty.

Modern systems may combine many of the above multiprocessing techniques to achieve high performance. From the server to the Internet of Things (IoT) market, all the spectra of computing systems implement multiprocessing in their various forms.

A major problem of using multi-processing and interconnecting processing units

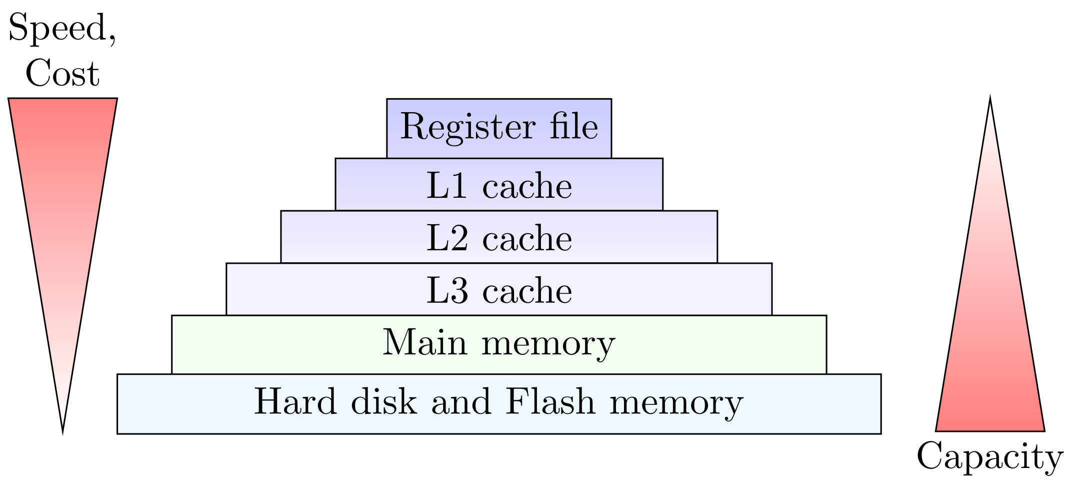


Figure 2.4: Typical memory hierarchy. The faster the memory, the higher the cost.

where each of them have their own internal memory is to ensure that all these elements have the same “view” of the shared memory content, the data must be coherent between all elements. The next section elaborates on this topic.

2.4 Memory Architecture

In the last section, different microarchitecture organizations related to computation were introduced without considering any memory-related aspect. This section elaborates on the matter. The first part of this section describes different memory hierarchies, while the second part discusses cache-coherency problems when there exists private and shared centralized memory space.

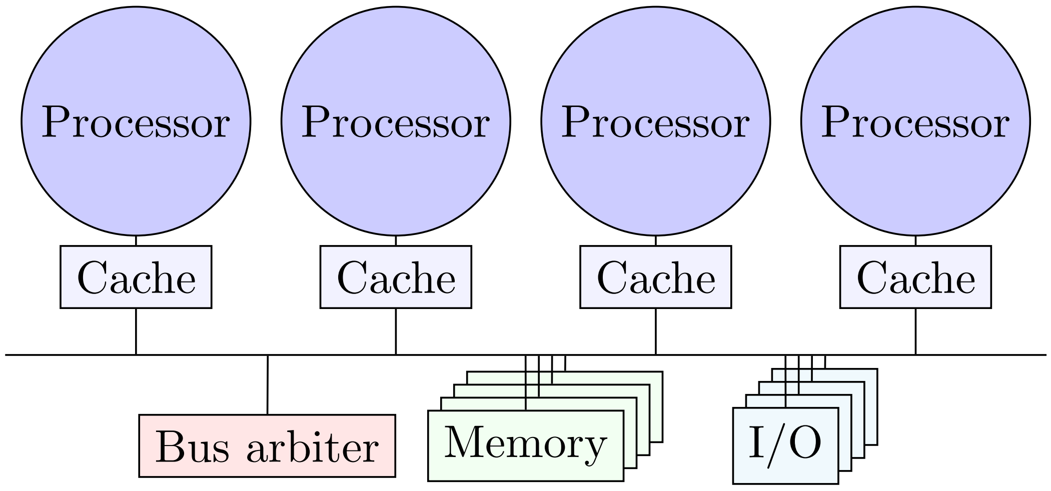
2.4.1 Memory hierarchy

In order to operate, data must be provided to execution units in a processing element. An important fundamental observation of program execution is the principle of locality: programs tend to reuse data and instructions they have used recently[39, 40, 41]. As such, the memory is organized in a hierarchy. This section describes some organizations of common memory hierarchies employed in computing system designs. Figure 2.4 shows a typical memory hierarchy.

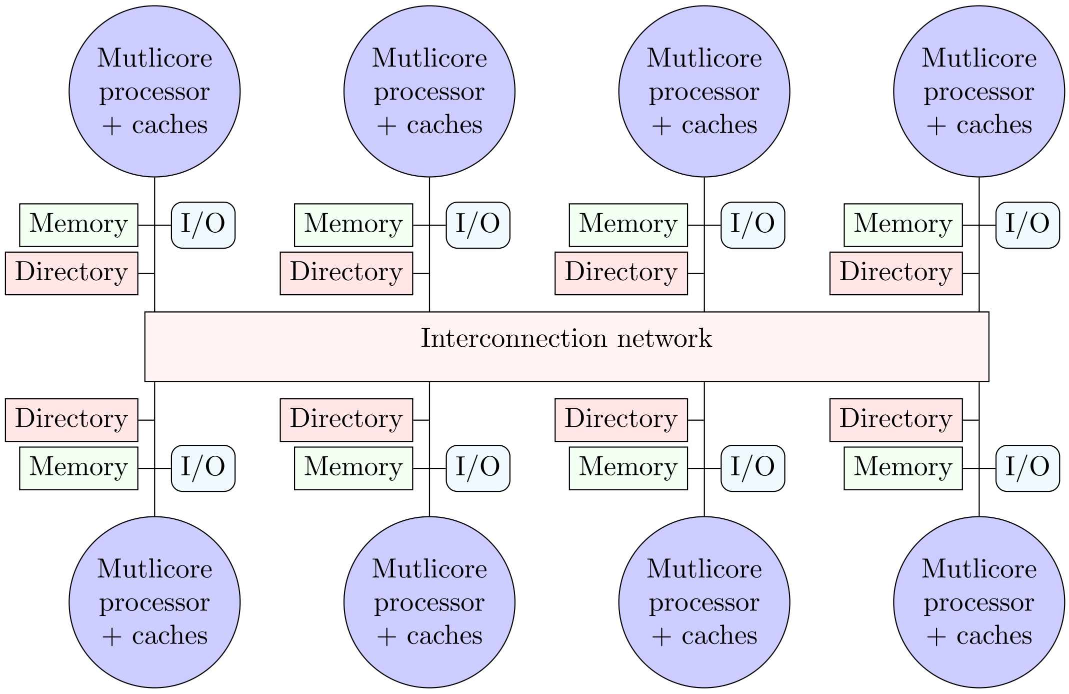
At the closest level to execution units, most instructions of a processing element work on data stored in a *register*. In a RISC architecture, computation is performed exclusively using registers, with specific *load from* and *store to* instructions targeting feeding and backing up registers from and to persistent storage for later use. In general, CPU registers are limited in size (usually less than 64-bit) and number (usually less than 32). They are often implemented to be very fast (in the order of the picosecond) using Static Random-Access Memory (SRAM) technology, while persistent storage is much larger (it is rare to have less than 1 MB) but much slower (in the order of milliseconds for magnetic storage disk or microseconds for flash memory). Also, because the fast memory of SRAM is more expensive than the slow memory of persistent storage, the memory is organized around a multi-level hierarchy between CPU registers and persistent storage that act as working memory. Nowadays, CPUs implement between 1 and 3 levels (rarely 4) of intermediate caches, that could be shared or private to CPU cores. Finally, between CPU caches and persistent storage there is the main memory implemented with Dynamic Random-Access Memory (DRAM) technology.

When the number of cores is low (typically 32 or fewer), they are interconnected to the main memory and share the same memory space. This topology is called Symmetric MultiProcessing (SMP). In such topology, each processor has equal access rights and time to the memory (and inputs/outputs), thus the term symmetric. In a single-chip multicore, the interconnection network is simply the memory bus. When the processor count increases, communication between processors and the main memory can be a limiting factor for performance. In this configuration a local memory to the processor is generally present and this topology is called Distributed Shared Memory (DSM). Figure 2.5 shows these two architectures.

As in an SMP architecture, all processors are interconnected to the shared memory via some sort of a bus. Each processor has a uniform latency to access



(a) Architecture of a symmetric multiprocessor system. A bus arbiter is used to determine which CPU gets to use the bus. Data coherency is ensured by a *bus-snoop protocol*.



(b) Architecture of a distributed shared memory system. Data coherency is ensured by a *directory protocol*.

Figure 2.5: Memory architectures.

Time	Event	Cache contents for processor A	Cache contents for processor B	Memory contents for location X
0		1		1
1	A reads X	1		1
2	B reads X	1	1	1
3	A writes 0 into X	0	1	0

Figure 2.6: The cache coherence problem for a single memory location (X), read and written by two processors (A and B), assuming a *write-through* cache policy. In a write-through cache policy, the cache and the main memory are simultaneously updated with the new value. With a *write-back* cache policy, though the memory content is updated only when the memory block is evicted from the cache, the cache coherency problem persists.

the memory, they are also called Uniform Memory Access (UMA) multiprocessors. On the other hand, in a DSM architecture, the access time to reach a memory block depends on the distance between the core that requests the memory block and the current location of that memory block. Thus, they are also referenced as Non-Uniform Memory Access (NUMA) systems.

2.4.2 Data coherency

With the presence of private (or local) memory living in a shared (or global) memory space as in the case when each core has its own cache, comes the problem of cache coherency. Figure 2.6 is an example of the problem. When core A is working on a particular memory block X in its private cache (at time 1), and core B attempts to work on this same memory block X (at time 2), the memory block X will be present on both core A and B. The problem arises when one of the cores modifies the content of the memory block X (at time 3). Without a coherency mechanism, the two processors A and B will see different content in their private cache, which breaks coherency and can cause software to run incorrectly.

To solve this coherency problem, multiple strategies have been proposed. The key concept is to keep track of the status (or state) of any sharing of memory blocks currently in caches and define operations transitioning from one state to another described in a finite state machine.

The states of a memory block can be as follows:

- M(odified): the cache has the most up-to-date value and the only valid copy of the memory block and is potentially non-coherent with the main memory.
- S(hared): the cache has a read-only copy of the memory block. Other caches may have valid, read-only copies of the memory block.
- I(nvalid): the memory block is invalid. The cache either does not contain the memory block or it contains a potentially stale copy.
- O(wned): the cache has the memory block and may be dirty (non-coherent with the main memory). Other caches may share the memory block in the shared state without write permission. Other caches may request a read operation. In that case, the data is supplied by the "owner" instead of the main memory.
- E(xclusive): the cache has the memory block and is coherent with the memory. No other copies of the memory block exist within other caches.

With these different states, one can design multiple state machines to describe what to do when processors request to read or write from and into a memory block. Multiple coherency protocols exist such as MSI, MEI, MES, MESI, MOSI and MOESI, as well as peculiar extensions taking these protocols as the basis for particular optimization to targeted workload or server and warehouse architectures [5, 61, 103].

There are two main classes of coherence protocols in use, each of which uses different techniques to track the sharing status: snooping and directory.

- In a *directory protocol*, the sharing status of a particular memory block is kept in one location called a directory. When one core wants to read a memory block, the cache controller of the core sends a request to the memory controller of the directory that is managing the memory block (the *home* directory). The directory has the information of the core that is the current owner, as well as other sharers of the memory block. If the memory controller is the current owner of the memory block, the memory controller completes the transaction by sending the data to the requester. If the memory block is owned by another cache controller, the memory controller forwards the request to the concerned cache controller that will complete the transaction by sending the data to the requester. The request and acknowledgement must transit first to the memory controller that manages the directory of the memory block, and possibly another cache controller if the memory block is currently owned by another cache controller, thus it is referred to as a 2-hop protocol in the literature.

- In a *snooping protocol*, the coherency is performed by broadcasting requests to the bus. When a core wants to operate on a memory location, the cache controller of the core initiates a request for the memory block by broadcasting a request message to all other cache controllers. Each processor's cache controller monitors or *snoops* the bus and act in accordance with the request. The cache controller will change the status of the memory block if that memory block is present and acknowledges the request. The request and acknowledgement messages are direct from cache-to-cache and are often referred to as a 1-hop protocol in the literature.

One of the major problems with a snooping protocol is that the broadcasting of coherency transactions becomes a bottleneck for performance for a large count of processors. Another issue with broadcasting is energy consumption. When one cache controller initiates a coherency transaction, the broadcast will target all other cache controllers, even if they do not share the memory block concerned by the transaction. On the other hand, a directory protocol is scalable because it unicasts, but many coherency transactions take more time because of a possible indirection when the home directory is not the owner of the memory block. An intermediate solution between a snooping protocol and a directory protocol is an insertion of the directory concept at the interconnect network of a snooping protocol. This directory structure in a snooping protocol is called a *snoop filter*. Instead of statically mapping a range of memory blocks managed by one directory like in a directory protocol, the snoop filter is a dynamic map that references memory blocks with the identifiers of other sharers and the state of the memory block in a similar manner to that of a directory protocol. The map is inclusive of the upward caches.

Besides the choice of the available memory block state model and the class of the coherency protocol, there is the choice of action to perform when a core writes to a memory block, *invalidate* or *update* sharers of that memory block. In an invalidate protocol, when a core wants to write to a memory block, it first initiates an invalidation request to all other caches before transiting to the M state when the invalidation request is acknowledged. In an update protocol, when a core wants to write to a memory block it initiates a coherence transaction to update copies in all other caches. Comparing these two choices, update protocols reduce the latency for a core to read a newly written memory block because the core does not require to initiate and wait for a bus transaction to complete when it was already a sharer of the memory block; the data is always up-to-date. However, update protocols consume more bandwidth than invalidate protocols as an update message is larger than an invalidate message (address + data instead of just an address). However, if the workload never reuses the memory block and it is not evicted, the memory block is continuously updated for no reason. In comparison to an invalidate protocol, the updated memory block is brought back in the cache only when required.

2.5 Summary and putting all together

In its simple form, general-purpose multicore processors consist of identical cores: either large, complex and powerful ones consisting of a superscalar out-of-order pipeline, or small low-power ones implementing an in-order pipeline. The Arm big.LITTLE system [55] is an alternative to this design and proposes the use of a "big" processor paired with a "LITTLE" processor to create a system that can deliver high performance for the most demanding tasks while providing energy efficiency for the less intensive tasks. This architecture is particularly interesting for battery-powered mobile or tablet devices to accommodate high performance for intensive tasks such as gaming or web browsing while delivering long battery life for less demanding tasks such as texting, e-mail and audio.

This thesis studies scheduling strategies for single-ISA Heterogeneous MultiProcessing (HMP) systems based on, but not limited to, an Arm big.LITTLE system. The test platform used throughout the thesis is a HardKernel Odroid-XU3 which has the Samsung Exynos5422 [120] System-on-Chip (SoC) implemented in a small form factor using a 28 nm HKMG process node. This SoC is used on the Samsung Galaxy S5 smartphone. Due to the small form factor of the SoC and the small-sized passive cooling device of the smartphone, power dissipation and thus thermal regulation becomes critical to achieve good performance. The scheduling strategies studied in this thesis can be applied during any situation. However, the scheduling strategies studied in this thesis are evaluated in the context of heavy thermal pressure which is a common concern for embedded devices [49].

The big.LITTLE system on the SoC is composed of 4 Cortex-A15 cores as the big processor while the LITTLE processor uses 4 Cortex-A7 cores. The main features of both processors are summarized in Table 2.1. The terminology used by Arm to refer to a multicore processor taking the role of either big or LITTLE is a *cluster*. The rest of this thesis will use big clusters or LITTLE clusters to refer to them.

Both clusters are attached to an Arm CoreLink CCI-400 Cache Coherent Interconnect. The interconnect guarantees data-coherency using a bus snoop protocol that does not contain a snoop filter to limit snoop traffic between the two processors. The interconnect uses a MOESI state machine for cross-cluster coherency. Figure 2.7 shows a block diagram of the SoC architecture.

Role		big	LITTLE
Processor		Cortex-A15	Cortex-A7
Core count		4	4
Pipeline	integer	15 stage	8 stage
	floating point	17-24 stage	10 stage
Execution		out-of-order	in-order
Issue width		3	2
Fetch width		3	2
L1 I-cache		32 KB/2-way private	32 KB/2-way private
L1 D-cache		32 KB/2-way private	32 KB/4-way private
L2 unified cache		2 MB/16-way shared inclusive	512 KB/8-way shared exclusive
Frequency range		0.2 to 2 GHz	0.2 to 1.4 GHz

Table 2.1: big.LITTLE system constitution.

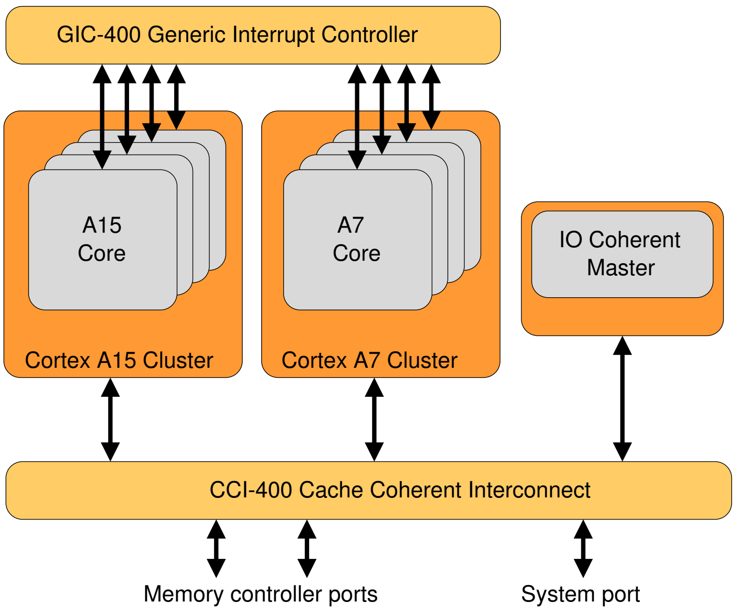


Figure 2.7: big.LITTLE architecture implementation on our platform. Full data coherency is assured by a bus-snooping protocol.

Chapter 3

Related work

This section reviews multiple works that are in direct relation to the subject of this thesis about scheduling strategies and thermal management for single-ISA AMP. In the literature, there exists different terminologies to identify the purpose of one CPU core in relation to other CPU cores whether it targets execution for performance of energy efficiency (i.e., big/LITTLE, big/small, fast/slow, complex/simple, strong/weak, aggressive/lightweight). To simplify the discussion, the terminology of big and LITTLE cores will be employed for the rest of this thesis to identify either performant cores or energy-efficient cores respectively.

The section 3.1 will focus on application scheduling without the notion of temperature. The following section 3.2 will continue on scheduling strategies towards thermal management using thermal models for reactive and proactive approaches.

Finally, the summary in section 3.3 will show an experiment that demonstrates the difficulty of thermal management in computing systems.

3.1 Scheduling on single-ISA AMP

This section reviews some work that focuses on scheduling for single-ISA AMP systems. To help frame work in this thesis table 3.1 presents an overview of work in this area.

Table 3.1: An overview of the work presented in section 3.1 on scheduling strategies for single-ISA AMP systems.

Paper	**Asymmetry**		**Scheduling criterion**	**Implementation**	**Offline**	**Online**
	micro-architecture	**frequency**				
[78]	x		IPC	simulation		x
[13]	x		IPC	simulation		x
[84]		x	task time	simulation		x
[81]		x	task remaining time	simulation		x
[122, 123, 113]		x	IPC and memory profile	simulation	x	
[74, 114]	x	x	IPC and memory accesses	simulation	x	
[136]	x		new PMCs to expose the length of the instruction dependency chain	simulation		x
[109]	x		IPC and memory accesses	physical	x	x
[56, 91]	x	x	IPC, memory accesses, branch misses and prepared formula	physical	x[a]	x
[110, 111]	x	x	IPC, memory accesses, branch misses and prepared formula	physical	x[b]	x
This thesis	x	x	IPCs[c] and memory profile	physical	x[d]	x

[a]exhaustive offline workloads characterization

[b]exhaustive offline workloads characterization, the work in [111] partially remove the offline characterization

[c]IPC is computed at both dispatch and retirement pipeline stage level

[d]offline analysis to compute memory profile only for better scheduling, not a strict requirement

In [78] and [13], authors propose a very similar scheduling approach that uses Performance Monitoring Unit (PMU) counters to track Instruction Per Cycle (IPC). Their strategies consist of executing the application thread for a small amount of time on each core type to determine respective IPC performance. Once measured, a thread will be mapped on a LITTLE core if it achieves only modest performance improvement running on a big core and the thread that benefits significantly from a big core is executed on a big core.

In [84], authors propose a load-balancing scheduler which gives more threads to the big core. As this work considers asymmetry in core clock frequency only, the number of threads that is allowed to run on the big cluster is computed as a scaling factor from the little core clock frequency to the big core clock frequency. The scheduler uses a "faster-core-first" strategy, favouring the use of the big core when it is underutilized.

In [81], authors propose a scheduling scheme which schedules tasks with longer remaining execution-time on big cores. This work considers multithreaded applications and it relies on the assumption that threads have symmetric work to do between synchronization points (i.e. barrier or termination). Tasks remaining execution time to the next synchronization point is determined by using the instruction retired PMU counter.

In [122, 123, 113], authors use the principle of the signature of an application to guide scheduling decisions. The signature consists of the miss-rate profile based on the reuse-distance profiles. The platform heterogeneity is only on frequency scaling.

In [74, 114], to perform scheduling, authors consider a Speedup Factor (SF) of a single thread application and its extension to multithreaded applications termed as "utility factor". A speedup factor is "how much quicker an application retires instructions on a fast core relative to a slow core". In the case of frequency-based heterogeneity, the SF is the miss rate profile as in their prior work [113]. In the case of micro-architectural heterogeneity, authors use a regression model considering IPC and other PMU counters related to misses in cache memory.

In [136], authors propose Performance Impact Estimation (PIE), a model to predict the best task-to-core mapping. In particular, the proposed runtime tracks the ILP and MLP of a task running on one core type to predict performance on the other core type. In this work, authors make some simplifying assumptions such as the presence of an identical cache hierarchy on both core types. In addition, a set of new hardware counters to record inter-instruction dependency distance distribution is required and are not available in existing processors. The proposed solution is evaluated on a simulator.

In [109], authors propose performance and power estimation models using several offline analyses of workloads. Both models are inserted into the binary of the workload. These models are instantiated at runtime to perform application scheduling. Models consist of prepared formulas considering PMU counters based on multiple offline analyses to determine the Cycle Per Instruction (CPI) stack from both computing intensity and latency penalty from memory accesses. Their study is performed only on a single thread application, with unused CPU cores and clusters offline. As such, this strategy does not capture thread memory interference such as data-coherency mechanisms and thrashing of shared cache memory.

In the XNU kernel (for macOS and iOS), the scheduler is guided by a QoS recommendation from the programmer of an application [3]. Internally, the scheduler uses this recommendation to schedule tasks on big cores or LITTLE cores (named P and E cores in the source code). Kernel tasks are assigned to E-cores, while the mapping of users' tasks forming an application follows the suggestion of QoS run queues. If too many tasks are suggested to run on big cores, low-priority tasks are spilt to LITTLE cores, or if there are unused LITTLE cores, these cores could steal tasks from big cores. A rebalance logic will readjust tasks to their recommendation.[1]

In the Linux kernel (and therefore Android and derivatives), the scheduler

[1]Source code can be found at: `https://github.com/apple-oss-distributions/xnu/blob/main/osfmk/kern/sched_amp.c`, `https://github.com/apple-oss-distributions/xnu/blob/main/osfmk/kern/sched_amp_common.h` and `https://github.com/apple-oss-distributions/xnu/blob/main/osfmk/kern/sched_amp_common.c`

migrates tasks between CPU cores considering a computing capacity model that represents the performance of the CPU related to the others. This work will be discussed in depth in section 5.2.1 of chapter 5 as the work of this thesis is directly compared to it.

In [56, 91], authors propose to dissect applications via a compiler approach snippet which consists of a set of basic blocks using LLVM. Each snippet contains calls to monitor PMU counters to determine instruction and memory intensiveness of the snippet. An offline analysis is performed to stress each snippet exhaustively over configurations of allowed CPU cores and frequencies. They determine Pareto-optimal configuration for different objective functions to minimize energy consumption or maximize performance. The output of the offline analysis is stored in a file that is reused at runtime for further runs of the application. The offline analysis seems intractable for a large number of applications as generated models are tailored to the device under scrutiny.

In [110], authors propose a static scheduling strategy from an offline study. The offline study thoroughly explores every combination of core mapping on a big.LITTLE system that led to the least execution time. Once the static scheduling is determined, the operating frequencies of clusters are adjusted using a Memory Reads Per Instruction (MRPI) metric. The runtime uses a low frequency for applications that are considered memory intensive, as these applications will suffer less performance degradation compared to computationally intensive tasks.

In [111], authors continue the work of [110] to limit the requirement of the exhaustive application's performance estimation by using a performance prediction at runtime via a machine learning model using PMU counters trained offline on a subset of workloads. At runtime, CPU core resources are adjusted at workload creation/completion using the performance prediction model.

3.2 Thermal management

This section reviews some work related to thermal management for computing systems. The next section introduces a thermal model that is used to a great extent to predict the temperature of a system. The following sections review work on thermal management for uni-processor, SMP and single-ISA AMP.

3.2.1 RC-network thermal model

There exists a well-known duality between heat transfer and electrical phenomena [121] which provides a convenient basis for modelling the chip temperature using a dynamic compact thermal model [44, 126]:

$$\frac{1}{R_{th}}T(t) + C_{th}\frac{dT(t)}{dt} = P(t)$$

where C_{th} and R_{th} are the thermal capacitance and thermal resistance (the inverse of thermal conductance) respectively, T and P represent the current temperature and power consumption at the time t.

In practice, thermal management is performed at a regular sampling period. Therefore, this differential equation can be discretized and rearranged to predict the temperature at the next sampling period by:

$$T'(t) = \frac{P(t)}{C_{th}} - \frac{T(t)}{R_{th}C_{th}}$$

With the use of system identification or model fitting tools, one can derive C_{th} and R_{th} parameters using a set of workload showing different power consumption and monitoring the input temperature of the current sample, and output temperature at the next sample.

3.2.2 Thermal management on uni-processor and SMP

This section reviews work related to thermal management for uni-processor and SMP. These works may not be applied directly to single-ISA AMP without severe loss in performance, but are interesting thermal management approaches for computing systems.

In [75, 76], authors adjust the task scheduling time slices. Reducing the time of “hot” jobs leads to slightly degraded application performances but limits the need for more severe thermal throttling techniques such as frequency scaling.

In [28, 45], authors use a compiler approach to determine application phases. They adjust task priority and execution time to control the temperature with a fully utilized system.

In [154], authors organize the order of execution of tasks to control the temperature. Their approach is to interleave “cold” tasks while “hot” tasks are running to limit thermal throttling.

In [63], authors propose activity migration to control the temperature of CPU cores. They use PMU counters and an RC-network to predict CPU core temperatures.

In [152], authors propose a scheduling system to control the temperature. They use PMU counters and RC-network to predict the temperature. Two predictive models are used, one to predict application temperature, and another to predict CPU core temperatures. Another mechanism to adjust tasks priority is in place.

Recently, Arm introduced Intelligent Power Allocation (IPA) which maximizes the performance of the device within a thermal envelope [12]. It accomplishes this by integrating thermal closed-loop management with intelligent power distribution amongst system components. A system component is a block that supports frequency scaling which could be CPUs, GPUs, DSP, etc. At its heart, IPA uses frequency scaling on system components to distribute performance and power requests based on power budget and thermal headroom in the system. The power budget is estimated using a Proportional Integral Derivative (PID) controller. Each system component can request different performance levels. And by using a power model, IPA estimates their relative power consumption and thermal impact on the system. The power budget is then intelligently distributed among system components in the device to maximize performance while keeping the temperature under control.

Under Linux, the default thermal management employs a frequency scaling approach. This strategy will be presented and explored in detail in chapter 5.

3.2.3 Thermal management on single-ISA AMP

This section reviews some works specific to the thermal management for single-ISA AMP systems. To help frame work in this thesis table 3.2 presents an overview of work in this area.

In [102], authors propose a QoS-based resource management to schedule tasks on a cluster at a specific frequency to satisfy the QoS requested for the tasks. Thermal management is relative to the Thermal Design Power (TDP) budget of a cluster. They rely on the Linux CFS scheduler for task-to-core mapping and do not control specific CPU core assignment within a cluster.

In [125], authors use state-space modelling based on an RC-network model to predict the temperature and to determine a thermal budget to operate the device. The strategy consists of using the highest tolerable frequencies with all big cores according to the power budget. When the power budget cannot be met, individual CPU cores that have the highest temperature is powered off, leaving the scheduling responsibility to the operating system to reassign applications to other cores. The evaluation considers only single-threaded applications.

In [17, 20], authors predict power and temperature using state-space modelling with an RC-network model to allow the number of cores used and the maximal frequency of the big cluster that satisfy thermal constraints. If the prediction suggests 0 CPU core from the big cluster at the minimal frequency that satisfy thermal constraints, tasks are mapped to the LITTLE cluster and the hottest core of the big cluster is set to offline as a last resort. It seems that once a core is set to offline, it is never set to online later. The task-to-core mapping is not controlled precisely and the work rely on the Linux CFS scheduler. Therefore, if a workload

Table 3.2: An overview of the work presented in section 3.2.3 on thermal management for single-ISA AMP systems.

Paper	**Scheduler**	**Offline prediction model**
[102]	own scheduler for intra-cluster migration OS default for inter-cluster migration	none
[125]	OS default	RC-network state-space thermal model
[17, 20]	OS default	RC-network state-space thermal model and power model
[115, 118, 116, 117]	own scheduler using exhaustive offline workload characterization	light hardware thermal characterization
[138]	scheduling based on [110] (see section 3.1	ARMA model
[72, 73]	own scheduling strategy based on IPC of tasks	none
This thesis	own scheduling strategy based on IPC and memory profile of tasks	light hardware thermal characterization

has an uneven computing activity or dynamic phase activity change, this strategy may not find the best tasks mapping before reducing the performance of the system.

In [18, 19], authors focus on maximizing performance of foreground tasks that run on the GPU under an Android environment. When an intensive background task on the big cluster causes a thermal emergency, tasks are migrated to the LITTLE cluster. The frequency control of the big cluster is performed using the default DVFS governor. Tasks scheduling on the CPU are not controlled precisely. The main drawback of this strategy is when the foreground application depends on a background task. The proposed strategy does not capture dependency relationship of tasks and may deliver poor performance to a foreground application that is waiting for a background task that runs on the LITTLE cluster.

In [115, 118, 116, 117], authors propose a QoS-based thermal management strategy to satisfy user experience. To do so, they allow short burst workload to exhaust thermal headroom and focuses on slightly degrading the QoS for long-time running workload by using frequency scaling that employs a step-by-step frequency reduction if the device is over a thermal limit. An offline analysis is used to characterize the device for thermal coupling between CPU cores with the GPU, and also for application performance characterization. This offline analysis is used at runtime to schedule applications. In particular, offline thermal characterization is used to determine an order in which cores are likely to heat up rapidly relative to each other. This offline analysis is then reused for scheduling during a thermal emergency. On the other hand, application performance characterization helps in determining tasks and threads which are part of the critical path that contributes the most to the QoS metric and user experience. These tasks are then mapped to the big cluster, while the other tasks stay on the LITTLE cluster. If the temperature of the device cannot be reduced further after reaching the lowest available frequency on the big cluster, all applications are migrated to the LITTLE cluster. The offline analysis must be performed for every application before being usable under this thermal strategy. This step can be long and difficult to scale when applications contain a large number of threads, and each thread has erratic activities that contribute to the critical path.

In [138], authors propose a predictive thermal and power management approach based on the gradient of the last two temperature readings and a power model using an AutoRegressive Moving Average (ARMA) model. A dynamic error correction is applied to the thermal predictor at runtime. The frequency of a cluster is adjusted by using the information of Memory Reads Per Instruction Retired that the cluster is currently experiencing. The scheduling strategy is based on the work from [110].

In [72, 73], authors use performance models built at runtime to predict which is the best option between intra- and inter-cluster migration and frequency scaling to reduce the device temperature. When there is a thermal emergency on the big cluster, they first try to migrate tasks between cores if there are any that are not being used. Otherwise, they migrate all tasks from the big cluster to the LITTLE cluster. While doing so, they build two models that attempt to predict the cost of migrations between clusters, as well as the performance loss by using DVFS on the big bluster. Both models take IPC and cluster operating frequencies as input and produce some sort of cost factor. As the model is built at runtime, it requires testing the two approaches of migration-based and DVFS-based approach separately. Once these penalty models are built, they can identify what would be the best approach in the event of a subsequent thermal emergency. Both models are reset if the workload changes. This strategy is effective when the workload is regular and linear in its dynamic behaviour. However, when the workload is constantly changing this strategy may be inefficient. The evaluation considers multiple single-threaded applications and does not involve multithreaded applications nor relative penalties due to memory communication.

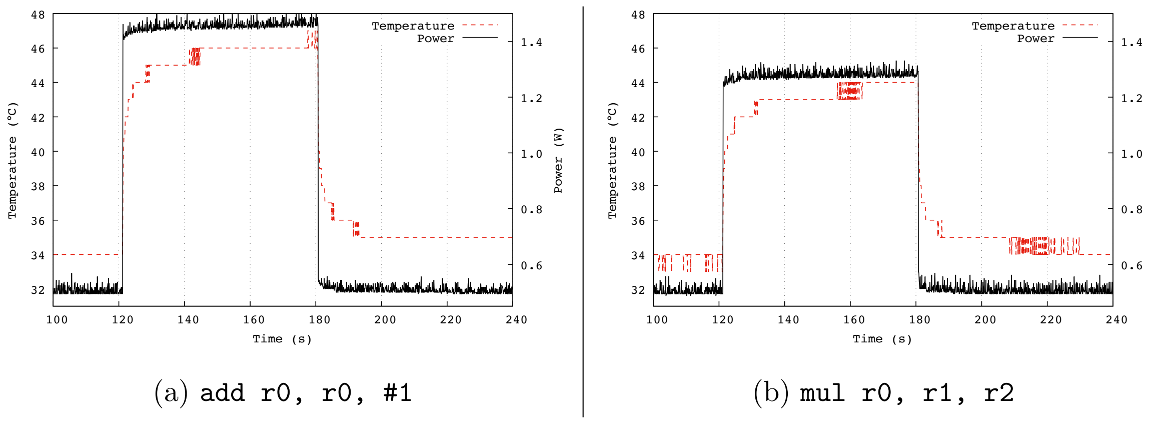
3.3 Summary

All related works presented in this chapter that attempt to predict values from power consumption and heat production show more or less errors between the predictive models and the truth.

This is due to the nature of the underlying hardware. In section 2.1, the relationship between the number of switched transistors and power consumption has been established. The more transistors involved, the greater the power consumption. As such, one could not extract the precise power consumption (therefore the temperature) without knowing the exact calculation beforehand. In [107], the authors conducted an experiment where they exhaustively exercise the `mul` instruction on all operand values on an 8-bit Atmel AVR processor. The results show that there is a 15% difference in power consumption depending on the value of the operands.

Moreover, the input of the model is generally a `util` metric provided by the operating system kernel which is based on the scheduling time slice. However, this metric does not encompass the actual physical use of the hardware considering micro-architectural specificities. Another approach is to use PMU counters to account for the number of instructions executed. Sadly, while being more fine-grained, this approach has its own drawback. figure 3.1 shows an experiment comparing two code constructions running on a Cortex-A15 consisting of “`add r0, r0, r0`” for figure 3.1a, or “`mul r0, r1, r2`” for figure 3.1b. Both code construction have an IPC value of one. However, the `add` code construction draws about 12% more power than a `mul` code construction. Thus, the non-injective non-surjective relation between IPC and power is not trivial and any model will be inaccurate by just considering PMU counters in isolation.

In addition, the nature of the memory hierarchy may result in different timing latencies depending on the actual data location. For instance, the same memory load instruction could have a low latency if the data is located in the local cache, and high latency if the data is residing in a remote cache or off-chip memory. Moreover, we will



(a) `add r0, r0, #1` (b) `mul r0, r1, r2`

Figure 3.1: Two different program that have an IPC value of one, but face different power consumption. The program exercise repeatedly an `add` instruction in figure 3.1a, and a `mul` instruction in figure 3.1b for one minute.

The rest of the code that drives the execution is the same and consists of a simple loop. The number of instructions in the loop is set to fit in a virtual memory page of 4 KB to limit TLB operations, the number of iterations per loop for both programs is the same and both experiments operate under the same setup on CPU frequency and core mapping. Therefore, the only difference in hardware usage is the instruction and registers used.

With a steady-state temperature of 46 °C and 1.46 W of power consumption for the `add` instruction and a steady-state temperature of 44 °C and 1.28 W of power consumption for the `mul` instruction, there are 4% difference in steady-state temperature and 12% difference in power consumption while the IPC is 1.

see in chapter 4 that the system under study shows a non linear latency depending on the frequencies of components of the system. This non-determinism will bring inaccuracy to any model that targets hardware comprising such a hierarchy.

Furthermore, the complexity of predictive models to consider transient temperature and the positive feedback loop nature between thermal and power leakage requires expensive code in space and time. Also, the utilization of such models directly on the device has an intrusive impact on the system.

For these reasons, the thermal management approach proposed in this thesis in chapter 5 comprising scheduling and frequency scaling does not involve a complex predictive thermal model. Instead, the approach uses directly available thermal sensors and acts reactively to thermal events.

Chapter 4

Multiprocessing and frequency scaling: when data takes its time.

Modern embedded platforms such as mobile and tablet devices have become a ubiquitous part of the modern computing ecosystem. Their battery-powered design has driven a new wave of hardware research, including the *asymmetric multiprocessor* (AMP). This is a System-on-Chip (SoC) which offers a cluster of CPU cores designed to be energy efficient, and another designed to offer high performance. This concept has been implemented in the ARM *big.LITTLE* architecture which is widely adopted in mobile platforms, including the Samsung Galaxy range. The big.LITTLE design has an energy efficient processor (named LITTLE) with a performant but more power-hungry processor (named big), where each processor also offers a range of frequency settings. This design exposes a large optimization space for software to trade performance against reduced energy consumption by choosing a processor depending on energy and time constraints.

In order to simplify software development for the platform, the hardware offers transparent data coherency between its processor clusters. On many big.LITTLE platforms, this is implemented via the ARM CoreLink CCI-400 interconnect [6] which uses a bus-snooping protocol: when a data access is issued by a processor, the interconnect will broadcast a message to all processors to check whether the data is present in their local cache before accessing RAM. Because the interconnect communicates with the processor, extra latency can be introduced in this procedure if the processor's clock frequency is low. Experimental exploration shows that `gcc` can suffer an 80% slowdown due to this mechanism. While newer big.LITTLE platforms include a hardware snoop filter to mitigate these effects, the popular CCI-400 interconnect remains in wide use across the world – a recent study by Facebook reports that **75% of smartphones** using their platform have CPU designs released before 2013 [150], before any big.LITTLE hardware snoop filters were designed.

This chapter presents a software solution to this problem with a novel `ondemand-anti-snoop` governor, a new Dynamic Voltage and Frequency Scaling (DVFS) governor which enhances the standard Linux `ondemand` governor to consider the memory traffic between processor units and main memory at a hardware-level. The proposed approach is highly generalized, working transparently across all software, and requires only a simple, generic, train-once model of real-time system activity to learn snoop effects in a range of scenarios. Evaluation of this work shows that performance improvements of up to 40% can be achieved with this new dynamic frequency governor on real-world software.

The main contributions of this work are:

- A methodology to characterize snooping latency effects on the bus-snoop protocol interconnect fabric.
- A simple but effective model of snoop latency using a microbenchmark; the model is trained once on this benchmark and then applies generically to all software.
- A new DVFS governor which uses the proposed model together with hardware-level information to mitigate snooping latency in real-time by locating the ideal holistic frequency configurations on the SoC.

This chapter is organized as follows. Section 4.1 first presents the hardware architecture in detail on which the study is conducted. Section 4.2 then presents a model of snooping effects, considering hardware-level data, and a new frequency scaling governor using this model to mitigate snooping latency. An evaluation of the new governor on a set of real-world workloads is presented in section 4.4. Finally, the chapter is summerised in section 4.5.

4.1 Memory architecture background

To understand the relative difference in execution time for different clock frequencies, this section first explains in detail how memory works on the platform used for this study.

The platform in question is an Odroid-XU3 from HardKernel [58], which implements the Exynos 5422 System-on-Chip (SoC) from Samsung [120]. The SoC itself is identical to that used in the Samsung Galaxy S5 smartphone, though the host board is slightly different. Section 2.5 provides more detail of the SoC architecture, which implements two CPU clusters of 4 cores each, for a total of 8 cores. One cluster targets energy efficiency (named LITTLE) and uses a Cortex-A7 [10], while the other cluster targets performance (named big) and uses a Cortex-A15 [9]. Each CPU core has its own private L1 cache and shares one L2 cache within the same cluster. Both CPUs have 32 KB instruction cache and 32 KB for data in L1 cache. The LITTLE cluster has a unified 512 KB L2 last-level cache, while the big cluster has 2 MB. Finally, CPU core clock frequency is shared between cores within the same cluster.

In this SoC, data cache coherency between the two clusters is managed by the ARM CoreLink CCI-400 Cache Coherent Interconnect [8]. Using this interconnect fabric, when a CPU core in a cluster performs a read/write memory operation for data that is not contained in its internal cache, the interconnect checks if the data is present in the cache of the other cluster, and if not, it then performs an access to off-chip main memory (RAM). This check is performed by an operation called snooping, for which further specification details can be found in the relevant ARM white paper [130].

The effect of snooping on this hardware is that extra latency can be introduced when a process performs several memory accesses and when the frequency of the other cluster is low. This is because the interconnect fabric communicates with the CPU cluster to check its cache status, and the cache status check is performed at a speed relative to the current clock frequency of the cluster. Therefore, a process running on one CPU cluster can stall on memory accesses because another cluster has a low clock frequency.

This effect interacts with the common dynamic CPU frequency scaling policy (DVFS) used in Linux, and present on the majority of Android smartphones (the popular Energy Aware Scheduler (EAS) for Android uses a frequency governor with a very similar design to that used in standard Linux). DVFS by default attempts to reduce the clock frequency of a CPU whenever it is idle, to correspondingly reduce the amount of energy consumed by that CPU. Whenever one particular CPU cluster on a big.LITTLE chip has a low workload intensity, the clock frequency of that cluster is reduced, which can in turn cause snoop-induced stalling on memory accesses across other clusters. Experimentation shows that this stalling can cause a slowdown of up to 80% for the most memory-intensive applications – which is a very significant impact for the wide range of smartphone models using this hardware architecture.

Obvious solutions to this problem are (i) to always run all CPUs at their highest clock frequency, or (ii) to try to completely power down CPUs that aren't being heavily used (as powering down a CPU cluster also removes snooping effects). Both approaches are problematic: running CPU clusters at high frequencies incur significant energy penalties and so will reduce battery lifetime while powering down a CPU cluster takes a significant amount of time to migrate tasks and deactivate cluster-bound kernel services (our measurements with `hotplug` show that Linux takes 300 ms to power down a low-workload cluster). A third approach for an idle CPU with no workload is to rely on periodically putting that CPU into sleep mode via the `cpuidle` framework (rather than forcing a full power down), and periodically waking the CPU to accommodate kernel maintenance routines. However, during all of the periods in which the CPU is awake this may incur snooping latency.

Therefore, neither of these approaches is attractive by themselves. Instead, a model of the hardwar snooping behaviour and its interaction with clock frequencies is used to discover how the snoop architecture behaves. In particular, this model is trained on a microbenchmark that stresses in isolation the memory subsystem. Using this trained model, a new dynamic frequency governor is proposed. This governor monitors the system in real-time to continually find a balance between raising clock frequencies to avoid snoop-induced stalling when needed while still keeping them as low as possible to conserve energy. This governor is most effective when multiple clusters are awake with at least some workload, but is also complementary to periodic CPU sleep protocols used on completely idle clusters during the times when that CPU wakes for maintenance procedures.

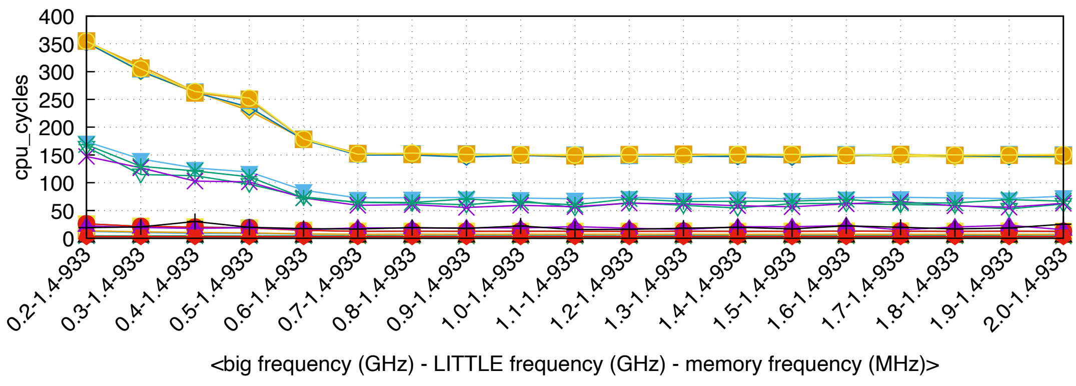
The next section introduces the design of this model and its training.

4.2 Modeling snooping latency

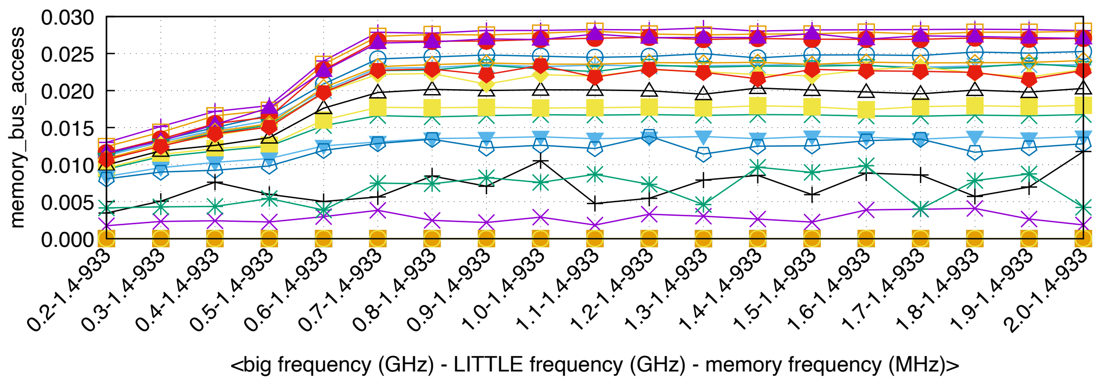
The first step is to develop a detailed model of the conditions under which snooping latency occurs and to understand which hardware monitors can be measured to enable the detection of this effect in real-time and correlate it with a model. This procedure is done by using a microbenchmark that targets a range of different memory access patterns.

The following subsections first present a study of snoop latency effects, in general, using this microbenchmark, then discuss how one could detect these effects in real-time using Performance Monitoring Counter (PMC) on both the CPU and the interconnect fabric.

(a) CPU cycles spent to perform a memory access.



(b) CPU PMCs `memory_bus_access` for the process.



(c) CCI PMCs `stall cycles per read` on the big port.

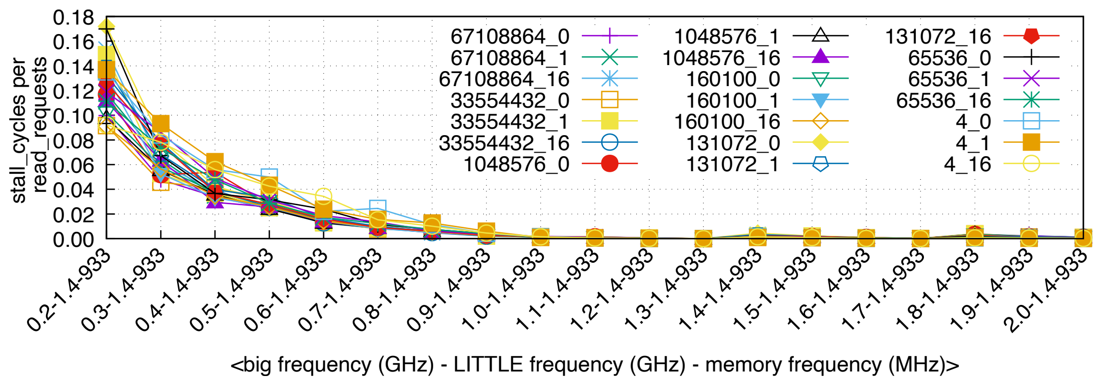
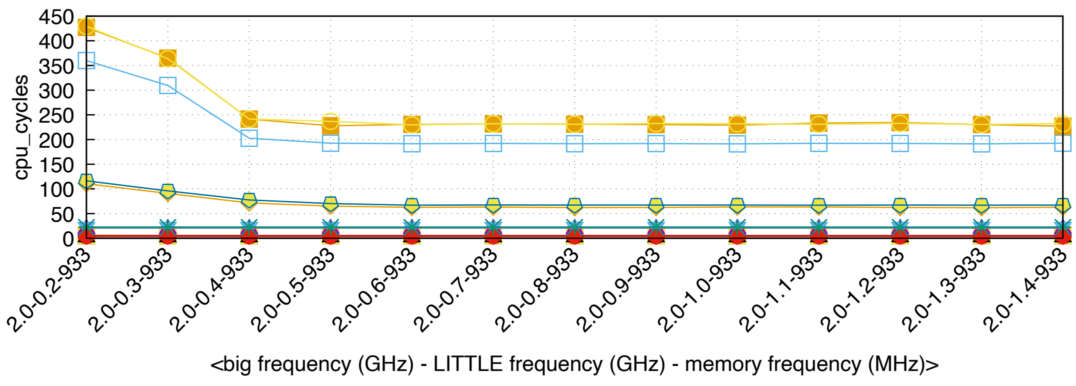
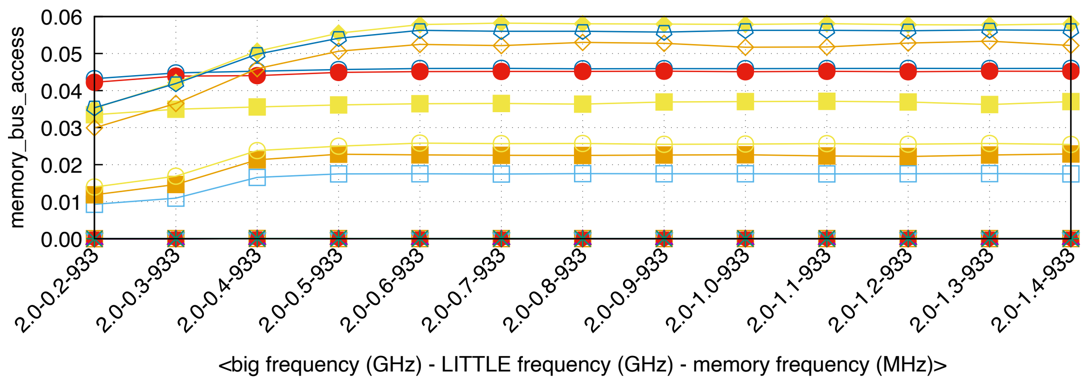


Figure 4.1: Benchmark on the LITTLE cluster, big cluster idle

(a) CPU cycles spent to perform a memory access.



(b) CPU PMCs `memory_bus_access` for the process.



(c) CCI PMCs `stall cycles per read` on the big port.

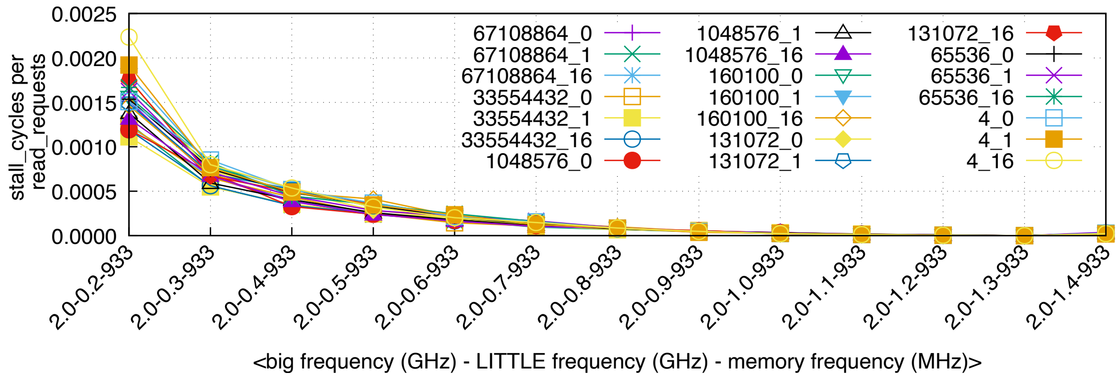


Figure 4.2: Benchmark on the big cluster, LITTLE cluster idle

4.2.1 Memory latency exploration

To study snooping latency on memory operation, a microbenchmark has been designed to stress the memory subsystem. This microbenchmark is largely inspired by other works on the subject [31, 94, 135]. However, those were lacking of generality and practicability for the purpose of this study. The benchmark runs a pointer-chasing loop over an array of a given size; values in the array represent the next index to follow in this same array for the next iteration. The benchmark can be configured to access the array using a range of different patterns, including a linear access of a unit stride, a stride of a cache line size, or a random pattern which prevents smart CPU memory prefetchers from being effective [1].

The benchmark is executed on one active cluster, keeping the other cluster idle, over each possible set of cluster frequencies and with different parameterizations to cover a large range of use cases. The parameterization comprises all three memory pattern accesses and multiple different array sizes. The array size itself is chosen to either fit or not fit in the different CPU internal cache sizes of both clusters, the latter case forcing off-chip memory access.

The figure 4.1 presents the overall results of running the benchmark on the LITTLE cluster and keeping the big cluster idle while the figure 4.2 presents the reverse scenario. Each graph shows all three memory access patterns with different array sizes. The graph legend shows the size (in number of indexes in the array) and pattern access for each series – where an access pattern 0 is the random pattern, 1 is a strictly sequential pattern, and 16 is a sequential pattern with a stride of the size of a data cache line of the CPU (both CPU clusters use a fixed line length of 64 bytes for their data cache at each cache level and array indexing via a `size_t` type which is an unsigned integer of 4 bytes on a 32-bit CPU).

First, let us begin by measuring the overall total number of CPU clock cycles spent to perform a single memory access in the benchmark (which reads the index of the next memory access in the array); the results of this are shown in figure 4.1a and

[1]The code source of the benchmarck is available at `https://github.com/wwilly/benchmark`

figure 4.2a for the benchmarks running on the LITTLE and big cluster respectively. The x-axes of all graphs show the CPU and main memory frequency configuration in the format {big_freq}-{LITTLE_freq}-{mem_freq}[2], and the y-axes show CPU cycles. figure 4.1a and figure 4.2a show that there is an increase in the number of cycles for any benchmark with an array of more than 131,072 elements[3], or when the memory access pattern is not strictly sequential. In both of these cases, there is an additional increase in cycles when the idle cluster has its frequency set below a certain level; in figure 4.1a this increase in cycles occurs when the idle big cluster drops below 0.7 GHz, while in figure 4.2a it occurs when the idle LITTLE cluster drops below 0.4 GHz.

One can note that, where a performance drift occurs, frequencies are at different levels depending on the cluster. This can be explained by differences in the microarchitecture design of both processors regarding pipelining. On the LITTLE cluster (Cortex-A7), the latency of an L1 cache hit is three cycles in the best case, while it is four cycles on the big cluster (Cortex-A15). As such, it is assumed that a snoop acknowledgement on the big cluster is deeper in the pipeline than on the LITTLE cluster, hence the different level of performance drift.

The graphs in figure 4.1b-c and figure 4.2b-c show information from the PMCs that are used to detect snooping latency cases in real-time. These PMC readings are taken across an identical set of frequencies and benchmarks to those used in figure 4.1a and figure 4.2a. In detail, figure 4.1c shows CCI stalling cycles per read request coming from the LITTLE cluster, while figure 4.2c shows CCI stalling cycles per read request coming from the big cluster. In both cases one can clearly see that this PMC correlates with the snoop mechanism effect: when clock frequencies are too low under certain memory access patterns, a higher stalling cycles is recorded on this PMC. Therefore, while this PMC shows part of the picture, it is only able

[2]CPU frequencies are shown in GHz, and memory in MHz. Dynamic adjustment of both memory and CPU frequencies is available by default on the Linux kernel on the tested platform.

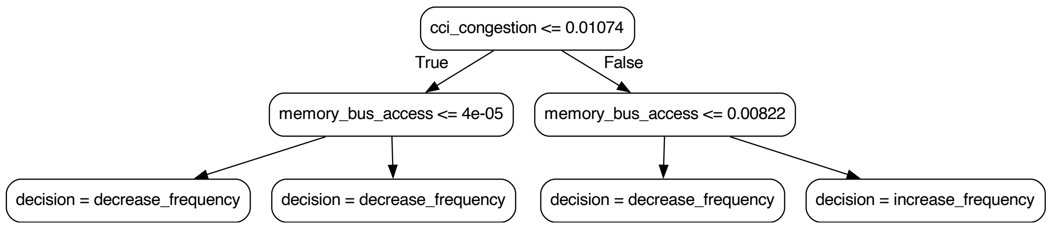
[3]This threshold is a result of the L2 cache size on the LITTLE cluster, which is 512 KB ($131,072 \times 4 bytes$).

to report all events from an entire cluster, involving all of its cores, and all of its processes.

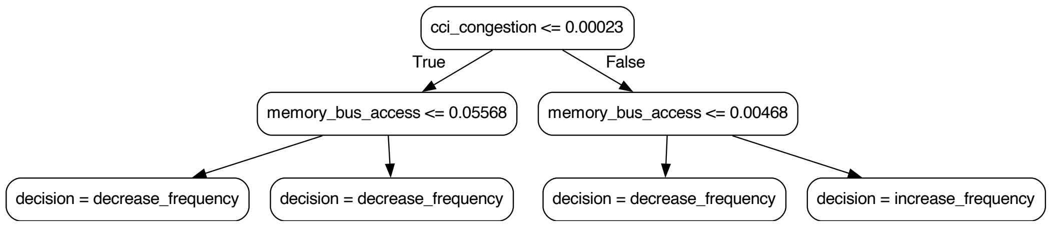
It is also useful to understand the memory profile of a particular process for which one would like to optimize, to figure out if the snoop latency measurements are actually affecting this process. Because the governor tends to be agnostic in any application-specific knowledge or static memory profiling, the governor uses a CPU PMC providing real-time memory bus usage for each core. figure 4.1b (and figure 4.2b) shows the `mem_bus_access` PMC for the microbenchmark running on the LITTLE and big cluster respectively which offers exactly this information in real-time. Here one can see that the number of memory accesses per cycle appears to *reduce* across benchmark configurations as the CPU frequency of the other cluster is lowered. This decrease is caused by the stalling effect itself, such that a process has an apparently lower amount of memory access as recorded by this PMC if that process is being affected by snoop-based stalling. This effectively reduces the speed with which that process executes and so reduces its apparent memory access volume per cycle.

The combined data from these PMCs indicates that stalling for a particular process of interest can be detected by reading the CCI PMC of its non-resident cluster, and the memory bus access profile CPU PMC of the process on its resident cluster. When the memory access profile exceeds a certain memory size or has a non-sequential pattern and begins to show lower memory accesses per cycle, *and* the CCI PMC of the non-resident cluster indicates stalling, one can assume that the clock frequency of the other cluster must be increased to reduce stalling effects for this particular process.

The next section proposes a formal way to use these PMCs to dynamically determine when the snooping mechanism could affect application performance.


cci_congestion <= 0.01074
True
False
memory_bus_access <= 4e-05
memory_bus_access <= 0.00822
decision = decrease_frequency
decision = decrease_frequency
decision = decrease_frequency
decision = increase_frequency

(a) Decision tree used to manage frequency of the big cluster.


cci_congestion <= 0.00023
True
False
memory_bus_access <= 0.05568
memory_bus_access <= 0.00468
decision = decrease_frequency
decision = decrease_frequency
decision = decrease_frequency
decision = increase_frequency

(b) Decision tree used to manage frequency of the LITTLE cluster.

Figure 4.3: Decision trees used to find CPU frequencies that limit snooping latency.

4.2.2 Detection of snooping latency

As shown in the section 4.2.1, for certain combinations of stalling cycles on the CCI PMC and memory bus access via the CPU's PMC, a clear trend of increase in latency can be seen. However, these increases show a non-linear relationship between the PMC values (specifically: the values reported in figure 4.1b and figure 4.1c relative to processes running on the LITTLE cluster; and the values in figure 4.2b and figure 4.2c for processes running on the big cluster). These non-linear relations are difficult to capture heuristically. Instead, an offline automatic modelling process has been developed to determine the ideal levels to configure the respective CPU frequencies.

There are two key questions to consider in solving this task: 1) which CPU frequency configuration has performance loss? and 2) what values do the relative PMCs report when there is known snooping latency?

A rigorous test for the first question would be to detect when the performance variation is *statistically significant* and not inherent noise due to the operating system's process management. To determine whether there is a statistically significant variation, a Student's t-test is performed on each execution trace (scanning each CPU configuration) against the highest possible frequency of both clusters. The Student's t-test is a standard statistical method to establish whether or not a difference between two data sets is significant. For this study, a problematic snooping latency is assumed when the p-value of this statistical test is higher than 0.05.

Once execution instances that have snooping latency issues are identified, the second question can be answered using a machine learning model. The goal of this step is to find thresholds on which there is a problematic snooping latency relative to the hardware-level monitoring points of the CPU PMC `memory_bus_access` and CCI PMC `stall cycles`. Since the objective is to find a way to make a quick decision at runtime to detect and mitigate snooping latency at any given time, a decision tree to model the situation is used. Decision trees are relatively simple models which are trained on a set of example data, for which execution instances are used to determine which input values (PMC levels in this case) should imply which output values (increase or decrease clock frequency). Once trained they can be automatically converted to simple C programs of if-else statements for rapid runtime decision-making.

As both cluster frequencies are managed independently, two decision trees are built by considering the CPU PMC of the process of the resident core cluster, and the CCI PMCs of the other non-resident cluster. In other words, when the frequency of the LITTLE cluster is controlled, `stalling cycles` per `read request` on the channel of the big cluster is considered, and vice versa.

Figure 4.3a and figure 4.3b show the trained decision trees to use while managing the frequency of the big and LITTLE cluster respectively. In the trees, the variable *cci_congestion* corresponds to the quotient of CCI PMCs, while *memory_access* corresponds to the sum of processes CPU PMC `memory_bus_access` of applications to optimize. At runtime, if PMC values are over these thresholds, the system is deemed to be in problematic snooping latency territory and CPU frequency may be increased to limit performance loss. Both steps of computing Student's t-test statistic and the generation of the decision trees were carried out using `scikit-learn`[108].

The next section presents the use of this model at runtime as part of the OS frequency management process.

4.3 A snoop-aware frequency governor

This section presents the runtime algorithm for frequency scaling which uses the decision trees derived from the offline modelling procedure. These decision trees are implemented at runtime as a set of `‘‘if-then-else’’` control flow construction. This helps in making almost instant decisions on which clock frequencies to use across both CPU clusters based on the current memory access characteristics of a process of interest, and the stalling behaviour of the inter-cluster cache interconnect.

This section will first present how the default Linux kernel algorithm to manage CPU frequencies is working. And then proposes a refined version which leverages runtime hardware-level information to alleviate snooping latency.

4.3.1 Linux DVFS governor

Algorithm 1: Linux DVFS `ondemand` governor.

```
1 if $load > up\_threshold$ then
2     $cpu\_freq = max\_cpu\_freq$
3 else
4     $cpu\_freq = min\_cpu\_freq + load * (max\_cpu\_freq - min\_cpu\_freq)/100$
5 end
```

To limit the energy consumption of a device when there is no CPU activity, Linux dynamically adjusts the speed and therefore power settings of all CPUs using DVFS. This is implemented in the logic of a module termed a *governor*.

The default governor used in many flavours of Linux is the `ondemand` governor [106] which works simply by adjusting the CPU frequency in direct proportion to the current load of the CPU, where the load level is defined as the amount of time for which the CPU is non-idle during the last sampling period. The algorithm 1 shows the main body of this standard governor [4], which is also very similar to that used with the EAS scheduling framework (`schedutil` governor) on Android.

This governor does not rely on any in-depth hardware-level information, basing its decision-making only on process activity levels over time. As such, the governor would choose the lowest frequency for a cluster when it is idle. However, as shown in the previous section, this decision may induce snooping latency combined with certain memory behaviour for a process running on another cluster.

To avoid this issue on AMP architectures which employ a bus-snoop cache coherency protocol, a more advanced governor which uses real-time hardware-level data is proposed in the next section.

4.3.2 DVFS ondemand-anti-snoop governor

The approach to avoid snooping latency is based on progressively finding the right frequency where the trained model does not detect any latency caused by the snooping mechanism. Using the model presented in section 4.2, a refinement of the `ondemand` governor is built by integrating information from both CCI PMCs and CPU PMCs of processes to avoid snooping latency when the model indicates that it is occurring. The algorithm 2 shows this enhanced `ondemand-anti-snoop` governor, which integrates the decision tree compiled out to C code as `‘‘if-then-else’’` control flow constructions.

[4]The full source code of the governor can be found at `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/drivers/cpufreq/cpufreq_ondemand.c`

Algorithm 2: Enhanced DVFS `ondemand-anti-snoop` governor.

```
1  if $load > up\_threshold$ then
2      $cpu\_freq = max\_cpu\_freq$
3  else
4      $cci\_congestion = update\_pmc\_cci\_congestion()$
5      $memory\_access = update\_pmc\_memory\_access()$
6      if $cci\_congestion > cci\_congestion\_threshold$
7          and $memory\_access > memory\_threshold$ then
8          $stall\_cpu\_freq = cpu\_freq + cpu\_freq\_step$
9      else
10         $stall\_cpu\_freq = cpu\_freq - cpu\_freq\_step$
11     end
12     $cpu\_freq = min\_cpu\_freq + load * (max\_cpu\_freq - min\_cpu\_freq)/100$
13     $cpu\_freq = clamp(max(cpu\_freq, stall\_cpu\_freq))$
14 end
```

This enhanced DVFS governor works as follows. The first step is to update the relevant hardware information by reading PMU counters (lines 4 and 5), then consults the trained decision tree (compiled out to C code, on lines 6 and 7) to determine whether the model suggests increasing the CPU frequency or reducing it for a given cluster. Following this, the procedure calculates both (a) the suggested CPU frequency based on the current system load, and (b) the recommended frequency based on the model, and set the actual frequency to whichever of (a) and (b) is higher (line 13)[5]. This comparison is necessary because the load-based frequency scaling approach may recommend a higher frequency than the model for scenarios in which CPU-intensive processes are running that incur no snooping latency. Likewise, the load-based approach may suggest a lower frequency for a less busy CPU cluster, when another cluster is actually stalling due to cache snooping, in which case the model will suggest the higher frequency.

4.3.3 Implementation details

The proposed approach is implemented directly in the Linux kernel v5.15 by modifying the `task_struct` to allow per-thread CPU PMCs readings. Also, the `ondemand-anti-snoop` governor reads CPU PMCs for each thread that has been active over the last period, and CCI PMCs[6] from the interconnect. Because of this additional periodic monitoring of PMCs, this approach incurs a small continuous overhead that will be discussed in detail in the section 4.4.3. All software and benchmarks have been compiled using GCC v10.2 and run on a Debian 11 Linux distribution.

[5]The `clamp` notation on line 13 includes a calculation of both the valid frequency range of the CPU according to its design specifications, and its maximum actual range advised by the thermal driver, taking into account automated thermal protection.

[6]Support for reading CCI-400 PMCs is not currently available in the mainline kernel source code; support has been added for these PMCs to the Linux kernel and a patch submitted to the Linux mainline maintainer is currently under review.

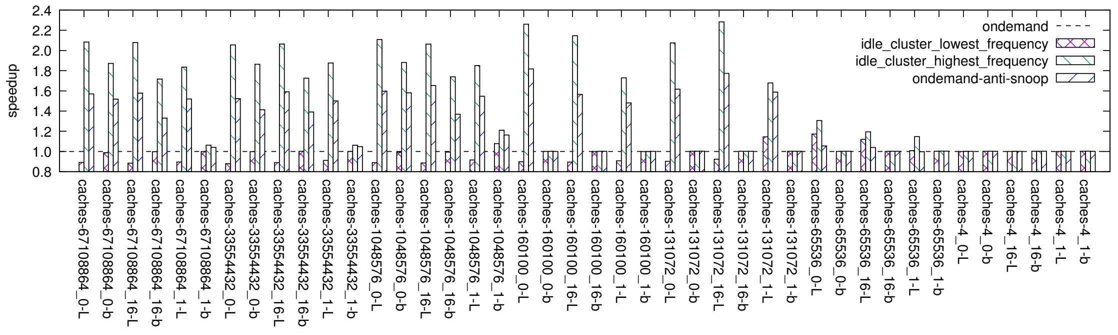


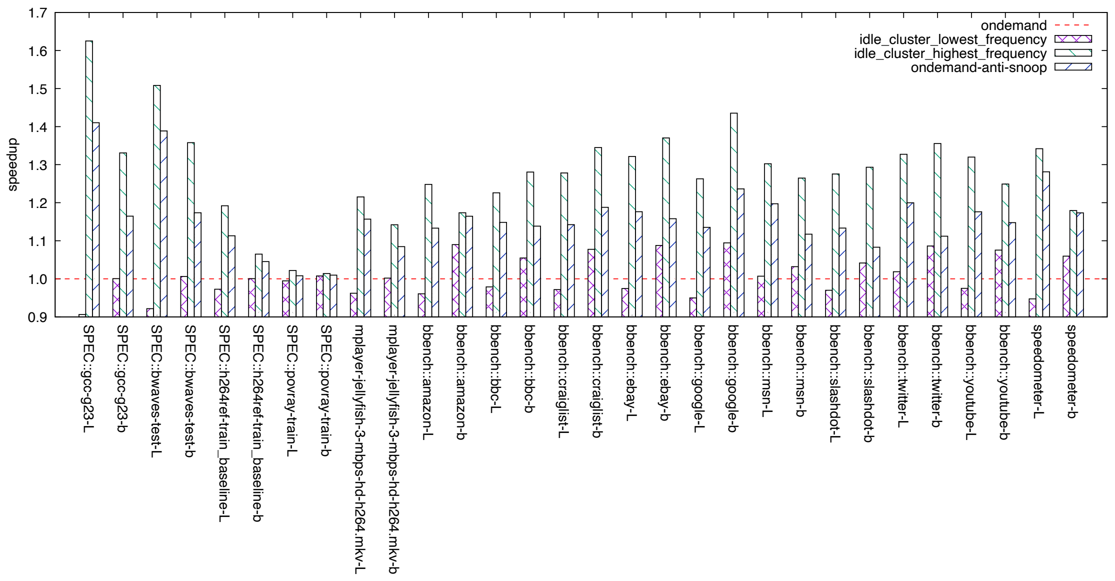
Figure 4.4: Time results on `microbe`, the microbenchmark used to train the decision trees. The baseline is the Linux `ondemand` governor.

4.4 Evaluation

To evaluate the performance of the software solution to avoid snooping latency, this section compares the new `ondemand-anti-snoop` governor against three alternatives using a set of benchmarks (discussed in the following section). The first alternative uses the default CPU DVFS `ondemand` governor and serves as a reference for other comparators. The second one sets the active cluster to its highest frequency while forcing the idle cluster to its lowest frequency. This setting is meant to determine the worst case scenario when snooping latency is not taken into account but exhibits low energy usage. The third one sets both clusters at their highest frequency, achieving the minimal time for the workload at the expense of higher energy costs.

Both metrics of *execution time* and *energy consumption* are considered and each benchmark are executed five times and the average (using a geometric mean [52]) of execution time and energy consumption are taken to normalize noise and reduce the effect of outliers. Energy consumption is measured using onboard energy sensors that measures power consumption where a reading is taken every 263808 μs and accumulated in a variable that represents the energy consumed during the last sampling period. This accumulator is reset before running a benchmark, and the first reading after the benchmark has finished is reported as energy.

(a) Speedup.



(b) Energy saving.

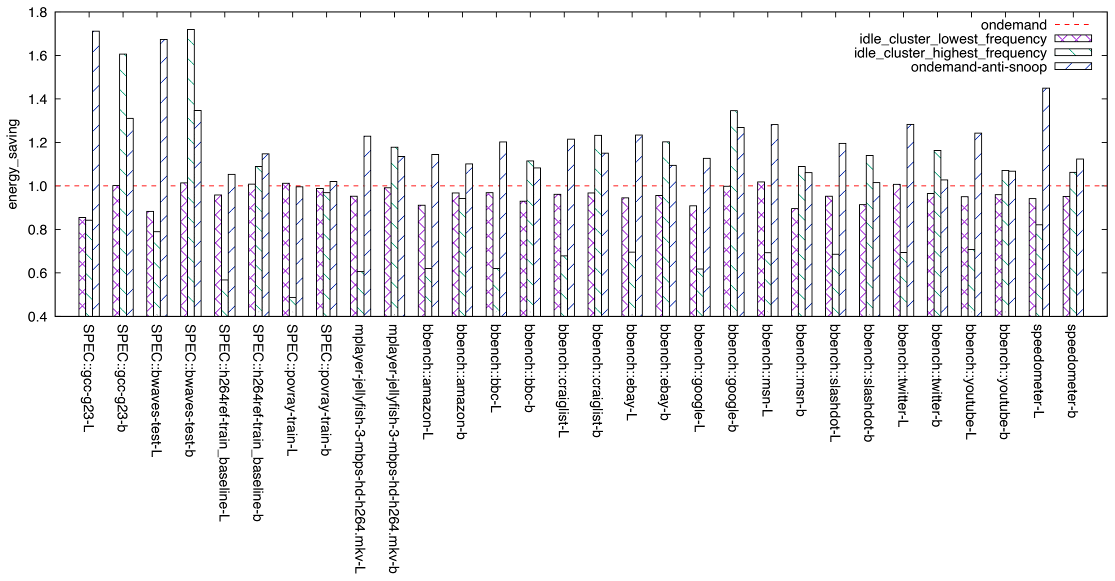


Figure 4.5: Results of experiments on real-world benchmarks, using the Linux `ondemand` governor as a baseline.

4.4.1 Benchmark selection

This work has been evaluated with a set of 15 different benchmarks. This includes popular benchmarks for CPU profiling chosen to demonstrate performance in the best, worst, and average case; and also benchmarks which are representative of the general use cases of mobile devices using this CPU architecture.

To evaluate the new governor on specific conditions that stress decision trees code blocks of the `ondemand-anti-snoop` governor, a set of benchmarks from the standard `SPEC2006` benchmark suite was selected [62]. The selection of specific tests is based on existing research by Jaleel [69] which studies this benchmark suite in detail to characterize it in terms of CPU cache memory misses per 1,000 instructions, a metric termed 'MPKI' which interacts with the hardware features that the new governor is designed to optimize. Benchmark selection comprises the `gcc` benchmark with `g23` input and `bwaves` with `test` input as these benchmarks face high MPKI for both clusters. For comparison, `povray` with `train` input was also selected as it faces very little MPKI. On the other hand, `h264ref` with `train` input is used as its MPKI appears below 2 MB, causing the LITTLE cluster to face high snoop latency while the big cluster should not suffer much. These benchmarks are chosen to provide a clear theoretical picture of the characteristics of the new governor in best, worst, and middle-ground scenarios. Finally, this work was also evaluated on the microbenchmark used to train the decisions trees as discussed in the section 4.2.1, to show how it performs at runtime when using the `ondemand-anti-snoop` governor.

The particular SoC present in the platform used in this work, the Exynos 5422, is mostly used on mobile devices including smartphones and tablet computers. To reflect two of the dominant end-user applications for these devices, a set of standard web-browsing benchmark suites and a video decoding benchmark is used, demonstrating the effects of the new `ondemand-anti-snoop` governor in a realistic end-user setting. In detail, `BBench` [57] and `Speedometer 2.0` [4] are used for web-browsing benchmarks. `BBench` is used to test general web browsing, which performs automatic browsing by loading and scrolling a selected web page. `Speedometer`

`2.0` is used to specifically test JavaScript performance in the browser to model highly interactive websites. Both aspects of this web browser benchmarking are performed using the `Chromium` browser controlled by `puppeteer` [35]. All of the performance measurements that are reported include the launch of `Chromium`, page loading, full JavaScript execution and taking a screenshot of the full rendered final web page. The server-side elements of the `Speedometer 2.0` benchmark are hosted on an isolated local Apache web server serviced by a 1 Gb Ethernet connection. For video decoding benchmarks the standard Linux `mplayer` application with the command-line parameters `-nosound -vo null -benchmark` options is used, using a specific video stored locally on the device and publicly available for replication [2].

4.4.2 Results

The experimental execution time results are shown in figure 4.4 for the microbenchmark used to train the decision trees used in `ondemand-anti-snoop` governor and figure 4.5a for real-world benchmarks. Figure 4.5b shows the energy results for real-world benchmarks. On both graphs the x-axis labels have the format {name}-{input}-{cluster} and show which benchmark name is being used, its input type, and the cluster (b/L) on which the benchmark is executed (the other cluster is kept idle). On both graphs data is reported in terms of how many times better or worse it is compared to the default Linux `ondemand` governor (thereby using this governor as a consistent baseline). All results involving `ondemand-anti-snoop` governor are achieved using real-time monitoring of each process, along with stalling effects on the cache interconnect to dynamically scan for the ideal clock frequencies of both clusters.

The first step of this evaluation is to check how the new governor behaves against the microbenchmark used to train the model as described in section 4.2. On average, in figure 4.4 one can see that the new governor outperforms the `ondemand` governor by 1.4x. `ondemand-anti-snoop` governor is also always at least as good as the `ondemand` governor across all benchmarks, indicating that the inherent overhead of using PMCs at runtime has limited impact on the system compared to its benefits. The varying performance of this governor across specific configurations of this benchmark is simply down to the relative memory intensity and access pattern of each particular configuration – those which incur more L2 cache misses see a higher benefit using `ondemand-anti-snoop` governor.

The next step is to evaluate this work on realistic benchmarks, including those designed to test specific aspects of the new governor and those representatives of common end-user activities. Considering execution time first, across all results, in figure 4.5a one can see that the configuration that keeps the idle cluster at its highest frequency gains the highest performance (at the cost of unnecessary energy consumption). The new governor consistently comes in second, followed by the `ondemand` governor. In some cases, this difference is very significant – against the `SPEC2006` benchmark the new governor delivers 1.4x speedup for the `gcc` test compared to the `ondemand` governor. The exact level of speedup is highly dependent on the memory usage characteristics of the benchmark, with the `povray` test yielding a very minor speedup due to its low level of main memory usage. Examining real-world applications, on web browsing benchmark against a series of different popular web pages gains between 1.1x and 1.25x speedup for page loading, while Speedometer JavaScript tests yield up to 1.3x speedup under `ondemand-anti-snoop` governor.

Considering the energy, shown in figure 4.5b, the graph demonstrates the benefit of `ondemand-anti-snoop` to the overall performance/energy profile of the device. As an example, one can see that `SPEC2006` benchmark's `gcc` test under `ondemand-anti-snoop` saves 1.7x the amount of energy compared to the default `ondemand` governor while also (from the previous graph) completing the benchmark 1.4x faster. This is also useful to compare against the configuration which runs the idle core at its highest frequency: although this configuration completes the benchmark faster than `ondemand-anti-snoop`, it also uses far more energy. However, this knowledge is to be balanced.

When the active cluster is the LITTLE cluster, setting the big cluster (which implements an out-of-order pipeline) to a higher frequency than necessary consumes a significant amount of unnecessary energy.

In the reverse scenario, when the active cluster is the big cluster and the LITTLE cluster (which implements an in-order pipeline) is kept idle, as this last consumes a small amount of energy overall, the amount of wasted energy while running this cluster to its highest frequency is not that significant.

In consequence, the energy saving of the `ondemand-anti-snoop` governor is less than the `idle_cluster_highest_frequency` governor when activated on the big cluster. This is prominent on some benchmarks such as `gcc` and `bwaves`, and visible on other benchmarks such as `mplayer` and `bbench`. The reason behind this is because `gcc` and `bwaves` have significantly more memory requirements than the other benchmarks.

Therefore, the `ondemand-anti-snoop` governor finds a useful balance between performance and energy when employed on the LITTLE cluster. This is demonstrated throughout all of the benchmarks running on the LITTLE cluster, where the `ondemand-anti-snoop` governor offers a significant energy saving over the `ondemand` governor while also yielding higher performance.

Finally, all of these benchmarks are subject to different dynamic activity phases over their execution lifetime and none have been seen before by the trained model. This indicates that this approach deals well with fluctuations of memory usage over time by dynamically adjusting frequencies on a continuous basis. Furthermore, its training on a single set of focused benchmarks around energy characteristics generalizes very well to good performance on a broad range of new benchmarks.

4.4.3 Discussion

This new `ondemand-anti-snoop` governor aims to avoid snooping latency using dynamic hardware-level information from PMCs, along with a simple trained model of how the CPU clusters behave at different frequencies and with different memory access scenarios.

This section discusses the overhead of the proposed approach and its broader implications. The runtime overhead of the governor comes from the fact that it reads PMCs for every individual process (thread) to understand in real-time how the memory accesses of a process of interest interact with CPU performance. The reading of one PMC took between 0.6 μs and 4.7 μs on average depending on which CPU the `kthread worker` and user tasks are mapped. The runtime performs readings of only five CCI PMCs (`cycle`, `read data stall cycle` and `read requests` for both clusters), and two CPU PMCs per task that has been active during the last period (`cycle` and `memory bus access`). This very low update time, coupled with the fact that PMC readings and decision-making do not require any application to stop their execution, even briefly, indicates that the overhead of this approach is far outweighed by the benefits it brings in overall performance and energy usage.

This new DVFS governor succeeds in limiting snoop latency with a pure-software solution. This approach is valuable in any heterogeneous multi-core design in which cache coherency checks are dependent on the relative clock frequencies of each different cluster. In these chip designs, a very simple model can be trained and used to capture the interaction between memory access and frequency, and combine this with real-time monitoring to configure the clock frequencies of all clusters in a processor to an ideal system-wide setting. While some newer processor designs include extra hardware support to aid with cache coherency, in which the interconnect maintains a list of which memory is currently in the cache of each cluster, a large number of existing devices do not have this capability and so will benefit from this approach. A recent study [150] suggests that as much as 75% of today's smartphone population use CPU designs that were released before 2013 and rely on a cache coherence interconnect with no hardware snoop filter support, making this approach very widely applicable across popular end-user devices today.

4.5 Summary

The increasing demand for performance and energy efficiency has led embedded systems such as mobile and tablet devices to employ heterogeneous multiprocessor system-on-chips. The combination of different kinds of core types and frequency configurations helps to fine-tune energy efficiency and/or performance at runtime. Thanks to full data coherency managed in hardware through an interconnect fabric, the software developer can ignore data cache coherency management as threads spread across processors. However, the interconnect fabric on some SoC can cause significant performance drops if processors are poorly configured. The work in this chapter has shown that these performance drops can be attributed to snooping latency which can occur when the software has large amounts of memory traffic *and* CPU frequencies are set too low aiming to save energy.

This chapter has presented an automated characterization of this snooping latency for any SoC that implements ARM CoreLink CCI-400 as its interconnect. A simple model has been built that takes into account hardware-level information in accordance with software memory usage to detect when snooping latency occurs and its extent. This simple model has been used to develop a new `ondemand-anti-snoop` dynamic frequency governor to manage CPU cluster frequencies and avoid snooping latency.

Evaluation of this governor shows that a speedup of more than 40%, with a 70% energy saving, can be achieved versus the default Linux `ondemand` governor on a real-world application. This new governor, based on hardware-level use of PMCs, does not depend on any particular software knowledge or modification to operate and resides directly in the operating system, fully transparent to application software.

This chapter has demonstrated one possible hardware-tuned enhancement to frequency scaling technique on asymmetric multiprocessor where data cache coherency is implemented using a snoop bus protocol; the next chapter will continue this work considering scheduling of applications on such systems.

Chapter 5

Scheduling on single-ISA asymmetric multiprocessing systems for embedded systems: when the temperature comes into play.

It is general knowledge that that high temperature causes unreliability and worse, physical damage to the hardware [82, 137, 92, 128, 68, 97, 153]. On the server and desktop market, machines are dimensioned to allow the use of cooling devices such as heat sinks, fans, or even liquid cooling to restrain the temperature from reaching high levels. However, the hardware considered in this work is of small form factors for pocket or tablet-sized devices and does not have room for active cooling apparatus. If the operating system is not able to limit the temperature, a guard is generally present to shut down the machine as a last resort to prevent permanent damage to the hardware and to the user of the device.

Suggested in section 2.1, the temperature of a CPU is in direct relation to the frequency, voltage, and the work performed by this CPU. To mitigate and protect the device, a simple approach to reduce the temperature of the CPU once it reaches a certain level is to reduce either the frequency, the voltage, the core activity, or a mix of them.

Generally, the manufacturer provides a voltage table for safe operation of the CPU operating at its different configurable frequencies. As this table already suggests the minimal voltage required to operate the CPU safely, there is no sensible optimization to bring by tweaking this parameter. However, there is plenty of room to optimize the temperature by scaling the frequencies and core activities by application scheduling.

The remainder of this chapter is as follows. Section 5.1 presents simple strategies to limit the temperature and their drawbacks regarding performance by touching frequency and core activity in isolation. Section 5.2 explores thermal management on AMP systems using the standard solution for Linux/Android operating system and its limitation. This section also introduces `TBASS`. `TBASS` is a new solution that proposes a unified approach to alleviate drawbacks and limitations discovered in the current state-of-the-art thermal management and scheduler in Linux. Both default Linux scheduler+thermal manager and `TBASS` will be compared on multithreaded workloads taken PARSEC-3.0 and Splash-3 benchmark suite in section 5.3. Finally, section 5.4 concludes this chapter.

5.1 Approach to reduce chip temperature

To understand the performance impact on applications and effectiveness in mitigating the temperature, the following sections study different strategies by manipulating the frequency and core activity of the CPU in isolation. Also, this section is treated as a foundation to understand different aspects of thermal management and consider homogeneous systems where each CPU core deliver the same computing performance. These different strategies will be combined and confronted to target heterogenous systems in section 5.2.

5.1.1 A simple approach for frequency scaling

Frequency scaling is a well-known technique and is common in operating systems. For instance, when the temperature of a CPU core increases above a specified threshold, the thermal manager of the kernel reduces the CPU frequency, to eventually increase it back when the thermal emergency is mitigated.

In Linux (and by extension Android), the thermal manager reduces the OPP of the chip by one step. Once the chip temperature has dropped below the threshold, the OPP of the chip is increased to improve the performance. This strategy is designated as the `step_wise` thermal governor and is presented in listing 5.1.[1]

Figure 5.1 shows a trace demonstrating the `step_wise` thermal governor in action. As can be seen, this simple strategy is effective in limiting the temperature to reach a critical level. However, there are multiple issues in its definition.

The first issue is that this simple thermal policy does not force the temperature to stay below the thermal limit. As such, it is possible to still reach and stay at a high temperature and face instability. For instance, during a workload burst, the temperature could reach a very high temperature. If after scaling down by one step the temperature becomes stable, no further cooling action is taken and thus the temperature is not reduced. To alleviate this issue, the `ENFORCE_STRICTNESS` option in listing 5.1 could be considered to enforce strict consideration of the thermal limit.

Another issue of this strategy is that it considers only temperature mitigation of the single CPU core covered by the thermal sensor, and does not consider the performance of the system as a whole. As such, when the frequency of a particular core is shared with other cores, this strategy will reduce the performance of all sibling cores. Moreover, other parts of the system may be impacted by the operating frequency of particular CPUs. For instance, chapter 4 studied a system where memory communication latency is tightly coupled with CPU frequency. Thus, when

[1]Linux source code of the `step_wise` thermal governor could be found at `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/drivers/thermal/gov_step_wise.c`

```c
void thermal_governor_stepwise_principle()
{
    while(true) {
        bool req_step_down = false;
        bool req_step_up = false;

        for_each_thermal_zone(tz) {
            if (tz->temperature >= th_limit) {
                switch (tz->trend) {
                case THERMAL_TREND_STABLE:
#ifdef ENFORCE_STRICTNESS
                    req_step_down |= true;
#endif /* ENFORCE_STRICTNESS */
                    break;

                case THERMAL_TREND_RAISING:
                    req_step_down |= true;
                    break;

                case THERMAL_TREND_DROPPING:
                    break;
                }
            } else {
                switch (tz->trend) {
                case THERMAL_TREND_STABLE:
                case THERMAL_TREND_RAISING:
                    break;

                case THERMAL_TREND_DROPPING:
                    req_step_up |= true;
                    break;
                }
            }
        }

        if (req_step_down) {
            freq_step_down();
        } else if (req_step_up) {
            freq_step_up();
        }

        sleep(polling_time);
    }
}
```

Listing 5.1: Thermal manager implementing a simple frequency scaling approach.

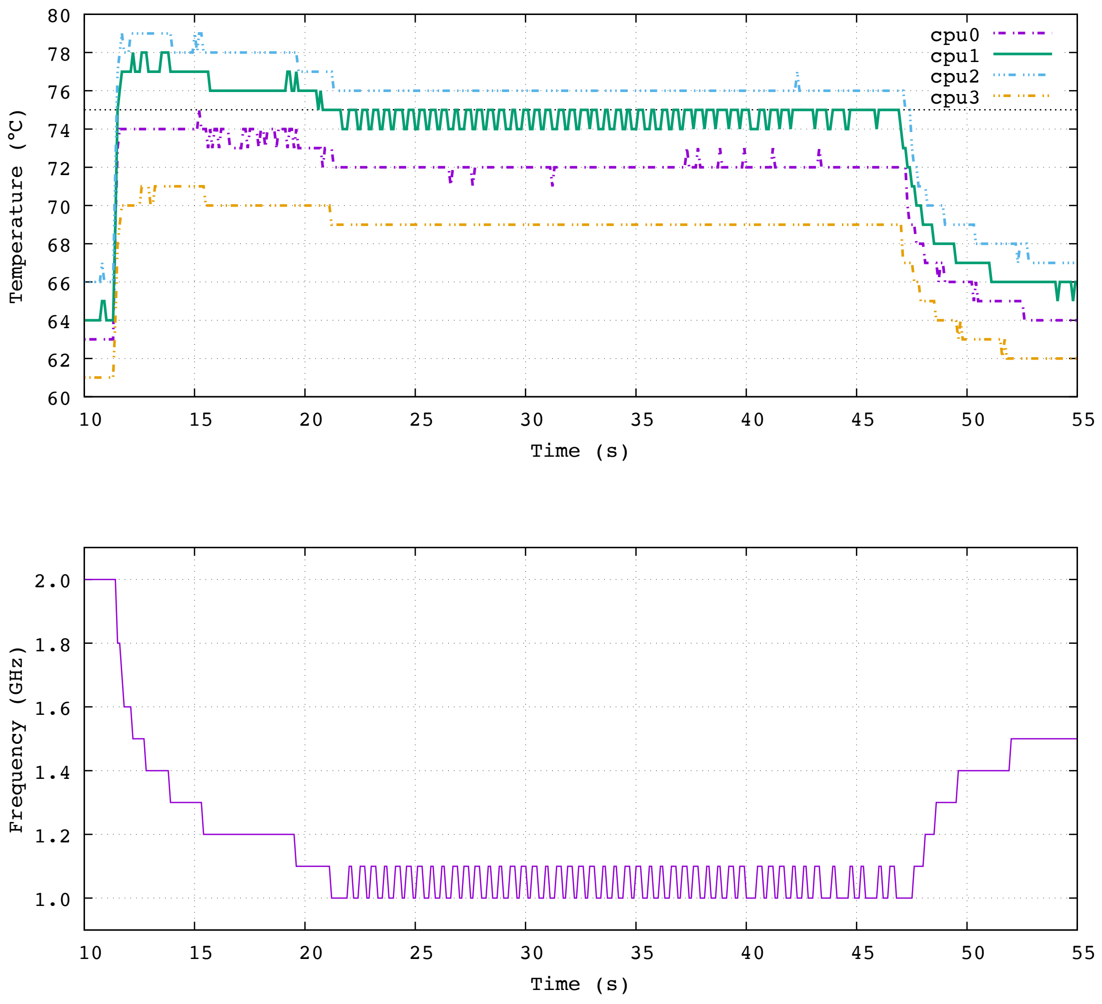


Figure 5.1: A workload using four threads under the `stepwise` thermal governor. The thermal limit is set to 75 °C. This thermal governor reacts considering the temperature trend, but does not enforce a strict thermal limit. When a temperature trend is stable but over the thermal limit, the governor does not react.

this aspect is not considered, this strategy in isolation could impact the performance of the entire system.

Lastly, this strategy does not consider any scheduling decision. Under Linux, the Completely Fair Scheduler (CFS) [88] is generally employed to manage the placement of tasks on the CPU. This scheduler is used to maximize overall CPU utilization while also maximizing interactive performance. However, it does not consider the thermal situation of CPU cores and relies on the thermal governor to manage device temperature. If the scheduler decides to use repeatedly hot cores, frequent cooling action has to be taken, resulting in performance degradation.

Unfortunately, the current situation of dissociating decisions from the scheduler and the thermal governor impacts both performance and thermal balance. Later in this chapter, a unified approach of frequency scaling and scheduler is proposed to maximize application performance while respecting a safe thermal operating range.

5.1.2 A simple approach to reduce core activity with SIGSTOP & SIGCONT

The POSIX standard allows to stop and resume any process by the use of signals. `SIGSTOP` suspends the process while `SIGCONT` resumes it. This mechanism could be used to reduce CPU core activity by suspending running tasks that are mapped on a CPU core which has its temperature above the thermal limit. Eventually, tasks are resumed when the temperature drops below the limit. When all tasks are suspended, frequency scaling could be used to reduce further CPU power consumption and eventually reduce the temperature.

One of the main drawbacks of this strategy is that it stops any progression of the suspended tasks. In a multithreaded context, this can have a particularly undesirable effect. For instance, in an application where a thread that has the role of a controller for multiple workers in a thread pool is mapped on the hottest core, and assuming that all worker threads get to a point where they are unable to proceed any further without a signal from the control thread, the application may lock for a very long time until the hottest core cools down enough to resume progression.

Other disadvantages of this strategy lie in its inability in reducing core temperature. Supposing there is a high thermal coupling between two cores. Considering two applications, one application may run on a core that is likely to get hot because of thermal coupling. This application may be stopped for a very long time if the other application is sparsely suspended. Another consideration is if the ambient temperature is warm, the core temperature may cool down at a very slow pace. These situations could be dramatic for performance.

5.1.3 A simple approach to reduce core activity with task migration

Another approach for reducing CPU core activity to limit the temperature is to migrate running tasks from hot cores to colder cores. A key point of this approach is to find a mapping that could be kept longer before re-engaging task migrations (limiting overhead of cold CPU cache) or involving other techniques to mitigate heat generation.

A simple heuristic to do task mapping is to assume that the more work the task performs, the more the task generates heat. As such, the load (i.e., the percentage of time the task has run during a scheduling time window) is generally a first metric to consider. With deeper insight and hardware support such as performance counters, Instruction Per Cycle (IPC) could be used to determine a mapping. Thus, a mapping such as assigning in descending order the hottest tasks to the ascending order of colder cores could be used. If there are no colder CPU cores available, whether all cores are used in a multithreaded context or temperatures of all free cores already exceed the thermal limit, other techniques are required to reduce the temperature.

The trace in figure 5.2 shows this strategy using a multithreaded application with 3 active threads. In this experiment, the criterion used to determine tasks' hotness order is $((instruction * 1000)/cycle) + 100 * load$. The rationale behind this criterion is that it is assumed that the more instructions per cycle, the higher probability of

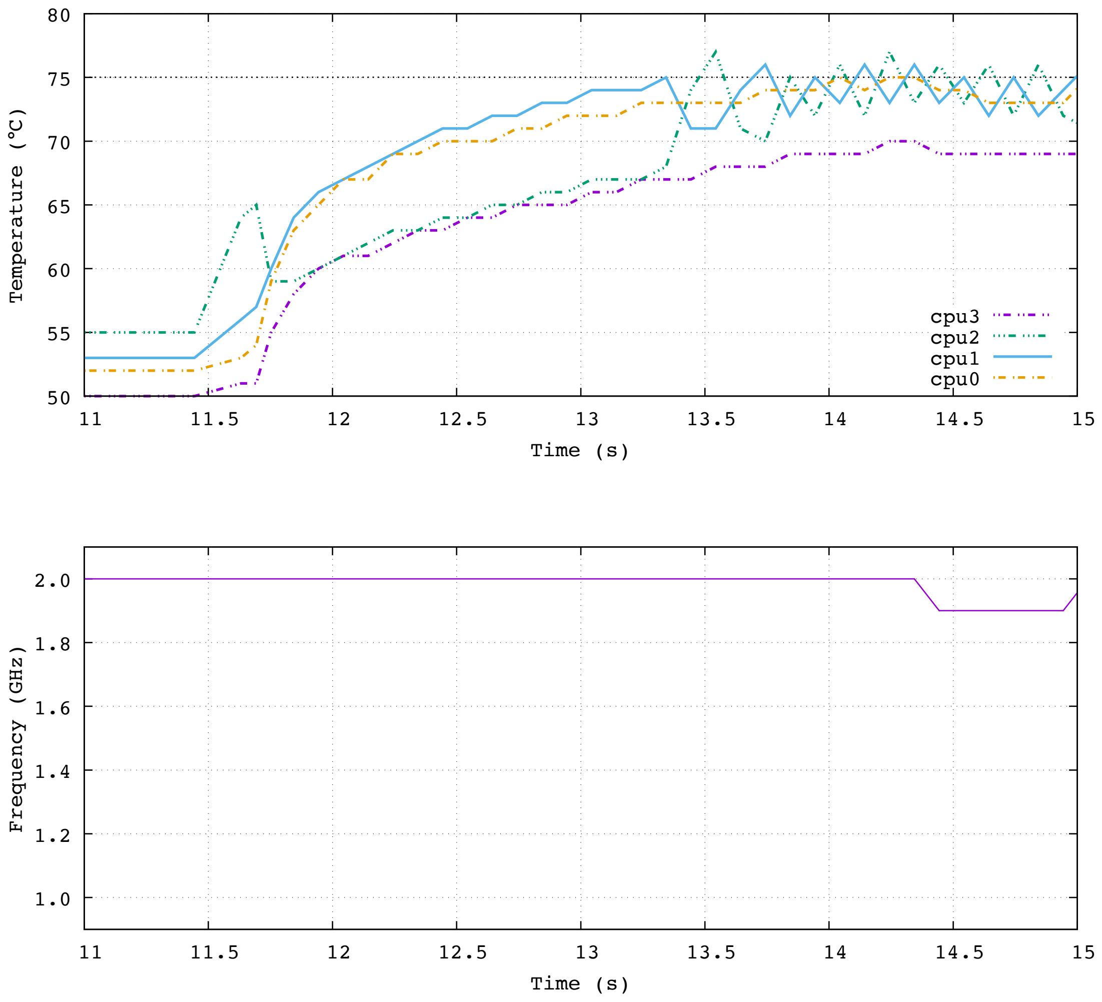


Figure 5.2: A workload using three threads under a thermal governor that does task migration first, followed by frequency scaling if optimal task mapping is assumed to be found. The thermal limit is set to 75 °C.

generating heat subsits and the longer a thread is running, the more it contributes to generating heat. Also, as the actual code is directly in the kernel, a factor of 1000 is used to keep enough precision in fixed-point arithmetic. When the best mapping is found per this heuristic but core temperature is still above the limit, frequency scaling is used.

There are two main problems about this strategy. The first problem is that the instruction performance counter is an aggregation of multiple micro-architecture operations. As shown in figure 3.1 in section 3.3 from the chapter 3, a program that performs one `add` instruction per cycle will have the same value of IPC as a program that performs one `mul` instruction per cycle. However, these two instructions involve different physical part of the chip resulting in `mul` instruction drawing less power than `add` instruction. Hence, this heuristic will not accurately discriminate the CPU burning tasks. Nonetheless, consideration of the load metric mitigates this inaccuracy by giving more weight to the running period of the application. The second problem is that this strategy assumes that all CPU cores deliver the same performance during task placement. Hence, in the context of heterogeneous architecture, as energy efficient CPUs draw less power (i.e., has cooler temperature) than performant CPU, this strategy will favour an energy efficient CPU whereas a performant CPU would be a better option performance-wise.

This section has demonstrated the complexity of thermal management, where each simple approach has significant shortcomings. In the next section, a new scheduling and thermal strategy that combines these simple approaches in a unified manner to target such heterogeneous system will be presented.

5.2 Thermal management on single-ISA heterogeneous multiprocessing systems

On AMP systems such as Arm big.LITTLE [55], not all cores deliver the same performance at the same core frequency. In this context, when employing a task migration approach to mitigate the temperature, the choice of a cooler CPU core becomes non-trivial.

This section presents two schedulers that exploit the heterogeneity of the CPU design to maximize energy efficiency and performance. The first scheduler presented is the Linux standard scheduler (CFS) and its enhancement introducing the knowledge of hardware performance asymmetry of different CPU type (CAS) with its variant that integrates energy consideration (EAS). This scheduler is used by default on smartphones equipped with heterogeneous architecture in recent Linux and Android kernels (Linux mainline v5.0 and android-4.4). The heart of its design is to fully consider the computing performance of individual CPU cores on a heterogeneous system and to deliver energy-efficient scheduling for each task with a minimal impact on throughput. However, it does not take into consideration the thermal characteristics of the device and may fail to deliver its full performance potential.

To address this flaw, a new scheduler is proposed that introduces thermal knowledge in its design and improves application performance while near the thermal limit. The scheduler favours the use of cold cores to limit thermal cooling action in the long term. Aside from this approach, it also takes a step further and suggests a surprising strategy of not using the most performant CPU core when available, even if its core temperature is below the thermal limit. This strategy is employed in the hope of using the CPU at a higher frequency to improve the overall performance of the device.

5.2.1 Capacity-Aware Scheduling and Energy-Aware Scheduling; Linux CFS on heterogeneous platform

Since version 2.6.23 of the Linux kernel, Completely Fair Scheduler (CFS) becomes the default scheduler [86]. In simple words, CFS tries to assign a fair proportion of

CPU processing time for each task on an “ideal, multitasking CPU”.

To do so, the scheduler uses the concept of a “load” metric, which is a metric of time duration, weighted by priority (the `nice` value). As on a real hardware, true multitasking is not achievable (i.e. on a CPU core, there is only one task running at any single time), the scheduler uses the concept of virtual runtime (`vruntime`) to determine the amount of time that a task would have run if there were no other tasks running.

The scheduler uses a time-ordered red-black tree to represent tasks that are to be executed on a CPU core (there are as many trees as CPU cores). In this tree, a node represents a task, and the task `vruntime` is used as an index in the tree, such as the leftmost node in the tree has the lowest `vruntime`, and this task deserves the most to execute when the currently running task goes to sleep or the scheduler timer expires. When the currently running task is preempted or goes to sleep, its `vruntime` is increased by the amount of time it had run, adjusted with its `nice` value.

In general, processes do not have a fixed and constant activity throughout their existence, and it is a possibility that cores might become idle when there are still work to do, which impact performance. As such, on a multiprocessor system, to improve application throughput, when a core becomes newly idle, the scheduler will try to pull tasks from other cores. If there are no pullable tasks, the core will enter on idle state (`C-state`). Also, the scheduler will regularly rebalance these trees which will be discussed later in this section.

Considering that each CPU cores are equal, CFS tries to spread, as much as possible, runnable tasks amongst CPU cores to maximize applications throughput. On heterogeneous platforms such as Arm big.LITTLE, it is wrong to consider all CPU cores equal. To make efficient use of this hardware topology, a capacity model is introduced in CFS that relates performance of a CPU in relation with other CPUs.

The model is constructed as follows: with frequencies of all CPUs set to their maximum, a workload is executed on each CPU core individually, producing a

$perf\_workload$ value per CPU core. Then, the `capacity` of a CPU core is computed as:

$$capacity_{cpu} = \frac{perf\_workload_{cpu}}{frequency(cpu)} * \frac{1024}{max(perf\_workload)}$$

For instance, considering a hypothetical dual processor where the first CPU `A` can retire 2 instructions per cycle, the second CPU `B` can retire 1 instruction per cycle and both running at the same fixed frequency, the CPU `A` will have a `capacity` of 1024, while the CPU `B` will have a `capacity` of 512. In practice, there is no "standard" workload to determine the `capacity`. However, it is suggested to use the `Dhrystone 2.0` benchmark [47].

With this addition, CFS becomes Capacity-Aware Scheduler (CAS) in the Linux literature [85] and introduces the concept of task "utilization" (`task_util`). `task_util` is a percentage meant to represent the throughput requirements of a task and can be considered as the task's duty cycle during a scheduling period. For example, in a typical homogeneous system, a task that has 100% of `task_util` is a task that never sleeps, while 10% suggests a small periodic task that spends more time sleeping than executing. To cope with dynamic hardware performance, this metric is also capacity and frequency agnostic and is computed as:

$$task\_util(p) = duty\_cycle(p) * \frac{curr\_frequency(cpu)}{max\_frequency(cpu)} * \frac{capacity(cpu)}{max\_capacity}$$

where `duty_cycle` is the ratio of the running time of the task `p` against the scheduling period; `curr_frequency`, `max_frequency` and `capacity` denote the current frequency, the maximum frequency and the capacity of the CPU core to which the task is currently assigned to, and finally `max_capacity` is the maximum capacity of the system, which is always 1024. Hence, `task_util` delineates the task utilization as if the task were running on the most performant CPU in the system at the highest frequency. Like the CPU capacity model, the scale of this metric spans from 0 to 1024.

Energy-Aware Scheduler (EAS) [87] can be considered as an extension to CFS-CAS in the sense that it replaces task placement decisions during the wakeup path (i.e. when a task becomes runnable after a sleep, mutex release, IO operation, etc.). EAS uses the capacity model, with the addition of an energy model to make task placement decisions. The energy model is based on static information of energy consumption of each CPU core at each frequency (`P-state`). During the wakeup path, the newly awaken task will be migrated if EAS finds a better mapping that reduces energy consumption without harming the systems throughput.

If during execution, any of the CPUs have a task where its `task_util` is higher than 80%, the system is considered as "overutilized", EAS is deactivated and the scheduler will regularly try to readjust task mapping to maximize the performance of the system with a load balancing procedure. To do so, the scheduler will use `task_util` and `load` metrics to balance tasks amongst CPUs to maximize system throughput and keep execution time fairness amongst tasks. Using the `task_util` metric, the scheduler will try to place tasks on CPU cores with enough spare capacity (i.e. `task_util(p) < capacity(cpu)`).

On top of that, a new mechanism is introduced that let the user clamp a `task_util` value for particular tasks. This can be used to reduce a `task_util` of a busy loop task (which should have 100% utilization), or on the contrary, boost a `task_util` of a small periodic task (e.g. 10% utilization) as to guide the scheduler and map tasks on a particular CPU type. For instance, this can help in forcing a small periodic latency-critical thread in a video game to run on the big cluster and improve its frame per second. Another use case is to force a non-sensitive background task to run on the LITTLE cluster and leave the big cluster to more important tasks for the user experience.

With the use of capacity and energy models, the scheduler can efficiently operate heterogeneous systems by scheduling tasks on an appropriate CPU type considering task computing demand and improve user experience. However, this scheduler does not consider CPU core temperature. As such, with the lack of core temperature knowledge, tasks may be assigned to cores that are already above the thermal limit. Thus, thermal actions such as performance degradation with frequency scaling may be taken often and could lead to poor performance.

The next section will elaborate more on the subject and propose a new scheduler that uses thermal information during task placement to maximize performance when the device is under thermal pressure.

5.2.2 Thermal Balance Aware System Scheduler

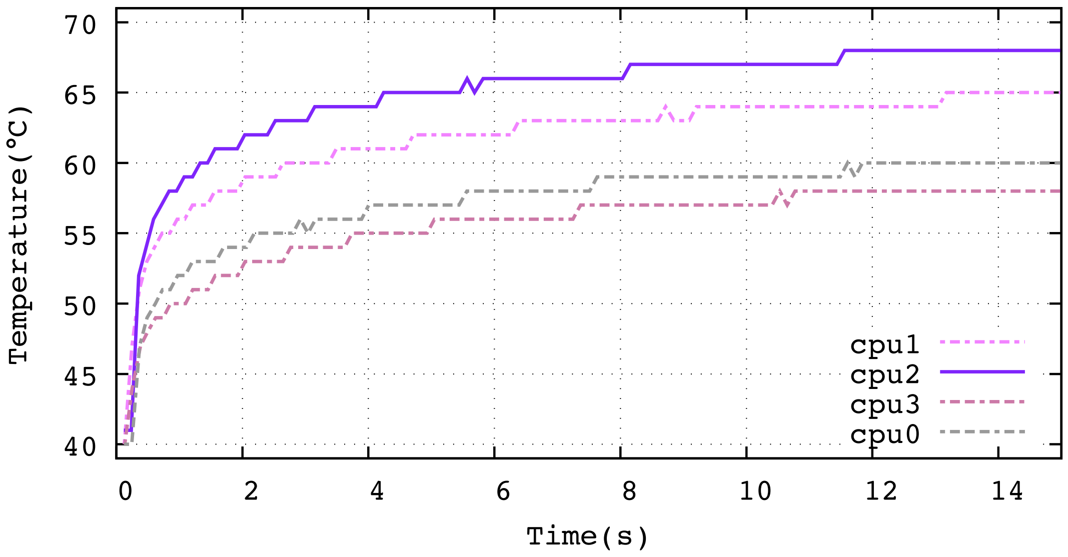


Figure 5.3: Thermal trace superposition of separate execution of `CoreMark` on each core. Peak temperature and increase rate are unique to each core.

As the manufacturing process and semiconductor are not perfect, process variations are frequent and tend to be more pronounced as process nodes shrink [34]. Process variation has been identified and studied in multiple prior works [46, 16, 65, 80, 155]. Similarly, the floorplanning of a design does affect CPU core thermal temperature [99, 36, 43, 117, 83].

The device used in this work is not protected from these effects. On the particular hardware used in this study, a Hardkernel Odroid-XU3 board that uses a Samsung Exynos5422 SoC[2], it appears that during a stable and steady thermal hot situation,

[2]The SoC used on this board, a Samsung Exynos 5422, is also present in the Samsung Galaxy

two CPU cores are relatively hotter than others by more than 5°C (see figure 5.3). If during execution and in a situation where all cores are near the thermal limit, avoidance of using these two CPU cores may lead to a higher CPU frequency resulting in higher application performance.

This section analyses this peculiar unbalanced thermal behaviour present in hardware in deeper detail. A new scheduler will then be proposed that exploits this behaviour. This new scheduler considers avoidance of CPUs that are likely to push the device into thermal imbalance to maximize CPU frequency in the hope of increasing overall application performance.

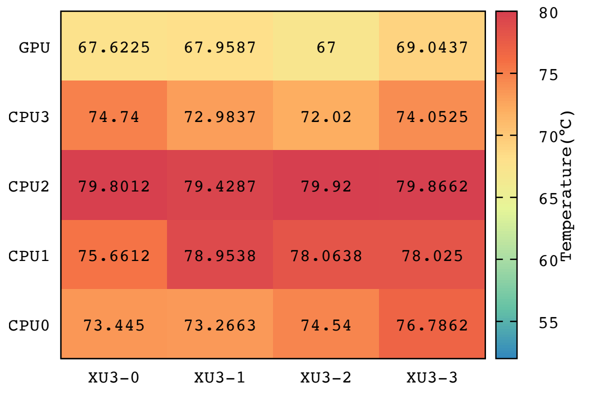
5.2.2.1 Imbalanced thermal behaviour

Like EAS, the proposed scheduler uses the capacity model presented above to make scheduling decisions. However, contrary to EAS, task placement is achieved using thermal information and makes the choice of not using performant CPU cores if they are likely to cause a high imbalance between CPU cores' temperatures depending on the current thermal situation.

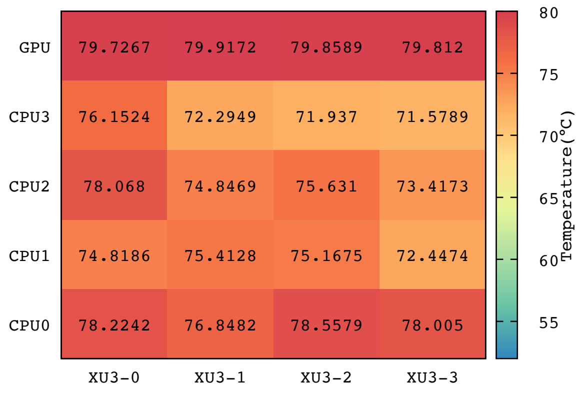
To do so, the scheduler employs a thermal heatmap model that helps identify cores that are likely to be hotter than others. The thermal heatmap model is derived during an offline stress test that tends to push the device above its thermal limit to discover peak temperatures of each CPU core and the GPU. To achieve good accuracy and avoid damaging the device, all experimentations are conducted with the device inside a thermal chamber where the thermal environment is under control.

The stress test consists of a synthetic workload written in assembly tailored to the specific CPU and performs instructions repeatedly without pause in the pipeline. To stress the GPU alongside, the benchmark `terrain` from the `glmark2_es2` OpenGL benchmark suite [54] is used as it shows the highest energy consumption and temperature increase of the benchmark suite. Figure 5.4 shows average CPU cores and GPU peak temperature (along the y-axis) on multiple boards (along the x-axis)

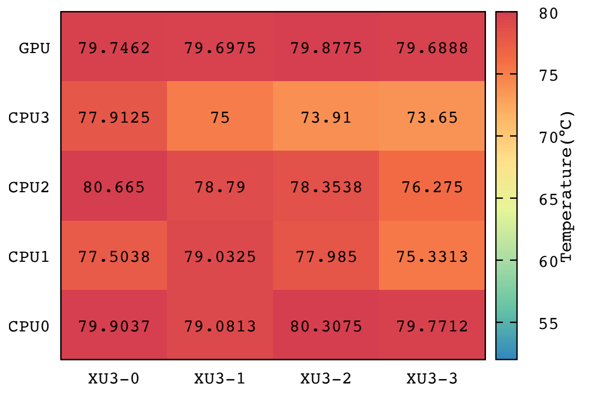
S5 smartphone as well as other devices.



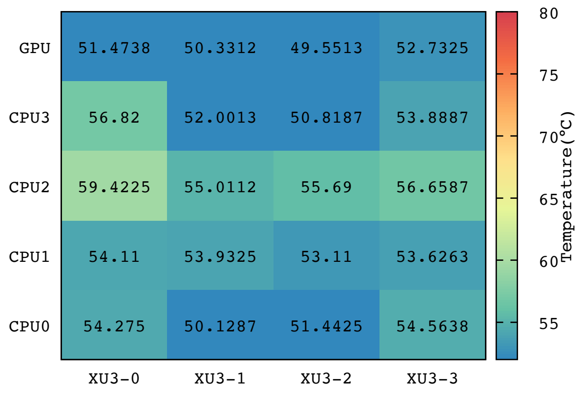
(a) CPU



(b) GPU



(c) CPU+GPU



(d) idle

Figure 5.4: Average core temperatures of the SoC while running a stress test on different boards while stressing only the CPU (a), the GPU (b), and both CPU+GPU (c). The SoC is idle in (d) with both CPU and GPU set to their minimum frequencies.

Even for the same board model revision, each computing core has its own thermal characteristics due to manufacturing process variations and thermal coupling.

using the stress test during steady temperature after 1 minute of execution[3]. In this experiment, figure 5.4a (figure 5.4b) shows the stress test with CPU (GPU) only with the GPU (CPU) frequency set to its minimum; in figure 5.4c both CPU and GPU are set to use their highest tolerable frequencies.

Overall, core 3 is always the coolest core. When the CPU is used while the GPU is idle 5.4a, cores 1 and 2 are the hottest CPU cores. When the GPU is used while the CPU is idle 5.4b, it can be seen that core 0 has the highest temperature. This indicates a strong thermal coupling between the CPU core 0 and the GPU. While both CPU and GPU are used at the same time 5.4c the strong thermal coupling is emphasized. It is interesting to note that CPU cores 1 and 2 are not the highest temperature in this last case.

The next section presents a new scheduler that takes this thermal behaviour into consideration.

5.2.2.2 Thermal Balance Aware System Scheduler: overview

The key point of this new scheduler is to exploit the imbalanced thermal behaviour between CPU cores, heterogeneity in the CPU design with its asynchronous CPU cores' computing and memory capacity of the device. This allows the scheduler to run all cores at higher frequencies for longer while balancing the different needs to CPU-bound vs. memory-bound tasks in the context of asynchronous CPU core characteristics.

The current Linux `step_wise` thermal governor tries to reduce the highest temperature sensed. On the board used in this study, each core within a cluster shares the same frequency. Hence `step_wise` will penalize all tasks that are running on the cluster, even if the temperature of a particular core is below the thermal threshold.

Rather than penalizing all tasks running on a cluster, this new scheduler takes a chance not to use cores from the cluster that are likely to get hotter than others

[3]At this time of execution, both CPU and GPU are throttled because of the temperature.

on the same cluster by deporting tasks on other clusters. In particular, this new scheduler takes a chance not to use cores from the big cluster that are likely to get hotter than others by dispatching tasks onto the LITTLE cluster and keeping these hottest cores from the big cluster idle. By doing so, it is possible to run the big cluster at a higher frequency. In other words, the scheduler increases raw CPU performance for fewer core. However, employing this strategy should not be taken naively as it may result in reduced performance. For instance, when the big cluster is above the thermal limit, but its computing capacity is still higher than the LITTLE cluster, it is preferable from a performance point of view to decrease the computing capacity of the big cluster than to dispatch onto the LITTLE cluster. Another problem could occur in the context of multithreaded applications (e.g. threads could suffer from memory communication if they do not reside on the same cluster). Hence, the scheduler deports tasks to the LITTLE cluster when both clusters reach seemingly the same performance considering running tasks.

To model differences in the computing performance of heterogeneous platforms, CFS-CAS-EAS uses a computing capacity model which is fixed at boot time of the Linux kernel and kept static for the lifetime of the running system. This model is derived using a single benchmark during an offline procedure. The documentation provided by Arm to extract this capacity model suggests the use of a benchmark with a small working set size, so that the executable binary and the dynamic data during its execution could both reside in the CPU cache. In particular, the documentation recommends `Dhrystone 2.0` as a benchmark [47]. This enables an easy and quick setup to compare CPU micro-architecture performance features. However, in general computing, this procedure fails to apprehend applications that use memory extensively that does not particularly fit in CPU caches [141].

To alleviate this concern, the new scheduler proposes the use of a dynamic computing capacity model that is adjusted at runtime considering running applications. The next section describes an approach to determine a dynamic model at runtime using performance monitoring counters.

5.2.2.3 Thermal Balance Aware System Scheduler: deeper details

In this section, the new scheduler is explained in detail.

The proposed new scheduler uses a new modelling procedure and adopts a dynamic computing capacity model that considers dynamic information during the execution of applications using Performance Monitoring Unit (PMU) counters and their memory requirements. This new approach enables captures dynamic information that helps to determine if the LITTLE cluster is good enough to execute a particular program with its dynamic behaviour. The scheduler uses two informations to make task mapping decisions. It first determines if the code flow could be executed on the LITTLE cluster without losing performance considering the current frequency of the big cluster. Secondly, it uses the knowledge of the application memory requirements. The following describes these two aspects of the scheduler.

5.2.2.3.1 Scheduler hint: application code flow

While not a being requirement in a big.LITTLE design, the big cluster generally employs a CPU that uses an out-of-order pipeline, whereas the LITTLE cluster uses an in-order pipeline. Other differences reside in the amount of memory in their CPU caches, with the LITTLE cluster benefiting from far less CPU cache memory than the big cluster.

The benefit of an out-of-order pipeline over an in-order resides in the capability of dynamically reordering instructions to exploit instruction-level parallelism and hide latency of some operation. For instance, when the program performs a memory operation, while an in-order pipeline has to stall execution until the memory operation has completed, an out-of-order pipeline has the possibility to execute instructions that are independent from the memory operation [61].

On the hardware used in this study, instruction fetch, decoding and dispatch stages of the pipeline are performed in-order, while instruction issue (i.e. execution) is performed out-of-order on the big cluster (Cortex-A15). Once the instruction is executed, the result is placed in a retirement buffer to eventually be retired from the instruction queue in the original program order (i.e. in-order retirement). In this architecture, the out-of-order execution is said to be speculative. The hardware includes PMC to count retired instructions, as well as the number of instructions speculatively executed [9].

In the general sense, assuming an identical instruction issue width, prefetch unit (i.e. memory prefetching and branch predictor), and memory subsystem characteristics (i.e. cache size, fetch latency, etc.) between different CPU micro-architecture; when the number of retired instructions is equal to the number of instructions speculatively executed, it could be assumed that the out-of-order machinery has not been involved and an out-of-order pipeline has no benefit over an in-order pipeline. Concretely, this is not exactly true because of the conditional execution of the instruction block after a branch. It is possible that results of speculated instructions could be discarded due to miss predicted branch, resulting in more instructions being speculatively executed than retired. With an in-order pipeline, the execution of the instruction block would have been stalled until the result of the conditional branch has been computed. However, if the scheduler is able to account for the number of discarded instructions speculatively executed, this metric can be refined and used to better predict code flow execution performance on an in-order pipeline that implements identical superscalar front-end and memory fetching latency.

To discover the specific condition of PMC values to determine when the scheduler can place applications on either the big or the LITTLE cluster efficiently considering their current performance, a set of micro-benchmark tailored to stress the out-of-order pipeline design has been developed. In particular, the set of micro-benchmark exhaustively explores the memory fetch latency and the number of independent instructions that could be performed in parallel while performing memory operations.

The micro-benchmark consists of memory operation (load and/or store) instructions with a succession of independent computing instructions. The set of computing instructions is defined to stress superscalar pipelines. The CPU used in this study (Cortex-A7 and Cortex-A15) can issue at most three instructions in one cycle, comprising at most two integer instructions (Integer ALU and Shifter), one `mul` or `div`. The A7 performs one NEON/SIMD instruction while the A15 can perform two instructions of this kind. The A7 can execute two load instructions (without store instruction in between, and only 32 bytes per load). The A15 can dispatch one load at a time, but the out-of-order pipeline is capable of dispatching another load at the next cycle, for a total of 11 memory readings in flight (data from an `ldr` instruction, but also TLB intermediate table fetch).

The micro-benchmark principle is rather simple, it consists of two loops, with the inner-loop as the main instruction executor and the outer-loop acting as a loop execution multiplier. At runtime, the inner-loop iterator is set to run for one second as to be easily multiplied for a determined execution time. The body of the inner-loop is written in assembly to restrain compiler optimization, and unrolled to limit loop overhead (i.e. increment upgrade and branching).

Figure 5.5 shows different instantiations of the micro-benchmark with different blocks of instructions after a memory access that does a pointer chasing. The pattern of the pointer chasing is fully random (i.e. non-prefetchable) over an array big enough to force off-chip memory access for every loop iteration. The $x - axis$ represents the number of repetitions of the instruction block in the loop. For example, `code_1` to `code_5` are instructions without memory access. Because each of these codes can be retired directly without reordering, 100% of speculative instructions are directly retired.

`code_10` to `code_50` represent the same code as above, but with a memory access performing a pointer chasing. The memory pattern of the pointer chasing is setup to be non-prefetchable, and forces off-chip memory access at each iteration. More precisely, in `code_10`, the independent instruction from the memory access is a

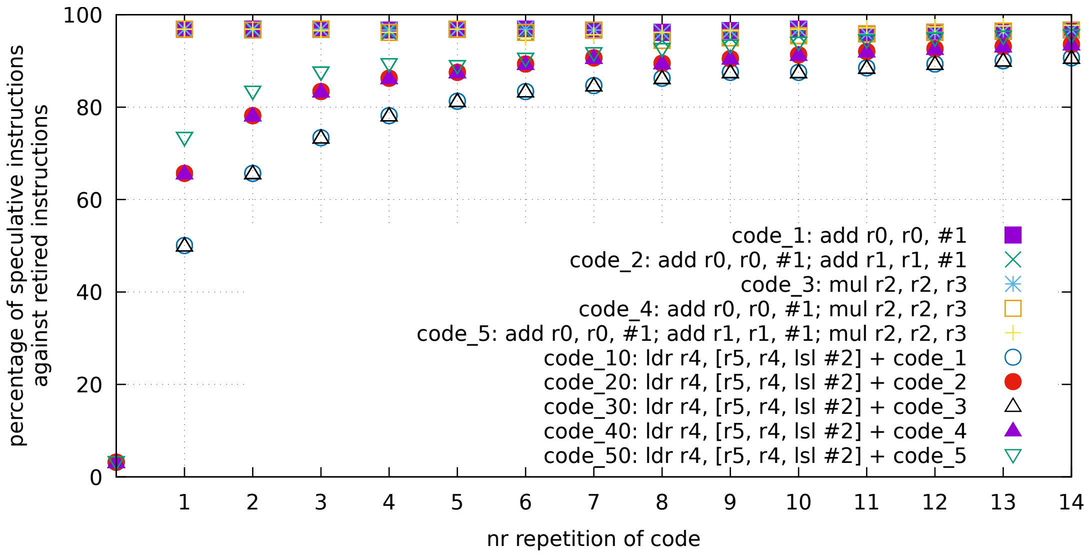


Figure 5.5: Percentage of instructions speculatively executed against instructions retired. The code flow is a loop which consists of one memory access that does a pointer chasing where each access must fetch data from off-chip memory, followed by x several independent instructions from the memory access.

simple `add r0, r0, #1` repeated x time. It can be seen that at 0 independent instruction (only the memory access), because there is no possible speculation, the percentage of speculative instructions against retired instructions is close to 0. With 1 independent instruction (one memory access followed by one `add`), this percentage increases to 50%, this is because while the out-of-order pipeline is able to compute all independent instructions while waiting for fetching memory for the memory access, the retirement is still performed in order. Hence, 50% of the instruction flow is executed speculatively. With 2 independent instructions, the percentage increases to 66%. For this particular set of codes, the percentage of speculation against retirement follows the law $100 - 100/(1 + dispatch\_slot\_used * x)$ where x is the number of independent instructions. It can be noted that the memory access `ldr` instruction is not counted in the speculative instructions. This enables discriminating actual work that is performed in parallel with fetching memory. As such, the metric of the percentage of instructions speculatively executed against instructions retired help in discovering how useful the out-of-order pipeline is compared to an in-order pipeline.

The higher the percentage of instruction that has already been executed at the speculative stage, the less useful an out-of-order pipeline is compared to an in-order pipeline. Hence, if one can determine the number of memory accesses with their latencies, it is possible to derive cycle timing execution of the code flow if it would have ran on an in-order pipeline. In practice, this becomes increasingly difficult to determine precisely. One of the reasons lies in the study from the prior chapter 4 where the latency of a single memory access shows a non-linear behaviour depending on the current hardware configuration. Another aspect to consider are differences in other parts of the underlying hardware. For instance, the prefetcher unit present in the different CPUs may implement a non-identical branch predictor and memory prefetcher. Thus, the described metric in this section does not manage to determine the exact performance between the two CPU designs. Nonetheless, this metric can be used as a hint during task mapping decisions.

The new scheduler uses this information at runtime to determine programs that benefit the most from the out-of-order pipeline with its speculative execution capability. When the performance of the big cluster is degraded due to temperature, the scheduler will use the LITTLE cluster to task map running applications that benefit the less from the out-of-order pipeline.

5.2.2.3.2 Scheduler hint: application memory requirement

Furthermore, as CPU cache sizes could be different between the two CPU types, to make a reasonable mapping decision the scheduler requires the memory requirement of the running applications, which is defined as the Working Set Size (WSS). To determine the WSS of a program, one could use the notion of reuse (or stack) distance which is the number of distinct data accesses location between two consecutive accesses to the same location [39, 42, 93]. If there are several distinct accesses between two same locations, the data may have been evicted from the cache due to the limited capacity of the cache. With a reuse distance histogram, the miss ratio can be estimated for a fully-associative cache with a least recently used replacement policy for any cache size d by $mr(d) = \sum_{i=d}^{M} histogram[i]/N$ where N is the total number of requests and M is the total number of unique requests. The current implementation determines this information by prior execution of the applications using a profiled version of the program.

To produce a profiled version of the program, a set of compiler passes using LLVM has been written to insert instrumentation code in the source code and consists of recording memory accesses that the program will perform at runtime. As a program could make several millions of memory accesses, storing the memory location trace to disk for further analysis is impractical, the instrumented code performs the analysis during the instrumented code execution. The instrumented program produces an analysis report at the end of the execution which includes memory location reuse distance histogram and expected miss rate considering different hardware cache sizes. figure 5.6 shows the result of this analysis on `art` from the SPEC CPU2000 benchmark suite with the test input size. If the WSS fits in the L2 cache of the LITTLE cluster, the scheduler is informed that there will be no performance penalty towards the memory requirements of the application. The current implementation to compute reuse distance histogram uses PARDA [104].

The proposed scheduler considers the WSS as a metric to determine if the workload and its memory requirements fit in the different cache size of either CPUs. This metric does not capture the actual characteristic of the memory subsystem of the CPU considering fetching latency and performance of the prefetcher. Moreover, the direct definition of the reuse distance yields miss rate expectation for a fully-associative cache using a least recently used replacement policy. The set of CPUs considered in this work implements a set/way associative cache with different replacement policies with the LITTLE cluster using a lower set/way associativity than the big cluster at both levels of CPU data cache. Studies show that a reduced

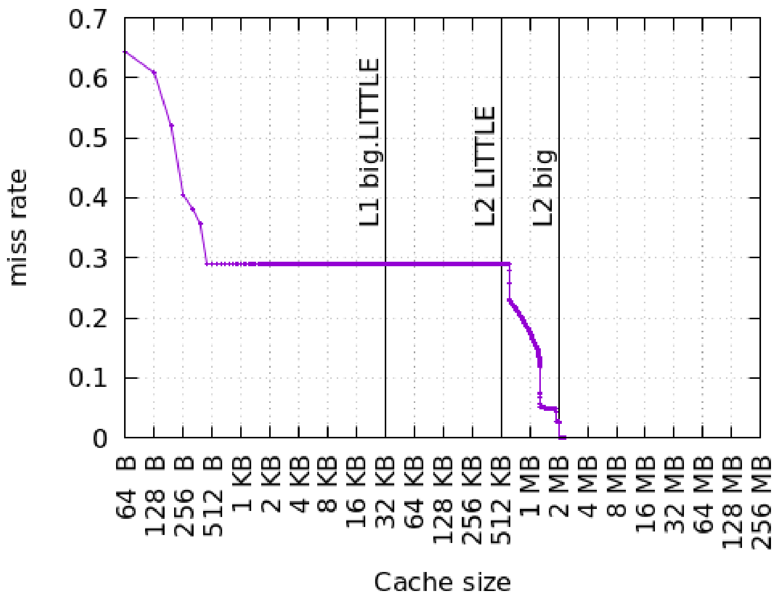


Figure 5.6: Miss ratio curve of `art` from SPEC CPU2000 benchmark suite considering a fully-associative cache with a least recently used replacement policy on 64-byte blocks.

cache associativity increases cache miss rate [29, 60, 59]. Nonetheless, this metric helps in determining a relative penalty performance when deciding which process to map on the LITTLE or big cluster.

Also, computing the WSS for a program is rather time-consuming as it requests to track each memory access, and perform the reuse distance computation for each memory access as it comes. A simple implementation suggested by Mattson [93] uses a linked list to represent a memory location as a node and nodes are linked in order of arrival. When a memory access is performed, the algorithm consists of traversing the linked list until finding the node representing the same memory location, counting the number of nodes from the head of the list, detachiing the node and finally reattaching it in front of the list. The counter represents the reuse distance for this memory access. If the node is not found, the reuse distance is infinite and corresponds to a cold miss. This algorithm has a complexity of $O(N * M)$ where N is the total number of references and M is the unique number of references. Multiple strategies have been suggested to improve this algorithm [27] using hashmap and tree data structures [14, 105, 104], or even to reduce accuracy for improving time and space complexity [48, 15, 139]. When the program comprises linear memory access over a regular data structure, advanced compiler techniques could be used to determine the WSS at compile time [30, 51, 33].

Another possibility would be to have a new set of PMCs that suggests the actual memory occupancy and reuse of cache lines in the CPU cache. While running on the big cluster, if the hardware can give insight on the number of cache lines that are regularly accessed, it could be possible to determine the actual working set size of the workload. It is then possible to determine if the LITTLE cluster can meet the memory requirements of the running program. These suggestions are not considered in this study and are open for future work.

With these two concepts, the scheduler has knowledge to differentiate which application would benefit the most from the big cluster. The next section presents their use for schedule running applications.

5.2.2.4 Thermal Balance Aware Scheduler: algorithm

When the temperature of a CPU core exceeds the limit, a per-core flag is set indicating that the core has overheated and a new mapping is searched using the heuristic presented in section 5.1.3. If the new mapping remains the same, it is assumed that it is the best mapping in respect of the heuristic. At this point, the only chance to cool-down the device is to decrease its operating frequency by following the `step_wise` thermal governor from section 5.1.1.

When the thermal pressure is strong enough to decrease to about the same performance between the two clusters considering the current workload, the scheduler takes a chance of not using the set of cores that has the overheated flag set by migrating some tasks to the LITTLE cluster. The analysis in section 5.2.2.1 suggests a difference of about 5 °C between the hottest core and the coolest core. As such, when the scheduler starts migrating tasks from the big cluster to the LITTLE cluster due to thermal pressure, the scheduler will not migrate back directly from the LITTLE cluster to the big cluster until the thermal sensor has reset the overheat flag. The overheat flag is reset when the core temperature reaches a temperature below the coolest core temperature of the cluster in the offline analysis shown in figure 5.4. This low-level threshold has been chosen by empirical evaluation. Refinement of the threshold to reset the per-core overheat flag will be explored in future work.

When a core is not available due to thermal pressure, a new mapping is then searched that tries to mix the use of both clusters. To do so, the scheduler organizes applications regarding a criterion of the out-of-order pipeline's effectiveness using knowledge from both code flow (see paragraph 5.2.2.3.1) and memory requirements (see paragraph 5.2.2.3.2) of tasks.

The mapping decision considers a multi-level criterion for each running task and is as follows. The first level considered is the area after the L2 data cache size of the big cluster of the MRC curve and the second level is the area after the L2 data cache size of the LITTLE cluster of the MRC curve. The third and last level is the percentage of instruction speculatively executed against instruction retired. The rationale of this criterion is that the latency penalty of fetching data from the off-chip memory to CPU registers has the most impact towards performance. On the evaluation machine, a penalty from a memory fetch is in the range of a few nanoseconds to 1.4 μs depending on whether the requested memory already resides in the CPU cache at high CPU and RAM frequencies, or incurs a memory page fault where a translation of virtual to physical address involves a full traversal during page walk at a low CPU and RAM frequencies. Moreover, stalling on an in-order pipeline is generally due to the access of off-chip memory. As such, favouring applications that require memory the most to run on the big cluster is considered a good choice.

When off-line memory analysis has not been supplied to the scheduler, or the MRC curve area at both levels of the cache hierarchy are similar, the scheduler uses instantaneous memory requirement information using PMU counters considering the last level cache miss rate. This criterion is then weighted with the effective execution time of the task during the current scheduling window. Also, to adapt to fast processes phase change and outliers of PMU counters sampling, the scheduler uses a moving average of the last samples to compute the criterion [13]. Finally, if this per-task criterion is not able to establish an order between tasks, a total order is imposed considering the time creation of the process.

Once this criterion is computed for each process, tasks benefiting the most from the out-of-order pipeline (i.e. the big cluster) have the highest criterion value. Using this criterion, the scheduler will then map in descending order tasks from the coldest to the hottest core, following the heuristic presented in subsection 5.1.3. CPU cores which have their `overheat` flag set are kept idle.

This approach has multiple benefits; it enables achieving higher performance in the short term, and also it helps cool down the set of hot CPU cores by keeping them idle. When CPU cores are cold enough, the frequency can be increased. The scheduler will eventually migrate tasks assigned to the LITTLE cluster onto the big cluster when the situation is deemed more favourable considering the thermal situation and workload requirement.

5.3 Evaluation

To demonstrate the effectiveness of the proposed solution to maximize performance under thermal pressure on an asymmetric multiprocessor, this section compares `TBASS` against the state of the art of the Linux kernel, using a set of multithreaded workloads discussed in the following section.

`TBASS` is compared against two strategies. The first strategy is the default Linux scheduler and frequency scaler (CFS-CAS-EAS) which is chosen when the CPU frequency scaling governor is set to `schedutil`[4] with the default thermal policy `step_wise`.

[4]When the cpufreq governor is set to other policies, the EAS scheduler extension is deactivated and default to CFS-CAS only.

As `step_wise` thermal governor does not enforce a strict thermal limit, the system may stay at a high temperature after a workload burst. The non-strict respect of the thermal limit may be critical in harsh thermal environments with a passive cooling device such as in smartphones and other small form factor devices and decrease reliability and limit the lifetime of the machine. In [121, 137], Viswanath noted that "small differences in operating temperature (order of 10-15°C) can result in a 2X difference in the lifespan of the devices". Thus, `TBASS` does respect a strict thermal policy against the thermal limit. As such, to have a fair comparison, this section evaluates a modified `step_wise` that enforces thermal action using a strict respect of the thermal limit as described in section 5.1.1 with the `ENFORCE_STRICTNESS` option. This thermal policy is named `step_wise_strict` in this section.

This evaluation reports both percentage improvement of `TBASS` on *execution time* and *energy consumption* of each benchmark against both comparison points; each benchmark is executed five times and the average (using a geometric mean [52]) of execution time and energy are taken to normalize noise and reduce the effect of outliers. Finally, the absolute value of the sum per core of the Root Mean Square Error (RMSE) over the thermal limit is reported to exhibit efficacy on the strictness of the thermal limit for all strategies.

All experiments are conducted under Debian 11 with a Linux kernel v5.15. All benchmarks are compiled with gcc 11.2.0. To simulate a smartphone environment, the FAN has been disabled, with a thermal limit set to 75°C. Experimentation is performed with devices placed in a thermal chamber with an ambient temperature of 25°C. This study explores strategies to maximize performance under thermal pressure. As such, the device is warmed before the execution of the benchmark by using a stress test that saturates the execution unit of the CPU by using only integer and floating-point instructions for one minute before the execution of the benchmark.

5.3.1 Benchmark selection

The work has been evaluated on a set of 18 multithreaded benchmarks from the PARSEC-3.0 [21] and Splash-3 [119] benchmark suite. PARSEC offers state-of-the-art computationally-intensive algorithms and very diverse workloads from different application domains. Splash-3 (an upgrade of the Splash-2 benchmark suite [149] that is shipped in PARSEC source tree) is composed of workloads targeting high-performance computing domains. In these two benchmark suites, parallelism is written using pthread library and features different parallel models (data-parallel and pipeline).

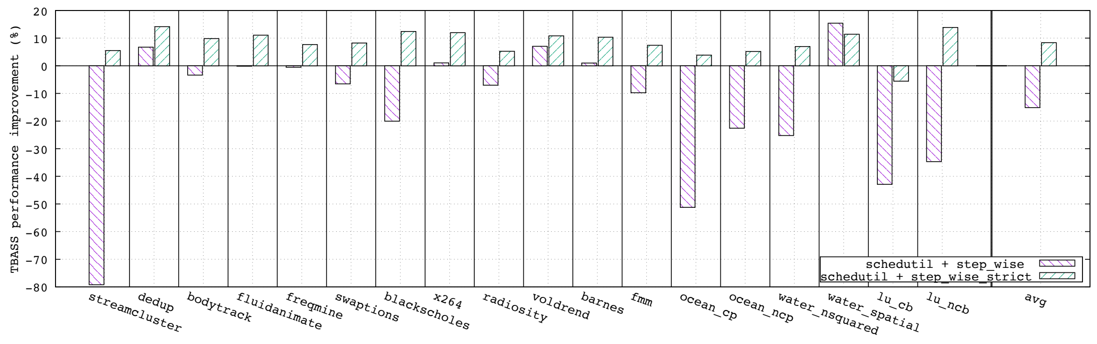
Some benchmarks have been discarded because they cannot be compiled for Arm architecture (PARSEC `raytrace` benchmark contains SSE specific instructions), or fail at runtime (`facesim`, `ferret`, `vips` and `canneal` from PARSEC). Finally, benchmarks that have a short execution time are omitted (`cholesky`, `fft` and `radix` kernels from Splash-3). All benchmarks are executed to completion using native datasets. Splash-3 benchmarks input sizes have been slightly reduced to fit in the RAM to avoid generating SEGFAULT errors.

Each benchmark is setup to run with four threads using the standard benchmark configuration. However, some benchmarks spawn more threads than requested [127].

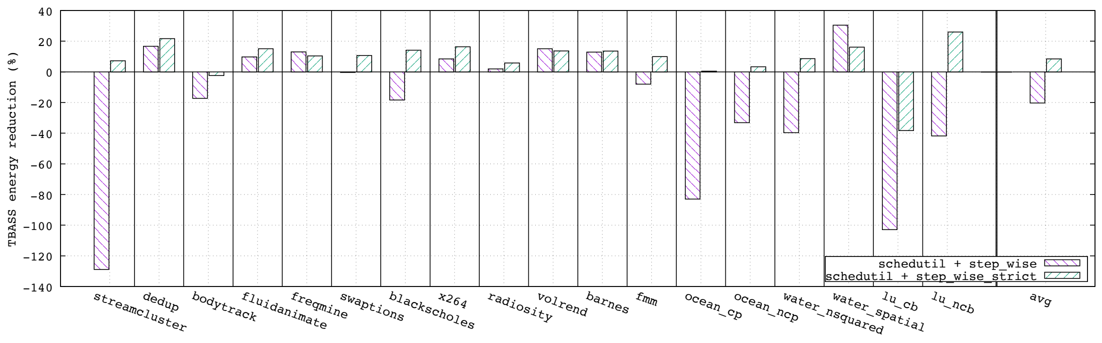
5.3.2 Results

As shown in figure 5.4, four different instances of the same Hardkernel Odroid-XU3 board present different thermal characteristics. While other results in this chapter used four different devices, this section uses only three of those devices as the fourth was unavailable. All evaluation results are shown in figure 5.7 (figure 5.8 and figure 5.9 respectively) for the `XU3-1` board (`XU3-2` and `XU3-3` respectively).

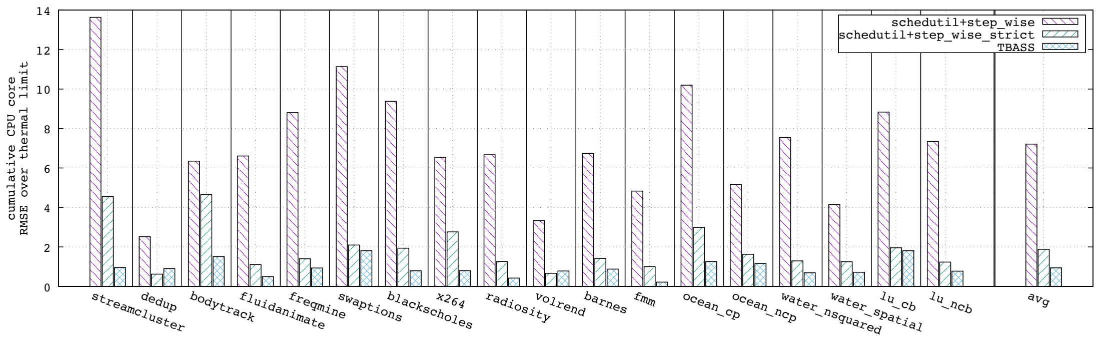
The percentage difference in the execution time of `TBASS` against both comparison points is shown in figure 5.7a (figure 5.8a and figure 5.9a respectively), while the percentage difference in energy consumption is shown in figure 5.7b (figure 5.8b and figure 5.9b respectively). A positive value on these two metrics is a speedup



(a) Improvement in performance of `TBASS`.

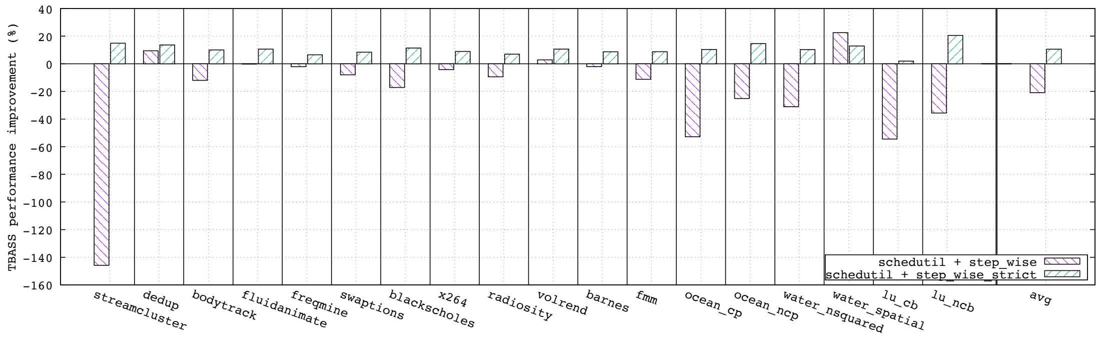


(b) Energy consumption reduction of `TBASS`.

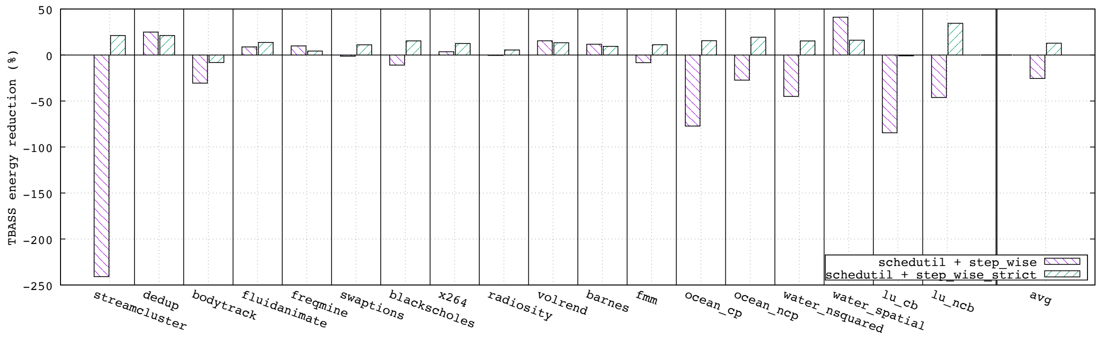


(c) Absolute cumulative RMSE over the thermal limit of each strategies.

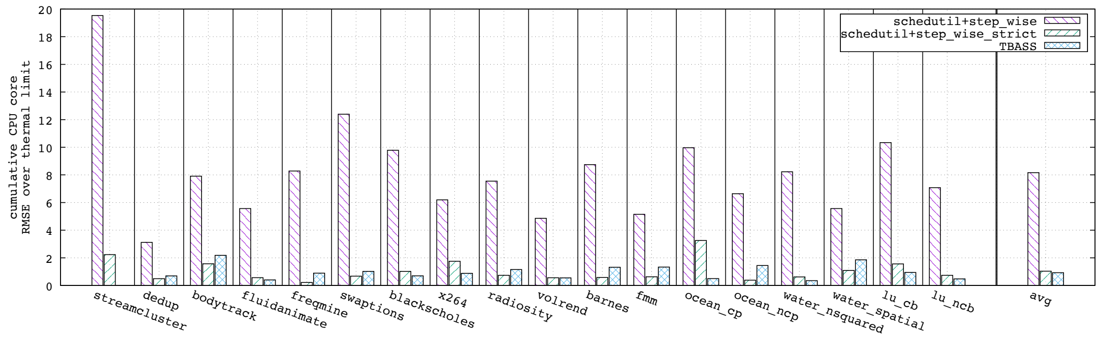
Figure 5.7: Results on `XU3_1` device.



(a) Improvement in performance of `TBASS`.

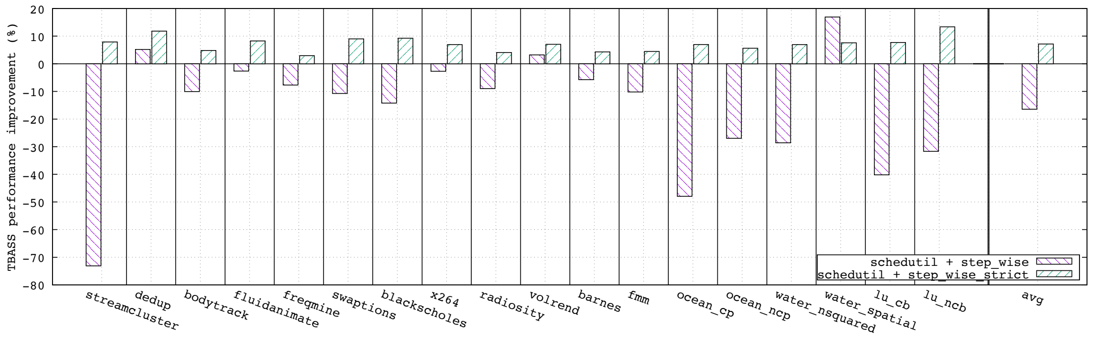


(b) Energy consumption reduction of `TBASS`.

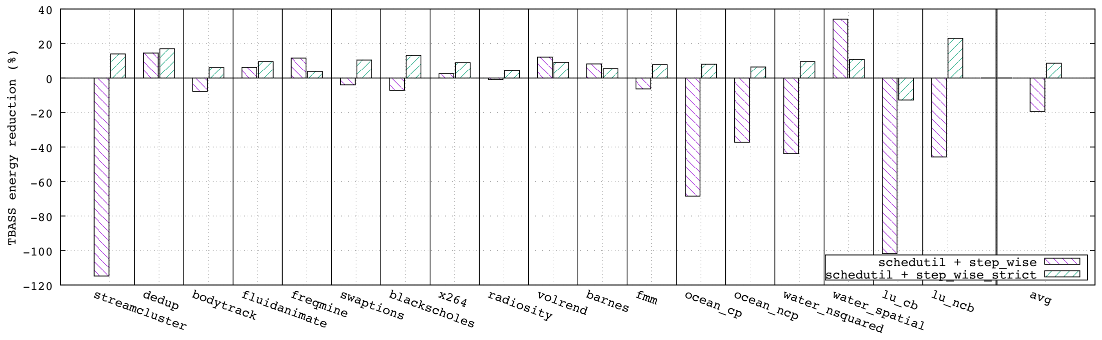


(c) Absolute cumulative RMSE over the thermal limit of each strategies.

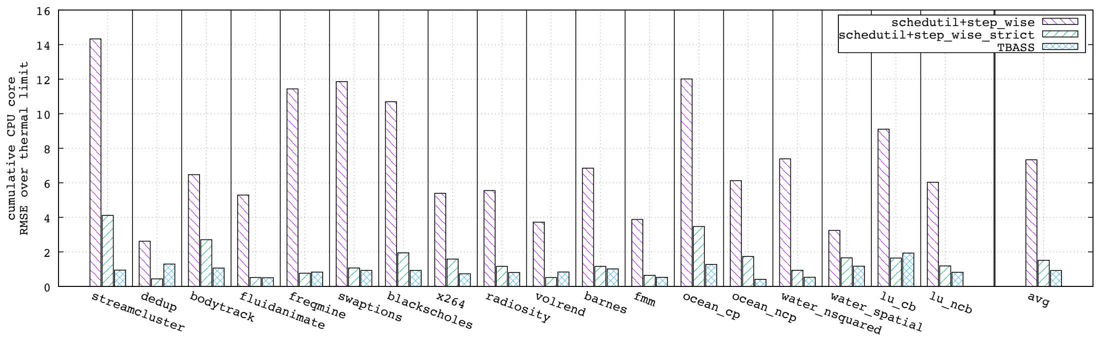
Figure 5.8: Results on `XU3_2` device.



(a) Improvement in performance of TBASS.



(b) Energy consumption reduction of TBASS.



(c) Absolute cumulative RMSE over the thermal limit of each strategies.

Figure 5.9: Results on XU3_3 device.

or an energy reduction in favour of `TBASS`. figure 5.7c (figure 5.8c and figure 5.9c respectively) shows the absolute RMSE over the thermal limit of each strategy.

First of all, `schedutil+step_wise` does not respect the thermal limit strictly and is largely visible in the error over the thermal limit (visible in figure 5.7c, figure 5.8c and figure 5.7c). As such, `TBASS` shows a large slowdown in performance against this strategy proposed as default in the Linux kernel. On the contrary, when enforcing a strict thermal limit to default strategy with `schedutil+step_wise_strict`, `TBASS` shows quite some improvement in both performance and energy reduction.

This behaviour is consistent across all three tested devices, with an average slowdown ranging from 15 to 20 % when comparing `TBASS` against `schedutil+step_wise`, with an absolute RMSE ranging from 7 to 9 °C. When comparing `TBASS` against `schedutil+step_wise_strict` that employs a strict thermal limit, `TBASS` shows speedups between 8 to 10 % across all devices. When enforcing a strict thermal limit, the RMSE ranges from 1 to 2 °C for `schedutil+step_wise_strict` while `TBASS` has an RMSE of less than 1 °C.

More particularly, on some benchmarks, `TBASS` that imposes a strict thermal limit faces a high slowdown compared to `schedutil+step_wise` which is less restrictive on the thermal limit.

For instance, considering `streamcluster` from PARSEC, `TBASS` shows between 80% to 140% slowdown depending on the board compared to `schedutil+step_wise`. The reason behind this is two-fold. First, this multithreaded application has a relatively high working set size and second, threads are working on shared memory and experience high traffic in terms of bytes per instruction [21]. While `schedutil+step_wise` tends to place more threads on the big cluster, `TBASS` tends to spread threads over both clusters while the heuristic deems that both clusters have similar computing performance. Thus, `TBASS` may increase latency on thread communication and could impact the performance of the application. However, when a strict policy on thermal limit is applied with the strategy `schedutil+step_wise_strict`, `TBASS` shows performance improvement. This is because, even if `TBASS` may increase latency, this strategy can increase the raw performance of some CPUs by favouring colder cores, leaving hotter cores idle.

In the case of `water-nsquared` and `water-spatial`, these two benchmarks solve the same problem, but with a different algorithm. The implementation of `water-nsquared` has a bigger WSS and higher thread communication than that of `water-spatial` [149]. Like for `streamcluster`, the communications latency of threads is reflected in the performance of `TBASS` against `schedutil+step_wise` with a slowdown ranging from 25 to 31 % when considering `water-nsquared` benchmark. However, `schedutil+step_wise` shows an RMSE over the thermal limit ranging from 3 to 9 °C, while `TBASS` is less than 1 °C.

On the other hand, with the `water-spatial` implementation that has less thread communication than `water-nsquared` implementation, `TBASS` shows a nice speedup ranging from 10 to 23 % over both alternatives amongst all devices.

`TBASS` currently does not capture thread communication traffic, and as such, does not try to pack threads of the same application on the same cluster. This aspect will be explored in future work.

Generally, the `XU3-2` board shows higher improvement in performance than the two other boards. This is expected as the difference between the two hottest cores and the two coldest cores are larger on this board (see figure 5.4). In the same manner, performance improvement on the `XU3-3` is generally lower, as this difference is smaller. When the difference between the hottest and coldest cores is large, by avoiding the hottest cores, `TBASS` allows the big cluster to run at a higher frequency.

On the other hand, energy improvement follows the same trend as performance improvement. This is due to finishing the workload earlier.

5.4 Summary

This chapter presented different approaches for thermal management employing techniques such as frequency scaling and scheduling. The particularity of the device in consideration is heterogeneity in computing performance between CPU cores, with cores implementing either an in-order or an out-of-order pipeline.

The current standard approach of Linux for thermal management is the use of frequency scaling to limit the temperature of the device when it reaches a thermal limit. Alongside this procedure, the scheduler place tasks using two algorithms. When tasks are not considered computationally intensive (load metric lower than 80%), it uses an energy and computing capacity model to select a CPU core for execution (EAS variant). When the system is considered over-utilized, the scheduler considers only the computing capacity model to do task placement. In the current implementation, both decisions of frequency scaling (for thermal management) and scheduling are isolated and target different purposes to manage the system.

On the board model under study, it appears that each board and each core have their own thermal characteristics, showing a steady-state temperature with more than 5 °C difference between the hottest and the coldest core. If the operating system is unaware of device thermal characteristics and CPU core raw performance capability, suboptimal scheduling and frequency scaling decisions could severely impact both thermal situation and performance.

This situation led to the development of the Thermal-Balance Aware System Scheduler (TBASS), a holistic approach for scheduling and frequency scaling to maximize performance under thermal pressure for single-ISA heterogeneous multicore processor architecture such as the Arm big.LITTLE architecture. To do so, `TBASS` considers the thermal characteristics of CPU cores, CPU cores performance hardware design and workload knowledge. The workload knowledge consists of static and dynamic information using performance monitoring counters at runtime and an offline analysis identifying the memory requirements of the workload.

The experimental results confirm the benefits of `TBASS` with average speedups between 8 to 10 % when compared to the default Linux kernel scheduler employing a strict thermal limit. When compared to the default Linux kernel scheduler without employing a strict thermal limit, `TBASS` shows an average slowdown ranging from 15 to 20 %. However, `TBASS` reduces the temperature of the device by 10 °C on average which may improve the lifespan of the devices [137].

One of the atypical approaches of `TBASS` is to avoid thermal imbalance of the device by avoiding the use of CPU cores that are likely to favour this imbalance and favour the LITTLE cluster when performance is deemed to be similar in performance considering the running workload. By doing so, the CPU can run at a higher frequency for the workload that could benefit the most of the big cluster. Moreover, this strategy helps in limiting snooping latency effect shown in chapter 4.

Chapter 6

Conclusion

This final chapter concludes the thesis with a summary and reviews the research undertaken and its findings. Then an examination of future research possibilities that could be pursued to expand upon and overcome the thesis's limitations will be presented. Finally, this thesis ends with some final remarks.

6.1 Thesis summary and contributions

This thesis began with a brief history of computing systems and exposed the motivation to follow the path of asymmetric multicore and multiprocessor architecture design to progress post Dennard scaling era while pursuing Moore's law.

The background chapter presented some aspects of micro-architectural design that make computing possible, and different ways to organize this computation capability around memory.

A meticulous literature review in chapter 3 regarding scheduling, frequency scaling and thermal management for single-ISA AMP, draws the inference that the current state-of-the-art does not consider low-level micro-architectural detail. This situation can lead to energy waste and suboptimal performance.

This thesis is an attempt to exploit low-level hardware knowledge to optimize frequency scaling and scheduling decisions for single-ISA AMP targeting small form factor devices such as smartphones and tablets.

Considering frequency scaling optimization technique, the thesis aimed to answer these two research questions:

RQ 1 To what extent frequency scaling could improve the energy efficiency of a computing device?

RQ 2 To what extent frequency scaling affects the system in terms of performance on multiprocessors-equipped devices where data-coherency is implemented using a bus-snoop protocol?

These research question has been addressed in chapter 4. In this chapter, a thorough exploration of the memory subsystem and data-coherency mechanism (which implements a bus-snoop protocol) has been carried out with the use of microbenchmarks tailored to precisely stress the memory subsystem. This exploration exposed a non-linear latency of the data-coherency mechanism when frequency scaling techniques are used to reduce the energy consumption of the system. In particular, this non-linear latency makes it particularly difficult to derive a precise performance model if not taken into account. However, the in-depth analysis enables the extraction of a simple model in the form of a decision tree which can be efficiently incorporated into any frequency scaling driver. Evaluation of the proposed solution shows an increase in application performance of up to 40% and reduces energy consumption by up to 70% compared to the default frequency scaling policy.

On the scheduling front, a goal of the thesis was to answer these two research questions:

RQ 3 How to schedule workloads for performance while minimizing energy consumption on systems that use asymmetric multicore multiprocessors?

RQ 4 How can thermal effects be effectively mitigated using software-based techniques, for small-form factor computing devices where active cooling (e.g. fan, liquid cooling) can not be employed?

To answer these research questions, in chapter 5, a set of microbenchmarks designed to comprehend low-level micro-architectural designs of processing elements that help in delivering computing performance and energy efficiency are presented. In addition to understanding the full potential of a workload, a thorough analysis of the workload memory requirement (i.e. WSS) using a profiled version of the workload has been carried out. The profiled version of the workload uses LLVM to monitor every memory accesses that the workload will perform at runtime. The output of this profile is the working set size of the workload. This knowledge is then used at runtime to guide workload scheduling, specifically with the device under thermal pressure, which is common to small form factor devices that rely on passive cooling. The evaluation shows an increase in application performance by 10% on average and reduces energy consumption by 12% on average compared to the default scheduling strategy that uses a strict thermal policy.

6.2 Future research directions

This section presents some limitations and possible research directions to improve the work conducted in this thesis.

To determine the overall memory requirement of a process, the current strategy employs an offline analysis that executes a profiled version of the workload. Though this step needs to be run only once with the analysis reusable for further execution, this strategy remains inefficient. Moreover, the current analysis considers the entire execution of the process without considering the workload phases. Some work in the compiler community studied reuse distance to guide code optimization [30, 51, 33]. These works could be reused to extract the memory profile of the workload faster than by executing a profiled version of the binary. Another approach would be to introduce a new set of PMU counters to identify memory portions that are frequently used by the workload. This would allow the refinement and speed up the computation of memory requirements of the workload during its different execution phases.

The proposed scheduler does not consider point-to-point memory communication in the case of a multithreaded application. As such, the solution may map two threads, one on a LITTLE core, and another one on a big core. If these two threads are working on shared memory and are working back-to-back on the same memory location, the proposed scheduler may increase memory latency and degrade performance. Note that the Linux scheduler is currently unaware of the notion of threads' point-to-point memory communication. It is the responsibility of the developer to group tasks in some sort of task pool via `cgroup` or `numactl` in the case of a NUMA topology. In the same manner, knowing the memory requirement and point-to-point communication between threads could help the scheduler avoid cache thrashing.

The experiment in figure 3.1 demonstrates that any power model will have errors by considering only PMU counters. However, the experiment is a rather peculiar use case as it stresses only one particular instruction. In the general case of a real workload, there will most probably be a mixture of several instruction types. In [140], the authors propose a strong power model technique considering PMU counters and thermal contributions to power leakage. The power model accuracy is as good as 3.8% and 2.8% on average to predict power consumption on an ARM Cortex-A7 and ARM Cortex-A15 respectively. A refined power model using compiler hints would help discriminate PMU counters aggregation to micro-architectural block. This strategy will help in better power and thermal predictions.

On a smartphone or tablet, the GPU is in constant use. However, the work of this thesis does not consider workloads that use this processing element type. Consideration of real-time tasks for media-processing applications incorporating GPU would be a great research direction.

6.3 Concluding Remarks

In summary, this thesis presented novel approaches to frequency scaling and scheduling decisions for asymmetric multiprocessing systems. By utilizing tailored microbenchmarks to discover micro-architectural specificities, multiple models were derived to find the best match of CPUs frequencies setting and workload scheduling on real-world applications.

Additionally, the work in chapter 4 and chapter 5 gave some strategies to alleviate memory-related issues regarding latency induced by hardware data-coherency protocol and the memory footprint of a workload. Several directions have been suggested to improve memory footprint computation and reach the full potential of this thesis. These works can be applied at a bigger scale for the server and warehouse market to cover the full spectrum of computing systems.

Furthermore, the work in chapter 5 explored scheduling strategies that encompass workload characteristics and effective use of the CPU core micro-architectural specificities. In particular, the scheduling decision was applied for thermal management on small form factor devices. However, knowledge acquired during this work can be generalized for regular load-balancing procedures. While the single-ISA asymmetric multiprocessing devices become more common, the work of this thesis enables better use of the underlying micro-architectural specificities for workload scheduling.

References

[1] Agarwal, A. "Leakage Power Analysis and Reduction for Nanoscale Circuits". In: *IEEE Micro* 26.2 (Mar. 2006), pp. 68–80. ISSN: 0272-1732. DOI: 10.1109/MM.2006.39.

[2] Allyn, S. *Jellyfish video.* 2020. URL: http://jell.yfish.us/media/jellyfish-3-mbps-hd-h264.mkv (visited on 04/01/2020).

[3] Apple Inc. *Tuning Your Code's Performance for Apple Silicon.* 2022. URL: https://developer.apple.com/documentation/apple-silicon/tuning-your-code-s-performance-for-apple-silicon (visited on 04/01/2022).

[4] Apple WebKit Team, *Speedometer2.0.* 2018. URL: https://browserbench.org/Speedometer2.0/ (visited on 04/01/2020).

[5] Archibald, J., Baer, J.-L., "Cache coherence protocols: evaluation using a multiprocessor simulation model". In: *ACM Transactions on Computer Systems* 4 (4 Sept. 1986), pp. 273–298. ISSN: 0734-2071. DOI: 10.1145/6513.6514.

[6] Arm Holdings, *White paper: big.LITTLE Technology: The Future of Mobile.* 2013.

[7] Arm Holdings, *Arm NEON.* 2020. URL: https://developer.arm.com/documentation/102474/latest (visited on 04/01/2020).

[8] Arm Holdings, *CCI-400.* 2020. URL: https://developer.arm.com/documentation/ddi0470/latest (visited on 04/01/2020).

[9] Arm Holdings, *Cortex-A15*. 2020. URL: https://developer.arm.com/documentation/ddi0438/latest (visited on 04/01/2020).

[10] Arm Holdings, *Cortex-A7*. 2020. URL: https://developer.arm.com/documentation/ddi0464/latest (visited on 04/01/2020).

[11] Arm Holdings, *Introduction to SVE*. 2020. URL: https://developer.arm.com/documentation/102476/latest/ (visited on 04/01/2020).

[12] Arm Holdings, *Intelligent Power Allocation*. 2021. URL: https://developer.arm.com/tools-and-software/open-source-software/linux-kernel/intelligent-power-allocation (visited on 04/11/2021).

[13] Becchi, M., Crowley, P., "Dynamic thread assignment on heterogeneous multiprocessor architectures". In: *Proceedings of the 3rd conference on Computing frontiers - CF '06*. New York, New York, USA: ACM Press, 2006, p. 29. ISBN: 1595933026. DOI: 10.1145/1128022.1128029.

[14] Bennett, B. T., Kruskal, V. J., "LRU Stack Processing". In: *IBM Journal of Research and Development* 19 (4 July 1975), pp. 353–357. ISSN: 0018-8646. DOI: 10.1147/rd.194.0353.

[15] Berg, E., Hagersten, E., "StatCache: A probabilistic approach to efficient and accurate data locality analysis". In: *IEEE International Symposium on - ISPASS Performance Analysis of Systems and Software, 2004*. IEEE, 2004, pp. 20–27. ISBN: 0-7803-8385-0. DOI: 10.1109/ISPASS.2004.1291352.

[16] Bernstein, K. "High-performance CMOS variability in the 65-nm regime and beyond". In: *IBM Journal of Research and Development* 50.4.5 (July 2006), pp. 433–449. ISSN: 0018-8646. DOI: 10.1147/rd.504.0433.

[17] Bhat, G., Gumussoy, S., Ogras, U. Y., "Power-Temperature Stability and Safety Analysis for Multiprocessor Systems". In: *ACM Transactions on Embedded Computing Systems* 16.5s (Oct. 2017), pp. 1–19. ISSN: 1539-9087. DOI: 10.1145/3126567.

[18] Bhat, G., Gumussoy, S., Ogras, U. Y., "Power and Thermal Analysis of Commercial Mobile Platforms: Experiments and Case Studies". In: *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE).* IEEE, Mar. 2019, pp. 144–149. ISBN: 978-3-9819263-2-3. DOI: `10.23919/DATE.2019.8714831`.

[19] Bhat, G., Gumussoy, S., Ogras, U. Y., "Analysis and Control of Power–Temperature Dynamics in Heterogeneous Multiprocessors". In: *IEEE Transactions on Control Systems Technology* 29.1 (Jan. 2020), pp. 329–341. ISSN: 1063-6536. DOI: `10.1109/TCST.2020.2974421`.

[20] Bhat, G. "Algorithmic Optimization of Thermal and Power Management for Heterogeneous Mobile Platforms". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26.3 (Mar. 2018), pp. 544–557. ISSN: 1063-8210. DOI: `10.1109/TVLSI.2017.2770163`.

[21] Bienia, C. "The PARSEC benchmark suite: Characterization and architectural implications". In: *Proceedings of the 17th international conference on Parallel architectures and compilation techniques - PACT '08.* New York, New York, USA: ACM Press, 2008, pp. 72–81. ISBN: 9781605582825. DOI: `10.1145/1454115.1454128`.

[22] Bohr, M. "A 30 Year Retrospective on Dennard's MOSFET Scaling Paper". In: *IEEE Solid-State Circuits Newsletter* 12.1 (2007), pp. 11–13. ISSN: 1098-4232. DOI: `10.1109/N-SSC.2007.4785534`.

[23] Bose, P. "Power Wall". In: *Encyclopedia of Parallel Computing.* Ed. by David Padua. Boston, MA: Springer US, 2011, pp. 1593–1608. ISBN: 978-0-387-09766-4. DOI: `10.1007/978-0-387-09766-4_499`.

[24] Bower, F. A., Sorin, D. J., Cox, L. P., "The Impact of Dynamically Heterogeneous Multicore Processors on Thread Scheduling". In: *IEEE Micro* 28.3 (May 2008), pp. 17–25. ISSN: 0272-1732. DOI: `10.1109/MM.2008.46`.

[25] Butts, J. A., Sohi, G. S., “A static power model for architects”. In: *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture - MICRO 33*. New York, New York, USA: ACM Press, 2000, pp. 191–201. ISBN: 1581131968. DOI: `10.1145/360128.360148`.

[26] Butzen, P. F., Ribas, R. P., *Leakage Current in Sub-Micrometer CMOS Gates*. 2008. URL: `https://www.inf.ufrgs.br/logics/docman/book_emicro_butzen.pdf` (visited on 04/01/2020).

[27] Byrne, D. *A Survey of Miss-Ratio Curve Construction Techniques*. Apr. 2018. URL: `http://arxiv.org/abs/1804.01972` (visited on 03/19/2021).

[28] Cameron, K. W., Pyla, H. K., Varadarajan, S., “Tempest: A portable tool to identify hot spots in parallel code”. In: *2007 International Conference on Parallel Processing (ICPP 2007)*. IEEE, Aug. 2007, pp. 37–37. ISBN: 0-7695-2933-X. DOI: `10.1109/ICPP.2007.77`.

[29] Cantin, J. F., Hill, M. D., “Cache performance for selected SPEC CPU2000 benchmarks”. In: *ACM SIGARCH Computer Architecture News* 29 (4 Sept. 2001), pp. 13–18. ISSN: 0163-5964. DOI: `10.1145/563519.563522`.

[30] Cascaval, C. “Compile-Time Based Performance Prediction”. In: *Languages and Compilers for Parallel Computing*. Ed. by Larry Carter and Jeanne Ferrante. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 365–379. ISBN: 978-3-540-44905-8. DOI: `10.1007/3-540-44905-1_23`.

[31] Celio, C. *Characterizing Multi-Core Processors Using Micro-benchmarks*. 2009. URL: `https://github.com/ucb-bar/ccbench/wiki` (visited on 04/01/2020).

[32] Chandrakasan, A., Sheng, S., Brodersen, R., “Low-power CMOS digital design”. In: *IEEE Journal of Solid-State Circuits* 27.4 (Apr. 1992), pp. 473–484. ISSN: 00189200. DOI: `10.1109/4.126534`.

[33] Chauhan, A., Shei, C.-Y., "Static reuse distances for locality-based optimizations in MATLAB". In: *Proceedings of the 24th ACM International Conference on Supercomputing - ICS '10*. New York, New York, USA: ACM Press, 2010, p. 295. ISBN: 9781450300186. DOI: 10.1145/1810085.1810125.

[34] Chen, T.-C. "Where CMOS is Going: Trendy Hype vs. Real Technology". In: *IEEE Solid-State Circuits Society Newsletter* 11.3 (Sept. 2006), pp. 5–9. ISSN: 1098-4232. DOI: 10.1109/N-SSC.2006.4785853.

[35] Chrome DevTools Team, *puppeteer*. 2020. URL: https://pptr.dev/ (visited on 04/01/2020).

[36] Cuesta, D., Risco-Martin, J. L., Ayala, J. L., "3D thermal-aware floorplanner using a MILP approximation". In: *Microprocessors and Microsystems* 36.5 (July 2012), pp. 344–354. ISSN: 01419331. DOI: 10.1016/j.micpro.2012.02.012.

[37] De, V., Borkar, S., "Technology and design challenges for low power and high performance [microprocessors]". In: *Proceedings. 1999 International Symposium on Low Power Electronics and Design (Cat. No.99TH8477)*. IEEE, 1999, pp. 163–168. ISBN: 1-58113-133-X. DOI: 10.1109/LPE.1999.799433.

[38] Dennard, R. "Design of ion-implanted MOSFET's with very small physical dimensions". In: *IEEE Journal of Solid-State Circuits* 9.5 (Oct. 1974), pp. 256–268. ISSN: 0018-9200. DOI: 10.1109/JSSC.1974.1050511.

[39] Denning, P. J. "The working set model for program behavior". In: *Communications of the ACM* 11 (5 May 1968), pp. 323–333. ISSN: 0001-0782. DOI: 10.1145/363095.363141.

[40] Denning, P. J. "On modeling program behavior". In: *Proceedings of the November 16-18, 1971, fall joint computer conference on - AFIPS '71 (Fall)*. New York, New York, USA: ACM Press, 1971, p. 937. DOI: 10.1145/1478873.1478998.

[41] Denning, P. J., Kahn, K. C., "A study of program locality and lifetime functions". In: *Proceedings of the fifth symposium on Operating systems principles - SOSP '75*. New York, New York, USA: ACM Press, 1975, pp. 207–216. DOI: `10.1145/800213.806539`.

[42] Denning, P. J., Schwartz, S. C., "Properties of the working-set model". In: *Communications of the ACM* 15 (3 Mar. 1972), pp. 191–198. ISSN: 0001-0782. DOI: `10.1145/361268.361281`.

[43] Dev, K. *Implications of Integrated CPU-GPU Processors on Thermal and Power Management Techniques*. Aug. 2018. URL: `http://arxiv.org/abs/1808.09651` (visited on 04/01/2020).

[44] Dhodapkar, A. "TEM2P2EST: A Thermal Enabled Multi-model Power/Performance ESTimator". In: *Power-Aware Computer Systems*. Ed. by Babak Falsafi and T. N. Vijaykumar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 112–125. ISBN: 978-3-540-44572-2. DOI: `10.1007/3-540-44572-2_9`.

[45] Dong Li, "System-level, thermal-aware, fully-loaded process scheduling". In: *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, Apr. 2008, pp. 1–7. ISBN: 978-1-4244-1693-6. DOI: `10.1109/IPDPS.2008.4536225`.

[46] Drennan, P., McAndrew, C., "Understanding MOSFET mismatch for analog design". In: *IEEE Journal of Solid-State Circuits* 38.3 (Mar. 2003), pp. 450–456. ISSN: 0018-9200. DOI: `10.1109/JSSC.2002.808305`.

[47] *EAS overview and integration guide (r1p6)*. 2018. URL: `https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Open%20Source/energy-aware-scheduling/eas_overview_and_integration_guide_r1p6.pdf` (visited on 04/11/2021).

[48] Eklov, D., Hagersten, E., “StatStack: Efficient modeling of LRU caches”. In: *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE, Mar. 2010, pp. 55–65. ISBN: 978-1-4244-6023-6. DOI: `10.1109/ISPASS.2010.5452069`.

[49] Esmaeilzadeh, H. “Dark silicon and the end of multicore scaling”. In: *Proceeding of the 38th annual international symposium on Computer architecture - ISCA ’11*. New York, New York, USA: ACM Press, 2011, p. 365. ISBN: 9781450304726. DOI: `10.1145/2000064.2000108`.

[50] Farkas, K., Jouppi, N. P., Ranganathan, P., “Heterogeneous processor core systems for improved throughput”. U.S. pat. 7996839B2. Hewlett Packard Development Co LP. July 2011. URL: `https://patents.google.com/patent/US7996839`.

[51] Fauzia, N. “Beyond reuse distance analysis: Dynamic Analysis for Characterization of Data Locality Potential”. In: *ACM Transactions on Architecture and Code Optimization* 10 (4 Dec. 2013), pp. 1–29. ISSN: 15443566. DOI: `10.1145/2555289.2555309`.

[52] Fleming, P. J., Wallace, J. J., “How not to lie with statistics: the correct way to summarize benchmark results”. In: *Communications of the ACM* 29.3 (Mar. 1986), pp. 218–221. ISSN: 0001-0782. DOI: `10.1145/5666.5673`. URL: `https://dl.acm.org/doi/10.1145/5666.5673`.

[53] Flynn, M. “Very high-speed computing systems”. In: *Proceedings of the IEEE* 54 (12 1966), pp. 1901–1909. DOI: `10.1109/PROC.1966.5273`.

[54] Frantzis, A., Barker, J., *GLmark2*. 2010. URL: `https://github.com/glmark2/glmark2` (visited on 04/01/2020).

[55] Greenhalgh, P. *big.LITTLE Processing with ARM Cortex™-A15 & Cortex-A7*. 2011.

[56] Gupta, U. "DyPO: Dynamic Pareto-Optimal Configuration Selection for Heterogeneous MpSoCs". In: *ACM Transactions on Embedded Computing Systems* 16.5s (Oct. 2017), pp. 1–20. ISSN: 1539-9087. DOI: `10.1145/3126530`.

[57] Gutierrez, A. "Full-system analysis and characterization of interactive smartphone applications". In: *2011 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2011, pp. 81–90. DOI: `10.1109/IISWC.2011.6114205`.

[58] HardKernel, *Odroid-XU3*. 2014. URL: `http://www.hardkernel.com/` (visited on 04/01/2020).

[59] Harris, S. L., Harris, D. M., "8 - Memory Systems". In: *Digital Design and Computer Architecture*. Ed. by Sarah L. Harris and David Money Harris. Boston: Morgan Kaufmann, 2016, pp. 486–529. ISBN: 978-0-12-800056-4. DOI: `https://doi.org/10.1016/B978-0-12-800056-4.00008-X`.

[60] Hartstein, A. "Cache miss behavior: Is It sqrt(2)?" In: *Proceedings of the 3rd conference on Computing frontiers - CF '06*. New York, New York, USA: ACM Press, 2006, p. 313. ISBN: 1595933026. DOI: `10.1145/1128022.1128064`.

[61] Hennessy, J. L., Patterson, D. A., *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 2019. ISBN: 978-0-12-811905-1.

[62] Henning, J. L. "SPEC CPU2006 benchmark descriptions". In: *ACM SIGARCH Computer Architecture News* 34.4 (2006), pp. 1–17. DOI: `10.1145/1186736.1186737`.

[63] Heo, S., Barr, K., Asanović, K., "Reducing power density through activity migration". In: *Proceedings of the 2003 international symposium on Low power electronics and design - ISLPED '03*. New York, New York, USA: ACM Press, 2003, p. 217. ISBN: 158113682X. DOI: `10.1145/871506.871561`.

[64] Horowitz, M. “Scaling, power, and the future of CMOS”. In: *IEEE InternationalElectron Devices Meeting, 2005. IEDM Technical Digest.* IEEE, 2008, pp. 9–15. ISBN: 0-7803-9268-X. DOI: 10.1109/IEDM.2005.1609253.

[65] Humenay, E., Tarjan, D., Skadron, K., “Impact of Process Variations on Multicore Performance Symmetry”. In: *2007 Design, Automation & Test in Europe Conference & Exhibition.* IEEE, Apr. 2007, pp. 1–6. ISBN: 978-3-9810801-2-4. DOI: 10.1109/DATE.2007.364539.

[66] Hwang, K. *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* McGraw-Hill Higher Education, 1992. ISBN: 978-0070316225.

[67] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer’s Manual.* 2021. URL: https://cdrdv2.intel.com/v1/dl/getContent/671200 (visited on 01/01/2022).

[68] Jagannathan, S. “Temperature dependence of soft error rate in flip-flop designs”. In: *2012 IEEE International Reliability Physics Symposium (IRPS).* IEEE, Apr. 2012, SE.2.1–SE.2.6. ISBN: 978-1-4577-1680-5. DOI: 10.1109/IRPS.2012.6241927.

[69] Jaleel, A. *Memory characterization of workloads using instrumentation-driven simulation.* 2010. URL: http://www.jaleels.org/ajaleel/workload/SPECanalysis.pdf (visited on 04/01/2020).

[70] Keshavarzi, A., Roy, K., Hawkins, C., “Intrinsic leakage in low power deep submicron CMOS ICs”. In: *Proceedings International Test Conference 1997.* Int. Test Conference, 1997, pp. 146–155. ISBN: 0-7803-4209-7. DOI: 10.1109/TEST.1997.639607.

[71] Khushu, S., Gomes, W., “Lakefield: Hybrid cores in 3D Package”. In: *2019 IEEE Hot Chips 31 Symposium (HCS).* IEEE, Aug. 2019, pp. 1–20. ISBN: 978-1-7281-2089-8. DOI: 10.1109/HOTCHIPS.2019.8875641.

[72] Kim, Y. G. "M-DTM: Migration-based Dynamic Thermal Management for Heterogeneous Mobile Multi-core Processors". In: *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2015, pp. 1533–1538. URL: `https://ieeexplore.ieee.org/document/7092632`.

[73] Kim, Y. G. "An Adaptive Thermal Management Framework for Heterogeneous Multi-Core Processors". In: *IEEE Transactions on Computers* 69.6 (June 2020), pp. 894–906. ISSN: 0018-9340. DOI: `10.1109/TC.2020.2970062`.

[74] Koufaty, D., Reddy, D., Hahn, S., "Bias scheduling in heterogeneous multi-core architectures". In: *Proceedings of the 5th European conference on Computer systems - EuroSys '10*. New York, New York, USA: ACM Press, 2010, p. 125. ISBN: 9781605585772. DOI: `10.1145/1755913.1755928`.

[75] Kumar, A. "HybDTM: a coordinated hardware-software approach for dynamic thermal management". In: *2006 43rd ACM/IEEE Design Automation Conference*. IEEE, 2006, pp. 548–553. ISBN: 1-59593-381-6. DOI: `10.1109/DAC.2006.229219`.

[76] Kumar, A. "System-Level Dynamic Thermal Management for High-Performance Microprocessors". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.1 (Jan. 2008), pp. 96–108. ISSN: 0278-0070. DOI: `10.1109/TCAD.2007.907062`.

[77] Kumar, R. "Processor Power Reduction Via Single-ISA Heterogeneous Multi-Core Architectures". In: *IEEE Computer Architecture Letters* 2.1 (Jan. 2003), pp. 2–2. ISSN: 1556-6056. DOI: `10.1109/L-CA.2003.6`.

[78] Kumar, R. "Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance". In: *ACM SIGARCH Computer Architecture News* 32.2 (Mar. 2004), p. 64. ISSN: 0163-5964. DOI: `10.1145/1028176.1006707`.

[79] Kumar, R. "Heterogeneous chip multiprocessors". In: *Computer* 38.11 (Nov. 2005), pp. 32–38. ISSN: 0018-9162. DOI: `10.1109/MC.2005.379`.

[80] Kursun, E., Chen-Yong Cher, “Variation-aware thermal characterization and management of multi-core architectures”. In: *2008 IEEE International Conference on Computer Design.* IEEE, Oct. 2008, pp. 280–285. ISBN: 978-1-4244-2657-7. DOI: 10.1109/ICCD.2008.4751874.

[81] Lakshminarayana, N. B., Lee, J., Kim, H., “Age based scheduling for asymmetric multiprocessors”. In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis - SC ’09.* New York, New York, USA: ACM Press, 2009, p. 1. ISBN: 9781605587448. DOI: 10.1145/1654059.1654085.

[82] Lall, P., Pecht, M. G., Hakim, E. B., *Influence of Tempemture on Microelectronics and System Reliability.* CRC Press, July 2020. ISBN: 9780138750879. DOI: 10.1201/9780138750879.

[83] Lee, Y., Shin, K. G., Chwa, H. S., “Thermal-Aware Scheduling for Integrated CPUs–GPU Platforms”. In: *ACM Transactions on Embedded Computing Systems* 18.5s (Oct. 2019), pp. 1–25. ISSN: 1539-9087. DOI: 10.1145/3358235.

[84] Li, T. “Efficient operating system scheduling for performance-asymmetric multi-core architectures”. In: *Proceedings of the 2007 ACM/IEEE conference on Supercomputing - SC ’07.* New York, New York, USA: ACM Press, 2007, p. 1. ISBN: 9781595937643. DOI: 10.1145/1362622.1362694.

[85] *Linux documentation: CAS.* 2021. URL: https://www.kernel.org/doc/html/latest/scheduler/sched-capacity.html (visited on 04/11/2021).

[86] *Linux documentation: CFS.* 2021. URL: https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html (visited on 04/11/2021).

[87] *Linux documentation: EAS.* 2021. URL: https://www.kernel.org/doc/html/latest/scheduler/sched-energy.html (visited on 04/11/2021).

[88] Lozi, J.-P. “The Linux scheduler”. In: *Proceedings of the Eleventh European Conference on Computer Systems.* New York, NY, USA: ACM, Apr. 2016, pp. 1–16. ISBN: 9781450342407. DOI: 10.1145/2901318.2901326.

[89] Luba, L. *memory: samsung: exynos5422-dmc: Add module param to control IRQ mode.* 2020. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=4fc9a0470d2dc370289e9d883feb41e5dd2c6303.

[90] Luba, L. *memory: samsung: exynos5422-dmc: Adjust polling interval and uptreshold.* 2020. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=74ca9e46107879551e2625bfbfbfd71da1881b82.

[91] Mandal, S. K. "Dynamic Resource Management of Heterogeneous Mobile Platforms via Imitation Learning". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.12 (Dec. 2019), pp. 2842–2854. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2019.2926106.

[92] Markoff, J. *Intel's Big Shift After Hitting Technical Wall.* 2004. URL: https://www.nytimes.com/2004/05/17/business/technology-intel-s-big-shift-after-hitting-technical-wall.html (visited on 03/19/2021).

[93] Mattson, R. "Evaluation techniques for storage hierarchies". In: *IBM Systems Journal* 9 (2 1970), pp. 78–117. ISSN: 0018-8670. DOI: 10.1147/sj.92.0078.

[94] McVoy, L., Staelin, C., "Lmbench: Portable Tools for Performance Analysis". In: *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference.* ATEC '96. San Diego, CA: USENIX Association, 1996, p. 23.

[95] Mead, C. A. "Scaling of MOS technology to submicrometer feature sizes". In: *Analog Integrated Circuits and Signal Processing* 6.1 (July 1994), pp. 9–25. ISSN: 0925-1030. DOI: 10.1007/BF01250732.

[96] Mednick, E. H., McLellan, E., "Instruction subset implementation for low power operation". U.S. pat. 10698472B2. Advanced Micro Devices Inc. June 2020. URL: https://patents.google.com/patent/US10698472B2/.

[97] Meza, J. "A Large-Scale Study of Flash Memory Failures in the Field". In: *ACM SIGMETRICS Performance Evaluation Review* 43.1 (June 2015), pp. 177–190. ISSN: 0163-5999. DOI: `10.1145/2796314.2745848`.

[98] Mihailescu, M. *ARM: dts: exynos: Add CPU perf counters to Exynos54xx boards*. 2017. URL: `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=c4f2fc00defc65950dfabce7a4c70cd2a289111d`.

[99] Monchiero, M., Canal, R., González, A., "Design space exploration for multicore architectures". In: *Proceedings of the 20th annual international conference on Supercomputing - ICS '06*. New York, New York, USA: ACM Press, 2006, p. 177. ISBN: 1595932828. DOI: `10.1145/1183401.1183428`.

[100] Moore, G. E. "Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff." In: *IEEE Solid-State Circuits Society Newsletter* 11.3 (Sept. 2006), pp. 33–35. ISSN: 1098-4232. DOI: `10.1109/N-SSC.2006.4785860`.

[101] Moore, G. E. "Progress in digital integrated electronics [Technical literaiture, Copyright 1975 IEEE. Reprinted, with permission. Technical Digest. International Electron Devices Meeting, IEEE, 1975, pp. 11-13.]" In: *IEEE Solid-State Circuits Society Newsletter* 11.3 (Sept. 2006), pp. 36–37. ISSN: 1098-4232. DOI: `10.1109/N-SSC.2006.4804410`.

[102] Muthukaruppan, T. S. "Hierarchical power management for asymmetric multi-core in dark silicon era". In: *Proceedings of the 50th Annual Design Automation Conference on - DAC '13*. New York, New York, USA: ACM Press, 2013, p. 1. ISBN: 9781450320719. DOI: `10.1145/2463209.2488949`.

[103] Nagarajan, V. "A Primer on Memory Consistency and Cache Coherence". In: *Synthesis Lectures on Computer Architecture* 15 (1 Feb. 2020), pp. 1–294. ISSN: 1935-3235. DOI: `10.2200/S00962ED2V01Y201910CAC049`.

[104] Niu, Q. “PARDA: A Fast Parallel Reuse Distance Analysis Algorithm”. In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. IEEE, May 2012, pp. 1284–1294. ISBN: 978-1-4673-0975-2. DOI: `10.1109/IPDPS.2012.117`.

[105] Olken, F. “Efficient methods for calculating the success function of fixed-space replacement policies”. MA thesis. Berkeley, CA (United States): Lawrence Berkeley National Laboratory (LBNL), May 1981. DOI: `10.2172/6051879`.

[106] Pallipadi, V., Starikovskiy, A., “The ondemand governor: past, present and future”. In: *Proceedings of the Linux Symposium*. 2006, pp. 215–230. URL: `https://www.kernel.org/doc/ols/2006/ols2006v2-pages-223-238.pdf` (visited on 04/01/2020).

[107] Pallister, J. “Data Dependent Energy Modeling for Worst Case Energy Consumption Analysis”. In: *Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems*. New York, NY, USA: ACM, June 2017, pp. 51–59. ISBN: 9781450350396. DOI: `10.1145/3078659.3078666`.

[108] Pedregosa, F. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830. DOI: `10.48550/arXiv.1201.0490`.

[109] Pricopi, M. “Power-performance modeling on asymmetric multi-cores”. In: *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. IEEE, Sept. 2013, pp. 1–10. ISBN: 978-1-4799-1400-5. DOI: `10.1109/CASES.2013.6662519`.

[110] Reddy, B. K. “Inter-Cluster Thread-to-Core Mapping and DVFS on Heterogeneous Multi-Cores”. In: *IEEE Transactions on Multi-Scale Computing Systems* 4.3 (July 2018), pp. 369–382. ISSN: 2332-7766. DOI: `10.1109/TMSCS.2017.2755619`.

[111] Reddy, B. K. “AdaMD: Adaptive Mapping and DVFS for Energy-Efficient Heterogeneous Multicores”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.10 (Oct. 2020), pp. 2206–2217. ISSN: 0278-0070. DOI: 10.1109/TCAD.2019.2935065.

[112] Roy, K., Mukhopadhyay, S., Mahmoodi-Meimand, H., “Leakage current mechanisms and leakage reduction techniques in deep-submicrometer CMOS circuits”. In: *Proceedings of the IEEE* 91.2 (Feb. 2003), pp. 305–327. ISSN: 0018-9219. DOI: 10.1109/JPROC.2002.808156.

[113] Saez, J. C. “Leveraging workload diversity through OS scheduling to maximize performance on single-ISA heterogeneous multicore systems”. In: *Journal of Parallel and Distributed Computing* 71.1 (Jan. 2011), pp. 114–131. ISSN: 07437315. DOI: 10.1016/j.jpdc.2010.08.020.

[114] Saez, J. C. “Leveraging Core Specialization via OS Scheduling to Improve Performance on Asymmetric Multicore Systems”. In: *ACM Transactions on Computer Systems* 30.2 (Apr. 2012), pp. 1–38. ISSN: 0734-2071. DOI: 10.1145/2166879.2166880.

[115] Sahin, O., Coskun, A. K., “On the Impacts of Greedy Thermal Management in Mobile Devices”. In: *IEEE Embedded Systems Letters* 7.2 (June 2015), pp. 55–58. ISSN: 1943-0663. DOI: 10.1109/LES.2015.2420664.

[116] Sahin, O., Coskun, A. K., “QScale: Thermally-Efficient QoS Management on Heterogeneous Mobile Platforms”. In: *Proceedings of the 35th International Conference on Computer-Aided Design.* New York, NY, USA: ACM, Nov. 2016, pp. 1–8. ISBN: 9781450344661. DOI: 10.1145/2966986.2967066.

[117] Sahin, O., Thiele, L., Coskun, A. K., “Maestro: Autonomous QoS Management for Mobile Applications Under Thermal Constraints”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.8 (Aug. 2019), pp. 1557–1570. ISSN: 0278-0070. DOI: 10.1109/TCAD.2018.2855180.

[118] Sahin, O., Varghese, P. T., Coskun, A. K., "Just enough is more: Achieving sustainable performance in mobile devices under thermal limitations". In: *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, Nov. 2015, pp. 839–846. ISBN: 978-1-4673-8388-2. DOI: `10.1109/ICCAD.2015.7372658`.

[119] Sakalis, C. "Splash-3: A properly synchronized benchmark suite for contemporary research". In: *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, Apr. 2016, pp. 101–111. ISBN: 978-1-5090-1953-3. DOI: `10.1109/ISPASS.2016.7482078`.

[120] Samsung, *Exynos 5 Octa (5422)*. 2014. URL: `https://www.samsung.com/semiconductor/minisite/exynos/products/mobileprocessor/exynos-5-octa-5422/` (visited on 04/01/2020).

[121] Sergent, J. E., Krum, A., *Thermal Management Handbook: For Electronic Assemblies*. Electronic packaging and interconnection series. McGraw-Hill Education, June 1998. ISBN: 9780070266995. URL: `https://books.google.fr/books?id=J3nty7eHkhcC`.

[122] Shelepov, D., Fedorova, A., "Scheduling on Heterogeneous Multicore Processors Using Architectural Signatures". In: *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture*. 2008, pp. 21–25. URL: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.217.4834`.

[123] Shelepov, D. "HASS: A Scheduler for Heterogeneous Multicore Systems". In: *ACM SIGOPS Operating Systems Review* 43.2 (Apr. 2009), pp. 66–75. ISSN: 0163-5980. DOI: `10.1145/1531793.1531804`.

[124] Shen, J. P., Lipasti, M. H., *Modern processor design: Fundamentals of superscalar processors*. 2013. ISBN: 978-1-4786-0783-0.

[125] Singla, G. "Predictive Dynamic Thermal and Power Management for Heterogeneous Mobile Platforms". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015*. New Jersey: IEEE Conference Publications, 2015, pp. 960–965. ISBN: 9783981537048. DOI: `10.7873/DATE.2015.1036`.

[126] Skadron, K., Abdelzaher, T., Stan, M., "Control-theoretic techniques and thermal-RC modeling for accurate and localized dynamic thermal management". In: *Proceedings Eighth International Symposium on High Performance Computer Architecture*. IEEE Computer. Soc, 2002, pp. 17–28. ISBN: 0-7695-1525-8. DOI: `10.1109/HPCA.2002.995695`.

[127] Southern, G., Renau, J., "Analysis of PARSEC workload scalability". In: *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, Apr. 2016, pp. 133–142. ISBN: 978-1-5090-1953-3. DOI: `10.1109/ISPASS.2016.7482081`.

[128] Srinivasan, J. "Exploiting Structural Duplication for Lifetime Reliability Enhancement". In: *32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE, 2005, pp. 520–531. ISBN: 0-7695-2270-X. DOI: `10.1109/ISCA.2005.28`.

[129] Stephens, N. "The ARM Scalable Vector Extension". In: *IEEE Micro* 37.2 (Mar. 2017), pp. 26–39. ISSN: 0272-1732. DOI: `10.1109/MM.2017.35`.

[130] Stevens, A. *White paper: Introduction to AMBA® 4 ACE™ and big.LITTLE™ Processing Technology*. 2013.

[131] Szyprowski, M., Kozlowski, K., Wolff, W., *ARM: dts: exynos: Disable frequency scaling for FSYS bus on Odroid XU3 family*. 2020. URL: `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=9ff416cf45a08f28167b75045222c762a0347930`.

[132] Szyprowski, M., Wolff, W., Kozlowski, K., *ARM: dts: exynos: Disable frequency scaling for FSYS bus on Odroid XU3 family*. 2020. URL: `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=9ff416cf45a08f28167b75045222c762a0347930`.

[133] Taylor, B. "Adaptive deep learning model selection on embedded systems". In: *Proceedings of the 19th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. New York, NY, USA: ACM, June 2018, pp. 31–43. ISBN: 9781450358033. DOI: `10.1145/3211332.3211336`.

[134] Taylor, M. B. "Is dark silicon useful? Harnessing the Four Horsemen of the Coming Dark Silicon Apocalypse". In: *Proceedings of the 49th Annual Design Automation Conference on - DAC '12*. New York, New York, USA: ACM Press, 2012, p. 1131. ISBN: 9781450311991. DOI: `10.1145/2228360.2228567`.

[135] Torvalds, L. *test-tlb*. 2017. URL: `https://github.com/torvalds/test-tlb` (visited on 04/01/2020).

[136] Van Craeynest, K. "Scheduling heterogeneous multi-cores through performance impact estimation (PIE)". In: *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, June 2012, pp. 213–224. ISBN: 978-1-4673-0476-4. DOI: `10.1109/ISCA.2012.6237019`.

[137] Viswanath, R. "Thermal Performance Challenges from Silicon to Systems". In: *Intel Technology Journal* 4.3 (2000), pp. 1–16. URL: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.8322`.

[138] Wachter, E. W. "Predictive Thermal Management for Energy-Efficient Execution of Concurrent Applications on Heterogeneous Multicores". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.6 (June 2019), pp. 1404–1415. ISSN: 1063-8210. DOI: `10.1109/TVLSI.2019.2896776`.

[139] Waldspurger, C. A. “Efficient MRC Construction with SHARDS”. In: *Proceedings of the 13th USENIX Conference on File and Storage Technologies.* FAST’15. Santa Clara, CA: USENIX Association, 2015, pp. 95–110. ISBN: 9781931971201. URL: `https://www.usenix.org/conference/fast15/technical-sessions/presentation/waldspurger` (visited on 03/19/2021).

[140] Walker, M. J. “Accurate and Stable Run-Time Power Modeling for Mobile and Embedded CPUs”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 36.1 (Jan. 2017), pp. 106–119. ISSN: 0278-0070. DOI: `10.1109/TCAD.2016.2562920`.

[141] Weiss, A. R. *Dhrystone Benchmark: History, analysis, scores and recommendations.* Tech. rep. EEMBC Certification Laboratories, LLC, Nov. 2002. URL: `https://www.johnloomis.org/NiosII/dhrystone/ECLDhrystoneWhitePaper.pdf` (visited on 03/19/2021).

[142] Wolff, W. *ARM: dts: exynos: add CCI-400 PMU nodes support to Exynos542x SoCs.* Under review. 2019. URL: `https://lore.kernel.org/patchwork/patch/1061141/`.

[143] Wolff, W., Kozlowski, K., Zolnierkiewicz, B., *ARM: dts: exynos: fix incomplete Odroid-XU3/4 thermal-zones definition.* 2017. URL: `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=e740731dae9470f7fb86efa643ec881a66d4e4c0`.

[144] Wolff, W., Lezcano, D., Kumar, V., *thermal/drivers/cpufreq_cooling: Fix return of cpufreq_set_cur_state.* 2020. URL: `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=ff44f672d74178b3be19d41a169b98b3e391d4ce`.

[145] Wolff, W., Porter, B., “Performance Optimization on big.LITTLE Architectures: A Memory-latency Aware Approach”. In: *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Em-*

bedded Systems. New York, NY, USA: ACM, June 2020, pp. 51–61. ISBN: 9781450370943. DOI: `10.1145/3372799.3394370`.

[146] Wolff, W., Porter, B., *What am I waiting for? Energy and Performance Optimization on big.LITTLE Architectures: A Memory-latency Aware Approach*. Dec. 2020.

[147] Wolff, W., Rui, Z., *thermal: fix source code documentation for parameters*. 2017. URL: `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=0d76d6e1eede5f2aa13695cb4c9d763bb3555e3e`.

[148] Wolff, W. *gem5: config, arm: memoryMode test*. 2019. URL: `https://gem5.googlesource.com/public/gem5/+/ea088f5150d03d4481555ecbbfa2afba3a87468a`.

[149] Woo, S. "The SPLASH-2 programs: characterization and methodological considerations". In: *Proceedings 22nd Annual International Symposium on Computer Architecture*. ACM, 1995, pp. 24–36. ISBN: 0-89791-698-0. DOI: `10.1109/ISCA.1995.524546`.

[150] Wu, C.-J. "Machine Learning at Facebook: Understanding Inference at the Edge". In: *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 331–344. DOI: `10.1109/HPCA.2019.00048`.

[151] Ye, G. "A Video-based Attack for Android Pattern Lock". In: *ACM Transactions on Privacy and Security* 21.4 (Nov. 2018), pp. 1–31. ISSN: 2471-2566. DOI: `10.1145/3230740`.

[152] Yeo, I., Liu, C. C., Kim, E. J., "Predictive dynamic thermal management for multicore systems". In: *Proceedings of the 45th annual conference on Design automation - DAC '08*. New York, New York, USA: ACM Press, 2008, p. 734. ISBN: 9781605581156. DOI: `10.1145/1391469.1391658`.

[153] Zhang, H. “Temperature dependence of soft-error rates for FF designs in 20-nm bulk planar and 16-nm bulk FinFET technologies”. In: *2016 IEEE International Reliability Physics Symposium (IRPS)*. IEEE, Apr. 2016, pp. 5C–3–1–5C–3–5. ISBN: 978-1-4673-9137-5. DOI: `10.1109/IRPS.2016.7574554`.

[154] Zhou, X. “Performance-aware thermal management via task scheduling”. In: *ACM Transactions on Architecture and Code Optimization TACO* 7.1 (2010), pp. 1–31. ISSN: 15443566. DOI: `10.1145/1746065.1736070`.

[155] Zhou, Z., Gu, J., Qu, G., “Scheduling for Multi-core Processor under Process and Temperature Variation”. In: *2012 IEEE 6th International Symposium on Embedded Multicore SoCs*. IEEE, Sept. 2012, pp. 113–120. ISBN: 978-1-4673-2535-6. DOI: `10.1109/MCSoC.2012.9`.