# Efficient Power Management for Heterogeneous Multi-Core Architectures

Thannirmalai Muthukaruppan Somu

*(B.S, State University of New York, Buffalo, 2009)*

**A THESIS SUBMITTED**

**FOR THE DEGREE OF DOCTOR OF PHILOSOPHY**

**DEPARTMENT OF COMPUTER SCIENCE**

**SCHOOL OF COMPUTING**

**NATIONAL UNIVERSITY OF SINGAPORE**

**2014**

# Declaration

- I hereby declare that this thesis is my original work and it has been written by me in its entirety.
- I have duly acknowledged all the sources of information which have been used in the thesis.
- This thesis has also not been submitted for any degree in any university previously.

---

Thannirmalai Muthukaruppan Somu

July 2014

# *Acknowledgements*

The research contributions in this thesis would not have been possible without the guidance, advice, mentorship and supervision of several people.

First and foremost, I would like to express my deepest gratitude to my advisor Tulika Mitra. I still remember the day when I joined her group. I was raw and lacking all the essential skills required to become a good researcher. Not to mention that my knowledge in computer architecture was near zero. She was patient and successful in transforming me into a decent researcher despite being a high-maintenance student. Throughout the years, I have acquired valuable knowledge from her, which helped me grow both professionally and personally. Her vision, commitment and ambition towards good quality contributions have shaped my personality in many ways. Her level of commitment to the most intricate details of a problem has always amazed me. She has always been supportive and caring, especially during my difficult times. I feel eternally indebted to her and respect her as a son respects a caring mother.

Besides my advisor, I would like to thank Prof. Wong Weng Fai and Prof. Colin Tan for their invaluable and intriguing comments that has shaped this research work. I am highly indebted to Cambridge Silicon Radio plc (CSR) for their generous financial and logistics (board) support without which this thesis would not have been possible. I am also very thankful to thank Sanjay Vishin from CSR for all the productive discussions. His critical thinking and intellectual foundation have influenced the contributions in this thesis in many ways.

There is no shortage of my fellow colleagues and collaborators to thank. First, I would like to sincerely thank Haris Javaid from UNSW. Haris has made me understand on how to present an idea to a wider audience in a convincing manner. I will remember his mentorship and guidance for life. From the day I joined the eCO lab (that is what we call ourselves now), Mihai has always been there to

listen, comment, positively criticize and support in all my research endeavours. I thank him for showing me how a researcher should quantitatively evaluate an idea in an effective manner. I would like to thank Vanchi for patiently listening to my rants, crazy ideas and philosophical believes. And more importantly he was instrumental in keeping me sane in the lab. The best time of my PhD was during my collaborations with Mihai and Vanchi. Thanks guys for giving an awesome and memorable time. I am grateful for Chen Liang for traveling this journey of PhD together in all the ups and downs. I would also like to thank Anuj for his support. His eagerness to develop numerous ideas in very short span is astonishing. I would like to thank all my lab mates: Huping, Chundong, Sudipta, Alok, Lee Kee, Tan Cheng, Henry and Jiao Qing for keeping an healthy research environment. A special thanks goes to Mahesh, without whom I would have never met my advisor.

I was fortunate enough to meet lots of nice people in Singapore. Their friendship and kindness helped me sail through the ups and downs of my life in Singapore. Each and everyone one of them have touched my heart in a very positive manner. Thanks to P-boy, SK, Director, Kauntz, Raaju, Poli samiyar, PM and Gii. I thank all the mamis (SK, P-boy and TKB wives) for providing enough home cooked and healthy food. My sincere thanks goes to Badri Mama, Manavalan Mama and TKB for enriching the spiritual side of my life. A special thanks goes to Ancy Alexander for his guidance about life in general.

Last, but certainly not the least, I would like to acknowledge my family. I would not be who I am today without their support. I owe everything to my family. Mani, my brother, has been instrumental in supporting and guiding me in all the major crucial phases of my life. I am always grateful for his passion to see me grow in life. Appa and Amma have always trusted and encouraged me in numerous ways. Appa, you have always been a great role model for me right

from my childhood. Amma, as a token of your immeasurable love and support I would like to dedicate this thesis to you.

# Contents

# *Abstract*

Relentless Complementary Metal-Oxide Semiconductor (CMOS) scaling at deep sub-micron level has resulted in increased power density in microprocessor, which forced the computing systems to move in the direction of parallel architectures with homogeneous multi-cores. However, the emergence of dynamic and diverse workloads combined with the failure of Dennard Scaling facilitated the growth of heterogeneous multi-cores. The presence of heterogeneity enables better match between application demand and computation capabilities leading to substantially improved performance and energy-efficiency. In spite of significant benefits in terms of both performance and energy consumption, the heterogeneous multi-core systems introduce many of design and scheduling challenges. In this thesis, we address various challenges involved in designing heterogeneous multi-cores.

In the first part of this thesis, we focus on developing power management schemes for heterogeneous multi-cores that can satisfy application's demand with low energy consumption under the Thermal Design Power (TDP) constraint. First, we develop a performance and power model of heterogeneous cores having different performance and power consumption characteristics that can be used in any predictive scheduling approach. Second, we propose two reactive power management frameworks: *Hierarchical Power Management* (HPM) and *Price theory based Power Management* (PPM). All the aforementioned dynamic power management frameworks were evaluated on a real Advance RISC Machines (ARM) big.LITTLE heterogeneous multi-core platform. Our experimental evaluations establish the superiority of the power management schemes compared to the existing state-of-the-art techniques. Lastly, we propose a power-aware dynamic reliability management technique that can meet both reliability and thermal/power constraints, while optimizing the performance.

In the second part of this thesis, we propose a comprehensive framework that help to design the most energy-efficient application-specific Multi-Processor System on Chips (MPSoCs). We model the synthesis of energy-efficient MPSoC as a design space exploration problem involving four design parameters: DVFS, processor customization, cache customization and task mapping. Experiments reveal that our framework can reduce energy consumption compared to solutions obtained from a combination of existing techniques.

Overall in this thesis, we address power consumption related challenges exhibited in heterogeneous multi-core systems by proposing both static and dynamic power management techniques. While the first part of the thesis focuses on the dynamic techniques, the second part elaborates the static solutions.

# List of Tables

# List of Figures

*For Amma, Appa and Mani. . .*

# Related Publications

T. S. Muthukaruppan, and T. Mitra. Lifetime Reliability Aware Architectural Adaptation. In *IEEE International Conference on VLSI Design and 2013 12th International Conference on Embedded Systems (**VLSID**)*, 2013.

T. S. Muthukaruppan, M. Pricopi, V. Venkataramani, T. Mitra and S. Vishin. Hierarchical power management for asymmetric multi-core in dark silicon era. In *ACM Proceedings of the 50th Annual Design Automation Conference (**DAC**)*, 2013.

M. Pricopi, T. S. Muthukaruppan, V. Venkataramani, T. Mitra and S. Vishin. Power-performance modeling on asymmetric multi-cores. In *IEEE International Conference on Compilers, Architecture and Synthesis for Embedded Systems (**CASES**)*, 2013.

T. S. Muthukaruppan, H. Javaid, T. Mitra and S. Parameswaran. Energy-aware synthesis of application specific MPSoCs. In *IEEE International Conference on Computer Design (**ICCD**)*, 2013.

T. S. Muthukaruppan, A. Pathania and T. Mitra. Price theory based power management for heterogeneous multi-cores. In *ACM Architectural Support for Programming Languages & Operating Systems (**ASPLOS**)*, 2014.

# Chapter 1

# Introduction

In the modern era, computers have penetrated all facets of human life. They have revolutionized the way we think, interact and perform our day-to-day activities. One of the reasons for this indispensable addiction is the variety of features that they offer such as recreation, health-care, transportation etc. We use computers in various forms and sizes such as laptops, tablets, smart phones, etc. even being oblivious to their presence at times. The increasing number of computing devices have inevitably led to an increasing demand on energy resources. Hence, it is crucial to develop energy-efficient computers – a design choice that helps in designing computers that are small, fast, efficient and generate less heat. *Heterogeneous computing* has emerged as a popular design option for realizing energy-efficient computers. In this thesis, we discuss and develop heterogeneous systems that have a positive impact on the energy consumption.

## 1.1 Motivation and Objective

The significant compound annual growth rate of 14% [6] for the microprocessor industry in the past 40 years is heavily attributed to the success of Moore's law [85], which states that the number of transistors doubles once every eighteen

| | |
|---|---|
| Dimensions | 0.7x ↓ |
| Voltage | 0.7x ↓ |
| Doping Concentrations | 0.7x ↓ |
| Area | 0.5x ↓ |
| Capacitance | 0.7x ↓ |
| Frequency | 1.4x ↑ |
| Power | 0.5x ↓ |

FIGURE 1.1: Dennard's constant field scaling.

months. This is achieved by scaling various transistor dimensions like channel length, channel width, and oxide thickness. The main challenge in integrating more transistors across generations is to prevent the chip from melting. It is in fact the Dennard Scaling [30] that has enabled the success of Moore's law for the past 40 years. Figure 1.1 shows the scaling factor for Dennard's constant electric field scaling. According to Dennard Scaling, for complementary metal–oxide–semiconductor (CMOS) transistors, scaling the dimensions, voltage and doping concentrations by 0.7 times results in an area reduction of 0.5 times of the original transistor. Similarly, the capacitance reduces by a factor of 0.7 times, while the frequency increases by a factor of 1.4 times. The dynamic power consumption of a transistor is given by the formula $Capacitance \times Frequency \times Voltage^2$. Therefore, for constant electric field, ideally the power consumption of the transistor reduces by the factor of 0.5 times. Therefore, at every new process technology, the power consumption scales by the same factor as the area, which results in constant power density in the chip. It is the fusion of Moore's law with Dennard Scaling that resulted in exponential performance increase in microprocessors.

Unfortunately, Dennard Scaling has started failing in recent generations due to the relatively slow scaling of supply voltage, resulting in increased dynamic power density. The non-ideal scaling of supply voltage is attributed to the following reasons: a) need for higher performance, which can be obtained only at high

supply voltage and b) relatively stagnant threshold voltage to control the static power consumption. Thus, as more and more of transistors are integrated in the same area in the future generations, the power density will increase rapidly. The increase in power density has resulted in increase in on-chip temperature of the microprocessors. High on-chip temperatures can affect the following features:

- **Leakage Power:** There exists positive feedback relationship between the leakage power and the temperature [77, 112]. Increase in temperature results in increasing the leakage power, which in turn can increase the temperature resulting in a thermal runway.
- **Reliability:** Extensive studies [108] have shown that the lifetime reliability of microprocessors is significantly affected by the high on-chip temperatures. The advent of various failure mechanisms like electro migration, stress migration, gate oxide breakdown, and thermal cycles surges with high on-chip temperature.

Traditionally, the researchers have relied upon packaging and cooling technologies (heat sink, convection resistance, fan etc.) to bring down the high temperatures in modern microprocessors. The maximum power dissipation handled by the given packaging and cooling solutions is defined as Thermal Design Power (TDP). The chips with higher TDP limits have better cooling solutions. Unfortunately, as we are already in the era of mobility, integrating advanced cooling solutions to mobile devices is both expensive and infeasible. From the above discussions, it is clear that reducing power dissipation to lower on-chip temperature is the most important design goal in modern high performance microprocessors.

For continued adherence to Moore's law and to combat the increase in power consumption, the computing systems have made an irreversible transition towards parallel architectures with multi-cores and many cores. From the virtue of the power model described in [86], for the same compute capability, the power

consumption of a dual core reduces by four times compared to that of a single microprocessor. However, with continued non-ideal CMOS scaling, power and thermal limits are rapidly bringing the computing community to another cross-road where a chip can have many cores but a significant fraction of them are left un-powered, or dark, at any point in time [37]. This phenomenon, known as dark silicon, is immediately visible in the computing space due to the increasing cooling costs of the chip. Furthermore, the emergence of sophisticated and power hungry mobile applications like speech processing, pattern recognition, audio/video editing etc. have further exacerbated the power challenges in the mobile devices.

The dark silicon era is driving the emergence of heterogeneous multi-cores, which exhibit diverse power/performance characteristics. Unlike homogeneous multi-cores, exploiting the potential of heterogeneous multi-cores is not straightforward. First, the major challenge in designing heterogeneous multi-cores is how to efficiently explore the complex design space so as to improve the efficiency of the power-performance tradeoff. Secondly, for static and pre-designed heterogeneous multi-cores, the capability can only be fully exploited with a proper online scheduling support. Hence, it is imperative that both the design of heterogeneous multi-core and scheduling should be prudently crafted.

The most popular choice of mechanism for power reduction is dynamic voltage and frequency scaling (DVFS). Few recent works [96, 108] have claimed that there is a decrease in overall lifetime reliability of the microprocessors due to aggressive power management policies. For example, frequent voltage-frequency (v-f) levels transition can introduce thermal cycling, which can significantly reduce the mean time to failure (MTTF) of the microprocessors. Hence, it is also important to design power management scheme that has minimal impact on the lifetime reliability.

The above discussions motivate the need for efficient power management schemes for heterogeneous multi-cores that can exhibit following desirable features:

- The power should not be allowed to exceed the power budget defined by TDP.
- The performance requirements of various applications have to be met under the power budget with minimal energy consumption.
- The reduction in power consumption should not come at the expense of sacrificing the lifetime reliability of the microprocessor.

To meet the above challenges and fulfill the objectives, we propose efficient power management schemes in this thesis. This work investigates various power management schemes like DVFS, task migrations, load balancing, custom instruction selection etc. in a detailed manner.

## 1.2 Contributions

This thesis makes following key contributions (as shown in Figure 1.2):

- We develop a power-performance model [92] for commercial heterogeneous multi-core: ARM big.LITTLE. Our model can be deployed with any prediction based dynamic power management scheme.
- We propose two reactive dynamic power management schemes based on the strong foundations of *control theory* [90] and *price theory* [89].
- We explore the effect of heterogeneity in terms of micro-architectural adaptation on the lifetime reliability of microprocessors [88].
- We also propose a comprehensive framework for synthesis of application specific MPSoC for multimedia applications. Our framework searches for

a design with minimum energy consumption under area and period constraints [87].

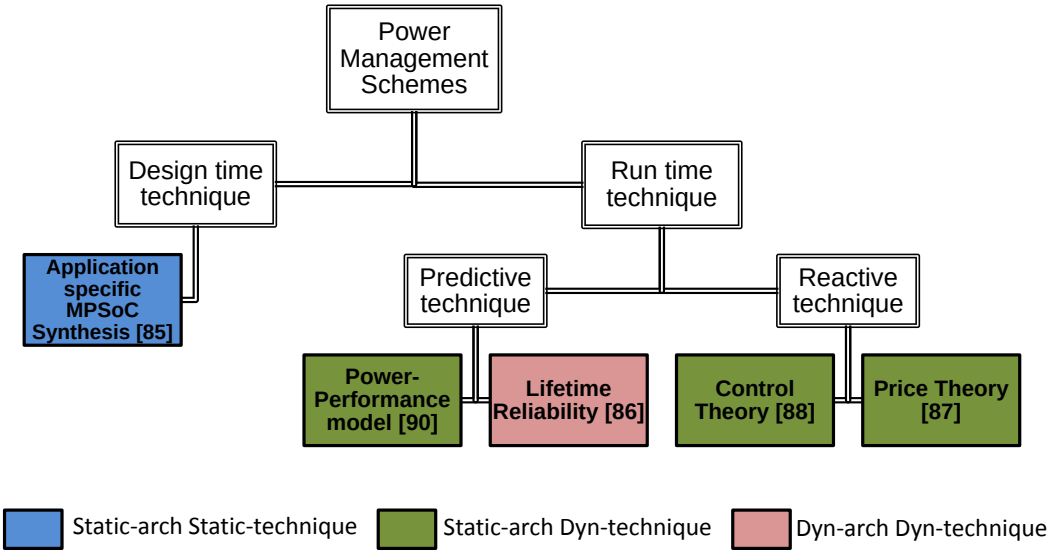



FIGURE 1.2: Overall Contributions of the thesis.

### 1.2.1 Run-time technique

#### 1.2.1.1 Predictive power management

The ability to estimate the performance/power characteristics for various workloads for each core type in heterogeneous multi-cores can solve the scheduling challenges in determining the best workload-to-core mapping. Hence, in the first contribution, we develop power-performance model for ARM big.LITTLE. While an application is executing on ARM Cortex-A7 (alternatively ARM Cortex-A15), we collect profile information provided by hardware counters, and estimate power and performance characteristics of the same application on ARM Cortex-A15 (alternatively ARM Cortex-A7). We evaluate the accuracy of our estimation on real ARM big.LITTLE hardware platform. Our evaluations clearly states the accuracy of our power-performance model. We also develop a scheduling algorithm

based on the proposed estimation model for ARM big.LITTLE heterogeneous multi-core.

#### 1.2.1.2 Reactive power management

The second contribution of this thesis is to propose a dynamic power management framework for heterogeneous multi-cores like ARM big.LITTLE in mobile platforms, that can satisfy application's demand expressed in terms of Quality of Service (QoS) with low energy consumption under Thermal Design Power (TDP) constraint. We propose two reactive run-time power management frameworks.

First, we propose *Hierarchical Power Management* (HPM) [90] for heterogeneous multi-cores – in particular ARM big.LITTLE [7] (as shown in Figure 1.3) architecture in the context of mobile embedded platforms — that can provide satisfactory user experience while minimizing energy consumption within the Thermal Design Power (TDP) constraint. Our HPM framework is based on the solid foundation of control theory and integrates multiple controllers to collectively achieve the goal of optimal energy-performance tradeoff under restricted power budget. Second, we propose *Price theory based Power Management* (PPM) [89] for heterogeneous multi-cores that can contain any number of clusters of different core types (unlike HPM which can handle only at most two clusters with each containing different core types). Our PPM framework borrows strong basics from the concept of *price theory* from economics, which makes the technique scalable, holistic and priority-driven.

Aforementioned techniques (HPM and PPM) have been build as an extension of *Linux completely-fair scheduler* while preserving all of its desirable properties such as fairness, non- starvation etc. Finally, both the frameworks have been implemented on a test version of the ARM big.LITTLE heterogeneous multi-core architecture and we report power, performance results from this real chip (as

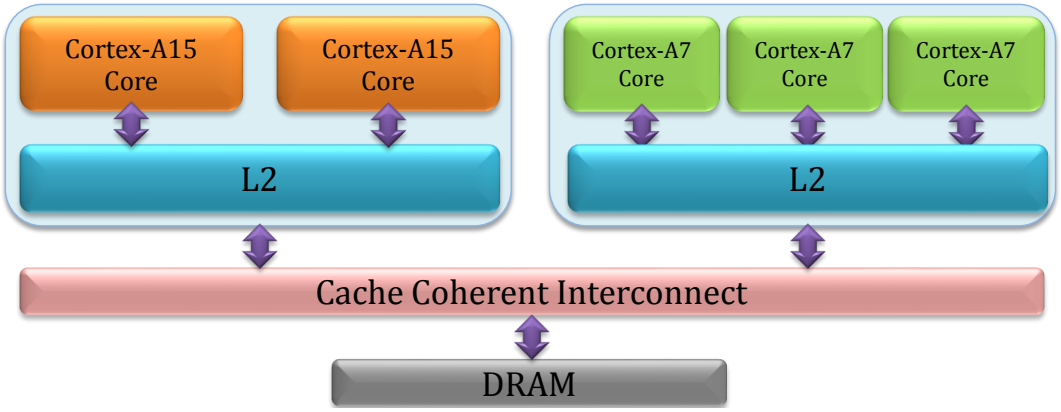



FIGURE 1.3: ARM big.LITTLE asymmetric multi-core.

opposed to simulation). We experimentally evaluate and establish the superiority of our approaches compared to the existing state-of-the-art.

#### 1.2.1.3 Lifetime-reliability aware power management

The third contribution of this thesis is to propose a dynamic reliability management technique for lifetime reliability enhancement via micro-architectural adaptations. We propose a dynamic reliability management (DRM) technique that exploits architectural adaptation in conjunction with dynamic voltage/frequency scaling (DVFS). In this contribution, the heterogeneity is evident from the dynamic architectural adaptation. We employ an online Bayesian classifier that can efficiently detect the reliable configurations, while a performance prediction model selects the one with best performance among all the reliable configurations. We later extend our approach to meet both reliability and thermal constraints. The thermal constraints act as proxy for power constraints.

### 1.2.2 Design-time technique

The final contribution of this thesis is a framework for design of heterogeneous application-specific MPSoC for multimedia applications [87]. Modern MPSoCs

for multimedia applications have to deliver a certain performance to provide reasonable quality of service to the users (performance constraint), must have area smaller than a certain limit due to the size of the portable devices (area constraint), and should have low energy consumption to increase the battery life. Therefore, application specific MPSoCs are deployed in portable devices [41] where an MPSoC is (extremely) customized for a given application under an objective function and various constraints. This contribution focuses on customization of MPSoCs for multimedia applications with the objective of minimum energy consumption under performance and area constraints.

To summarize, the run time techniques [89, 90, 92] proposed in thesis are dynamic techniques on a static heterogeneous architecture except for the one proposed in [88] (which is a dynamic technique on a dynamic heterogeneous architecture), while the design time technique[87] proposed is a static technique engaged on a static heterogeneous architecture.

## 1.3 Organization

The rest of this thesis is organized as follows. Chapter 2 discusses related work. Chapter 3 discusses the power-performance estimation model for heterogeneous multi-core. Chapter 4 and 5 elaborates the various reactive based run-time power management framework for heterogeneous multi-cores. Chapter 4 proposes control theory based power management framework in detail. Chapter 5 proposes price theory based power management framework that improves on the technique explained in Chapter 4. Chapter 6 proposes a dynamic reliability management technique for microprocessors. Chapter 7 describes the static design time technique for synthesizing energy-efficient application specific MPSoC. Chapter 8 describes the conclusion of this thesis and Chapter 9 explains possible avenues of future work.

# Chapter 2

# Related Work

In this chapter, we briefly present the overview of the previously published work on power management based on the categories described in Figure 1.2. The categorization is based on the type of architecture and technique, which can be either *static* or *dynamic*. For static techniques, the mechanisms are determined at the design time. Unlike static techniques, the dynamic techniques adapt according to the workload at run-time. Similarly, in terms of architecture, static architectures are fixed at design time (for example, ARM big.LITTLE). In this thesis, we adapt micro-architectural parameters like issue-width, window size and cache sizes at run-time to emulate dynamic heterogeneous architectures.

## 2.1 Static technique - Static architecture

Power management techniques can be built into the system at the design time either in software or hardware. Static techniques are mostly applicable for embedded domain, where the hardware-software co-design is very relevant. In recent years, application specific MPSoCs have become a promising option for designing embedded portable devices, because of their high performance and low energy consumption. There is a plethora of work on designing of application specific

MPSoCs, where researchers have considered different objective functions, constraints and design parameters. We report the most relevant works categorized according to the four design parameters: DVFS, processor customization, cache customization and task mapping.

### 2.1.1 DVFS

The authors of [38, 39] used DVFS to balance workload across processors connected in a pipeline, in order to reduce their energy consumption. They proposed feedback controllers to monitor the occupancy levels of buffers in the pipeline, and either increased or decreased the v-f level of a processor accordingly. Chen et al. [23] also considered a pipeline of processors with the availability of DVFS; however, they minimized the energy consumption of the system under an end-to-end application deadline using quadratic programming.

### 2.1.2 Processor customization

Bonzini et al. [18] studied the effects on energy consumption and performance due to addition of custom instructions in an ASIP. They built an estimation model for a simplescalar-like processor to quickly evaluate different custom instructions. In [17], the authors characterized the energy benefits of extending the baseline instruction set architecture of an FPGA based soft processor. Lin et al. [76] targeted multiobjective optimization of an ASIP where custom instructions are added considering area and energy consumption. They used mixed integer linear programming for an optimal solution and a simulated annealing based heuristic for a near-optimal solution.

### 2.1.3 Cache customization

The authors of [48, 125] explored the design space of a cache (cache size, line size, associativity) to select a cache configuration with minimum energy consumption. The authors proposed a heuristic to quickly search through complex design space of cache configurations for a near-optimal solution. Rawlins et al. [95] targeted run-time tuning of L1 data cache to minimize energy consumption of a heterogeneous MPSoC architecture. They proposed a heuristic to quickly search through the design space with minimal run-time overhead.

### 2.1.4 DVFS and processor customization

Jung et al. [63] customized an MPSoC, where custom instructions and different v-f levels were used for the ASIPs in the system. They employed mixed integer linear programming to find the design point with minimum dynamic energy consumption under an area constraint.

### 2.1.5 DVFS and task mapping

Ruggiero et al. [99] considered an MPSoC with variable number of processors and DVFS. They used a design space exploration algorithm to determine the optimal number of processors and v-f levels for a given application to minimize the MPSoC's power consumption under quality of service constraints. The authors of [14] considered resource allocation and voltage selection problem in an MPSoC. They minimized MPSoC's energy consumption with the use of integer programming and constraint programming. Lu et al. [78] considered the problem of task mapping/scheduling and DVFS in homogeneous MPSoCs. They proposed a processor utilization based algorithm for task mapping and exploited the slacks available in periodic tasks to minimize energy consumption.

### 2.1.6 Processor customization and task mapping

Sun et al. [113] proposed an iterative algorithm to select custom instructions for ASIPs in an MPSoC along with the mapping and scheduling of tasks to maximally improve performance under an area constraint. A dynamic programming based algorithm was introduced in [25] to find optimal mapping of tasks on ASIPs of an MPSoC under a period constraint, where custom instructions for ASIPs and interval-based mapping were considered.

### 2.1.7 Processor customization and cache customization

The works in [59, 60, 103] considered a pipeline of ASIPs for multimedia applications. They maximized performance improvement per unit area [103] or minimized area under performance constraints [59, 60] while exploring custom instructions and cache configurations. Pruning algorithms, heuristics and integer linear programming based approaches were proposed in these works.

It is clear that none of the above works considered combined use of DVFS, processor customization, cache customization and task mapping, which has a potential to save significant amounts of energy. To the best of our knowledge, our contribution of designing heterogeneous MPSoC is the first to use these techniques together for energy minimization under performance and area constraints in application specific MPSoCs for multimedia applications.

## 2.2 Dynamic technique - Static architecture

Design time techniques are beneficial for static architectures when the workloads are known a priori. On the other hand, dynamic techniques are required for applications exhibiting phase behaviours [53] (which is difficult to capture in static

techniques). Most of the commercial mobile platforms, which are not application-specific have static architectures. Examples include NVIDIA's Tegra [28], Qualcomm's Snapdragon [56] and Samsung's Exynos [29] platforms. We discuss different types of dynamic techniques on static architectures in detail.

### 2.2.1 Homogeneous Multi-cores

There exists plenty of prior works on dynamic power management on homogeneous multi-core systems. Most of the works focus on power management using any combination of techniques like DVFS, load balancing and task migrations. Few recent works [26, 80, 82, 122] focuses on power management of homogeneous multi-core systems based on the control theory. [82] allocates the chip power budget to each of the power islands, which is in turn distributed to the individual cores by employing DVFS. The authors in [93] proposed a hierarchical feedback-based control system for power management in server farms. Isci et al. [58] evaluate a DVFS based global power management policy with various objectives like prioritization, power balancing and throughput for different combinations of benchmarks. Rangan et al. [94] explore the use of thread migration in power management compared to the traditional DVFS scheme. The authors in [115] proposed a power management technique based on linear programming using DVFS and thread mapping. In [122], the authors present a control theory based power management framework using per-core DVFS capability and dynamic cache resizing. Ma et al. [80] present a scalable power management solution for workloads that contain a mix of multi-threaded and single-threaded applications in homogeneous chip multiprocessor. However, these solutions are designed for homogeneous multi-core systems and require non-trivial modifications to adapt them to heterogeneous multi-cores.

### 2.2.2 Heterogeneous Multi-cores

The potentials of heterogeneous multi-cores in terms of power-performance efficiency have been illustrated in [12, 24, 69, 70, 118]. However, the heterogeneity introduces additional complexity to the dynamic/runtime scheduler [27, 70]. [74] proposed a scheduling algorithm for heterogeneous cores that incorporates the following techniques: a) asymmetric aware load balancing, b) fast-core first scheduling and c) NUMA-aware migrations. Similarly, the authors in [100] proposed an asymmetric-aware scheduler, where ILP intensive and TLP intensive threads are scheduled in fast and small cores, respectively. In both the works, the heterogeneous cores are simply symmetric cores using different frequency levels without any micro-architectural differences. [68] identified the key metrics such as external and internal stalls, for mapping a task to the appropriate core type to improve performance. The heterogeneity is achieved by limiting the instruction retirement bandwidth. Operating system support for heterogeneous architecture with non-identical but overlapping ISA was proposed in [75]. Craeynest et al. [118] propose a scheduling technique for asymmetric multi-cores using online performance estimation across different core types. Similarly, Koufaty et al. [69] propose a dynamic heterogeneous aware scheduler, which schedules tasks with very low memory stalls on complex cores for higher performance. However, none of these techniques consider power management as an optimization criteria.

A study by Winter et al. [123] evaluates various scheduling and power management techniques for heterogeneous multi-cores with special considerations to the scalability of the approaches. They propose a thread scheduling algorithm called *Steepest Drop*, which has a light overhead and completely ignores the DVFS technique. The technique Pack & Cap proposed in [26] uses thread packing and DVFS to maximize performance under a TDP constraint. Schranzhofer et al. [101] introduce a static solution for task to core mapping problem in heterogeneous MPSoC. [27] developed energy-aware scheduling for a single task on

Intel QuickIA heterogeneous platform with two cores. Our work dynamically incorporates all the three techniques (load balancing, task migration and DVFS) in both *HPM* and *PPM* frameworks to meet performance demands at minimum energy consumption under a power budget.

### 2.2.3 Computational Economics

One of the dynamic power management technique (*PPM*) proposed in this thesis is based on price theory, which borrows lots of inspiration from computational economics. Few existing works [9, 22, 34–36, 50, 79, 98] borrow economic theory ideas to develop power or thermal management schemes. Ebi et al. [34] propose an agent-based power distribution scheme for multi-cores, where the trading commodity is the power units. Agent based dynamic thermal management techniques are proposed in [9, 47], where negotiations are made in the market to make efficient task migration decisions. Roy et al. [98] propose an energy management technique for mobile devices based on abstractions such as isolation, delegation and subdivision. This technique requires building an offline energy model for a system, which consists of a multi-core that uses two different ISA (ARM11 and ARM9).

Some prior works [22, 50, 79] employ welfare economics in datacenters to improve power efficiency. [50, 79] employ Mixed Integer Linear Programming (MILP) technique for determining the optimal allocation of resources. Lubin et al. [79] present power management in homogeneous multi-core datacenters. This approach is extended to heterogeneous systems in [50]. The solving time is quite high ($800ms$) for MILP formulation. This is only suitable for datacenter workloads exhibiting relatively stable phases so that allocation decisions can be made at long intervals (e.g., 10-minute interval). But such high overhead cannot be tolerated in a mobile platform with dynamic workloads where the allocation decisions need to be revised multiple times per second.

### 2.2.4 Power-Performance Model

We also propose a power-performance estimation model for heterogeneous multi-core. Considerable number of prior works [19, 44, 66] have developed analytical performance models for processors. The two predominant approaches employed in building performance models are mechanistic modeling and empirical modeling. Mechanistic models are purely based on the insights of the target processor architecture. In [64, 66], the authors developed a simple interval based mechanistic model for out-of-order cores that assumes a sustained background performance level, which is punctuated by transient miss-events. The models from [64, 66] was further improved in [43] by weighing the dispatch stage in detail. Eyerman et al. [19] propose mechanistic model for superscalar in-order processors. In empirical modeling, the performance model is considered as a black box and typically inferred using statistical/regression techniques. Joseph et al. [62] use non-linear regression performance modeling. In [72], the authors employ spline-based regression modeling for performance and power across different micro-architectural configurations. The authors in [44] propose hybrid mechanistic-empirical modeling for commercial processor cores with few simplistic assumptions. However, the model proposed in this thesis uses the combination of compile-time analysis, mechanistic modeling and empirical modeling to construct performance models for both out-of-order and in-order cores with better accuracy on a real platform.

## 2.3 Dynamic technique - Dynamic architecture

Our last contribution is a power/thermal aware dynamic reliability management technique. Traditionally, dynamic thermal management techniques were employed as a convenient proxy to improve the lifetime reliability of the processors [109]. Commonly employed mechanisms that reduce temperature include

DVFS, activity migration [33, 107], fetch gating and clock gating. However, these techniques do not consider the lifetime reliability problem explicitly. Donald et al [33] weighed the efficiency of various combinations of DVFS, clock gating and migration for thermal management.

Several techniques have been proposed for lifetime reliability (also known as hard errors) management. Srinivasan et al. [109] proposed an architectural level analytical model, called Reliability-Aware Micro-Processor (RAMP), for temperature induced lifetime reliability. They explore the effectiveness of optimizing the architectural configurations and the voltage/frequency settings statically to meet the reliability target. Karl et al. [67] proposed the use of a proportional-integral-derivative (PID) controller based DRM technique. The most common technique employed for DRM is DVFS, possibly with a feedback controller. Dynamic wearout centric job scheduling in chip multiprocessor proposed in [45] employs a fine grained reliability management at the module-level of the cores. As these approaches focus only on the lifetime reliability, the peak temperature constraint is not considered.

We show that dynamically adapting architectural configurations along with DVFS can provide better performance and meet both reliability and/or thermal constraints. Also, while previous works are mostly reactive in nature, i.e., the performance is throttled only when reliability constraint is violated, we propose a predictive DRM technique. Also, we extend our technique to accommodate both reliability and thermal/power constraints.

# Chapter 3

# Power-Performance Modeling on Heterogeneous Multi-cores

A predictive technique that can estimate power-performance across different core types in heterogeneous multi-cores can solve the challenge of scheduling the workload to the appropriate core types. In this chapter, we propose a power-performance estimation model for heterogeneous multi-cores that can be efficiently employed in any prediction based scheduling power management technique. We also develop an online predictive scheduling algorithm which leverages the benefits of the developed power-performance model.

Earlier proposals [13, 69, 74] employed a simple strategy of scheduling memory-intensive workloads on the small core and compute-intensive workloads on the big core. Recently [119] has shown that this strategy may lead to sub-optimal mappings and it is imperative to accurately estimate the power-performance characteristics of a workload on different core types. The Performance Impact Estimation (PIE) mechanism proposed in [119] is a dynamic technique that collects profile information while executing the application on any one core type, and estimates the performance on the other core type. This estimation allows

the scheduler to make appropriate adjustments to the application-core mapping at runtime. However, the PIE mechanism [119] has few shortcomings that renders it difficult, if not impossible, to be deployed on real hardware. First, the estimation is based on a number of simplifying assumptions such as the presence of identical cache hierarchy and branch prediction on both core types, which are unrealistic for commercial asymmetric multi-cores. Second, the PIE mechanism requires profile information, such as the inter-instruction dependency distance distribution, that cannot be collected on existing cores and requires specialized hardware support. Third, power estimation is completely missing as [119] focuses on throughput oriented server workload. Finally, and most importantly, the mechanism is evaluated using simulator where one has complete flexibility in choosing the core configurations.

*We develop power-performance model for commercial heterogeneous multi-core: ARM big.LITTLE.* While an application is executing on ARM Cortex-A7 (alternatively ARM Cortex-A15), we collect profile information provided by hardware counters, and estimate power and performance characteristics of the same application on ARM Cortex-A15 (alternatively ARM Cortex-A7). *We evaluate the accuracy of our estimation on real ARM big.LITTLE hardware platform.* We also construct a runtime scheduler that uses the estimation model for meeting the performance goals of an application under minimal energy consumption.

Our modeling and estimation on real hardware are challenging in many ways. First, the big core and the small core are dramatically different, not just in the pipeline organization, but also in terms of memory hierarchy and the branch predictor — a reality that is ignored in all previous works [69, 100, 119]. These differences render the power, performance estimation from one core type to another considerably more difficult. Second, we are constrained by the performance counters available on the cores and their idiosyncrasies; for example, while the big core provides the *L2 cache write access* counter, it is unavailable on the

small core. More importantly, in contrast to simulation based modeling work, we cannot rely on additional profiling information, such as inter-instruction dependency [119], that can only be collected by introducing extra hardware.

We overcome the challenges outlined above using a combination of static (compile time) program analysis, mechanistic modeling [64, 66], which builds analytical model from an understanding of the underlying architecture, and empirical modeling [62, 72], which employs statistical inferencing techniques like regression to create an analytical model.

Our performance model for any core centers around the CPI (cycles per instruction) stack that quantifies the impact of different architectural events (such as data dependency, cache miss, branch misprediction etc.) on the execution time. While we can obtain information about certain events (e.g., cache miss, branch misprediction) from the hardware counters, other information such as data dependency are not readily available. We rely on compile time static program analysis technique to capture the data dependency information and its impact on pipeline stalls.

Once we develop the CPI stack based performance model for each core, we proceed to estimate the CPI stack of the second core given the CPI stack of the first core. We employ regression modeling to estimate the architectural events (cache miss, branch misprediction) on the second core given information about the architectural events on the first core. These estimates of architectural events can be plugged into the CPI stack model of the second core to derive the CPI value and hence the performance estimate. Finally, our power model uses the CPI value along with additional information, such as instruction mix, memory behavior etc., to estimate the power behavior of the core.

Our concrete contributions in this chapter are the following.

- We propose a combination of static program analysis, analytical modeling, and statistical techniques to model the performance of individual cores and estimate power, performance across different cores on single-ISA heterogeneous multi-core platforms.

- Ours is the first work towards performance estimation across asymmetric cores on real hardware. Estimation on real hardware is challenging compared to simulation based studies [119] due to distinctly different configurations of the cores, memory hierarchy, and unavailability of some of the required hardware counters.

- Ours is the first work to model CPI stack on real out-of-order and in-order cores. [44] is the only existing work that models CPI stack for commercial out-of-order processors; but does not consider in-order processors. We demonstrate that our CPI stack model is more accurate as we combine the strengths of static program analysis and runtime analytical modeling.

- Ours is the only work to derive power estimation on the second core solely based on the execution profile on the first core. Existing works [27] require execution of the application on both cores to estimate power, an assumption that is unrealistic when migration cost from one core type to another is relatively high, as is the case in our setting.

- We implement a runtime predictive scheduler that integrates the power-performance estimation model for single-ISA heterogeneous multi-core platform. The online scheduler achieves the performance goals with minimal energy consumption for an application.

## 3.1 ARM big.LITTLE architecture

We first describe the micro-architectural features of the ARM big.LITTLE heterogenous multi-core that we model for power, performance estimation. The

| **Parameter** | **Cortex-A7** | **Cortex-A15** |
|---|---|---|
| Pipe-line | In-order | Out-Of-Order |
| Issue Width | 2 | 3 |
| Fetch Width | 2 | 3 |
| Pipeline Stages | 8-10 | 15-24 |
| Branch Predictor | 512-entry BTB 2-way | 2K-entry BTB 2-way |
| L1 I-cache | 32KB/2-way/32B | 32KB/2-way/64B |
| L1 D-cache | 32KB/4-way/64B | 32KB/2-way/64B |
| L2 Unified-cache | 512KB/8-way/64B | 1MB/16-way/64B |
| Frequency Levels | 8 | 8 |
| Frequency Range(MHz) | 350-1000 | 500-1200 |
| Voltage Range(mV) | 900 - 1050 | 900 - 1050 |

TABLE 3.1: Architectural Parameters of Cortex-A7 and Cortex-A15

single-ISA heterogeneous architecture consists of high performance Cortex-A15 cluster and power efficient Cortex-A7 cluster, as shown in Figure 1.3. The evaluation platform we use in this work contains a prototype chip with two Cortex-A15 cores and three Cortex-A7 cores at 45nm technology. All the cores implement ARM v7A ISA. The Cortex-A15 is complex out-of-order superscalar core that can execute high intensity workloads, while Cortex-A7 is a power efficient in-order core meant for low intensity workloads. While each core has private L1 instruction and data caches, the L2 cache is shared across all the cores within a cluster. The L2 caches across clusters are kept seamlessly coherent via the CCI-400 cache coherent interconnect.

Table 3.1 summarizes the micro-architectural parameters of Cortex-A15 and Cortex-A7, obtained from publicly released data. It should be evident that the cores are genuinely asymmetric in nature. The 2-way issue in-order pipeline of A7 containing 8-10 stages is dramatically different from the 3-way issue out-of-order pipeline of A15 containing 15-24 pipeline stages. Moreover, even the cache configurations and branch predictors are distinctly different in A15 compared to A7. Most previous works [69, 100, 119] assume that the memory parameters are identical across different core types.

The architecture provides DVFS feature per cluster. The A7 cluster provides

eight discrete frequency levels between 350MHz – 1GHz, while the A15 cluster also provides eight discrete frequency levels between 500MHz – 1.2 GHz. Note that all the cores within a cluster should run at the same frequency level. Moreover an idle cluster can be powered down if necessary. As our focus is on power, performance estimation across core types, we conduct the experiments by setting the same voltage (1.05 Volt) and frequency (1 GHz) for the two clusters. Estimating power, performance for different frequency levels is left as future work. We also consider execution of a sequential application on either A7 or A15, that is, we only use one core at a time and the idle cluster is powered down.

The heterogeneous cores exhibit different power and performance characteristics across workloads. Figure 3.1 shows the performance speedup, energy consumption ratio, and EDP (Energy-Delay product) ratio for 15 selected benchmarks on A15 in comparison to A7. Clearly, A15 has significant performance improvement compared to A7 (average speedup of 1.86); more importantly, the speedup varies significantly across benchmarks from 1.45 to 2.30. In terms of power, it is expected that A7 has lower average power compared to A15 for all the benchmarks. While average power on A7 is 1.44Watt, the average power on A15 varies from 4.20Watt to 5.15Watt. Even though A7 has worse performance, it can completely make up for it in terms of power to achieve far superior energy efficiency compared to A15 (1.78 times lower energy on average). A7 is also more energy efficient for all the benchmarks.

But in embedded systems, especially in interactive systems such as smartphones, we are more interested in the combination of energy and delay to decide on workload-to-core mapping because both battery life and response time are equally important. This metric is captured as Energy-Delay product (EDP). Interestingly, in terms of EDP, there is no clear winner: A15 is more efficient than A7 for 8 benchmarks due to faster execution that overcomes the power inefficiency,

while A7 is superior for the remaining 7 benchmarks due to lower power consumption. Thus, the scheduler needs both power and performance behavior on a core type to decide on the appropriate mapping.

As observed in [119] and validated in our experiments, it is impossible to predict the power, performance characteristics of an application on different core types based on simple metrics such as memory access intensity. We also observe that the average migration cost across clusters is quite high: 2.10ms to move a task from A7 to A15, and 3.75ms to move from A15 to A7. This renders it unrealistic to first execute a workload on each cluster separately and then make the workload-core mapping decision as proposed in [27]. Thus it is essential to accurately estimate the CPI for performance and use the CPI to estimate power. We do so through power, performance modeling in the next section.

## 3.2 Performance Modeling

The aim of performance modeling is to estimate the performance of an application on a second core type (small/big) given its execution profile on the first core (big/small) type. Our model centers around CPI stacks. The basic observation

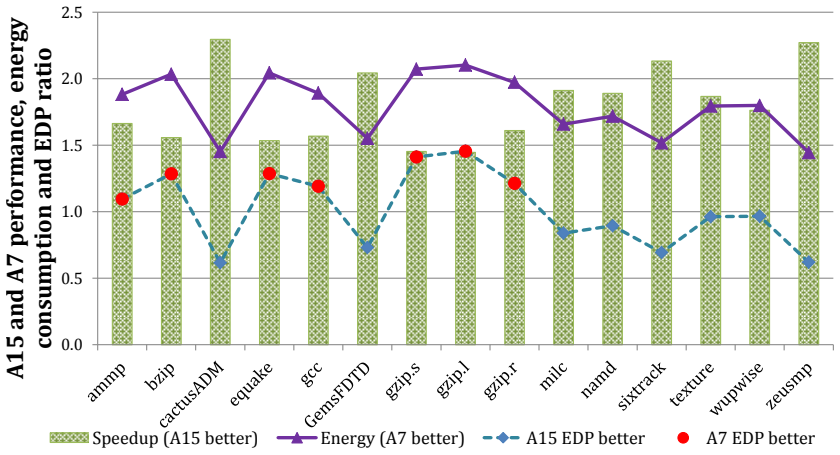



FIGURE 3.1: Performance improvement, energy consumption ratio and EDP ratio of A15 in comparison to A7.

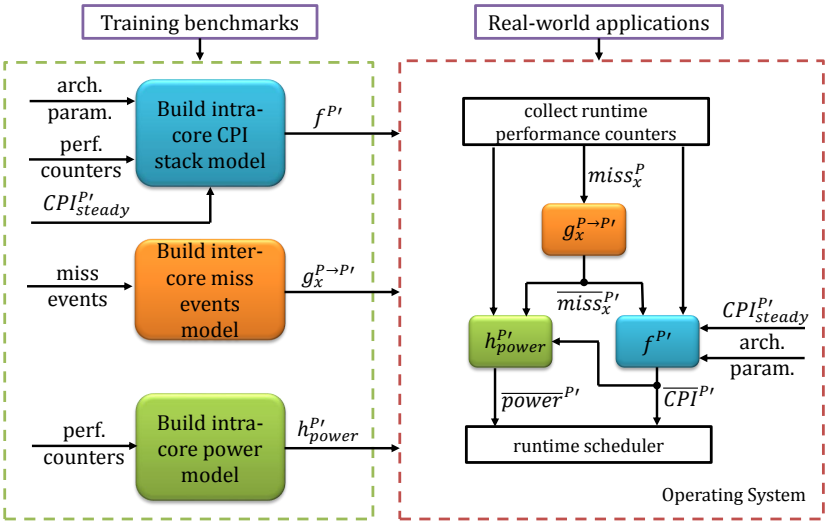



FIGURE 3.2: Inter-core performance, power estimation from $P$ to $P'$.

behind the model is that the CPI follows a sustained background level performance $CPI_{steady}$ punctuated by miss events that show up as temporary peaks. $CPI_{steady}$ captures the cycles spent in the architectural events tightly coupled to the pipeline such as data dependency among instructions and structural hazards, while $CPI_{misses}$ represents the cycles spent due to the external events such as cache miss and branch mispredicton.

$$CPI = CPI_{steady} + CPI_{miss} \tag{3.1}$$

The performance estimation framework shown in Figure 3.2 comprises of three major steps. The first step is an off-line procedure where we build *intra-core CPI stack model* for each core type.

While $CPI_{miss}$ can be expressed in terms of miss events and their latencies, computing $CPI_{steady}$ requires presence of elaborate hardware mechanisms [119] that can collect inter-instruction dependencies and are not available in existing processors. We avoid additional hardware mechanism by observing that $CPI_{steady}$ is an intrinsic characteristics of a program on a core type and is stable across different program inputs, whereas $CPI_{miss}$ is highly dependent on the program

inputs. For example, Figure 3.3 shows the estimated $CPI_{steady}$ and $CPI_{miss}$ values of *bzip* benchmark for different program inputs on A7 and A15. Note that expectedly $CPI_{steady}$ is higher on A7 than A15 because A15 with out-of-order execution engine can better exploit instruction-level parallelism in the presence of data dependencies and structural hazards. The estimated CPI is the summation of the estimated $CPI_{steady}$ and $CPI_{miss}$. For reference, we have also plotted the measured CPI. Our assumption that $CPI_{steady}$ of an application on a core type is stable across different program inputs is validated here as the variation in CPI has been captured accurately only through variation in $CPI_{miss}$.

We exploit this observation to estimate $CPI_{steady}$ of a program on both core types at compile time (see Section 3.2.1) and encode this information with the binary executable. In other words, we estimate both $CPI^{big}_{steady}$ and $CPI^{small}_{steady}$ for a program at compile time. Most modern compilers have an optimization pass that takes care of instruction scheduling based on the hardware description of the processor pipeline. We modify the compiler in the instruction scheduling phase to estimate the $CPI_{steady}$. For applications with distinct phases, i.e., multiple computation kernels with different behavior, we estimate separate $CPI_{steady}$ value for each phase.

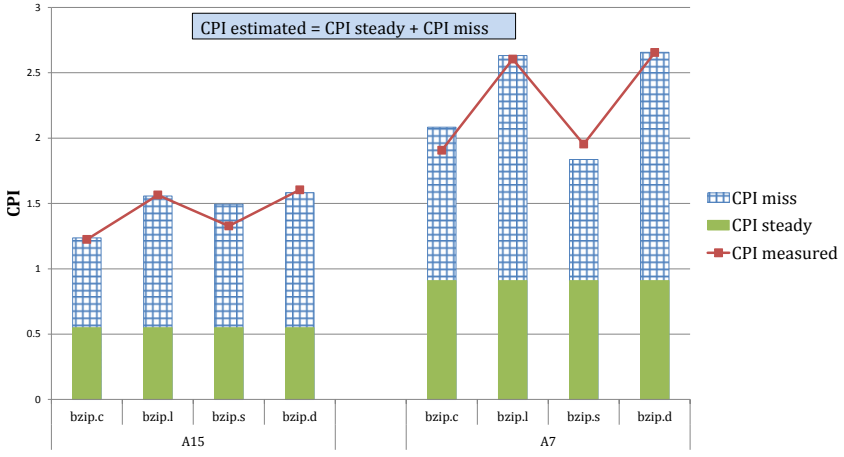



FIGURE 3.3: Estimated $CPI_{steady}$ and $CPI_{miss}$ of different inputs for the same benchmark on A7 and A15.

To build the CPI stack on a core $P$, we collect the execution profiles of a set of training benchmarks through the hardware counters. We then combine analytical modeling with linear and non-linear regressions to derive the CPI stack model that accurately captures the contributions of the different events to performance. The CPI stack model can thus be expressed as the function $f^P$ where

$$CPI^P = f^P(CPI^P_{steady}, miss^P_X, latency^P_X) \tag{3.2}$$

where $miss^P_X$ and $latency^P_X$ are the number of occurrences and latency of each occurrence of the miss event $X$ on processor $P$.

The second step is another offline procedure where we develop regression models that estimate the occurrence of different miss events on processor $P'$ given the frequency of the miss events on processor $P$. These *inter-core miss event estimation models* are built by collecting and correlating corresponding miss events on both cores using a set of training benchmarks. The inter-core estimation model from $P$ to $P'$ for an event $X$ can be expressed by a function $g^{P\rightarrow P'}_X$ where

$$\overline{miss}^{P'}_X = g^{P\rightarrow P'}_X\left(miss^P_X\right) \tag{3.3}$$

where $\overline{miss}^{P'}_X$ is the predicted occurrence of miss event $X$ on $P'$[1].

At runtime, when a new application is running on core $P$, the operating system collects the counter values at regular intervals to get information about the miss events on $P$. For each miss event $X$, it uses inter-core miss event estimation model to predict $\overline{miss}^{P'}_X$ on core type $P'$. Finally, it plugs in the estimated miss event counter values in the CPI stack model of $P'$ to predict $\overline{CPI}^{P'}$, which is inter-core estimated performance.

$$\overline{CPI}^{P'} = f^{P'}(CPI^{P'}_{steady}, \overline{miss}^{P'}_X, latency^{P'}_X) \tag{3.4}$$

[1]We use $\overline{M}$ to indicate the estimated value of a metric across cores.

where $CPI^{P'}_{steady}$ is estimated using analytical models, while $\overline{miss}^{P'}_{X}$ is estimated using linear and non-linear regression models. It has been well established in literature [44] that analytical models are far more accurate than regression models. However, building analytical models for miss events are extremely difficult due to the presence of heterogeneous memory hierarchy and branch predictor.

### 3.2.1 $CPI_{steady}$ estimation

Computing the $CPI_{steady}$ value of a program on real hardware is challenging due to limited information that is exposed through the performance counters. While [119] proposes hardware counters that can count dynamic data dependencies and structural hazards for this purpose, the overhead of such counters is quite high due to the increased amount of book-keeping. An alternative is to simply assume $CPI^{P}_{steady} = 1/D$ where $D$ is the dispatch width of processor $P$ [44]. This assumption only holds true for perfectly balanced pipelines where the number of functional units for each type of operation is equal to the dispatch width and hence there is no structural hazards. It is not realistic as commercial processors do have unbalanced number of functional units. More importantly, the assumption completely ignores the dependency of $CPI^{P}_{steady}$ on the characteristics of the program, in particular, inter-instruction data dependencies. We sidestep this problem by computing $CPI^{P}_{steady}$ of a program on core $P$ at compile time.

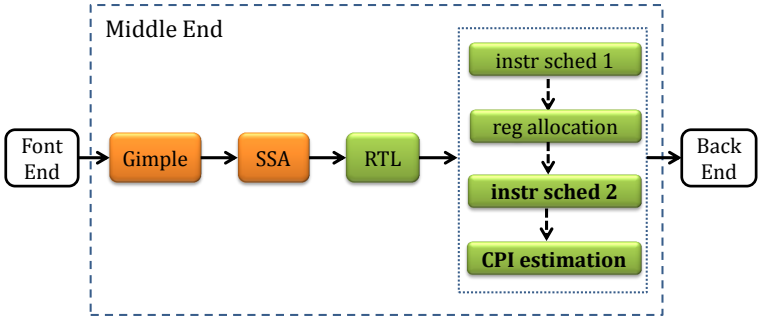



FIGURE 3.4: Estimation of steady state CPI of a program using gcc.

We use *gcc* compiler, where instruction scheduling optimization pass is performed twice, before and after register allocation pass (see Figure 3.4). When scheduling instructions, the algorithm uses a detailed description of the target processor pipeline. At this stage, the compiler is aware of the data dependencies among instructions and the structural hazards due to the limited number of the functional units in the processor pipeline.

We include our $CPI_{steady}$ estimation pass after the second instruction scheduling pass. For each basic block $B$ of the application, we extract the estimated number of cycles $cycle_B^P$, number of instructions $instr_B$, and the estimated frequency $freq_B$. Traditionally, the frequency values are obtained by profiling the application across different inputs but when the profile information is not available, the compiler can predict the behavior of each branch in the program using a set of heuristics and can compute estimated frequencies of each basic block by propagating the probabilities over the control graph. This estimate is used in our equation and it captures rather an average behavior of the application regardless of the input.

We define $CPI_{steady}^P$ of an application $\mathcal{A}$ on core $P$ as

$$CPI_{steady}^P\Big|_{\mathcal{A}} = \frac{\sum_B freq_B \cdot cycles_B^P}{\sum_B freq_B \cdot instr_B}\Big|_{\mathcal{A}} \tag{3.5}$$

Note that only $cycles_B^P$ depends on the core type and leads to different steady state CPI values for different core types. The $CPI_{steady}$ values thus computed for the small and big core are embedded into the application binary.

### 3.2.2 CPI stack model of big core

We extensively employ linear and non-linear regression models in our performance and power estimation framework. Our CPI stack model for the big core

extends and adapts the mechanistic-empirical model proposed in [44] to Cortex-A15 core. Our model estimates the total number of clock cycles $C$ required to execute an application on the big core as:

$$\begin{aligned} C^{big} &= \beta_0 \cdot N \cdot CPI_{steady} + miss_{L1I} \cdot c_{L2} \\ &+ miss_{br} \cdot (c_{br} + c_{fe}) + dmiss_{L2} \cdot \frac{c_{mem}}{MLP} \end{aligned} \tag{3.6}$$

Once the total number of cycles are estimated, the CPI value can be easily computed by dividing the cycles by the total number of instructions $N$.

$$CPI^{big} = \frac{C^{big}}{N} \tag{3.7}$$

This parameterized model sums the number of cycles consumed due to internal and external events. The first term, $CPI_{steady}$ is converted into the corresponding number of cycles by multiplying it with the total number of instructions $N$. The $\beta_i$ parameters are unknown and are fitted through non-linear regression.

The next term represents the miss event cycles due to the instruction misses in first level of cache. The penalty paid for an instruction miss in L1 cache is $c_{L2}$ and represents the number of cycles spent to access L2 cache and is micro-architecture dependant.

The next term of the equation quantifies the cycles spent during the branch misprediction events. The branch misprediction penalty is a function of the front-end length of the pipeline $c_{fe}$ and the back-end of the processor where the branch is resolved in a branch resolution time $c_{br}$. The branch resolution time represents the number of cycles spent between the arrival time of the mispredicted branch in the dispatch queue and the moment when the branch is actually resolved in the execution unit. The branch resolution time is dependent on inter-instruction dependency, long-latency instructions and L1 data misses.

The next term of the Equation 3.6 represents the cycles spent due to the misses in the last level of data cache. The big core is an out-of-order core which takes advantage of the memory level parallelism such that part of an L2 cache miss latency overlaps with other independent L2 cache misses. Thus, we reduce the overall penalty by a factor $MLP$ which is described as follows:

$$MLP = \beta_1 \cdot \left( \frac{dmiss_{L2}}{N} \right)^{\beta_2}$$

This equation assumes that the L2 data misses are uniformly distributed and the amount of parallelism that can be extracted has a power law relation with the window of misses per instruction from which the parallelism is to be extracted.

We recommend the reader to consult the work in [19] for more details about the intuition behind the presented equations.

### 3.2.3 CPI stack model of small core

Modeling the CPI stack for the small in-order core is simpler. We start from Equation 3.6 and remove the terms that are specific to out-of-order processors. The total number of cycles for the small core can be modeled using linear regression as follows:

$$\begin{aligned} C^{small} = \beta_3 + \beta_4 \cdot N \cdot CPI_{steady} + miss_{L1I} \cdot c_{L2} \\ + miss_{br} \cdot c_{fe} + dmiss_{L2} \cdot c_{mem} \end{aligned} \tag{3.8}$$

$$CPI^{small} = \frac{C^{small}}{N} \tag{3.9}$$

| **Parameter** | **Var name** | **Cortex-A15** | **Cortex-A7** |
|---|---|---|---|
| Pipeline front-end | $c_{fe}$ | 4 | 13 |
| L2$ access | $c_{L2}$ | 19 | 13 |
| Main memory access | $c_{mem}$ | 140 | 100 |

TABLE 3.2: Estimated latency in cycles for miss events on A15 and A7

In case of in-order processor, the branch resolution time in the back-end pipeline is not relevant because there is no reorder buffer structure present in an in-order processor. Once there is branch misprediction, the entire pipeline has to be drained. Similarly, the MLP correction factor is not used as cache missses cannot overlap.

| **Parameter** | **Var name** | **Cortex-A15** | **Cortex-A7** |
|---|---|---|---|
| Cycles | $C$ | ✓ | ✓ |
| Instructions | $N$ | ✓ | ✓ |
| Branch instr | $N_{br}$ | ✓ | ✓ |
| Branch misses | $miss_{br}$ | ✓ | ✓ |
| Load instr | $N_{ld}$ | ✓ | ✓ |
| Store instr | $N_{st}$ | ✓ | ✓ |
| Integer instr | $N_{int}$ | ✓ | ✓ |
| Float instr | $N_{fp}$ | ✓ | ✓ |
| L1I$ access | $access_{L1I}$ | ✓ | ✓ |
| L1D$ access | $access_{L1D}$ | ✓ | ✓ |
| L1I$ misses | $miss_{L1I}$ | ✓ | ✓ |
| L1D$ misses | $miss_{L1D}$ | ✓ | ✓ |
| L2$ data miss | $dmiss_{L2}$ | ✓ | ✓ |
| L2$ write access | $dwaccess_{L2}$ | ✓ | |
| L2$ write back | $WB_{L2}$ | ✓ | ✓ |
| Power sensor | $Power$ | ✓ | ✓ |
| Energy sensor | $Energy$ | ✓ | ✓ |

TABLE 3.3: Hardware Performance Counters on A15 and A7

### 3.2.4 Latency of miss events and performance counters

The performance models for both big and small core use a number of hardware performance counters and the latencies corresponding to each individual miss event. Table 3.3 enumerates all the performance counters that are used in our work and their availability on A7 and A15 cores.

While information about the pipeline structure and memory hierarchy configurations of A7 and A15 are available from publicly released data as well as processor internal registers, the cache miss and memory access latencies are not released. To estimate the access latencies to L2 cache and main memory we use the *lmbench* [81] micro-benchmark. Table 3.2 summarises the penalties in cycles for the different miss events used in our models for A7 and A15.

### 3.2.5 Contribution of CPI stack components

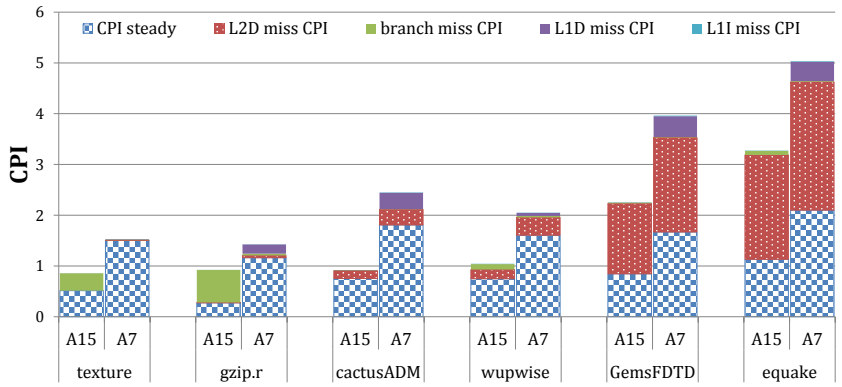



FIGURE 3.5: Estimated CPI stack components on A7 and A15 for a subset of benchmarks.

The miss events used in both the models are branch misprediction, L1 and L2 cache miss. We chose only these events as they contribute most to the overall CPI of a processor. In order to support our claim we conducted several experiments on a set of benchmarks that expose different computational behaviour. Figure 3.5 plots the estimated CPI stack on both small and big cores. We chose two compute intensive benchmarks (*texture* and *gzip.r*), two average compute intensive benchmarks (*cactusADM* and *wupwise*), and two memory bound benchmarks (*GemsFDTD* and *equake*). The benchmarks were selected from Vision [120], SPEC2000 and SPEC2006 [3] benchmark suites. In case of the memory bound applications, the impact of misses in L1 and L2 caches on the overall CPI is considerably higher compared to the compute intensive applications for which the $CPI_{steady}$ and branch mispredictions are impacting the CPI mostly. Note that

branch misprediction impacts A15 substantially more than A7, because A15 has an aggressive back-end pipeline that suffers more from squashing of instructions. We also observe that the speedup on A15 compared to A7 is dependent on a lot of factors, such as $CPI_{steady}$, branch misprediction cost, and cache miss cost.

## 3.3 Inter-core miss estimation

The real challenge in inter-core performance and power estimation on asymmetric cores is that the memory hierarchy and the branch predictors may not be identical across different core types, as is the case in ARM big.LITTLE (see Table 3.1). The small cores are connected to a simpler cache system in order to increase the power efficiency, while the big cores are connected to a more complex memory that supports higher memory throughput, which increases the overall performance. Recent related works [119] assumed that the asymmetric systems have identical memory hierarchy. *The innovation in our approach is that we develop mechanistic-empirical models that can predict the occurrences of miss events* $miss_X^{P'}$ *on processor* $P'$*, given their occurrences* $miss_X^P$ *on* $P$ *obtained through hardware performance counters.*

In order to predict the CPI value of core $P'$ while running on core $P$, we need to predict the values of the performance counters used in Equation 3.6 and Equation 3.8 depending on whether we are predicting the CPI of big core or small core, respectively. These counters are: number of first level data and instruction cache miss ($miss_{L1D}, miss_{L1I}$), number of last level cache miss ($miss_{L2}$) and the number of branch mispredictions ($miss_{br}$).

**Inter-core branch misprediction estimation.** The big core A15 has significantly more aggressive branch predictor compared to A7 to ensure sustained supply of instructions to the high-throughput back end. We observe that the

branch misprediction rate on $P'$ (big or small) is correlated to three metrics on $P$: the branch misprediction rate, the CPI, and the number of branches per instruction. The last metric signifies the rate of branch prediction — the higher the rate, the more is the benefit from a complex predictor. Similarly, the higher the instructions per cycle (or lower the CPI), the more is the need for aggressive branch predictor. Thus we define the inter-core branch misprediction estimation model as follows:

$$\overline{miss_{br}^{P'}} = \beta_5 + \beta_6 \cdot miss_{br} + \left(\frac{1}{CPI}\right)^{\beta_7} + \beta_8 \cdot \left(\frac{N_{br}}{N}\right)^{\beta_9} \Bigg|_P$$

**Inter-core L1 instruction cache miss estimation.** The L1 instruction caches on both cores have the same size and associativity. But the line size on A15 is 64 bytes, while the line size on A7 is 32 bytes. Thus A15 can exploit more spatial locality leading to reduced cold miss. But A7 has twice the number of sets compared to A15, which may lead to reduce conflict miss in A7. As we do not have information about cold and conflict miss, we attempt to estimate them. We assume that the number of cold misses *cold* on processor $P$ is the code size divided by the line size. To predict cold miss on $P'$, the cold miss obtained from $P$ is scaled by the average size of basic blocks $\frac{N}{N_{br}}$. The rationale is that the larger the basic block size, the more likely the cache benefits from larger line size due to spatial locality. Thus, our inter-core L1 instruction cache miss estimation model is

$$\overline{miss_{L1I}^{P'}} = \beta_{10} + \beta_{11} \cdot cold \cdot \left(\frac{N}{N_{br}}\right)^{\beta_{12}} + \beta_{13} \cdot conflict \Bigg|_P$$

$$cold = \frac{code_{size}}{line_{size}} \Bigg|_P ; \quad conflict = miss_{L1I} - cold \Bigg|_P$$

**Inter-core L1 data cache miss estimation.** The L1 data cache has the the same size but different associativity on A7 and A15. However, across a large range of benchmarks, we observe that there is very little difference between the number of L1 data cache miss on A7 and A15. So we employ a simple linear regression model for inter-core miss prediction of L1 data cache.

$$\overline{miss_{L1D}^{P'}} = \beta_{14} + \beta_{15} \cdot miss_{L1D}\Big|_{P}$$

**Inter-core L2 cache miss estimation.** The L2 is a unified data plus instruction cache on both A7 and A15. Even though both L1I and L1D miss filter down and access the unified L2 cache, the instruction accesses have higher spatial and temporal locality leading to negligible miss rate for instruction accesses in L2. Thus instruction miss in L2 does not influence the CPI stack on either A7 or A15 and can be safely ignored. This is fortunate because both A7 and A15 provide performance counters for only L2 data access miss and not L2 instruction access miss. We denote L2 data access miss as $dmiss_{L2}$ and it has significant influence on CPI stack as shown in Figure 3.5. Thus for accurate inter-core performance and power estimation, it is absolutely essential to predict $dmiss_{L2}$ correctly.

For our architecture, L2 cache is distinctly different in A15 compared to A7. Not only the L2 in A15 has twice the associativity of A7 (16-way versus 8-way); but also the size is doubled in A15 (1MB compared to 512KB). This also implies that the number of sets (1024) in L2 is exactly the same for both A7 and A15 and A15 is likely to have significantly less conflict miss due to higher associativity, whereas cold misses should be similar because the line size is identical.

How do we determine the number of conflict miss for L2 data access? We use the number of write backs to estimate conflict miss in L2. A write back indicates conflict miss because a memory line is being evicted from the cache due to conflict with another memory line. But not all conflict miss are captured via write backs.

If the memory line being replaced in the cache is clean (i.e., contains read data), we cannot observe the conflict in terms of write back. We make the assumption that the rate of conflict miss is the same for both read data and write data. Thus we scale the write back by the fraction of write access to estimate the conflict miss $conflict$.

$$conflict = \left.\frac{WB_{L2}}{(wfrac_{L2})^{\beta_{16}}}\right|_P ; \quad cold = dmiss_{L2} - conflict\Big|_P$$

While predicting L2 data miss from the big core (A15) to the small core (A7), we have measured value for $wfrac$ from performance counters: number of L2 access (which is same as the sum of L1D and L1I miss) and number of L2 data write access ($dwaccess_{L2}$). Thus

$$wfrac_{L2} = \frac{dwaccess_{L2}}{miss_{L1I} + miss_{L2D}}$$

While predicting from the small core to the big core, however, we are challenged by the lack of performance counters for write access. So we estimate L2 write access as the L1D miss scaled by the fraction of store instructions over total memory instructions.

$$wfrac_{L2} = \frac{\frac{N_{st}}{N_{st}+N_{ld}} \cdot miss_{L1D}}{miss_{L1I} + miss_{L2D}}$$

We are now ready to predict $dmiss_{L2}$ across cores. We use linear regression of cold miss and conflict miss on $P$ to predict the total miss on $P'$. We observe that while L2 instruction access miss is negligible, if the number of instruction access in L2 is high compared to total L2 access, there is higher chance of instructions evicting data through conflict in unified cache. Thus we scale the conflict miss by L2 instruction access fraction to obtain more accurate inter-core conflict miss

prediction.

$$\overline{dmiss_{L2}^{P'}} = \beta_{17} \cdot conflict \cdot \frac{miss_{L1I}}{miss_{L1I} + miss_{L1D}} + \beta_{18} \cdot cold \Big|_{P}$$

**Inter-core CPI estimation.** Once we have estimated the miss events on core $P'$, given the miss event information on core $P$, it is straightforward to obtain CPI estimate on $P'$. We simply need to compute $CPI_{miss}^{P'}$ by plugging in the estimated miss event values in the CPI stack of $P'$ as defined by Equation 3.6 or Equation 3.8.

## 3.4 Power Modeling

We now describe our modeling technique to estimate power on asymmetric multi-core. Unlike performance modeling, which required a combination of mechanistic and empirical modeling, power can be modeled purely based on regression analysis. We used a simple linear regression model to estimate the power consumption in terms of available performance counters.

**Modeling power of small core.** In big.LITTLE platform, the small cores are superscalar in-order, power efficient Cortex-A7 processors. We observe that the average power consumption of the small core is quite similar across all the benchmarks. This is because, the benchmarks exhibit similar performance characteristics in small core. The only variability observed in the performance across the benchmarks is the $L2Dmiss$. Our power consumption does not capture $L2Dmiss$ i.e., access to main memory. The min and max power consumption measured across training benchmarks (from Table 3.4) are 1.385 watts and 1.506 watts respectively. Thus, there is no need to model power for the small cores.

**Modeling power of big core.** While power consumption on the small A7 core is stable across benchmarks, the big core (power-hungry, out-of-order A15) shows significant variation in power consumption within (due to phase behavior in programs) and across benchmarks. The observed min and max power consumption on A15 across training benchmarks (from Table 3.4) are 4.535 watts and 5.155 watts respectively. This is because complex out-of-order cores exhibit different access profiles of various micro-architectural components across the benchmarks. Thus, it is imperative to model application-specific power consumption on A15.

The power consumption of A15 depends on the pipeline behavior and the memory behavior of the application. In particular, the instruction mix of an application is expected to influence the access profile of different architectural components such as ALU, floating-point unit, branch predictor etc, which in turn, determines the power consumed in the pipeline. The power consumption in the memory hierarchy is determined by the number of L1I, L1D, L2, and memory access. So we are looking for the function $h^P$ in Figure 3.2 that models the power consumption

$$Power^P = h^P(N_X, miss_X^P, CPI^P)$$

where $N_X$ is fraction of instructions of type $X$ in instruction mix.

Given a set of training benchmarks (described in Section 3.6), we first collect the performance counter values on A15 that captures the instruction mix and the access at different levels of the memory hierarchy. We also measure the power consumption on A15 (power measurement setup will be presented in Section 3.6). Next we employ correlation analysis to identify the important performance counter that are most related to power consumption. The total power consumption of the big core can be expressed in terms of the following linear

regression model:

$$Power = \beta_{19} + \beta_{20} \cdot \frac{N_{int}}{N} + \beta_{21} \cdot \frac{N_{fp}}{N} + \beta_{22} \cdot \frac{1}{CPI} + \beta_{23} \cdot \frac{access_{L1D}}{N} + \beta_{24} \cdot \frac{access_{L2}}{N} + \beta_{25} \cdot \frac{dmiss_{L2}}{N} \tag{3.10}$$

The first three terms capture the power consumption in the pipeline, which is influenced by the proportion of integer instructions ($N_{int}$), the proportion of floating point instructions ($N_{int}$), and the instructions per cycle IPC (the inverse of CPI).

The power consumption is also linearly related to the rate of access to the various levels of the memory hierarchy, which is captured using the next three terms. Notice that we do not include L1 instruction cache access here because it is already included in terms of CPI. The higher the CPI, the lower the rate of access to L1 instruction cache.

**Estimating power of big core from small core.** In the previous section, we described the methodology for estimating the performance of an application on the big core while running it on the small core. In this section, we provide the models to estimate the power consumption. The major challenge in estimating the power consumption of an application on the big core while running it on the small core is that we have to predict the access profile. In Equation 3.10, the instruction mix ($N$, $N_{int}$ and $N_{fp}$) remains unchanged across cores. The inter-core miss event prediction model given in Section 3.3 estimates $\overline{CPI}$, $\overline{dmiss_{L2}}$, $\overline{miss_{L1D}}$, $\overline{miss_{L1I}}$ on the big core from the corresponding values on the small core (see also in Figure 3.2). We can then define

$$\overline{access(L2)} = \overline{miss_{L1D}} + \overline{miss_{L1I}}$$

These estimated values can be plugged into Equation 3.10 to estimate the power consumption on the big core.

## 3.5 Runtime Scheduler

We now present our runtime scheduler that leverages the power-performance estimation model for single-ISA heterogeneous multi-core. Figure 3.6 presents the various components of our online scheduler. The goal of the runtime scheduler is to schedule the application in a big or small core such that the performance goals are met at a minimal energy consumption.

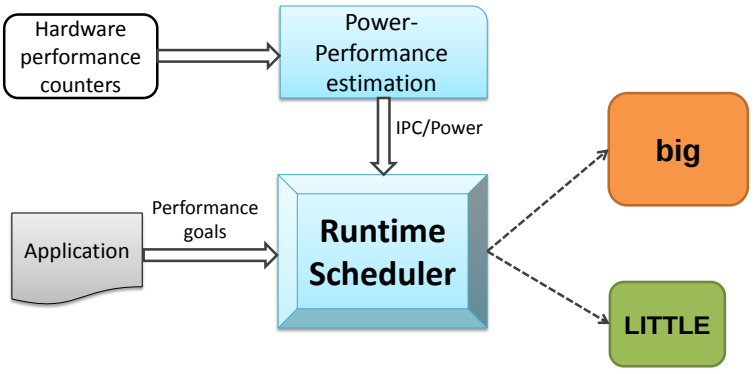



FIGURE 3.6: Online scheduler with power-performance estimation.

The application provides the performance goals in terms of the target throughput rate. We employ Heart Rate Monitor [54] infrastructure to set the performance goal. Heart rate is defined as the throughput of the critical kernel of a task, for example, number of frames per second for a video encoder.

Every 100 ms, the runtime scheduler collects the hardware performance counters and estimates the IPC and power consumption of the other core type. Since the migration across clusters is expensive (2-4 ms), the runtime scheduler is called infrequently. At every scheduling epoch, the runtime scheduler maps the application to the core that meets the performance goal with minimal energy consumption. We explain in detail on how the runtime scheduler uses the power-performance estimation model to achieve the target goal.

### 3.5.1 Performance Estimation

As mentioned before, the application defines the performance goals in terms of heart rate, which is number of heart beats per second. Our performance estimation model computes the CPI of the target core type. We assume a linear correlation between the heart rate and CPI. The conversion between heart rate and CPI is illustrated using the following example. Let us assume an application is running on a small core with CPI of 1.0 and heart rate of 50 heart beats per second (hb/s). For a CPI of 0.5 (in big core), the heart rate in the big core would be 100 hb/s.

### 3.5.2 Energy Estimation

Our power estimation model computes the power consumption of the target core type. From the heart rate and power consumption, one can easily calculate the energy consumption per heart beat.

The scheduler chooses the core that meets the target performance goal in terms of heart rate with minimal energy consumption per heart beat. Currently, our runtime scheduler can handle only one application at any given time. This means that only a big core or a small core is active at any give time. The unused cluster is always turned off.

## 3.6 Experimental Evaluation

We now evaluate our power, performance estimation framework for asymmetric multi-core. We first present the experimental setup, followed by fitting errors of our model on training benchmarks, and finally a validation of our models within and across cores for a new set of test benchmarks. Then, we evaluate the superiority of our runtime scheduler in exploiting the heterogeneity of the cores.

**Experimental setup.** We use the Versatile Express development platform [7] comprising of a motherboard on which the big.LITTLE prototype chip is mounted as part of a daughter board. The motherboard handles the interconnection between the daughter board and the peripherals using an FPGA bus interconnection network. The board boots Ubuntu 13.02 Linaro with the Linux kernel release 3.7.0 for Versatile Express [8]. The platform firmware runs on an ARM controller, which is embedded on the motherboard. The Linux file system is installed on the Secure Digital (SD) card where all our benchmark applications are located.

We collect the hardware performance counter values using ARM Streamline gator kernel module and daemon [1]. We compile and configure Linux kernel to support the gator driver. The gator driver is a dynamic kernel module that interrupts the core at periodic intervals to collect the performance counters. The average CPU utilization of gator daemon is less that 0.5%, which indicates that the overhead of running gator daemon in the background is minimal. We use Matlab [84] to develop our regression models offline.

The prototype big.LITTLE chip consists of one A15 cluster and one A7 cluster at 45nm technology. The individual clusters are equipped with sensors to measure the frequency, voltage, current, power and energy consumption at the cluster level and not at the core level. Moreover, we can only power down a cluster; but not individual cores within a cluster. In our experiments, we utilize only one A15 core and one A7 core. The remaining cores in the clusters are logically turned off using system calls, such that no tasks are scheduled on them. Finally, we set the voltage and frequency for both the clusters at 1.05V and 1GHz, respectively.

The runtime scheduler is implemented within the Linux kernel source. The scheduler is invoked every 100 ms to collect the hardware performance counters using the gator kernel module and daemon [1]. Then, it employs the estimation model to predict the power and performance on a different core type. Finally, it

migrates the application to the appropriate core type using the *sched_setaffinity* interface in the Linux scheduler.

| | |
|---|---|
| Training Benchmarks | *ammp, cactusADM, equake, gcc GemsFDTD, gzip.s, gzip.l, gzip.r, milc namd, sift, sixtrack, texture, wupwise, zeusmp* |
| Test Benchmarks | *apsi, calculix, gamess, gzip.p gzip.g, h264, lbm, leslie3d, mcf, mgrid, mser, omnetpp parser, swim, tonto* |

TABLE 3.4: Training and Test Benchmarks

**Compiler setup.** We implement our $CPI_{steady}$ estimation pass in the GCC Linaro version 4.7.3. The GCC instruction scheduler [2] uses a very efficient pipeline hazard recognizer to estimate the possibility of issuing an instruction on a given core in a given cycle. The processor pipeline descriptions can be expressed in terms of a deterministic finite automaton, which in turn is used to generate pipeline hazard recognizer. The latest version of Linaro GCC compiler includes the processor pipeline descriptions for Cortex-A7 and Cortex-A15 cores. We exploit the hazard recognizer to estimate the data dependencies and structural hazards for a program on A7 and A15, which leads to the steady state CPI estimate as presented in Section 3.2.1.

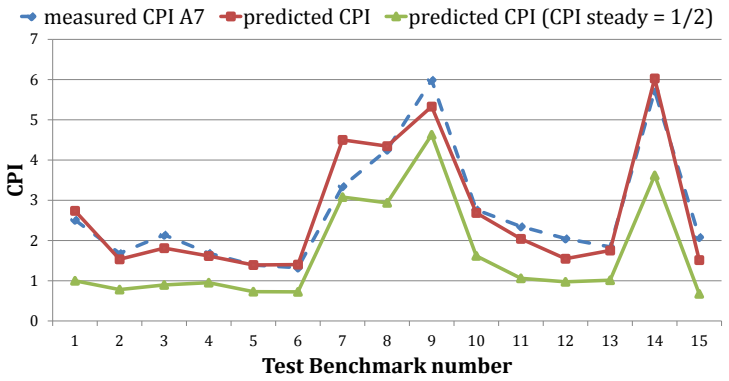
We compile all the benchmarks with -O2 optimization flag. This ensures that the instruction scheduling optimization pass and the $CPI_{steady}$ estimation pass are invoked. We disable both the hardware and the software prefetcher in all our experiments. The Cortex-A15 cores comprise of Level 2 hardware prefetcher, while Cortex-A7 contains Level 1 data cache hardware prefetcher. We disable all the hardware prefetcher because of lack of documentation on the working of the prefetcher. Hence, modeling of the prefetchers were not made possible. All the hardware prefetcher are disabled by writing to the CP15 auxiliary control register. All the benchmarks are compiled with *-fno-prefetch-loop-arrays* flag to disable software prefetching.

**Training and Test benchmarks.** For our experiments, we use Vision [120], SPEC CPU2000 and CPU2006 [3] benchmark suites with reference inputs. Table 3.4 lists the set of benchmarks used in our training and tests set. We categorize all the benchmarks into three types based on the memory behaviour: memory intensive benchmarks, compute intensive benchmarks and intermediate benchmarks. The fraction of L2 miss per instruction is given by $frac_{L2} = (\frac{dmiss_{L2}}{N} \cdot 100)$. For memory intensive benchmarks, we chose the $frac_{L2}$>1.5%, while the compute intensive benchmarks have $frac_{L2}$<0.5% and the remaining are classified as intermediate benchmarks. We randomly select five benchmarks from each category to capture diverse behavior in our training set. The training set is used to develop our regression models for power and performance, while the test set is used to cross-validate the model. As shown in Table 3.4, we keep the test benchmark set consisting of 15 benchmarks completely disjoint from the training set.

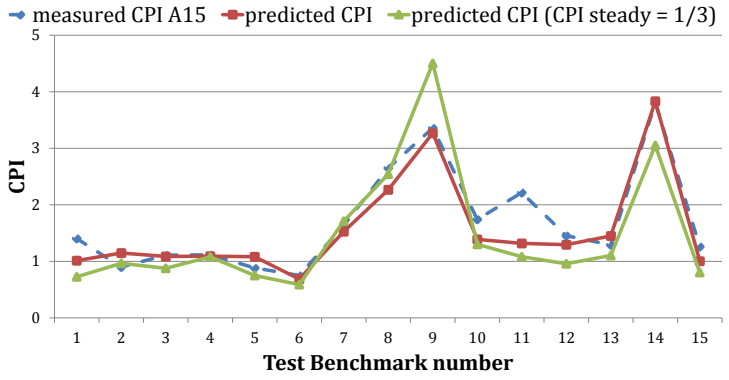
**Benchmarks for runtime scheduler.** For our runtime scheduler evaluation, we use *x264, bodytrack, swaptions* and *bodytrack* from PARSEC [16] benchmark suites with native input. The reason behind choosing the aforementioned benchmarks for evaluating our runtime scheduler is two-folds: a) it is straightforward to implement the heart rate monitor infrastructure [54] and b) the selected benchmarks are not part of the training benchmarks.

### 3.6.1 Performance estimation accuracy

We validate our performance and power estimation models using three sets of experiments. In the first experiment, we compute the fitting error for our regression models on the training benchmarks. It is important to get a good fitting in order to build an accurate model. However, with over-fitting, we run at the risk of large errors for new applications, for which the model was not




(A) CPI prediction on A7




(B) CPI prediction on A15

FIGURE 3.7: Intra-core model validation accuracy using $CPI_{steady}$ obtained through compile-time analysis compared to the accuracy assuming $CPI_{steady} = 1/D$

trained. Thus, our second experiment computes error in intra-core performance and power estimation for the test benchmark set using the model derived from training benchmarks. This shows the robustness of the model, i.e., how well the model behaves for new applications. Finally, we validate the accuracy of our inter-core estimation models on test benchmarks. This challenging task requires both accurate CPI stack models and inter-core miss event estimation models to achieve good accuracy.

**Fitting error in regression for training benchmarks.** In Figure 3.8, the benchmarks are numbered in the same order as it is enumerated in Table 3.4. Figures 3.8a and 3.8d show the measured and estimated CPI for small and big core, respectively, on the training set. The average fitting errors observed are

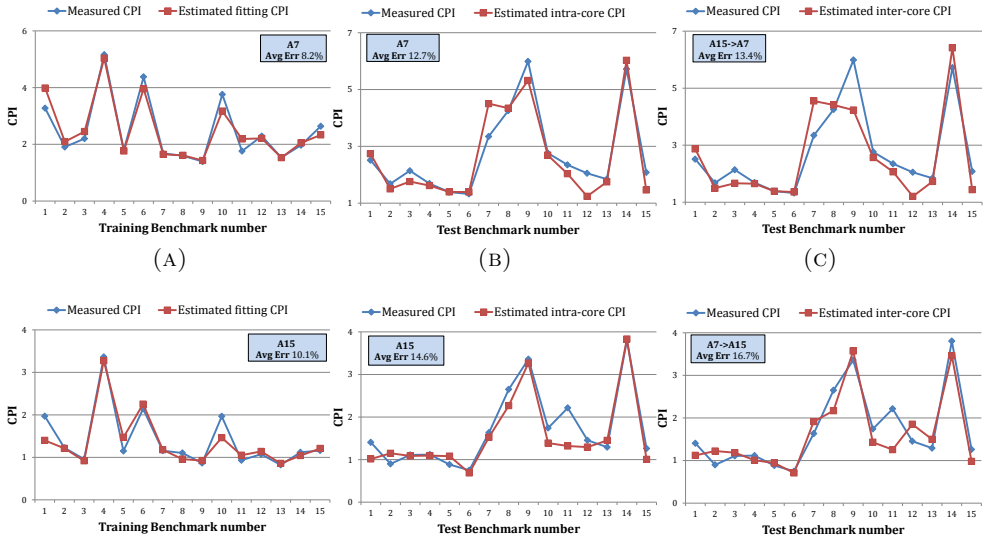



FIGURE 3.8: CPI stack model fitting error on training benchmarks, intra-core model validation error using test benchmarks and inter-core CPI estimation error for Cortex-A7 (top row) and Cortex A-15 (bottom row).

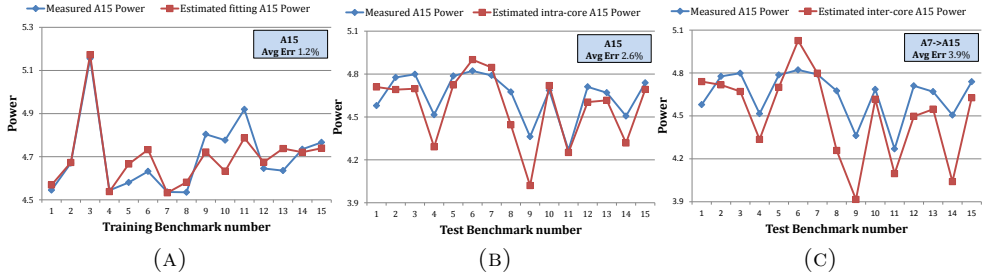



FIGURE 3.9: Power model fitting error on training benchmarks, intra-core model validation error using test benchmarks and inter-core power estimation error for Cortex-A15.

8.2% for small core and 10.1% for big core, respectively. A7 CPI stack model has better accuracy because it is easier to build the CPI stack for in-order cores in comparison to complex out-of-order cores.

Given an application, we obtain $CPI_{steady}$ on big and small core at compile time. This is in contrast to the technique proposed in [44] that assumes $CPI_{steady}$ to be equal to $\frac{1}{D}$, where $D$ is the dispatch width of the core ($D = 2$ for A7 and $D = 3$ for A15). In other words, the model in [44] completely ignores the impact of program characteristics on steady state CPI. Figure 3.7 shows the advantage of compile-time estimation on CPI prediction for both A7 and A15. Our technique

reduces the prediction error by 33.3% on A7 and 8.1% on A15, on an average, in comparison to [44].

**Intra-core validation for test benchmarks.** In order to further evaluate the accuracy and robustness of our intra-core CPI stack model, we compare the measured CPI and the estimated CPI for a completely new set of benchmarks (i.e., test benchmarks) from Table 3.4. From Figures 3.8b amd 3.8e, we observe that the average intra-core prediction errors are 12.7% for small core and 14.6% for big core, respectively. For both small and big cores, 80% of all test benchmarks have prediction error less than 25%. The error increases slightly compared to the fitting error, which is expected given the diverse characteristics of our test benchmarks. The intra-core validation with test benchmarks confirms that our CPI stack model is robust and we have avoided over-fitting.

**Inter-core validation for test benchmarks.** It is challenging to estimate the CPI for one core type, while executing on the other core type. The estimation is further exacerbated by the presence of highly dissimilar cache hierarchy. We perform *Inter-core validations* using the test benchmarks. For this set of experiments, we execute each test benchmark on A7 (alternatively A15), collect the execution profile, and estimate its CPI on A15 (alternatively A7) using our regression models explained in Section 3.2. We then compare the predicted CPI on the target core with the measured CPI to evaluate the accuracy of our estimation.

Figure 3.8c shows the measured CPI and the estimated CPI on small core using the performance counters from big core. The average *Inter-core validation* error in predicting small core CPI from big core is 13.4%; the maximum error is 43.2%. The comparison between the measured CPI and the estimated CPI on big core using the performance counters from small core is shown in Figure 3.8f. We observe average *Inter-core validation* error in predicting big core from small

core is 16.7% and the maximum error is 41.3%. As inter-core estimation depends on both CPI stack model and inter-core miss event estimation models, this experiment validates the accuracy of both the models.

### 3.6.2 Power estimation accuracy

Similar to the evaluation of our performance modeling, we evaluate the power modeling in terms of fitting error, intra core validation and inter core validation. As discussed earlier, we do not need to build a power model for the small core due to insignificant variance in power consumption across the benchmarks. Figure 3.9a shows the measured and estimated fitting power for the training set on big core. Similarly, Figure 3.9b and 3.9c compare the measured and estimated power for intra-core and inter-core validation on test benchmarks. The average prediction error is fairly low even for inter-core validation (3.9%) (y-axis is scaled to capture the small difference between the measured and estimated values). The power estimation across cores rely more on memory access behavior, which we predict fairly accurately leading to high acuracy.

### 3.6.3 Phase behavior

So far we have shown the accuracy of our power and performance estimation models for whole benchmarks. In reality, some benchmarks exhibit phase behavior in their execution. We envision that our estimation framework can be used in such contexts to continuously monitor the execution profile on one core and estimate the power, performance on the other core. This will allow the scheduler to migrate the task back and forth between the cores depending on which phase the program is currently in and the appropriate core type for that phase.

We conduct a case study experiment with *astar* benchmark to evaluate the accuracy and robustness of our model in detecting phase changes and accurately

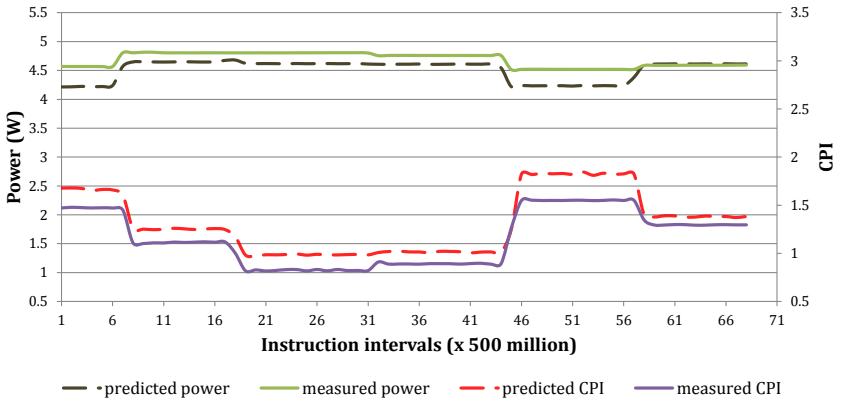



FIGURE 3.10: Contiuous CPI and power estimation from A7 to A15 for *astar* benchmark.

predicting the behavior on the target core for each phase. Figure 3.10 shows the estimated power, performance on A15 predicted from executing the application on A7. For references, we also show the measure power, performance on A15. The X-axis shows the number of committed instructions as time progresses. We set our sampling interval at 500ms, which roughly corresponds to 500 million instructions on A7 at 1GHz. The application demonstrates clear phase behavior. Our estimations are fairly close to the measured values. Thus we can track the phase changes accurately and present performance speedup and energy efficiency on A15 compared to A7 for each phase.

### 3.6.4 Asymmetric vs Symmetric multi-core

Our runtime scheduler exploits the benefits of heterogeneity present in ARM big.LITTLE architecture. We compare the benefits of having our runtime scheduler for asymmetric multi-core to a scheduler present in a symmetric multi-core. We emulate symmetric multi-core by using only big or small core.

Figure 3.11 plots percentage of time heart rate was met on big, small (LITTLE) and big.LITTLE. Similarly, figure 3.12 plots the energy consumption of big, small and big.LITTLE. It is evident from the Figures 3.11 and 3.12 that the runtime scheduler exploits the heterogeneity efficiently by having performance very close

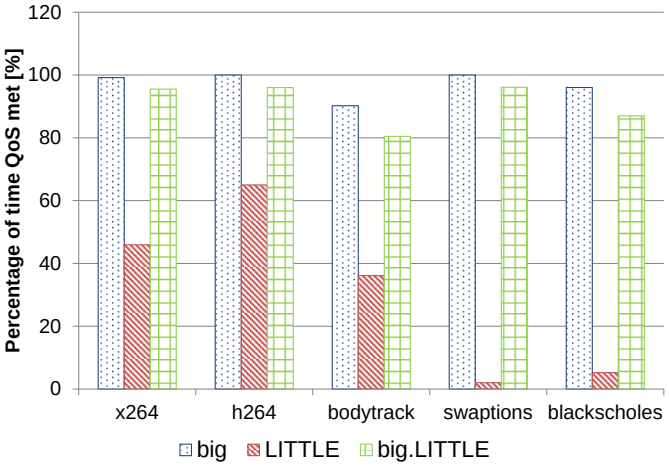



FIGURE 3.11: Comparison of percentage of time heart rate was met between symmetric and asymmetric multi-core.

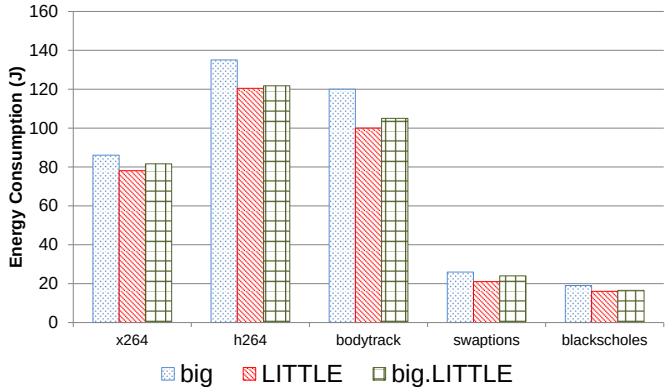



FIGURE 3.12: Comparison of energy consumption between symmetric and asymmetric multi-core

to big core with minimal energy consumption. The runtime scheduler is able to achieve better performance-energy efficiency because of the power-performance estimation model.

## 3.7 Summary

We developed accurate models to estimate the power and performance on asymmetric multi-core architectures. Our aim is to predict, at runtime, the power, performance behavior of an application on a target core, given its execution

profile on the current core, where the cores share the same ISA but has heterogeneous micro-architecture. We overcome the challenges of distinctly different micro-architecture, memory hierarchy, and branch predictor on commercial asymmetric multi-cores through a combination of compile-time analysis, mechanistic modeling, and linear/non-linear regressions. One of the key contribution of our work is an accurate model that estimates the cache miss and branch misprediction rates on the target core, solely from the information available on the current core. We also implemented a runtime scheduler for asymmetric multi-core that employs the estimation model. Unlike almost all previous modeling works, we design and evaluate our estimation framework on a real asymmetric multi-core – ARM big.LITTLE.

# Chapter 4

# Hierarchical Power Management

In the previous chapter, we discussed about power-performance estimation models for asymmetric multi-cores. We also developed a simple runtime scheduler that leverages the estimation model to schedule a single application on ARM big.LITTLE multi-core. In this chapter, we present a comprehensive dynamic power management framework for heterogeneous single-ISA multi-cores architecture in the context of mobile embedded platforms — that can provide satisfactory user experience while minimizing energy consumption within the Thermal Design Power (TDP) constraint. With comparison to homogeneous multi-cores, power management is challenging on heterogeneous multi-cores under limited TDP budget. Unlike the scheduler proposed in chapter 3, the power management technique proposed in this chapter can handle multiple applications. We set out to design our framework with the following objectives.

- The dramatically different power-performance behavior of the cores implies that we need to identify the right core for the right task at runtime and migrate the tasks accordingly.

- The power hungry complex cores should be employed sparingly and only when absolutely necessary.
- Dynamic Voltage and Frequency Scaling (DVFS) as a control knob is available per cluster rather than per core within a cluster necessitating appropriate load balancing strategies. A cluster should run at the minimum frequency level required for adequate user experience so as to conserve energy.
- The restricted TDP budget precludes certain combination of frequencies for the different clusters. For example, it may be necessary to power down A7 cluster when A15 cluster is running at maximum frequency, a canonical example that illustrates the impact of the dark silicon era. Thus power budget has to be allocated opportunistically among the clusters.
- If a system exceeds the power budget, the quality-of-service (QoS) of the tasks should degrade gracefully.
- The framework should be integrated in a commodity operating system without altering any of its desirable properties.

While, there exists solutions in the literature focusing on at least a subset of the objectives mentioned earlier, each of these solution have been generally designed to operate independently. It should be clear that deploying them together requires a carefully coordinated approach that is aware of the complex interplay among the individual solutions. For example, once the system exceeds the TDP of the entire chip, the power budgets for the clusters have to be reduced, which implies scaling down the voltage and frequency levels of the clusters, and consequently degrading the QoS of the tasks that triggered the thermal emergency in the first place. However, once the system load decreases (e.g., some tasks leave the system), this process has to be reversed and the QoS of the tasks should be restored back to the original level. This requires synergistic interaction among

the different solutions so as to ensure *safety* (operate under power budget) and *efficiency* (optimal tradeoff between power and performance), while maintaining *stability*, i.e., avoiding oscillation between different operating points.

We design a *hierarchical power management framework* that is based on the solid foundation of *control theory* and integrates multiple controllers to collectively achieve the goal of optimal energy-performance tradeoff under restricted power budget in asymmetric multi-core architectures. Moreover, we build our framework as an extension of *Linux completely-fair scheduler* while preserving all of its desirable properties such as fairness, non-starvation etc. We take advantage of Heart Rate Monitor (HRM) [54] infrastructure in Linux to set the performance goal for a task and to monitor its execution progress as a measure of QoS. Finally, our Linux-based hierarchical power management framework is implemented on *real ARM big.LITTLE platform* [7] exploiting all the control knobs provided on the platform, namely, per cluster DVFS, cluster power down, and task migration within and across clusters.

To the best of our knowledge, the work presented in this chapter is the first one to provide a comprehensive power management approach for heterogeneous multi-cores under limited power budget and definitely the first one to integrate the solution in a commodity operating system (Linux) running on real platform (ARM big.LITTLE). In this chapter, following are there key contributions:

- Our power management framework successfully achieves all the objective enumerated earlier.
- Our solution builds on a formal control-theoretic approach that provides guarantees for safety, efficiency, and stability.
- Our hierarchical framework carefully coordinates the controllers to avoid inter-controller interference.

- We integrate our power management framework within the confines of Linux completely fair scheduler.
- We implement our Linux based power management solution on a test version of the ARM Big.Little asymmetric multi-core architecture and report power, performance results from this real chip (as opposed to simulation).
- We experimentally evaluate and establish the superiority of our approach compared to the state-of-the-art.

## 4.1 ARM big.LITTLE architecture

We have provided a detailed micro-architectural features of the ARM big.LITTLE in Section 3.1. The power-performance tradeoff in ARM big.LITTLE has been comprehensively discussed in Section 3.1. In this section, we discuss the impact of DVFS on performance and power consumption. Second, we evaluate the impact of the number of active cores on each cluster. Finally, we discuss the migration cost within and across the clusters.

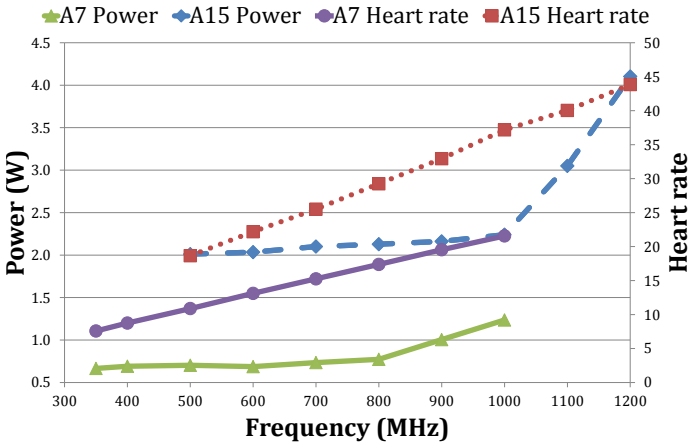
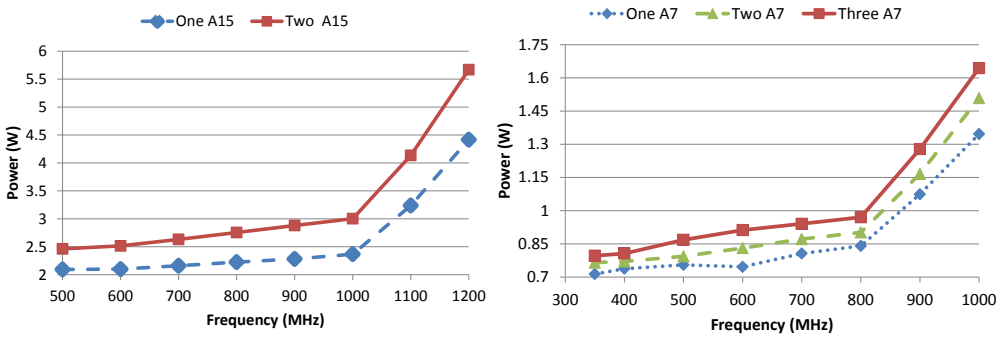



FIGURE 4.1: Power and heart rate with varying frequency.

### 4.1.1 Impact of DVFS

As mentioned earlier, our objective is to provide satisfactory user experience or QoS at minimal energy. We employ Heart Rate Monitor [54] infrastructure to set the performance goal and measure the QoS of a task. Heart rate is defined as the throughput of the critical kernel of a task, for example, number of frames per second for a video encoder. Figure 4.1 plots the heart rate and power for `blackscholes` from PARSEC benchmark suite on A7 and A15 at difference frequency levels. We observe that the heart rate increases linearly with increasing frequency on a core. Also as the IPC of A15 is better than A7, the heart rate can be improved by migrating a task from A7 to A15 at the same frequency level (but at higher power cost). Finally, the power generally increases linearly with increasingly frequency on a core; but there is a sudden jump at 800MHz for A7 and 1GHz for A15 due to change in voltage level. For A7, the voltage level remains constant for frequency levels 350-800MHz. Similarly, for A15 the voltage level is same for for frequency levels 500-1000MHz.

### 4.1.2 Impact of active cores on cluster power




(A) A15 Cluster Power (B) A7 Cluster Power

FIGURE 4.2: Impact of number of active cores on cluster power.

As noted earlier, we can set frequency and measure power only at cluster level. Also we can only power down a cluster, but not individual cores. Thus, even

if a core is idle, it still consumes power. Here we evaluate the impact of active cores on power consumption of the cluster. For this experiment, we run the same benchmark application on one, two, and three cores in A7 cluster as well as one and two cores on A15 cluster at different frequency levels. It is interesting to observe(Figure 4.9) that the A7 and A15 cluster have completely different power behavior with respect to the number of active cores.

In A7 cluster, even at the highest frequency level (1GHz), there is only 0.3 Watt difference between one active core and three active cores. In the A15 cluster, on the other hand, there is roughly 1.5 Watt difference in power between one active core and two active cores. For both clusters, it is important to perform load balancing and run all the cores at the lowest possible frequency level.

### 4.1.3 Migration Cost

Task migration across clusters is important to exploit the unique advantage of asymmetric multi-cores. We perform a set of experiments to quantize the migration cost within and across clusters.

Table 4.1 summarizes the migration costs within the cluster. The migration cost among cores within A15 cluster is 54 $\mu$sec – 105 $\mu$sec depending on the frequency level, while the cost within A7 cluster is 71 $\mu$sec – 167 $\mu$sec. Table 4.2 and 4.3 shows the migration costs across the clusters. The migration costs between clusters are somewhat high: 1.88ms – 2.16ms for moving from A7 to A15 cluster at different frequency levels, and 3.54ms – 3.83ms for a move from A15 to A7 cluster. The migration cost is proportional to the amount of architectural state being transferred.

| **Frequency(in MHz)** | 350 | 400 | 500 | 600 | 700 | 800 | 900 | 1000 | 1100 | 1200 |
|---|---|---|---|---|---|---|---|---|---|---|
| **Within A15 cluster** | 167 | 149 | 131 | 119 | 95 | 85 | 79 | 71 | - | - |
| **Within A7 cluster** | - | - | 105 | 91 | 79 | 72 | 63 | 58 | 58 | 54 |

TABLE 4.1: Migration Cost within cluster in usec.

| A15(MHz) / A7(MHz) | 500 | 600 | 700 | 800 | 900 | 1000 | 1100 | 1200 |
|---|---|---|---|---|---|---|---|---|
| **350** | 2.00 | 1.99 | 1.95 | 1.95 | 1.94 | 1.94 | 2.17 | 2.17 |
| **400** | 1.96 | 1.95 | 1.95 | 1.93 | 1.93 | 1.93 | 2.16 | 2.16 |
| **500** | 1.96 | 1.94 | 1.93 | 1.93 | 1.92 | 1.91 | 2.15 | 2.16 |
| **600** | 1.94 | 1.93 | 1.92 | 1.92 | 1.91 | 1.91 | 2.14 | 2.14 |
| **700** | 1.93 | 1.92 | 1.92 | 1.92 | 1.90 | 1.90 | 2.13 | 2.13 |
| **800** | 1.94 | 1.94 | 1.92 | 1.90 | 1.90 | 1.89 | 2.13 | 2.13 |
| **900** | 1.93 | 1.93 | 1.92 | 1.90 | 1.89 | 1.89 | 2.13 | 2.12 |
| **1000** | 1.93 | 1.92 | 1.90 | 1.90 | 1.89 | 1.88 | 2.13 | 2.12 |

TABLE 4.2: Migration Cost in msec from A7 to A15 cluster.

| A7(MHz) / A15(MHz) | 350 | 400 | 500 | 600 | 700 | 800 | 900 | 1000 |
|---|---|---|---|---|---|---|---|---|
| **500** | 3.66 | 3.65 | 3.63 | 3.62 | 3.60 | 3.59 | 3.82 | 3.83 |
| **600** | 3.65 | 3.63 | 3.61 | 3.60 | 3.58 | 3.55 | 3.81 | 3.82 |
| **700** | 3.64 | 3.63 | 3.59 | 3.58 | 3.56 | 3.55 | 3.79 | 3.79 |
| **800** | 3.65 | 3.62 | 3.59 | 3.57 | 3.55 | 3.55 | 3.78 | 3.79 |
| **900** | 3.64 | 3.62 | 3.61 | 3.59 | 3.56 | 3.55 | 3.78 | 3.79 |
| **1000** | 3.65 | 3.62 | 3.59 | 3.57 | 3.55 | 3.54 | 3.77 | 3.78 |
| **1100** | 3.70 | 3.58 | 3.57 | 3.56 | 3.55 | 3.54 | 3.78 | 3.80 |
| **1200** | 3.70 | 3.59 | 3.57 | 3.56 | 3.55 | 3.54 | 3.79 | 3.80 |

TABLE 4.3: Migration Cost in msec from A15 to A7 cluster.

Therefore, task migration for load balancing within a cluster can be performed more frequently, whereas migration decisions across clusters should be done infrequently.

## 4.2 Power Management Framework

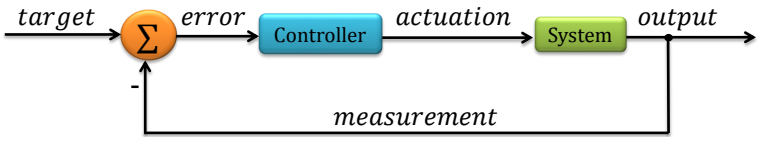



FIGURE 4.3: Feedback based Controller.

An overview of our hierarchical power management framework is presented in Figure 4.4. We incorporate several feedback based controllers in our framework. A controller measures the output metric and compares it with the reference or target metric as shown in Figure 4.3. The error is minimized by manipulating the actuators of the target system. The actuation policy is determined by the model

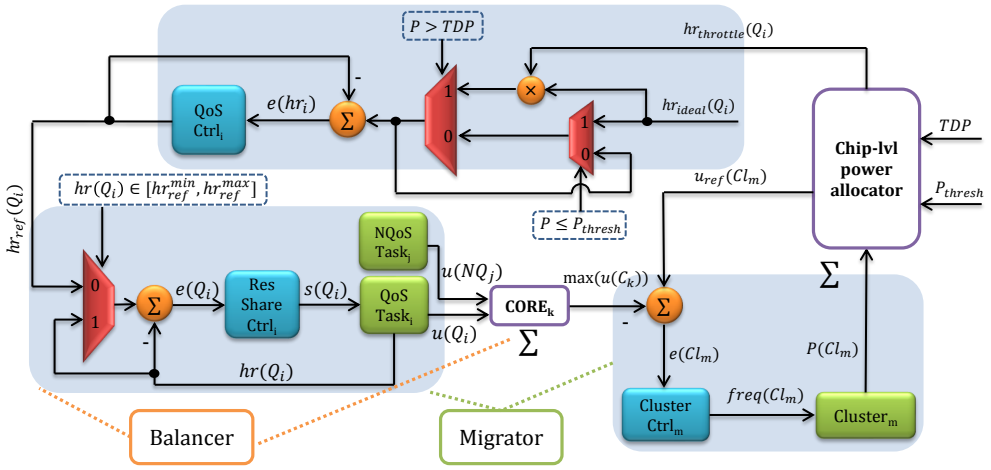



FIGURE 4.4: Overview of the hierarchical power management system coordinating multiple controllers.

of the target system being designed. We employ PID (Proportional-Integral-Derivative) controllers $z(t) = K_p e(t) + K_i \int e(t)dt + K_d \frac{de(t)}{dt}$, where $z(t)$, $e(t)$, $K_p$, $K_i$ and $K_d$ are the output of the controller, error, proportional gain, integral gain, and derivative gain, respectively. Table 4.4 summarizes the process variable, actuator and set point for the various controllers employed in our hierarchical power management framework.

| Controller Name | Process Variable | Actuator | Set Point |
|---|---|---|---|
| Resource Share Controller | heart rate | time slices | target heart rate |
| DVFS Controller | cpu utilization | voltage-frequency | target cpu utilization |
| QoS Controller | target heart rate | heart rate | throttled heart rate |

TABLE 4.4: Controller Features.

We have two types of tasks in our system; *QoS* and *non-QoS* tasks. A *QoS* task is one that demands certain user-defined throughput (e.g., video encoder, music player), while the *non-QoS* tasks do not specify any QoS requirement. As noted in Section 4.1.1, we specify the QoS of a task in terms of its heart rate.

The framework consists of three different types of controllers: per-task resource share controller, per-cluster DVFS controller, and per-task QoS controller. Each

QoS task in the system is assigned a resource share controller and a QoS controller. The resource share controller (bottom left in Figure 4.4) of a QoS task $Q_i$ manipulates the CPU share available to $Q_i$ so that it can meet the target heart rate $hr_{ref}(Q_i)$. The per-task QoS controller (top in Figure 4.4) is inactive when the entire system is lightly loaded. However, when the total power of the chip exceeds the TDP, the QoS controller slowly throttles the target heart rate $hr_{ref}(Q_i)$ so that the workload in the system decreases to a sustainable level and brings it back to the user- defined level when the thermal emergency is over.

We have two cluster controllers corresponding to A7 and A15 clusters. The objective of the controller for cluster $Cl_m$ (bottom right in Figure 4.4) is to apply DVFS such that the utilization remains close to the target utilization $u_{ref}(Cl_m)$. The utilization of a cluster is determined by the maximum utilization of its cores. Thus, we periodically invoke a load balancer to ensure even utilization among the cores within a cluster. We also invoke a migrator periodically (at a much longer interval compared to the load balancer) to migrate the tasks between the clusters if necessary. Finally, we have a chip-level power allocator (extreme right in Figure 4.4) that throttles the frequency of the clusters and forces QoS controller to degrade target heart rates when the total power exceeds the TDP.

The key challenge here is to coordinate the various controllers, load balancer, migrator, and chip- level power allocator. We achieve a synergistic coordination with two mechanisms. First, the different components in our framework are invoked at different timescales. The per-task resource share controller and load balancer are invoked most frequently, followed by per-cluster DVFS controller and per-task QoS controller, then the migrator, and finally the chip- level power allocator. This ensures that a task attempts to reach its QoS target by first manipulating its share in a core or through migration within a cluster. If this fails, then it tries to change the frequency of the cluster. As a last resort, the

task is migrated to another cluster. The thermal emergency takes quite a long time to develop; hence the power allocator is invoked least frequently.

Second, the controllers communicate with each other through designated channels. For example, the resource shares of the tasks within a core (both QoS and non-QoS) determines its utilization, which is provided as input to the cluster controller. More interestingly, when the power exceeds TDP, the power allocator increases the target utilization levels of the clusters $u_{ref}(Cl_m)$. This indirectly achieves the goal of decreasing power as the cluster controller is forced to lower its frequency in order to meet the increased target utilization. In parallel, the power allocator also sends a heart rate throttling factor ($hr_{throttle}(Q_i)$) to each QoS controller which makes them slowly degrade their target heart rate. This reduced heart rate is communicated to the resource share controller, who in turn, reduces the CPU share of the QoS tasks and hence the processor utilization to a more sustainable level. Overall, the system stabilizes to a level where the total power is just below the TDP.

### 4.2.1 Per-Task Resource Share Controller

We employ one resource share controller per QoS task. The target heart rate of a task $Q_i$ is defined as a range $hr_{ref}(Q_i) = [hr^{min}_{ref}(Q_i), hr^{max}_{ref}(Q_i)]$ and is set by the QoS controller. The objective of the resource share controller is to keep the measured heart rate $hr(Q_i)$ in the target heart rate range. This is achieved by regulating the slice $s(Q_i)$ of time provided to the task $Q_i$ in the scheduler. For example, a task that does not meet the minimum heart rate would demand more resource, which translates to more slices of time. The manipulation of the slice value of a task within Linux completely fair scheduler is explained in detail in Section 4.3. If the measured heart rate is within the reference range, then the controller does not need any action and hence the target heart rate $hr_{ref}(Q_i)$ is set to the measured heart rate $hr(Q_i)$ so that error is zero in the controller.

### 4.2.2 Per-Cluster DVFS Controller

Let a core $C_k$ consist of $N$ QoS and $P$ non-QoS tasks. Then its current utilization is

$$u(C_k) = \sum_{i=0}^{N} u(Q_i) + \sum_{j=0}^{P} u(NQ_j) \tag{4.1}$$

where $u(Q_i)$ and $u(NQ_j)$ are the utilizations of the QoS task $Q_i$ and non-QoS tasks $NQ_j$, respectively. The *core* component in Figure 4.4 is responsible for measuring the utilization of each individual core. As the frequency can be controlled only at cluster level, the utilization of cluster $Cl_m$ defined as $u(Cl_m)$ is set to the maximum utilization $max(u(C_k))$ across all the cores within the cluster.

The DVFS controller attempts to achieve the target utilization

$$u_{ref}(Cl_m) = max(u_{ideal}, u_{target}(Cl_m)) \tag{4.2}$$

where $u_{ideal}$ is a constant specifying the ideal target utilization and $u_{target}(Cl_m)$ is the target utilization set by the power allocator under thermal emergency. Using $max(u(C_k))$ as the measured metric and $u_{ref}(Cl_m)$ as the reference metric, the cluster-level PID controller actuates the frequency of the cluster.

### 4.2.3 Chip-Level Power Allocator

When the total power of the chip exceeds the TDP, the power allocator needs to throttle the frequency of the clusters and the QoS of the tasks. Let $P_m$ be the current power measured for cluster $Cl_m$. The target power $\overline{P_m}$ for cluster $Cl_m$ is calculated using the following equation

$$\overline{P_m} = P_m - \left( (P - TDP) \times \frac{(T^{qos} - T_m^{qos})}{T^{qos}} \right) \tag{4.3}$$

where $P$ is the total power of the chip given by $P = \sum_{m=0}^{M} P_m$, $T^{qos}$ is the total number of QoS tasks in the system and $T_m^{qos}$ is the total number of QoS tasks in the cluster $Cl_m$. From Equation 4.3, it is evident that the reference power allocated to the cluster is proportional to the number of QoS tasks in the cluster. From the reference power budget allocated to each cluster, the power allocator computes $u_{target}(Cl_m)$ using the following equation,

$$u_{target}(Cl_m) = u_{ideal} + u_{ideal} \times \frac{P_m - \overline{P_m}}{P_m} \tag{4.4}$$

In the event of TDP violation, the power allocator increases the target utilization $u_{ref}(Cl_m)$ of the cluster, which in turn causes cluster-level DVFS controller to set a lower frequency for the cluster. As our controllers are reactive in nature, the power may exceed the TDP for a short time interval. The gain factors within the DVFS controller are set appropriately so that it stabilities the power below the TDP within the specified time interval (typically few seconds [97]) as demonstrated in Section 4.3.

When TDP is violated, the power allocator also sets a throttle factor $hr_{throttle}(Q_i)$ for each QoS task $Q_i$ in a hierarchical manner. The throttle factor $hr_{throttle}(Cl_m)$ for a cluster is proportional to its penalty factor as defined via higher than ideal utilization.

$$hr_{throttle}(Cl_m) = 1 - \frac{(u_{target}(Cl_m) - u_{ideal})}{u_{target}(Cl_m)} \tag{4.5}$$

The cluster throttle factor is further divided among the cores

$$hr_{throttle}(C_k) = hr_{throttle}(Cl_m) \times \frac{u(C_k)}{u_{avg}(Cl_m)} \tag{4.6}$$

where $u_{avg}(Cl_m)$ is the average utilization in cluster $Cl_m$ across all the cores. Finally, the throttle factor of a QoS task in a core ensures that the penalty of a

task is proportional to its utilization.

$$hr_{throttle}(Q_i) = hr_{throttle}(C_k) \times \frac{\sum_{i=0}^{N} u(Q_i)}{u(C_k)} \tag{4.7}$$

Once the system escapes from the thermal emergency, the power allocator needs to set back $hr_{throttle}(Q_i) = 1$. During the thermal emergency, the clusters reduce their frequency and the QoS tasks reduce their workload, the power decreases just below the TDP. However, the QoS of the tasks cannot be brought back to their ideal QoS level as the system will again oscillate back to thermal emergency. The QoS of the tasks can be restored only when the workload decreases because (a) one or more tasks leave the system and/or b) the tasks exhibit phase behavior. This is reflected in the drop in power consumption of the system. Thus, we chose an empirically determined power threshold $P_{thresh}$ below which the $hr_{throttle}$ is set to one (as shown in Figure 4.4).

### 4.2.4 Per-Task QoS Controller

The QoS controller provides the graceful degradation of the QoS measure in case of thermal emergency by manipulating the target heart rate $hr_{ref}(Q_i)$. The input to this controller is the user-defined ideal heart rate range $hr_{ideal}(Q_i) = [hr^{min}_{ideal}(Q_i), hr^{max}_{ideal}(Q_i)]$. When the power is below the TDP, the power allocator sets $hr_{throttle}(Q_i) = 1$ and this controller sets $hr_{ref}(Q_i) = hr_{ideal}(Q_i)$. In case of thermal emergency, the controller strives to set the reference heart rate $hr_{ref}(Q_i) = hr_{throttle}(Q_i) \times hr_{ideal}(Q_i)$.

### 4.2.5 Load Balancer and Migrator

In our framework, the *Balancer* ensures that the cores within the cluster are evenly load balanced in terms of the utilization. The *Migrator* migrates the set

of tasks that do not achieve their target heart rate at maximum frequency in the A7 cluster to the A15 cluster. Similarly, a task is migrated from A15 cluster to A7 cluster when the measured heart rate $hr(Q_i)$ is above the maximum target heart rate $h_{ref}^{min}(Q_i)$ at the minimum frequency in the A15 cluster.

## 4.3 Experimental Evaluation

### 4.3.1 Implementation Details

**big.LITTLE platform with Linux.** In our evaluation, we use the real Versatile Express development platform [7] as shown in Figure 4.5. It is a flexible, configurable and modular developing platform that allows quick prototyping of hardware and software projects. The system comprises a motherboard on which modular daughter boards can be plugged. The big.LITTLE processor is part of the daughter board (TC2) pointed in the Figure 4.5. The motherboard handles the interconnection between the daughter board and the peripherals by using a FPGA bus interconnection network. The process technology of the TC2 board is at 45nm Generic Process (GP). The TC2 test chip consists of two core Cortex-A15 (big) cluster and three core Cortex-A7 (LITTLE) cluster. Both big and LITTLE cores implement ARM v7A ISA and are connected together by CCI-400 cache coherent interconnect. The frequency can only be modified at the cluster level in the target platform.

The board boots an Ubuntu 12.10 Linaro release for Versatile Express [8]. The platform firmware runs on an ARM controller (MCC) embedded on the motherboard and handles the load of the Linux kernel while booting. The Linux file system is installed on the Secure Digital (SD) card where all our benchmarks are saved. The TC2 daughter board is also equipped with sensors for measuring the frequency, voltage, current, power and energy consumption per cluster. The board also supports the change of voltage and frequency per cluster.

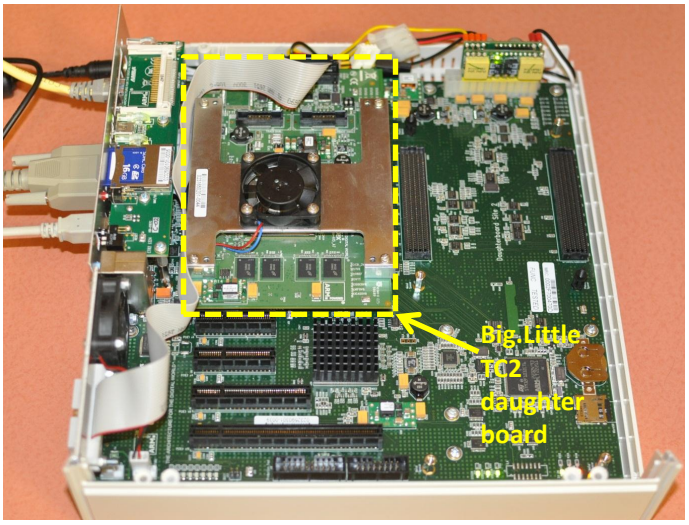



FIGURE 4.5: Picture of the Vexpress board.

**Profiling Section.** The Linux version [8] provides hardware monitor (*hwmon*) interface to communicate with the sensors located in the test chip. We use ARM Streamline gator kernel module and daemon [1] to obtain performance related metrics like instructions per cycle (IPC). Powering off the cluster and adjusting the clock frequency were made possible by accessing the oscillator related drivers provided in the kernel. The legal voltage and frequency ranges for the clusters are shown in Table 3.1.

**Managing task slice and Migrator.** Linux scheduler uses the notion of time slicing for allocating the resources to the running tasks in the system. At every system tick (10ms in our experiments), the kernel computes the time slice that the next tasks should receive. By default, the CFS scheduler fairly divides a relatively fixed period of time (6ms in our experiments) and allocates the slice to the task. The slice dictates the duration for which the task can consume the core. Our Resource Share Controller manipulates the computed slice for the QoS task by the original Linux mechanism by gradually increasing the time slice when a higher utilization is required or reducing the slice when less utilization is required. For non-QoS tasks, the CFS scheduler will try to fairly share the remaining time

period among them. Linux kernel uses cpumask to decide the affinity of the tasks. Migrator component in our HPM alters the cpumask associated with each task to attain the desired scheduling decision.

In Table 4.5 we show the minor modifications that we did in the Linux kernel in order to implement our HPM scheduler.

| Function | Description | # lines |
|---|---|---|
| scheduler_tick() | Fire controllers based on system tick. | 30 |
| load_balance() | HPM Balancer within the cluster. | 12 |
| run_rebalance_domains() | HPM Migrator across the clusters. | 47 |
| sched_slice() | Manipulate the QoS time slices. | 5 |

TABLE 4.5: Linux kernel modifications.

**Heart Rate Monitor and Benchmarks.** We use the Application Heartbeats framework proposed in [54] as a mechanism to measure the performance of an application. The API provided in this framework provides a QoS metric in terms of *heartbeats* which are periodic signals sent by an application to track its progress. The QoS metric provided by the framework is called *heartrate* (i.e, the number of heartbeats per second). For example, in video encoding applications the heartbeats can be registered every frame. Thus, the heart rate measured would be the number of frames per second. The interested reader is referred to [54] for more information on Heartbeats Framework. In the absence of HRM infrastructure, an approximate way to determine the demand in Linux operating system is to measure the time a task spends in the run-queue in a given epoch of scheduling. This per-entity load tracking proposed by Paul Turner [117] in kernels higher than 3.7 can be used in lieu of heartbeats.

Table 5.5 describes the benchmarks used in our experiments together with the inputs. Table 4.7 summarizes heartbeat insertions in the benchmarks [54].

**Controller Features.** We deploy PID controllers for per-task resource share controllers, per-task QoS controllers, and per-cluster DVFS controller. Table

| Benchmark | Benchmarks suite | Description | Inputs |
|---|---|---|---|
| swaptions | PARSEC | QoS<br>Monte Carlo (MC) simulation. | sim_native |
| bodytrack | PARSEC | QoS<br>Tracks a human body with images. | sim_native |
| x264 | PARSEC | QoS<br>Video encoder. | sim_native |
| blackscholes | PARSEC | QoS<br>Solves partial differential equation. | sim_native |
| h264 | SPEC 2006 | QoS<br>Video encoder. | foreman |
| disparity | Vision | non-QoS<br>Motion, tracking and stereo vision. | fullhd |
| sift | Vision | non-QoS<br>Image Analysis. | fullhd |
| tracking | Vision | non-QoS<br>Motion, tracking and stereo vision. | fullhd |

TABLE 4.6: Benchmarks description.

| Benchmark | Heartbeat Location |
|---|---|
| swaptions | Every "swaption" |
| h264 | Every frame |
| bodytrack | Every frame |
| x264 | Every frame |
| blackscholes | Every 25000 options |

TABLE 4.7: Heartbeats in QoS benchmarks.

4.8 summarizes our HPM framework, describes the terminologies and provides the gain factor values associated with each of the controllers employed in our experiments. The invocation period of RSC is a user-defined value. For example, for video encoding it can typically be 30 frames per second, which translates to RSC being evoked every 30 frames. The invoke period was chosen in such a way that the per-task resource share controller and load balancer are invoked most frequently followed by DVFS controller, per-task QoS controller, migrator and finally the chip-level power allocator.

### 4.3.2 Results

We use the sequential version of the PARSEC benchmarks as QoS tasks. We specify and track the heart rates for the QoS tasks using Heart Rate Monitor infrastructure [54] integrated with our Linux kernel. Note that some of the benchmarks are computationally demanding (e.g., *x264*) and requires hardware

| Controller name | Metrics | Symbol | Value |
|---|---|---|---|
| **Resource Share Controller (RSC)** | target heart rate | $hr_{ref}$ | tuned by QoSC |
| | measured heart rate | $hr$ | measured by the task |
| | slice | $s$ | actuator tuned by RSC |
| | proportional gain | $K_p^{RSC}$ | 0.8512 |
| | integral gain | $K_i^{RSC}$ | 0.01241 |
| | derivative gain | $K_d^{RSC}$ | 0.00941 |
| | invoked period | $T^{RSC} = \beta \times \frac{1}{hr_{ideal}}$ | determined by the $hr_{ideal}$ |
| | heart rate measurement frequency factor | $\beta$ | user-defined |
| **CORE component** | core utilization | $u_k$ | measured by each core |
| | invoked period | $T^c = 4 \times max(T^{RSC}(Q_i))$ | determined by the task with max $hr_{ideal}$ |
| **DVFS Controller (DVFSC)** | target cluster utilization | $u_{ref}$ | estimated by CHIP component |
| | measured cluster utilization | $max(u(C_k))$ | measured by CORE component |
| | cluster frequency | $freq$ | tuned by DVFSC |
| | proportional gain | $K_p^{DVFSC}$ | 0.9533 |
| | integral gain | $K_i^{DVFSC}$ | 0.2572 |
| | derivative gain | $K_d^{DVFSC}$ | 0.0014 |
| | invoked period | $T^{DVFSC} = 5 * T^c$ | slower than the CORE component |
| **QoS Controller (QoSC)** | ideal hr | $hr_{ideal}$ | user-defined |
| | throttle factor | $hr_{throttle}$ | estimated by CHIP component |
| | target reference hr | $hr_{ideal} \times hr_{throttle}$ | product of ideal hr and throttle factor |
| | measured reference hr | $hr_{ref}$ | measured by the task |
| | proportional gain | $K_p^{QoSC}$ | 0.74175 |
| | integral gain | $K_i^{QoSC}$ | 0.0214 |
| | derivative gain | $K_d^{QoSC}$ | 0.0045 |
| | invoked period | $T^{QoSC} = 30 * T^{RSC}$ | much slower than RSC |
| **CHIP level power allocator** | thermal design power | $TDP$ | user-defined |
| | threshold power | $P_{thresh}$ | user-defined |
| | throttle factor | $hr_{throttle}$ | estimated by CHIP component |
| | invoked frequency | $T^{ch} = 2 \times T^{QoSC}$ | slower than DVFSC |
| **Balancer** | invoked period | $T^b = 2 \times T^{RSC}$ | faster than DVFSC |
| **Migrator** | invoked period | $T^m = 4 \times T^{QoSC}$ | slower than QoSC |

TABLE 4.8: Controller Parameters.

accelerators for execution. As we run software-only versions of these benchmark, they achieve low heart rate even on A15 core at highest frequency.

The evaluations are designed to demonstrate that HPM achieves the following objectives: (1) HPM can exploit asymmetry to provide significant energy savings compared to symmetric multi-cores, (2) HPM performs better than the Linaro scheduler, (3) HPM can respond to thermal emergency in a graceful manner, and (4) HPM does not interfere with the desired properties of Linux CFS, namely, fairness and non-starvation of the non-QoS tasks.

**Asymmetric versus symmetric multi-core.** We use *x264* benchmark that exhibits phases with varying performance requirements during execution. The symmetric architectures are emulated using only A7 cluster or A15 cluster. We run *x264* benchmark on each of these configuration. All the configurations use HPM framework; but inter-cluster migration is disabled for symmetric architectures. Figure 4.6 plots the heart rate on the asymmetric and symmetric configurations. The heart rate line type specifies the cluster on which the task is

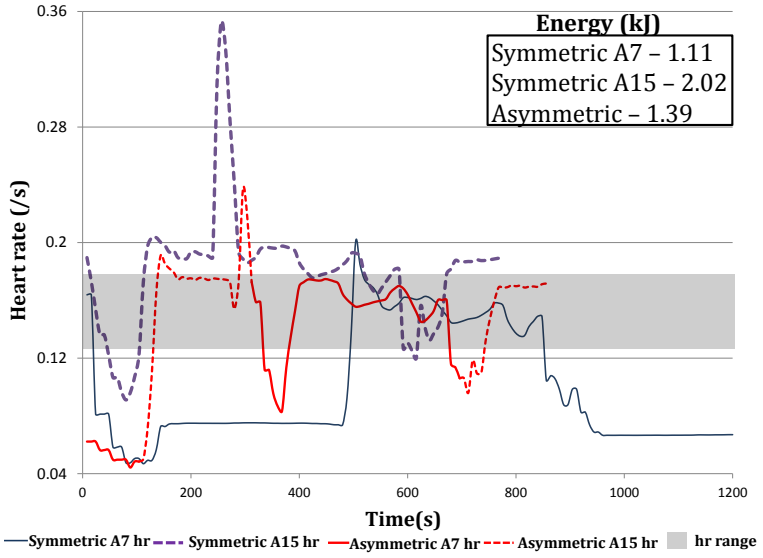



FIGURE 4.6: x264: Heart rate on symmetric & asymmetric multi-core.

running: continuous line corresponds to A7 cluster and dashed line corresponds to A15 cluster. The gray shaded area shows the specified heart rate range.

On symmetric configurations, the measured heart rate is below the minimum heart rate most of the time when executing on A7 cluster, while the heart reate mostly exceeds the maximum heart rate when running on A15 cluster. As expected, the energy consumption is very low (1.11kJ) in A7 cluster and quite high in A15 cluster (2.02kJ). The asymmetric multi-core provides the best of both worlds. On the asymmetric architecture, we can see that the application migrates to A15 cluster for the demanding phases and moves back to A7 cluster as the computational demand decreases. The HPM manages to maintain the heart rate within the reference range with a very low energy consumption (1.39kJ), which is 68% less than the energy consumption on A15 cluster alone.

**HPM versus Linaro scheduler.** We compare HPM scheduler with Linaro scheduler kernel release 3.6.1, where we activate the power conservative governor. The Linaro scheduler is aware of the different performance capabilities of the asymmetric cores, but it does not react to different performance requirements

of the QoS tasks. Once the task load (defined as time spent on the runqueue of the processor) increases above a predefined threshold, the Linaro scheduler moves the task to the more powerful core. However, it never migrates the task back to the weaker core when workload reduces. We launch three QoS tasks, *x264*, *bodytrack*, *h264*, on three A7 cores. The results are shown in Figure 4.7. In all the subgraphs the X-axis shows the time in seconds. The Y-axis in the first three subgraphs shows the measured heart rate of the QoS tasks under HPM and Linaro. Additionally, the figure shows the specified heart rate range for the tasks as grey shared area. The last subgraph shows power comparison between the two approaches.

*bodytrack* and *h264* meet their specified heart rate on A7 cluster. As *x264* does not meet its heart rate on A7 all the time, it is migrated to A15 cluster by HPM when necessary. All the while, HPM keeps the heart rate of all the applications within the specified range. The Linaro scheduler, on the other hand, migrates all the tasks to the A15 cluster based on task load. As a result, the tasks complete execution much earlier compared to HPM; but exceeds the heart rate by a large margin consuming significantly more energy. On an average, the system consumes 2.27W using our scheduler compared with 5.83W consumed under Linaro scheduler.

The average frequencies of the A7 and A15 clusters are 780MHz and 157MHz when using our scheduler. Due to aggressive migration by Linaro scheduler, the average A7 frequency is 170MHz and the average A15 frequency is 1044MHz. We plot the frequency change along with the power values in Figure 4.8 where frequency 0 corresponds to a switched off cluster.

Table 4.9 quantitatively shows the average power consumption and heart rate miss percentage (i.e., how much time a QoS task spends below its minimum specified heart rate) for HPM and Linaro scheduler using identical experimental setup but five different combination of QoS benchmarks. In general, a small loss

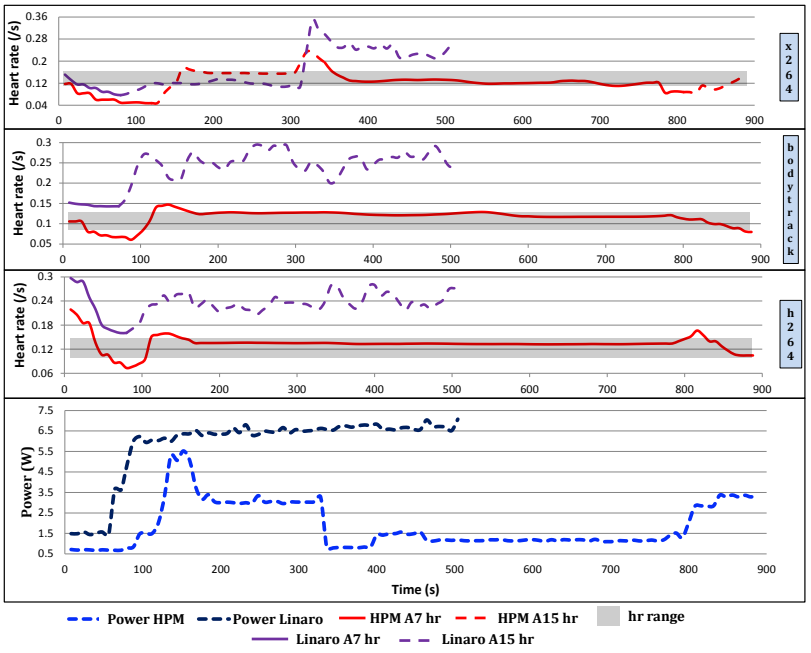



FIGURE 4.7: HPM versus stock Linaro scheduler equipped with DVFS governor and inter-cluster migration.

in performance of the QoS tasks in our framework is heavily compensated by the average power reduction. The Linaro scheduler performs quite badly even in terms of performance in the two highlighted experiments. This is because the benchmarks are very demanding. Linaro scheduler moves them all to the A15 cluster, where they suffer from lack of resources, even at the highest frequency level. HPM uses the resources more efficiently and miss rate is reduced along with considerable reduction in power consumption. The results clearly demonstrate that HPM exploits the asymmetric architecture much more efficiently than current Linux scheduler.

| | **HPM scheduler** | | **Linaro scheduler** | |
|---|---|---|---|---|
| **Benchmarks** | **Avg Power(W)** | **hr miss %** | **Avg Power(W)** | **hr miss %** |
| swap_h264_x264 | 3.35 | 8.27 | 6.18 | 5.95 |
| swap_h264_body | 3.88 | 13.39 | 6.06 | 9.80 |
| **h264_body_black** | **4.19** | **15.65** | **6.00** | **33.99** |
| **black_x264_h264** | **4.21** | **19.93** | **6.19** | **29.76** |
| x264_body_h264 | 2.27 | 9.61 | 5.83 | 7.41 |

TABLE 4.9: Quantitative comparison of HPM with Linaro scheduler.

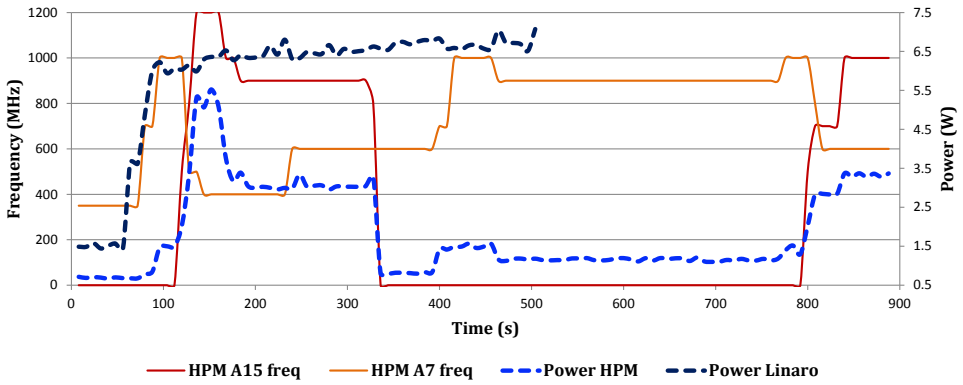
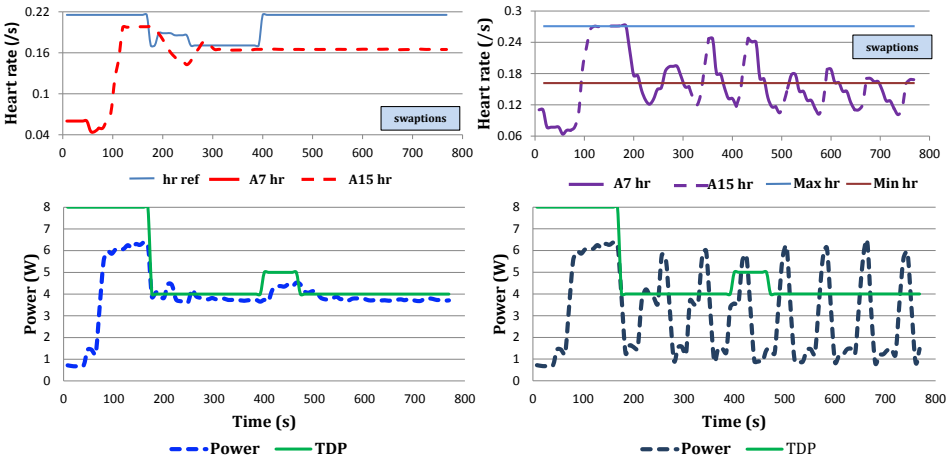



FIGURE 4.8: Frequency and power consumption plot (HPM versus stock Linaro scheduler).

**Response under TDP constraint.** This experiment evaluates the efficiency of HPM in managing the chip power below the TDP through DVFS and graceful degradation of the QoS of the tasks if necessary. For fair comparison, we add a feature to the Linaro scheduler that switches off the A15 cluster once the power exceeds the TDP threshold. We use *bodytrack*, *swaptions*, and *h264* where the first two benchmarks have high workload and are migrated to A15 cluster. *swaptions* is the most demanding one and sets the frequency of the A15 cluster to the highest value. As we cannot control the frequency of individual cores, the core with *bodytrack* is forced to run at a higher frequency than required and hence its heart rate exceeds the target. Figure 4.9a shows the heart rate of *swaptions* (the application with maximum impact on power) together with the median value of the target heart rate range. The subgraph at the bottom of Figure 4.9a shows the chip power and the specified TDP cap. In this experiment, we dynamically change the TDP cap between 4-8W to demonstrate how the scheduler adapts to TDP budget. Once the chip power exceeds the TDP, the power allocator immediately increases the target utilization value of the clusters, which forces the DVFS controllers to decrease the frequency, and thereby reduce total chip power. Meanwhile, the power allocator also sets the heart rate throttle values, which in turn makes the QoS controllers reduce the target heart rates correspondingly bringing down the workload to a more sustainable level. HPM




(A) HPM (B) Cluster switch off

FIGURE 4.9: Comparison of HPM and Linaro extended with cluster switch-off policy under TDP constraint.

always maintains the heart rate of *swaptions* at the target value. Note that the target heart rate is decreased by the QoS controller when the power is above the TDP, thereby degrading the performance of the tasks. Once TDP is increased, the target heart rate switches back to the user-specified ideal value.

In case of the modified Linaro scheduler (Figure 4.9b) the A15 cluster is switched on and off frequently in response to increase in chip power beyond TDP. This oscillation happens because the workload is not throttled when the A15 cluster is switched off. As soon as A15 cluster is switched off, the power decreases much below the TDP, the tasks again migrate back to A15, the power increases above TDP, and the cycle continues. This frequent powering down of clusters and consequent migration makes *bodytrack* and *swaptions* run below their target heart rate most of the time under modified Linaro scheduler. This experiment confirms that a holistic approach is required to maintain the chip power below TDP; our approach not only decreases the frequency of the clusters but also solves the root cause of increased power by slowly degrading the QoS of the tasks. As a result, our approach reaches a stable and sustainable level both w.r.t. the heart rate and the chip power.

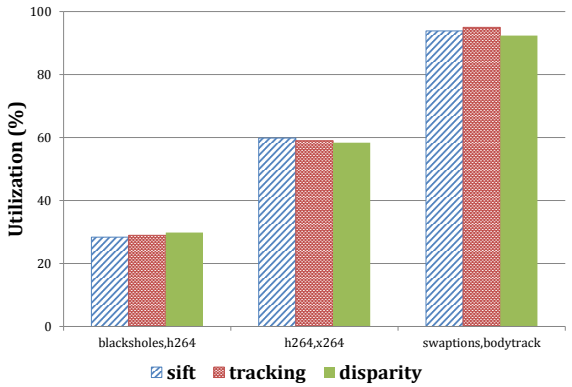



FIGURE 4.10: Fairness of non-QoS tasks.

**Fairness of Non-QoS tasks.** Our HPM framework is built on top of the existing Linux kernel scheduler. This set of experiments validate that we do not interfere with the scheduling of the non-QoS tasks handled by the Completely Fair Scheduler (CFS), which guarantees equal (fair) share of processor utilization among the tasks.

We run three experiments each with three non-QoS tasks (*sift*, *tracking*, *disparity*) and two QoS tasks. The behavior of the QoS tasks dictates the amount of A7 cluster utilization that CFS can provide to the non-QoS tasks. The first experiment uses *blackscholes* and *h264* as QoS tasks that satisfy the target heart rate on A7 cluster with close to maximum target utilization. Thus the CFS scheduler clusters together non-QoS tasks on the third available core. Figure 4.10 shows that the non-QoS tasks have equal share of utilization (33%).

The second experiment involves *h264* and *x264*; *x264* has a demanding execution phase where HPM migrates it to A15 cluster. Mostly the three non-QoS tasks run on their cores receiving 60% utilization, while *h264* runs on A7 cluster.

The final experiment uses *swaptions* and *bodytrack*, both of which migrate to A15 cluster and non-QoS tasks receive almost 100% of the A7 cluster utilization.

## 4.4 Summary

In this chapter, we presented a dynamic power management framework for heterogeneous multi-cores that carefully coordinates multiple controllers. It is integrated with Linux on ARM big.LITTLE platform. It exploits asymmetry among the cores through selective migration and employs DVFS to minimize energy consumption while satisfying QoS constraints. Our technique combines graceful QoS degradation at task level with power reduction through DVFS at core level to reach a stable and sustainable execution under TDP cap.

# Chapter 5

# Price Theory based Power Management

In this chapter, we improve the dynamic power management technique proposed in Chapter 4 by introducing features like scalability, priority-driven and priority consciousness. As in chapter 4, the goal is to develop a power management framework for heterogeneous multi-cores in mobile platforms, that can satisfy applications' demand expressed in terms of Quality of Service (QoS) with low energy consumption footprint under Thermal Design Power (TDP) constraint. We explain the features added to *HPM* technique in detail:

- Traditionally, centralized power management [26, 58, 122] have been employed in embedded mobile platform. However, centralized approaches suffer from scalability issues, specially in future many-core systems [5, 29] containing heterogeneity. For example, Samsung's Exynos 5 Octa contains 8 cores: 4 Cortex A-15 and 4 Cortex A-7 cores. Scalable power management technique for such platforms should have distributed decision making strategies.

- Incorporating priority-driven feature gives a better user experience in mobile platforms. For example, when power exceeds the TDP, it would be favorable to penalize the lower-priority tasks higher compared to high-priority tasks.

- Among the same priority tasks, it is favorable to allocate more computational resources to tasks with the bursty feature relative to a compute-bound and long running tasks. Most of the interactive mobile applications like browsers, speech recognition etc are bursty in nature.

To incorporate all objectives mentioned in Chapter 4 and fulfill the aforementioned challenges, we propose a framework based on the foundation of *price theory* [71] from economics. The *price theory* framework has strong applicability on resource management problems. All our evaluations are performed on a real heterogeneous multi-core chip ARM big.LITTLE.

## 5.1 System Overview

In this section, we present the models for the different entities in the heterogeneous multi-core system where our proposed price theory based power management scheme is employed.

**Architecture model.** We target single-ISA heterogeneous multi-core architectures, which exhibit power-performance heterogeneity as in big.LITTLE [49] and Tegra [5] platforms. The target system is comprised of a set of cores $\mathcal{C}$ grouped in a set of voltage-frequency clusters $\mathcal{V}$, with each cluster having a separate voltage and frequency regulator. Each cluster $v$ can operate at several discrete voltage-frequency (V-F) levels and consists of a set of cores $\mathcal{C}_v \subseteq \mathcal{C}$. All the cores within a cluster are symmetric in terms of micro-architecture and have to run at the same V-F level.

**Task model.** A task $t$ is a computational entity that can execute on a core. Each task $t$ is assigned a priority $r_t$ by the user, where higher value means higher priority. $\mathcal{T}$ represents the set of all tasks.

**Task to core mapping.** Our framework dynamically maps the tasks to the cores. A task $t$ is mapped to a core $c_t$. $\mathcal{T}_c \subseteq \mathcal{T}$ represents the set of tasks mapped to core $c$ and $\mathcal{T}_v = \cup_{c \in \mathcal{C}_v} \mathcal{T}_c$ denotes the set of tasks mapped to the cores in cluster $v$. The idle task $t_{idle}$ executes on a core without any active task. If there are no active tasks in an entire cluster, then we can power down that cluster.

We define $R_c$, $R_v$, $R$ as the sum of the priorities of all the tasks mapped to core $c$, cluster $v$, and the entire system, respectively.

**Supply Model.** Each core $c$ can supply certain amount of computational resources $S_c$, which is constrained by the maximum supply $\hat{S}_c$. The computational resource is defined in terms of of *Processing Units* (PU), where one PU is equivalent to one million processor cycles per second. The higher the frequency of a core $c$, the more it can supply PUs (i.e., higher the value of $S_c$) and the maximum supply of PUs $\hat{S}_c$ is determined by the maximum possible frequency of the core. For example, a core running at 1000MHz (or 350MHz) produces a supply of 1000PUs (or 350PUs). Note that the amount of work (instruction processing) that can be achieved with one PU on a small core is generally less than the amount of work that can be done with one PU on a big core; that is, one PU on a big core is more valuable than one PU on a small core.

The supply of a cluster $S_v$ is the same as the supply of any of the constituent cores, which have identical $S_c$ values. The supply of the entire chip $S$ is the summation of the cluster supply values. The current supply of PUs to task $t$ on core $c_t$ is represented by $s_t$ ($\equiv s_t^{c_t}$). The supply of PUs to a task $t$ has to be less than the supply produced on the core $c_t$ it is mapped to, that is, $s_t \leq S_{c_t}$.

**Demand Model.** Each task $t$ demands a certain amount of computational resources (PUs), which can vary dynamically during the course of the execution. In heterogenous multi-cores, a task demands different amount of PUs across different core types. For example, a task would demand more PUs on a small core compared to a big core to achieve the same application-level performance. The differing demands for computational resources on different core types model the **heterogeneity** of the architecture. The current demand of task $t$ on core $c_t$ is represented by $d_t$ ($\equiv d_t^{c_t}$).

Let $D_c$ represent the sum of demands of all the tasks mapped to core $c$. The core with the highest demand in a cluster is called the constrained core of the cluster. Let $\tilde{c}_v \in \mathcal{C}_v$ represent the constrained core of the cluster $v$. Then the demand of the cluster $D_v = D_{\tilde{c}_v}$ is defined as the demand of its constrained core $\tilde{c}_v$ and the demand of the entire chip $D$ is the summation of the demands of the clusters.

**Power model.** The power consumption of a core $c$ represented by $W_c$ depends on the core type, its V-F level, and the workload. The power consumption of a cluster $v$ is represented by $W_v$, while the entire chip power consumption is represented by $W$.

The quality of the cooling solution determines the value of the TDP constraint $W_{tdp}$. As mentioned before, our goal is to keep the total chip power consumption below the TDP ($W < W_{tdp}$) while meeting the task demands at minimal energy.

## 5.2 Power management Framework

In this section, we introduce our novel price theory based power management framework in heterogeneous multi-cores. Our power management framework is designed on the foundation of the *Price Theory* [71] and the *Quantity theory of money* [46]. The resource allocation, DVFS, task mapping and migration are all

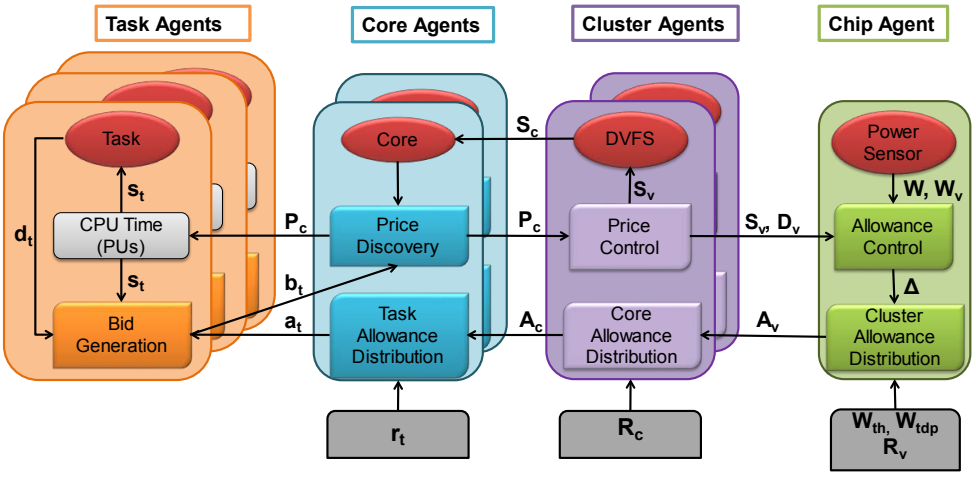



FIGURE 5.1: Agent Interaction Overview

controlled through the virtual market place, where the commodity being traded is processing unit (PU) using virtual money.

The framework is realized as a collection of autonomous entities called *agents*. Each agent represents a transactional body in the market. An agent can perform various functionalities such as earning, bidding, purchasing, and distribution of computational resources. Furthermore, the actions of an agent are prompted by the goals of the client(s) it represents. For example, an agent representing a task is motivated to meet the demand imposed by the task. Similarly, the agent representing the entire chip ensures that the thermal design power constraint is not violated.

The efficient functioning of the market is ensured through the regulations imposed on the actions of the agents. The regulations introduced in our market are reflections of the power management goals to be achieved. According to *Price Theory*, in a competitive market, the quantity and price at which a commodity is traded will be determined by point of intersection of its supply and demand curve, also known as economic equilibrium. Furthermore, based on *Quantity theory of money*, if we increase(or decrease) the money in circulation while keeping

the supply and demand fundamentals unchanged, we will observe inflation(or deflation) in equilibrium market price.

The economic market sans inflation (or deflation) exhibits a stable equilibrium price only when the demand is completely satisfied. Thus, controlling inflation (or deflation) is equivalent to providing enough supply to satisfy the current demand in the market. The supply can be modified by adjusting the v-f levels. In the market, high prices of the commodity is the reflection of high contention for the resource, which leads to inflation (increase in supply) and thus higher power consumption. Therefore, an efficient rearrangement of the market dynamics leads to lower prices, thus less spending is more desirable. Interested readers may refer to [46, 71] for more details on economic equilibrium, inflation and deflation concepts.

Our power management framework consists of two main components: *Supply-Demand module* and *Load-Balancing plus Task migration module (LBT)*. Given a task to core mapping, the supply-demand module attempts to satisfy the demands of all the tasks with minimal power consumption under the TDP constraint. It relies on the concept of regulating inflation-deflation. The LBT module aims to reach a power efficient task to core mapping through load balancing and task migrations and employs the concept of reduced spending. Both the modules work in tandem to achieve the final design goal of power management in heterogeneous multi-cores.

### 5.2.1 Agents Overview

Figure 5.1 shows the interaction among the agents.

**Task agent.** Each task is represented by an agent, who is a buyer in the market. A task agent can receive, spend, or save money to purchase the computational resources (PU). An agent corresponding to a task gets an allowance

(virtual money) that it uses to bid for the resources according to the demands of the task.

**Core agent.** Each core is represented by an agent who determines the price of the computational resources produced by the core. The price for PUs in a core emerges from the bids submitted by the task agents and the current supply of the core. The core agent then distributes the available resources among the task agents according to the bids. A core agent also distributes the allowances received from the cluster agent to the task agents.

**Cluster agent.** A cluster agent controls the price of the resources by manipulating the supply of PUs in the cores under its purview. The increase (or decrease) of the supply is achieved by varying the V-F levels of the cluster. A cluster agent also distributes the allowance received from the chip agent to the core agents.

**Chip agent.** The chip agent controls the amount of money in circulation in the system by manipulating the allowances, thereby ensuring that the total chip power does not exceed the TDP constraint. It then distributes the allowances to the cluster agents.

### 5.2.2 Supply-Demand Module

In this section, we explain the mechanisms employed by the supply-demand module in manipulating the V-F levels to meet the task demands at minimal power consumption. The supply-demand module requires all the task, core, cluster, and chip agents to work in synergy. In terms of price theory, the demands of all the tasks are satisfied only in an economic market without inflation (or deflation). Thus, controlling inflation (or deflation) is equivalent to providing enough supply to satisfy the current demand in the market. This is the basic regulatory principle employed in the supply-demand module.

#### 5.2.2.1 Task Dynamics

The main objective of the task agent is to sustain the demand of the task it represents. This objective is achieved through an iterative process consisting of three steps per round: bidding by the task agents, price discovery by the core agent, and purchase of the resources. The iterative process continues till the market stabilizes in an economic equilibrium.

The core agent gives an allowance $a_t$ (the virtual money) to each task agent according to the priority of the task. The task agent bids an amount $b_t$ to buy resources based on the current demand of the task $d_t$. If the bid is less than the allowance, then the difference $m_t = a_t - b_t$ is saved for future use. The bid cannot exceed the sum of allowance and savings. We also require the bid to be higher than a pre-defined minimum bid $b_{min}$. That is $b_{min} \leq b_t \leq a_t + m_t$.

For every round, each task agent submits a bid amount $b_t$ based on the experience in the previous round. The task agent increases (or decreases) the bid amount if the supply received was less (or more) than the demand in the previous round. The agents keep the bid unchanged when the demand is satisfied.

Given the bids from all the tasks mapped to core $c$, the core agent representing $c$ discovers the price per PU $P_c$ as follows

$$P_c = \frac{\sum_{t \in \mathcal{T}_c} b_t}{S_c} \tag{5.1}$$

Each task agent now purchases the resources at the value determined by the core agent and obtains its current supply $s_t$.

$$s_t = \left. \frac{b_t}{P_c} \right|_{t \in \mathcal{T}_c} \tag{5.2}$$

The bids in the $(N+1)^{th}$ round by the task agents depend on the supply, demand and prices observed in the $N^{th}$ round. As mentioned earlier, the bidding amount

is capped by the summation of allowance $a_t$ and savings $m_t$ for the task.

$$b_t^{N+1} = \max\left(a_t + m_t,\ \ b_t^N + (d_t - s_t) \times P_c\right)\Big|_{t \in \mathcal{T}_c} \tag{5.3}$$

TABLE 5.1: Task and Core Level Dynamics Example

| Round | $b_{t_a}$ | $b_{t_b}$ | $P_c$ | $s_{t_a}$ | $s_{t_b}$ | $S_c$ |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0.0066 | 150 | 150 | 300 |
| 2 | 1.33 | 0.66 | 0.0066 | 200 | 100 | 300 |

**Running Example** Table 5.1 illustrates the working of the agents of two tasks $t_a$ and $t_b$ executing on core $c$ with supply $S_c = 300$ PUs. The current demands are $d_{t_a} = 200$ PUs and $d_{t_b} = 100$ PUs. Both the task agents begin with the initial bid of \$1. In round 1, the task $t_b$ is over-supplied, while $t_a$ is under-supplied. In round 2, by adjusting bids based on the local supply-demand characteristics, the resources are effectively shared among the tasks according to their requirements.

#### 5.2.2.2 Cluster Dynamics

The cluster agents are responsible for controlling the price and preventing both price inflation and deflation in the cores. When a core is undersupplied (over-supplied), we observe price inflation (deflation). The cluster agent adjusts the supply using DVFS to avoid either over-supply or under-supply situations in the cores, which in turn is reflected in the stable price of the PU.

In our architecture, all the cores within a cluster have to run at the same V-F level. Thus the supply can be modified only at the cluster level and not at the core level. So the cluster agent observes and responds to price inflation (or deflation) of only the constrained core because the constrained core represents the highest demand among the cores within the cluster. Thus, the supply of the cluster is controlled by the most constrained core. Note that given a task mapping, a non-constrained core may suffer from deflation, while the constrained

core suffers from inflation. The cluster agent takes care of the inflation in the constrained core, which can further magnify the deflation in the non-constrained core. The LBT module (discussed in Section 5.2.3) is responsible for fixing the deflation in the non-constrained core through load balancing.

In order to identify inflation/deflation, we need a base price from which the relative changes can be observed. Every time the V-F level changes, we reset the base price to the new price observed in the market. While the V-F level is changing, we do not allow the task agents to change their bids until they have observed the effect of the new supply on their existing bids.

A user supplied parameter called tolerance factor $\delta$ defines the rate of inflation (or deflation) that the cluster agent can tolerate before increasing (or decreasing) the supply, i.e., DVFS by one level. Let $P_c$ and $PBase_c$ represent the current and base price of the resources in a constrained core $c$, respectively. The cluster agent increases the supply when the current price $P_c \geq PBase_c + PBase_c * \delta$. Similarly, a decrease in supply is observed when $P_c \leq PBase_c - PBase_c * \delta$. The tolerance factor $\delta$ determines the response sensitivity of the cluster agents. The lower the value of $\delta$, the faster the response of the cluster agent. The faster response results in frequent V-F level transitions, and hence thermal cycling [96], which can be detrimental to both the performance and the reliability of the hardware. Thus, it is important to carefully select the value of $\delta$ by taking into consideration the underlying hardware.

TABLE 5.2: Cluster Level Dynamics Example

| Round | $b_{t_a}$ | $b_{t_b}$ | $P_c$ | $PBase_c$ | $s_{t_a}$ | $s_{t_b}$ | $S_c$ |
|---|---|---|---|---|---|---|---|
| 3 | 1.99 | 0.66 | 0.0088 | 0.0066 | 225 | 75 | 300 |
| 4 | 1.99 | 0.66 | 0.0066 | 0.0066 | 300 | 100 | 400 |

**Running Example** We demonstrate the cluster level dynamics by extending the example from Table 5.1 to Table 5.2. In round 3, let us assume that the demand of $t_a$ increases from 200 PUs to 300 PUs. Let the tolerance factor $\delta$ be

0.2. In round 3, the price increases to \$0.0088, which is higher than the tolerable value of $\$0.00796 = \$(0.0066 + 0.0066 \times 0.2)$, thus causing inflation in the system. In round 3, the cluster agent responds by increasing the supply $S_c$ from 300 PUs to 400 PUs (highlighted in gray). At the new supply, both the tasks are satisfied and per unit price observed in fourth round is \$0.0066, which is set as new base price of $c$. Also, in round 4, the task agents do not change their bids as the new prices are determined only at the end of the round 4.

#### 5.2.2.3 Chip Dynamics

While the cluster agent attempts to set the V-F level at the minimum value so as to meet the demand of the tasks, the chip level agent is responsible to ensure that the overall chip power does not exceed the TDP budget. The chip agent indirectly controls the power consumption of the chip by manipulating the allowance. It decides on the global allowance value $A$ for the current round. The allowance $A$ is distributed hierarchically throughout the system using the different cluster and core agents.

The global allowance is distributed as cluster allowances ($A_v$) to the cluster agents and the distribution is inversely proportional to power consumption. The cluster consuming more power is given less allowance.

$$A_v = A \cdot \frac{W - W_v}{W} \tag{5.4}$$

The cluster allowance is distributed as core allowance ($A_c$) to the core agents of the cluster based on the priorities of task agents running on them.

$$A_c = A_v \cdot \frac{R_c}{R_v} \tag{5.5}$$

Finally, the core allowance is further distributed as task allowances ($a_t$) to the task agents proportional to their priorities.

$$a_t = A_c \cdot \frac{r_t}{R_c} \tag{5.6}$$

When the chip agent increases the global allowance $A$, the task agents receive additional money to generate higher bids for the resources. The task agents with unsatisfied demands increase their bid with the additional money. This causes inflation in the under supplied clusters, triggering the respective cluster agents to control the inflation by increasing the supply (increase V-F level). This increased supply in the cluster results in increased power consumption.

On the other hand, when the chip agent decreases the global allowance $A$, all the task agents have less money at their disposal and hence are forced to bid lower values. This causes deflation in the clusters, which prompts the cluster agents to decrease the supply (decrease V-F level) to control the deflation, resulting in reduced power consumption.

When the chip agent decides to keep the allowance $A$ constant, all the cores will reach stable equilibrium prices. With stable prices, neither inflation or deflation will be observed by the cluster agents resulting in a *steady-state* with no changes in V-F levels.

The global allowance for the $(N+1)^{th}$ round is set as follows

$$A^{N+1} = A^N + \Delta \tag{5.7}$$

where $A^{N+1}$ and $A^N$ are the current and previous round allowances, respectively and $\Delta$ is the change in the allowance. The key question is how to dynamically set the $\Delta$ value. The $\Delta$ value is set according to the current total power consumption of the chip.

When the chip power consumption $W$ is below the TDP, the primary goal of the chip agent is to meet the demands of the tasks. On the other hand, if the chip power exceeds TDP, then the chip agent is responsible to bring the power below TDP. In case the system has a demand that is unsatisfiable within the TDP, due to the discrete nature of the V-F levels the system will oscillate around the TDP. To stabilize the system near TDP when overloaded, we introduce a buffer zone near TDP where the system is ought to stabilize. The size of the buffer zone is decided by the parameter $W_{th}$. Thus, the spectrum of power consumption is divided into three regions:

**Normal State.** In the normal state, the power consumption of the entire chip is less than the pre-defined threshold $W < W_{th}$. In this state, the chip agent manipulates the $\Delta$ value based on the current total supply $S$ and total demand $D$ of the entire chip. When the demand is not satisfied in at least one of the clusters, the chip is under-utilized and the task agents need extra money to buy more resources. Therefore, the allowance is increased by an amount proportional to the difference between the supply and the demand.

$$\Delta = A^N \cdot \frac{D - S}{D} \tag{5.8}$$

**Threshold State.** In the threshold state, the power consumption of the chip is observed between $W_{th}$ and TDP $W_{tdp}$. Ideally, it is desirable for the power consumption of the chip to stabilize in threshold state when the system is overloaded. The stability is attained in the threshold state by keeping the allowance constant through $\Delta = 0$. With larger buffer zone $(W_{tdp} - W_{th})$, the number of oscillations around the TDP reduces and the stable state is reached quickly, but the chip might be severely under-utilized. On the contrary, a smaller buffer zone leads to frequent oscillations around the TDP, but achieves higher utilization. The idea of stability here is similar to the concept of hysteresis in control systems.

**Emergency State.** In the emergency state, the power consumption of the chip is above $W_{tdp}$ and must be brought down quickly. The allowances of the task agents have to be curbed to reduce the power consumption. In emergency state, the reduction in allowance is proportional to the deviation from the TDP.

$$\Delta = A^N \cdot \frac{W_{tdp} - W}{W_{tdp}} \tag{5.9}$$

Thus our system can achieve stability (stable equilibrium price) in either the normal state (supply meets demand), or the threshold state (when overloaded), but never in the emergency state.

**Savings** An important by-product of our price theory based power management scheme is the concept of allowance savings by the task agents in the form of non-zero $m_t$ values. How does a task end up with savings? The savings are incurred under two scenarios. First we note that the global allowance is increased by $\Delta$ when the demand is not satisfied in at least one of the clusters. Thus, the task agents belonging to the clusters in supply-demand equilibrium would have additional allowance that will be saved for future bidding. Second, the price per PU within a cluster is determined by the most constrained core. This leads to the savings of allowances by the task agents belonging to the non-constrained cores. The saved allowances would facilitate the task agents to outbid other task agents for more resources during the supply constrained situation in both the threshold state and the emergency state. The savings are especially beneficial for tasks with alternating high and low demand requirement. Such task agents save money during their dormant phase and use the saved allowance to outbid other tasks during their active phase.

We choose to cap the savings of a task agent at a fraction of its current allowance. This is because large amount of savings may allow the tasks to keep the system

TABLE 5.3: Chip Level Dynamics Example

| R | $A$ | $a_{t_a}$ | $a_{t_b}$ | $b_{t_a}$ | $b_{t_b}$ | $m_{t_a}$ | $m_{t_b}$ | $P_c$ | $PBase_c$ | $d_{t_a}$ | $d_{t_b}$ | $s_{t_a}$ | $s_{t_b}$ | $S_c$ | D | S | W |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 4 | 4.5 | 3.0 | 1.5 | 1.99 | 0.66 | 1.01 | 0.84 | 0.0066 | 0.0066 | 300 | 100 | 300 | 100 | 400 | 400 | 400 | .8W |
| 5 | 4.5 | 3.0 | 1.5 | 1.99 | 0.66 | 1.01 | 0.84 | 0.0066 | 0.0066 | 300 | 300 | 300 | 100 | 400 | 600 | 400 | .8W |
| | 6.0 | 4.0 | 2.0 | 1.99 | 1.98 | 3.02 | 0.85 | 0.0099 | 0.0066 | 300 | 300 | 200 | 200 | 400 | 600 | 400 | .8W |
| | 6.0 | 4.0 | 2.0 | 1.99 | 1.98 | 3.02 | 0.85 | 0.0099 | 0.0066 | 300 | 300 | 200 | 200 | 500 | 600 | 500 | .8W |
| 6 | 6.0 | 4.0 | 2.0 | 1.99 | 1.98 | 5.03 | 0.86 | 0.0079 | 0.0079 | 300 | 300 | 250 | 250 | 500 | 600 | 500 | 2W |
| 7 | 6.0 | 4.0 | 2.0 | 2.38 | 2.38 | 6.64 | 0.47 | 0.0095 | 0.0079 | 300 | 300 | 250 | 250 | 500 | 600 | 500 | 2W |
| | 6.0 | 4.0 | 2.0 | 2.38 | 2.38 | 6.64 | 0.47 | 0.0095 | 0.0079 | 300 | 300 | 250 | 250 | 600 | 600 | 600 | 2W |
| 8 | 6.0 | 4.0 | 2.0 | 2.38 | 2.38 | 8.25 | 0.10 | 0.0079 | 0.0079 | 300 | 300 | 300 | 300 | 600 | 600 | 600 | 3W |
| 9 | 4.0 | 2.67 | 1.33 | 2.38 | 1.42 | 8.53 | 0 | 0.0063 | 0.0079 | 300 | 300 | 375 | 225 | 600 | 600 | 600 | 3W |
| | 4.0 | 2.67 | 1.33 | 2.38 | 1.42 | 8.53 | 0 | 0.0063 | 0.0079 | 300 | 300 | 375 | 225 | 500 | 600 | 500 | 3W |
| 10 | 4.0 | 2.67 | 1.33 | 2.38 | 1.33 | 8.81 | 0 | 0.0074 | 0.0074 | 300 | 300 | 320 | 180 | 500 | 600 | 500 | 2W |
| 11 | 4.0 | 2.67 | 1.33 | 2.23 | 1.33 | 9.25 | 0 | 0.0071 | 0.0074 | 300 | 300 | 313 | 187 | 500 | 600 | 500 | 2W |
| . | | | | | | | | | | | | | | | | | |
| . | | | | | | | | | | | | | | | | | |
| 16 | 4.0 | 2.67 | 1.33 | 2.01 | 1.33 | 12.26 | 0 | 0.0067 | 0.0074 | 300 | 300 | 300 | 200 | 500 | 600 | 500 | 2W |

in an emergency state longer than permissible. The ideal factor for capping is determined by the designer with knowledge of the underlying hardware.

**Running Example** To illustrate the chip level dynamics, we further extend the example from Table 5.2 to Table 5.3. Let us set $W_{tdp}$ and $W_{th}$ to $2.25W$ and $1.75W$, respectively. Let us further assume that, for the given application, the system reaches the emergency state at 600 PUs supply ($3W$ power) and the threshold state at 500 PUs supply ($2W$).

The global allowance is \$4.5 in the beginning. Let the priorities of tasks $t_a$ and $t_b$ be 2 and 1 respectively. Thus task $t_a$ receives higher allowance relative to task $t_b$ due to the difference in priorities. In table 5.3, the change in values are highlighted in gray.

So far in round 4, we have met supply and demand of both the tasks by increasing the core supply to 400 PUs. Now the demand of task $t_b$ increases to 300 PUs in round 5. As the demand at the core level by the two tasks cannot be met with the current supply of 300 PUs, we observe an increase in allowance (from \$4.5 to \$6.0) as well as price inflation. This price inflation forces the cluster agent to increase the supply to 500 PUs, which brings the chip to threshold state consuming $2W$ power. In the threshold state, the allowance is kept constant

(observed in rounds 6-7). Meanwhile, the savings ($m_{t_a}$ and $m_{t_b}$) are calculated based on the allowance allocated and bids by the tasks in each round.

In round 7, the price inflation again causes the supply to increase to 600 PUs, which pushes the system to the emergency state. Now we need to stabilize the system in the threshold state. This is achieved by decreasing the system-level allowance from \$6.0 to \$4.0. This decreased allowance percolates all the way to the tasks. By round 9, the task $t_b$ has also used up all its savings. This is because the allowance was held constant in the threshold state (rounds 6-7). So with decreased allowance and zero savings, the task $t_b$ is now forced to lower its bids. The lower bids cause price deflation, resulting in reduction of supply from 600 PUs to 500 PUs.

As the supply is brought down to 500 PUs, the system reaches the threshold state again. This time the allowances are constant; hence the tasks cannot increase their bids preventing price inflation and subsequent increased supply (higher frequency) that takes the system back to emergency state. So the system stabilizes (round 16) in the threshold state where the power consumption is close to the TDP and the higher priority task ($t_a$) meets its demand, while the lower priority task ($t_b$) suffers.

#### 5.2.2.4 Stability of the Supply-Demand module

We show that given a fixed task-to-core mapping, the supply-demand module ensures that the system reaches a stable state. By stable state here, we imply that there are no changes in the V-F levels and the resources allocated to the tasks.

The principle of price theory states that the market is only stable at a price equilibrium, which is the price at which the supply is equal to the demand.

Once the supply meets the demand, it automatically prevents further inflation or deflation within the market.

Let us assume that we start off in a stable state. The stability is perturbed as tasks enter/exit the system, or the demand within a task changes due to phase behavior or change in the input conditions. We show that the system will reach a (possibly) different stable state assuming there is no task migration in between.

There are three possible scenarios when the demand changes. In the first scenario, the demand can be satisfied in the normal state. In this case, we observe price inflation/deflation till the supply is equal to the demand and the system reaches stability. We always round up the demand to the next supply value so as to prevent oscillation between two consecutive supply values. Note that the price equilibrium is reached in the constrained core, which determine the V-F level. For the non-constrained cores, the supply might be greater than the demand because all the cores have the same supply value (V-F level). In this case, the price in the core(s) with over-supply will fall till the bid price hits the minimal bid value $b_{min}$. The same situation happens when the demand on the constrained core is less than the minimum supply value (minimum frequency level) and the system stabilizes at the minimum frequency.

In the second scenario, the demand can be satisfied in the threshold state. Here, the allowance is kept constant, which eventually translates to fixed bid prices by the tasks (because bid price cannot exceed allowance), preventing further inflation/deflation. In the absence of inflation/deflation, the V-F levels are not modified and all the clusters in the system stabilize to fixed V-F levels.

The third scenario is the most interesting one where the system needs to be in the emergency state to meet the demand. But we clearly cannot keep the system in the emergency state for long and have to ensure that it quickly stabilizes in the threshold state. This is the scenario illustrated in the example in Table 5.3. The stability is ensured by defining a buffer zone near TDP (the difference between

$W_{tdp}$ and $W_{th}$). The buffer zone should be designed such that the system cannot move from the normal state to the emergency state or vice versa without passing through the threshold state. Once the system reaches the emergency state, the allowances are reduced, which moves the system to the threshold state. Once in threshold state, the allowance is kept constant at reduced value, which leads to fixed bidding prices and hence price equilibrium.

### 5.2.3 Load Balancing and Task migration (LBT) module

The supply-demand module achieves a steady-state with permissible power consumption for any given task mapping by manipulating the V-F levels. But the mapping itself may not be efficient in terms of performance and power leading to a sub-optimal solution. Thus, the goal of the LBT module is to find a task mapping that is superior in terms of both performance and/or power consumption relative to the current mapping. The LBT module first attempts to meet the task demands followed by improving power efficiency through load balancing within a cluster and task migration across the clusters. Load balancing within a cluster helps reduce the V-F level of the cluster, while task migration exploits the heterogeneity of the different clusters to potentially improve both the power and the performance behavior of the tasks. In the following, we first describe our task migration policy across heterogeneous clusters.

Give a fixed task-to-core mapping $\mathcal{M}$, let $s_t^{\mathcal{M}}$ and $b_t^{\mathcal{M}}$ be the steady-state supply and bid corresponding to the task $t$ on core $c_t^{\mathcal{M}}$. Also $d_t^{\mathcal{M}}$ be the demand of task $t$ on core $c_t^{\mathcal{M}}$. Recall that the demand of a task changes based on the core type; the demand is lower on a more powerful core compared to a simpler core for the same level of performance.

We define a metric $perf(\mathcal{M})$ to compare the performance of two different task mappings. Given two different mappings $\mathcal{M}$ and $\mathcal{M}'$, we define $perf(\mathcal{M}') >$

$perf(\mathcal{M})$ if and only if

$$\left(\exists t \in \mathcal{T} : \frac{s_t^{\mathcal{M}'}}{d_t^{\mathcal{M}'}} > \frac{s_t^{\mathcal{M}}}{d_t^{\mathcal{M}}}\right) \text{AND} \left(\forall t' \in \mathcal{T} \text{ s.t. } r_{t'} > r_t : \frac{s_{t'}^{\mathcal{M}'}}{d_{t'}^{\mathcal{M}'}} \geq \frac{s_{t'}^{\mathcal{M}}}{d_{t'}^{\mathcal{M}}}\right)$$

Basically we sort all the tasks $\mathcal{T}$ in the system according to their priority $r_t$. For the mapping $\mathcal{M}'$ to be better than the mapping $\mathcal{M}$ in terms of performance, we need two conditions to be satisfied. The first condition is that there should exist a task $t$ for which the supply-demand ratio in the mapping $\mathcal{M}'$ denoted as $\frac{s_t^{\mathcal{M}'}}{d_t^{\mathcal{M}'}}$ is higher than the ratio in $\mathcal{M}$. The second condition requires all the tasks with higher priority than $t$ to have either better or equal supply-demand ratio in $\mathcal{M}'$ than $\mathcal{M}$.

We also define another metric $spend(\mathcal{M})$ to capture the aggregate spending by the tasks in the steady-state for the mapping $\mathcal{M}$.

$$spend(\mathcal{M}) = \sum_{t \in \mathcal{T}} b_t^{\mathcal{M}}$$

From the power perspective, it is desirable if a task movement brings down the aggregate spending without affecting the performance. The aggregate spending $spend(\mathcal{M})$ reduces only when the steady-state bids from all the tasks combined together is lower. This reduction in bid price is observed only when the steady-state demand lowers due to appropriate load balancing across the cores within a cluster and the migration of the tasks to the most efficient cluster in terms of heterogeneity. The reduced bids cause deflation by lowering the price, which in turn brings down the supply, i.e., V-F levels and hence the power consumption. Therefore, any reduction in aggregate spending will translate to reduction in the power consumption, provided that the performance remains unchanged. Thus mapping $\mathcal{M}'$ is more power efficient than mapping $\mathcal{M}$, that is, $power(\mathcal{M}') <$

$power(\mathcal{M})$ if and only if

$$spend(\mathcal{M}') < spend(\mathcal{M}) \text{ AND } perf(\mathcal{M}') \geq perf(\mathcal{M})$$

In the LBT module, each task agent first estimates steady-state $perf(\mathcal{M})$ and $spend(\mathcal{M})$ for the current mapping $\mathcal{M}$. This is the baseline power-performance behavior that we want to improve upon. Next, each task agent estimates $perf(\mathcal{M}')$ and $spend(\mathcal{M}')$ for all possible mappings $\mathcal{M}'$ where only the task corresponding to this agent migrates to another cluster while the remaining tasks do not move.

Let us consider the potential migration of task $t$ from the cluster $v$ to the cluster $v'$. In order to estimate the performance and the spending of the current and the new mapping, we need the steady-state supply, demand, and bid price for the current cluster $v$ and the target cluster $v'$. This steady-state behavior is estimated as follows.

- *Demand*: In the current cluster $v$, the steady-state demand is assumed to be the currently observed demand. For the target cluster $v'$ (with different core type), the steady-state demand is estimated using off-line profiling, which is explained in detail in Section 5.3. In chapter 3, we developed a power-performance prediction model for single-ISA heterogeneous multi-core systems using a combination of program analysis, mechanistic modeling, and empirical modeling. In future, we plan to include this estimation model within our price theory based power management framework to eliminate the off-line profiling step.

- *Supply*: The steady-state supply of a cluster is estimated to be the same as the steady-state demand, unless the supply is constrained by the TDP constraint. The steady-state supply per task can be estimated by distributing the total supply among the tasks in proportion to their priorities.

- *Bids*: The steady-state bids are calculated by estimating the price in the steady-state. The price at a higher V-F level *Z+1* can be estimated from the price observed in the current V-F level $Z$ using the following equations,

$$P_{Z+1} = P_Z + (P_Z \times \delta) \tag{5.10}$$

  where $\delta$ is the tolerance factor. Using recursion, the price at the steady-state supply can be estimated. For example, let $P_Z$ and $\delta$ be \$10 and 0.02 respectively. Let us also assume that the V-F level has to increase by 3 levels to achieve the steady-state supply. Then, by using Equation 5.10 recursively for each level, the price can estimated to be \$10.612.

Note that the task agents perform performance and savings estimations in parallel, which enables the computational overhead to be distributed across the entire chip, thus ensuring scalability. All the information required for the estimation is hierarchically disseminated from the cluster agents to the chip agents and subsequently to the task agents and is kept consistent with periodic message passing. The overhead of the computation and communication increases with increasing number of tasks, cores, and clusters.

To reduce this overhead, only the task agents in the constrained core of each cluster contemplate movement of the tasks. Furthermore, the overhead is high if the task agents consider migration possibility to all the other cores in the system. We only let the task agents consider the most over-supplied unconstrained core in the target cluster as a potential candidate for migration. Thus there exists a trade-off between the overhead and the quality of the solutions obtained. We demonstrate in Section 5.3 that this simple heuristic works quite well in practice.

Figure 5.2 illustrates the flowchart of the task migration module in the constrained core $c$ of a cluster $v$. If the demands of all the tasks are expected to

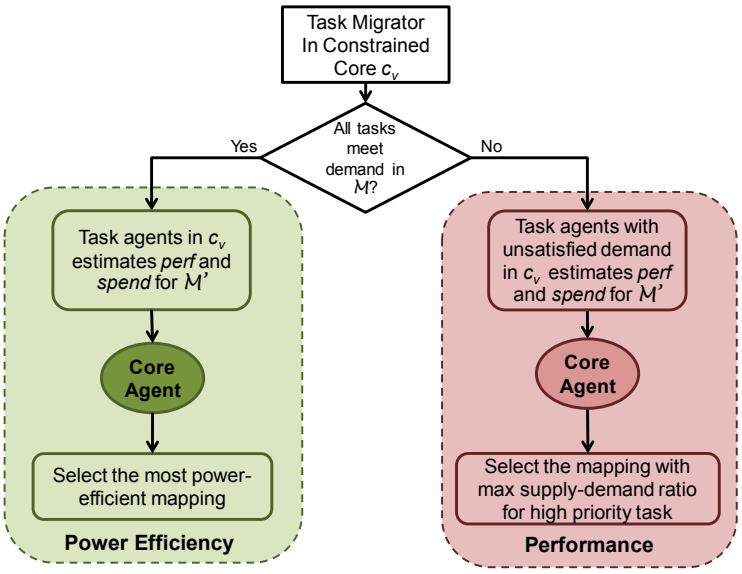



FIGURE 5.2: Task Migration in Constrained Core.

be satisfied in the steady-state of the current mapping $\mathcal{M}$, then the goal is to reduce the power consumption. In this case, all the task agents in $c$ estimate the performance and spending, if the task migrates to the most over-supplied unconstrained core in each of the other clusters, leading to a new mapping. Each task agent then sends the estimated $perf(\mathcal{M}')$ and $spend(\mathcal{M}')$ for the most power-efficient new mapping $\mathcal{M}'$ to the core agent. The core agent selects the most power-efficient mapping from the task agents and forwards this mapping to the chip agent through its cluster agent. The chip agent finally selects the most power-efficient mapping from all the clusters.

On the other hand, if some tasks are not expected to meet the demand in the steady-state in the constrained core, then only the clusters with unsatisfied demand consider possible task migration. For each such cluster, only the task agents with unsatisfied demand in the constrained core contemplate migration to the other clusters. In particular, at all levels (chip level, cluster level, and core level) we choose the highest priority task that improves its supply-demand ratio through migration without impacting the supply-demand ratio of the higher

priority tasks. If two mappings have the same performance, then we select the one with better spending, i.e., better power consumption.

The load balancing process is very similar to the task migration. The only difference is that the target core is not in a different cluster but the most over-supplied unconstrained core within the same cluster. Thus only the cluster agent and the core agent are involved in this process. The chip or the cluster agent approves only one task movement at any given time. The movement of the other tasks, if any, will be performed in future LBT invocation.

#### 5.2.3.1 Stability of the LBT module

We have shown earlier that given a task mapping, the supply-demand module leads the system to a steady-state. Now we need to show that the LBT module does not introduce any instability in the system. The LBT module performs task migration either within the cluster or across clusters. It can introduce instability if and only if there exists cyclic movement of the tasks. However, our heuristic ensures that the number of task migrations is finite for the following reasons.

If the demands of some tasks are not met in the current mapping, then we choose the mapping that improves the supply-demand ratio of the highest priority task with unsatisfied demand through migration. This, by definition, ensures that only the lower-priority tasks are impacted by this migration and the chain of task movements will end with the lowest priority tasks. Thus the number of migrations is bounded by $|\mathcal{T}|$.

On the other hand, if all the tasks meet their demands, then task migration reduces power consumption through reduced spending. This series of task migrations to reduce spending ends when the spending cannot be reduced further.

### 5.2.4 Invocation Frequency

The different modules involved in our framework have to be invoked at different rates to reduce the overhead in the system. As mentioned before, the bidding process by the task agents takes place in a series of rounds. The change in the supply level by the cluster agent occurs asynchronously based on inflation/deflation.

The load balancing within the cluster is invoked more frequently than the task migrator because task migration across clusters is expensive (2-4 ms) compared to migration within cluster (50-170 $\mu$s). The equations below summarize the periods for load balancing and task migration where linux scheduling epoch is 10ms.

$$task\_migration\_period = 2 \times load\_balancing\_period$$

$$load\_balancing\_period = 3 \times bid\_rounds\_period$$

$$bid\_rounds\_period = max\,(linux\_sched\_epoch,\ task\_period)$$

For workloads with periodic tasks, we set the bidding interval as the maximum of the linux scheduling epoch and the shortest period among the tasks. In our experiments, the shortest period for any task is 31.7ms; so we invoke the bidding round every 31.7ms, the load balancer every 95.1ms, and the task migrator every 190.2ms.

The LBT module is disabled in the emergency state as the immediate goal is to bring the power below TDP through the supply-demand module.

## 5.3 Experimental Evaluation

We now proceed to evaluate our price-theory based power management framework, called *PPM*, for asymmetric multi-cores. First, we present the experimental setup and the workload selection. We compare PPM with the hierarchical based power management technique (*HPM*) proposed in chapter 4 and the Linux asymmetry-aware scheduler [8]. Finally, we quantify the effects of savings and priorities followed by the overhead of PPM.

### 5.3.1 Experimental Setup

We use the Versatile Express development platform [7] for our experimental evaluation. The details of our evaluation platform (Versatile Express) is provided in Section 4.3. Section 3.1 summarizes the architectural features of big and LITTLE core. For heterogeneous multi-cores, it is important to discern the migration penalties across different clusters. Section 4.1.3 elaborates the migration costs observed in big.LITTLE.

In our framework, the agents are implemented in software as kernel modules. The core, cluster, and chip agents are instantiated during kernel boot process. The task agents are instantiated as and when the tasks are created. The communication between the tasks in user space and the agents in kernel space is performed using system calls. The cluster agent uses *cpufreq* utility to manipulate the frequency of the cluster. The voltage at each frequency level is automatically set by the hardware. The task migration is handled by the cluster and the chip agents using the task affinity through *sched_setaffinity* interface in the Linux scheduler.

The core agents are responsible for distributing the available resources among the tasks. This is achieved by manipulating the *nice* values of each task. In Linux kernel, *nice* values are the indirect indications of priorities for task management. For example, lower nice value manifests as higher priority and more resource

consumption. As we use *nice* values for resource allocation, the user-level priorities of the tasks in our framework are set in the context of the Linux kernel by adding a new member *prio* in the structure *task_struct*. The *prio* value can be modified from the user space using system calls. For the sake of simplicity, we do not allow dynamic modification of the priorities (*prio*) of the tasks.

### 5.3.2 Workload Selection

TABLE 5.4: Illustration of conversion from heart rate to demand with min and max heart rate being 24 hb/s and 30 hb/s respectively.

| Prog. phase | Current hr (hb/s) | Frequency (Mhz) | Utilization (%) | s (PU) | d (PU) |
|---|---|---|---|---|---|
| 1 | 15 | 500 | 100 | 500 | 900 |
| 2 | 10 | 800 | 50 | 400 | 1080 |
| 3 | 40 | 1000 | 100 | 1000 | 675 |

We use benchmarks from PARSEC [16], Vision [121] and SPEC 2006 [4] suites. At present, our framework requires the tasks to express the performance demand. We employ the *Heart Rate Monitor* (HRM) [55] infrastructure to capture this information. HRM provides a simple and effective way to measure the performance of a task in terms of heartbeats per second (hb/s), which is defined as the throughput of the critical kernel in a task. For example, number of frames processed per second defines the heart rate of a video encoder. For each application, the user can define the performance goal in terms of reference heart rate range and our goal is to maintain the heart rate within that range, while minimizing energy. Note that an application may have highly variable computation requirement due to phase behavior and hence may need different V-F levels or even migration to a different core type to keep the heart rate within the specified range. Table 5.4 illustrates the conversion of heart rate to demand for a particular task. Each row in table 5.4 represents the different program phases of the same application. The user defines the performance goals in terms of minimum and maximum heart rate (24 hb/s – 30 hb/s). From the current observed heart

rate, core frequency and task utilization, the demand of the task can be easily computed. For example, with the supply of 500 PUs, the current heart rate is 15 hb/s in program phase 1. It is clearly well below the reference range prescribed by the user. The required demand is estimated using the following equation,

$$d_t = \frac{target\ heart\ rate \times s_t}{current\ heart\ rate} \tag{5.11}$$

where *target heart rate* is the mean of the minimum and maximum heart rate range. In our example in table 5.4, the *target heart rate* = 27. Similarly, in program phase 3, the current heart rate exceeds the predefined range and here the demand is lowered.

In the absence of HRM infrastructure, an approximate way to determine the demand in Linux operating system is to measure the time a task spends in the run-queue in a given epoch of scheduling. This per-entity load tracking proposed by Paul Turner [117] in kernels higher than 3.7 can be used in lieu of heartbeats. Table 5.5 summarizes the benchmarks along with inputs and the heartbeat insertion point.

TABLE 5.5: Benchmarks description

| **Benchmark** | **Suite** | **Description** | **Inputs** | **Heartbeat Location** |
|---|---|---|---|---|
| swaptions | PARSEC | Monte Carlo (MC) simulation | native and large | every "swaption" |
| bodytrack | PARSEC | Tracks a human body with multiple images. | native and large | every frame |
| x264 | PARSEC | Video encoder. | native | every frame |
| blackscholes | PARSEC | Solves partial differential equation . | native and and large | every 50000 option |
| h264 | SPEC2006 | Video encoder. and bluesky | foreman, soccer | every frame |
| texture | Vision | Motion, tracking and stereo vision. | vga and fullhd | every frame |
| multicnt | Vision | Image Analysis | vga and fullhd | every frame |
| tracking | Vision | Motion, tracking and stereo vision. | vga and fullhd | every frame |

TABLE 5.6: Workload Sets

| | | |
|---|---|---|
| *light* | **l1** | texture_v, tracking_v, h264_s<br>swaptions_l, x264_l, blackscholes_l |
| | **l2** | texture_v, multicnt_v, h264_b<br>swaptions_l, bodytrack_l, blackscholes_l |
| | **l3** | tracking_v, multicnt_v, h264_s<br>x264_l, bodytrack_1, blackscholes_l |
| *medium* | **m1** | swaptions_l, bodytrack_l, blackscholes_l<br>texture_v, tracking_v, h264_b |
| | **m2** | texture_v, tracking_v, h262_s<br>swaptions_n, bodytrack_n, x264_n |
| | **m3** | tracking_v, multicnt_v, blackscholes_n<br>bodytrack_n, texture_f, h264_fo |
| *heavy* | **h1** | h264_fo, x264_n, blackscholes_n<br>texture_f, swaptions_n, multicnt_f |
| | **h2** | blackscholes_n, x264_n, tracking_f<br>bodytrack_n, texture_f, h264_s |
| | **h3** | h264_b, h264_fo, x264_n<br>swaptions_n, bodytrack_n, tracking_f |
| * v-vga, f-fullhd, n-native, l-large, s-soccer, b-bluesky, fo-foreman | | |

We create 9 different multiprogrammed workload sets from the benchmarks based on the following metric,

$$intensity = \frac{\sum_{t \in \mathcal{T}} d_t^{A7} - S_{A7}^{max\_freq}}{S_{A7}^{max\_freq}} \tag{5.12}$$

where $\sum_{t \in \mathcal{T}} d_t^{A7}$ is the total demand of all the tasks in the given workload and $S_{A7}^{max\_freq}$ is the supply at the maximum frequency in the A7 cluster. The metric *intensity* shows whether the demand of the entire task set in a workload can be accommodated in the A7 cluster at the highest frequency. If $intensity \leq 0$, the supply exceeds the demand and hence the demand from all the tasks can be satisfied in A7 cluster at highest frequency. On the other hand, if $intensity > 0$, some tasks will not meet their demand on A7 cluster and need to move to the more powerful A15 cluster. Therefore, based on the *intensity* metric, we classify the workload sets into three types: a) *light* ($metric \leq 0$), b) *medium*

($0 < metric \leq 0.30$) and c) *heavy* ($metric > 0.30$). Table 5.6 summarizes the workload sets and their classification based on the *intensity* value.

The *LBT* module requires the average demand and power consumption of a task in different core types for speculation during load balancing and task migration. We obtain the average demand and power consumption of the tasks in both Cortex-A7 and Cortex-A15 through off-line profiling. The average metrics (demand and power consumption) do not capture the dynamic phases of a task. Nevertheless, it leads to better speculation than the absence of any knowledge whatsoever and the supply-demand module can handle wrong speculations by manipulating the V-F levels. Moreover, as mentioned before, we plan to include the power-performance estimation model for big.LITTLE multi-core within our price theory based power management framework to eliminate the off-line profiling step in the future.

### 5.3.3 Comparative Study

We compare our price theory based power management framework *PPM* with Hierarchical Power Management (*HPM*) technique proposed in chapter 4 and Heterogeneous aware scheduler in Linux kernel (*HL*) [8]. As discussed in chapter 4, the HPM is a control-theory based power management framework that employs multiple PID controllers to meet the demand of tasks in asymmetric multi-cores under TDP constraint. However, the HPM scheduler uses naive load balancing and task migration strategy.

The HL scheduler released by Linaro in Linux kernel release 3.8 is aware of the heterogeneity in ARM big.LITTLE platform. The activeness of a task (the amount of time spent in the active task run-queue) is used as a proxy for migration decisions. For example, the HL scheduler migrates a task to A15 cluster (A7 cluster) once the time spent in the active run-queue exceeds (falls below) certain predefined threshold. Furthermore, the HL scheduler does not react to

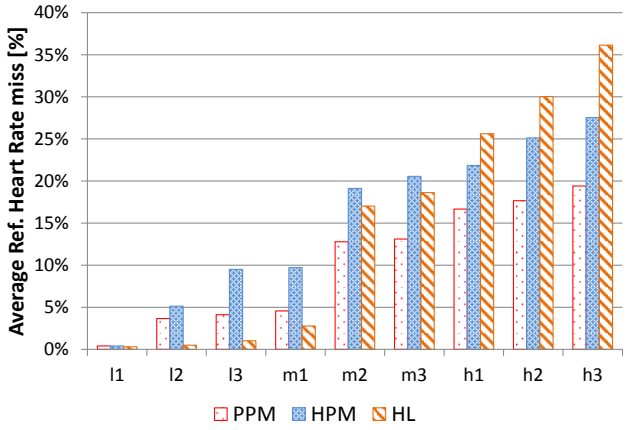



FIGURE 5.3: Comparison of the percentage of time the tasks do not meet the reference heart rate range (no TDP constraint).

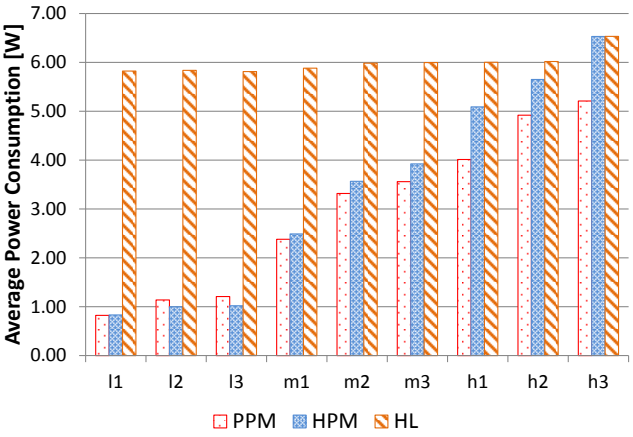



FIGURE 5.4: Comparison of power consumption (no TDP constraint).

the varying demands of the individual tasks. For the HL scheduler, we also employ *cpufreq on-demand* governor that changes the frequency value based on processor utilization. In all the experiments related to the comparative study, we set all the tasks to run at the same priority because HPM and HL do not take the priorities into consideration.

For the first comparative study, we assume that the system does not have any TDP constraint and hence can consume arbitrarily high power. Figure 5.3 plots the percentage of time the reference heart rate range of any task in the workload is not met, that is, the percentage of time the observed heart rate was smaller

than the minimum prescribed heart rate for any of the task in the workload. It is evident that the HL performs better under light workloads (*l1, l2, l3*). This is expected as the HL scheduler migrates the tasks to the powerful A15 cluster at the first opportunity while HPM and PPM both take a more judicious approach. The impact is shown as significantly higher average power consumption for HL compared to HPM and PPM as shown in Figure 5.4.

On the contrary, the PPM scheduler outperforms both HPM and HL for medium (*m1, m2, m3*) and heavy (*h1, h2, h3*) workloads. The HPM scheduler implements a relatively simple and non-speculative load balancer and task migrator that is oblivious to the utilizations in the other clusters. As the HL scheduler migrates all the tasks to the A15 cluster, it results in inefficient usage of the resources for the more demanding workloads.

Figure 5.4 plots the average power consumption for the different techniques with no TDP constraint. HPM and PPM have comparable average power consumption across all types of workloads. The HL scheduler with on-demand governor results in an average power consumption of 5.99W, which is much higher than that of HPM (3.43W) and PPM (2.96W) across all the workloads.

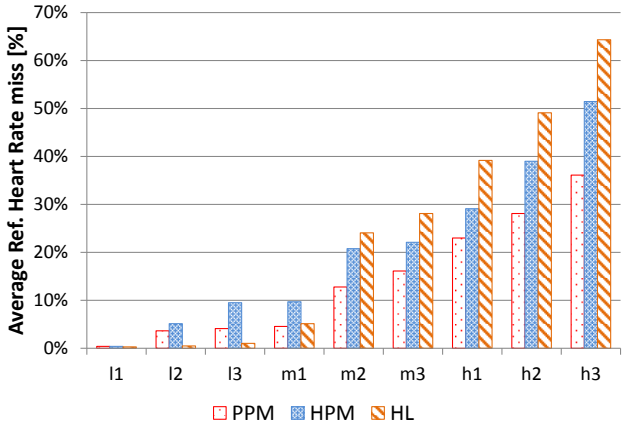
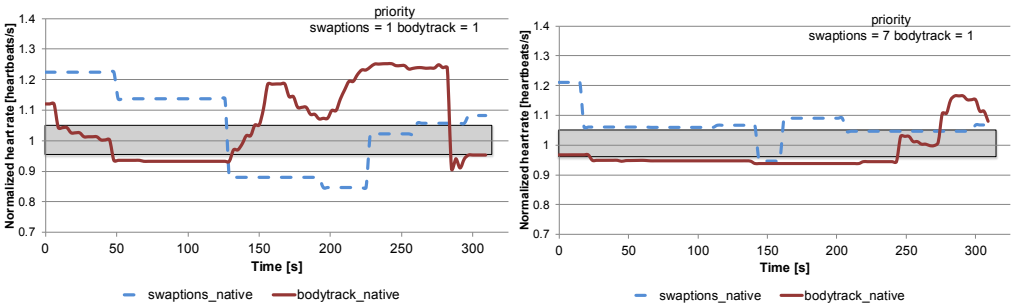



FIGURE 5.5: Comparison of the percentage of time the tasks do not meet the reference heart rate range under TDP constraint of 4W.




(A) (B)

FIGURE 5.6: Normalized performance of swaptions and bodytrack where [0.95,1.05] is the normalized performance goal.

Next we study how the different techniques cope with strict TDP constraints. We observed through a series of experiments that the TDP of our evaluation platform is 8W. To emulate a power-constrained environment, we artificially cap the power budget to 4W. For the HL scheduler, we switch off the A15 cluster once the power exceeds the TDP. This is because the observed maximum power in A7 cluster and A15 cluster are 2W and 6W, respectively. Thus powering down of the A15 cluster guarantees that the total power consumption will be well below the TDP constraint of 4W.

Figure 5.5 plots the percentage of time any task in the workload do not meet their reference heart rate range under TDP constraint of 4W. The tasks are able to meet their reference heart rate more often with PPM approach compared to HPM and HL. The improvements are 34% and 44% compared to HPM and HL, respectively under 4W TDP constraint.

### 5.3.4 Impact of priorities and savings

A unique aspect of our price theory-based solution is that we take into consideration the priorities of the tasks as well as their savings. To evaluate the effectiveness of these concepts, we schedule two demanding tasks on one core.

We disable load balancing and task migration to study the behavior of the tasks with different priorities and phases of execution (where savings become useful).

Figure 5.6a shows the dynamically changing performance in terms of heartbeats per second for *swaptions* and *bodytrack* with the same priority. The black shaded region shows the expected performance range. In this case, *swaptions* and *bodytrack* spend 29.7% and 31.1% of time outside the expected performance range. In Figure 5.6b, we change the priority level of *swaptions* to 7. As can be observed from the figure, *swaptions* is now allocated more resources and hence spends 7.5% of time outside the performance range, while *bodytrack* now spends 57% of time outside the range.

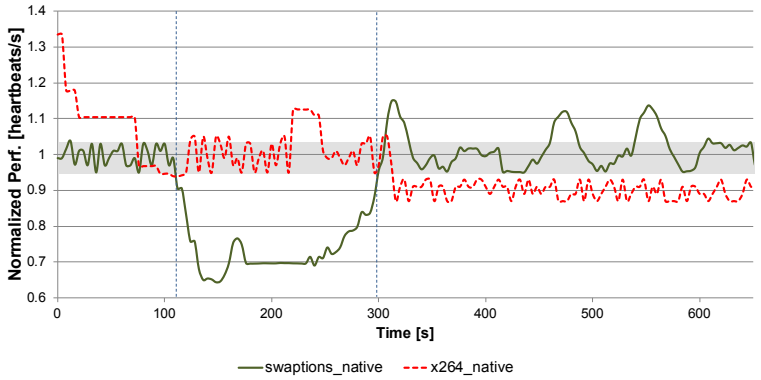



FIGURE 5.7: Normalized performance of swaptions and x264 when [0.95, 1.05] is the normalized performance goal.

To evaluate the advantage of savings, we choose *swaptions* and *x264*. Both the tasks are running at the same priority. In the initial phase (first 100s), *x264* exceeds the performance goals due to relatively less demand (dormant phase), while *swaptions* just about meets its demand. Thus *x264* manages to save a significant fraction of its allowance in this phase.

In the second phase (100 to 300 s), the performance demand of *x264* increases severely as it moves into its active phase. Now *x264* uses up its savings to buy more resources relative to *swaption*. At 300 s, the savings runs out and the high performance demand of *x264* cannot be sustained any further. This illustrate

that the concept of savings offers transient benefits to the tasks that spend long time in the dormant phases with very few active phases.

### 5.3.5 Scalability

Our price theory based power management framework is scalable due to the distributed nature of the agents that work in parallel. We first provide a computational complexity analysis of our framework followed by quantitative results from the implementation.

The computational overhead arises from the calculations performed in the supply-demand module to decide on the bids, price, and supply level. It is evident that these computations are straightforward with negligible overhead. The primary overhead comes from the LBT module that needs to estimate the performance, spending of current and candidate future task mappings.

Let $V$ be the number of clusters on chip, $C$ be the number of cores per cluster, and $T$ be the average number of tasks per core. Let $M$ be the average computation performed to estimate $perf(\mathcal{M}')$ and $spend(\mathcal{M}')$ for each possible task mapping $\mathcal{M}'$. We estimate the cost and benefit of migrating a task to the most over-supplied unconstrained core in each cluster. Thus for each task in the constrained core, we consider $V$ task mappings. So the worst-case computational overhead in the constrained core is $T \times V \times M$. Clearly, the overhead increases with number of tasks and clusters.

Our evaluation platform has two clusters and five cores. To measure the overhead of our approach, we inject all the 8 benchmarks in the system. The supply-demand module is invoked every 31.7ms. As discussed in Section 5.2.4, the load balancing and task migration are invoked every 95.1ms and 190.2ms, respectively. The overhead per invocation of the LBT module in the ARM big.LITTLE platform is only 0.003ms.

In order to quantitatively evaluate the scalability of our approach, we *emulate* systems with large number of cores and clusters. We randomly generate tasks with varying demands and feed this information to the A7 core running at the lowest frequency level (350MHz), which is a highly pessimistic scenario. We assume that this core is the constrained core. We randomly generate the supply/demand information for the other clusters (up to 256) and the cores (up to 16), and provide this information to the constrained core. The supply and demands are randomly chosen between 10–50 PUs, while the maximum supply of the cores in different clusters are between 350–3000 PUs. We then measure the time spent in the supply-demand module and the LBT module by this constrained core when the task migration module is triggered every 190ms (see in Section 5.2.4). Table 5.7 summarizes the overhead in the constrained core for varying number of tasks, cores, and clusters. As we measure the overhead in the small core at the lowest frequency, the overhead will be much smaller on big cores. As recommended for Linux kernel, we also compile the supply-demand module and the LBT module with -O2 compiler optimization flag. For a system with 256 clusters (16 cores per cluster, 32 tasks per core for a total of 131,072 tasks), we observe that the overhead drastically reduces from 11.4ms to 1ms with -O3 optimization flag.

Some existing works [50, 79] provide mixed integer-linear programming (MILP) formulation of the power management problem, which can provide the optimal solution. The MILP-based approach has low overhead for small systems with fewer core types, clusters, and tasks. The complexity of the MILP approach increases exponentially with increasing number of clusters. The average overhead reported in [79] for solving the MILP formulation is 29 minutes, whereas a greedy approximation takes 5.16 minutes for 1000 homogeneous nodes, each node consisting of four cores. All the experiments in [79] were performed on a 3.2GHz dual-processor dual-core platform with 8GB of memory. In [50], the authors report 800ms linear solver time per invocation at 10-minute interval in

TABLE 5.7: Computational overhead for varying number of clusters $V$, cores per cluster $C$, and tasks per core $T$.

| $V$ | $C$ | $T$ | Total Tasks | Avg. overhead [%] | Avg. overhead [ms] |
|---|---|---|---|---|---|
| 4 | 2 | 8 | 64 | 0.02 | 0.038 |
| | | 32 | 256 | 0.11 | 0.21 |
| | 4 | 8 | 128 | 0.03 | 0.057 |
| | | 32 | 512 | 0.16 | 0.30 |
| 16 | 8 | 8 | 1024 | 0.75 | 1.42 |
| | | 32 | 4096 | 0.96 | 1.82 |
| | 16 | 8 | 2048 | 0.81 | 1.54 |
| | | 32 | 8192 | 1.37 | 2.67 |
| 256 | 8 | 8 | 16384 | 3.48 | 6.62 |
| | | 32 | 65536 | 5.12 | 9.74 |
| | 16 | 8 | 32768 | 4.16 | 7.90 |
| | | 32 | 131072 | 6.0 | 11.4 |

a datacenter environment consisting of 160 Xeon nodes or 225 Atom nodes or some combination of them. Clearly, the MILP based approaches are infeasible in a modern embedded platform with dynamically varying workloads requiring frequent invocation of the solver. In contrast, the estimated overhead of our price theory-based approach in a 256 cluster system (16 cores per cluster, 32 tasks per core for a total of 131,072 tasks) is only 11.4ms per invocation at 190ms interval on a Cortex-A7 running at 350 MHz.

## 5.4 Summary

In this chapter, we propose a price theory based power management framework for heterogeneous multi-cores to minimize the power consumption while satisfying the performance goals under a power budget constraint. Our approach, which is highly scalable and distributive, is implemented in real ARM big.LITTLE heterogeneous multi-core platform. We incorporate various power management techniques like DVFS, load balancing and task migrations in a single, unified and comprehensive framework. Our solution is integrated within the Linux fair

scheduler with minimal modifications to the kernel. Empirical results confirm the superiority of our approach compared to the existing techniques.

## 5.5 Future Work

Our price theory based power management framework requires off-line profiling of the application to estimate the average demand and power consumption. This is not a feasible and scalable approach. Therefore, one can include the power-performance estimation model proposed in Chapter 3. The model proposed in Chapter 3 can handle only one core per cluster. On the other hand, the price theory framework can handle any number of cores (and clusters). Therefore, the power-performance model has to be scaled to incorporate it within the price theory framework.

# Chapter 6

# Dynamic Reliability Management

Non-ideal CMOS scaling has resulted in increased power density and on chip temperature, which directly impact the processor lifetime reliability. Extensive studies have demonstrated the detrimental impact of non-ideal scaling on permanent errors caused by wear out phenomena such as electro migration, stress migration, gate oxide breakdown and thermal cycles [110]. As the lifetime degradation at future technology generations is expected to increase, it has become important to design reliability solutions at the architectural level.

Most of the intrinsic failure mechanisms [111] have exponential dependency on the on-chip temperature consumption. In this chapter, temperature reduction acts as an indirect method for reducing power consumption in the microprocessors. The power reduction mechanisms like DVFS, task scheduling, task migration, etc., are still applicable to any reliability management schemes. However, research focussing on lifetime reliability [108, 110, 111] have shown that aggressive power management techniques can induce failures in circuits due to thermal cycling. Thus, it is imperative to design reliability management techniques that

can maximize performance under both lifetime reliability and power/temperature constraints. In this chapter, we propose dynamic reliability management (DRM technique that exploits architectural adaptation in conjunction with dynamic voltage/frequency scaling (DVFS). We employ an online Bayesian classifier that can efficiently detect the reliable configurations, while a performance prediction model selects the one with best performance among all the reliable configurations. We later extend our approach to meet both reliability and thermal (power) constraints.

The goal of any dynamic reliability management (DRM) technique is to improve the lifetime reliability of the microprocessor with minimal impact on performance. Most applications have limited instruction-level parallelism (ILP) and cannot take advantage of extensive ILP exploitation techniques present in current generation out-of-order architectures. When the ILP of an application is limited, various architectural parameters can be scaled down to the appropriate level such that performance remains the same but the power density and consequently the reliability can be improved.

We propose a dynamic reliability management (DRM) technique that adapts at runtime, in addition to the voltage and the frequency, a number of micro-architectural parameters to achieve fine-grained control over reliability with minimal negative impact on performance. The micro-architectural parameter we consider for runtime adaptation are as follows: (a) supply voltage and frequency, (b) fetch gating (fraction of cycles where instruction fetching is disabled), (c) issue width (number of instructions issued per cycle), (d) instruction window size (maximum number of instructions in-flight), and (e) cache way disabling (number of cache ways disabled in the set-associative cache). The detailed rationale behind the choice of these parameters will be presented in Section 6.1.

Figure 6.1 shows the impact of different adaptation mechanisms on both mean time to failure (MTTF) and performance corresponding to the benchmark *bzip2*.

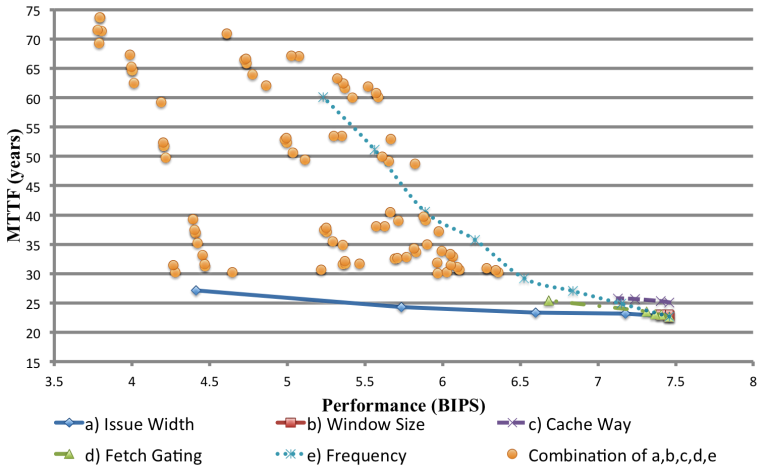



FIGURE 6.1: MTTF vs. Performance for different adaptation mechanisms for the benchmark bzip2

MTTF is defined as the mean time expected until the first failure of the processor. The experimental setup will be described in Section 6.3. We assume the lifetime reliability target (MTTF) as 30 years. The performance is plotted as billion instructions processed per second (BIPS). The baseline contains the highest performing value for each adaptation parameter and hence provides maximum performance with minimum reliability (7.5 BIPS with MTTF equal to 21 years). Then we adapt each parameter individually, while keeping the remaining parameters constant at the highest performing value. Clearly, voltage/frequency scaling has the largest steps and can improve reliability to more than 60 years at significantly lower performance of 5.25 BIPS. The other architectural mechanisms, in contrast, improve reliability moderately with little impact on performance. The figure also shows that adapting a combination of these parameters (the yellow points in Figure 1) can satisfy the reliability target with much better performance compared to DVFS alone. Our objective, thus, is to engage a combination of these parameters so as to reach the reliability target without sacrificing much performance.

It is challenging to design a DRM technique that exploits multiple different architectural mechanisms in conjunction with DVFS. We need to identify, in each

adaptation interval at runtime, the optimal configuration (choice of values for the different parameters) that meets the reliability target with the best performance for an application (or the phase of an application). To filter out unreliable configuration points, we design a software-based Bayesian classifier [126]. In order to identify the optimal configuration among the reliable ones, we develop an analytical model that can predict the performance of a configuration corresponding to the currently executing application.

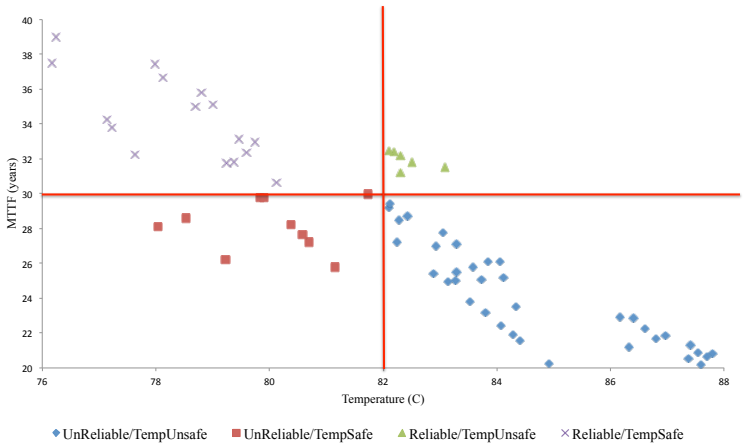



FIGURE 6.2: MTTF vs. temperature for different architectural configurations for the benchmark crafty

Due to the exponential dependency of lifetime reliability on the temperature [109], one can expect the dynamic thermal management solutions (DTM) to be employed for DRM. However, there exist subtle differences between temperature and reliability management goals. The objective of DTM techniques is to optimize performance while keeping peak temperature below certain threshold. On the other hand, DRM maximizes performance while meeting lifetime reliability target. Lifetime reliability is impacted through chip-wide higher temperature rather than just the peak temperature at a particular localized structure. Moreover, certain mechanism like DVFS used for thermal management might have negative impact on reliability [109]. The need for independent but synergistic DTM and DRM techniques is illustrated in Figure 6.2. We plot MTTF versus

peak temperature observed for different micro-architectural configurations corresponding to benchmark *crafty*. We assume peak temperature threshold of 82$^o$C and lifetime reliability target (MTTF) as 30 years. We partition the graph into four regions (clockwise): (1) thermally safe and reliable, (2) thermally unsafe and reliable, (3) thermally unsafe and unreliable, and (4) thermally safe and reliable. It is evident from Figure 6.2 that there exist configurations in all the four partitions. Therefore, it is imperative that we design customized techniques specially targeted at improving reliability.

## 6.1 Parameter Selection

We identify the following parameters as potential adaptation candidates in conjunction with DVFS: (a) fetch gating, (b) issue width, (c) instruction window size, and (d) selective cache way disabling. These parameters are easy to adapt at runtime and also have considerable impact on temperature and lifetime reliability. We choose eight different frequency levels (3.6GHz to 2.5GHz) for DVFS. We use five different fetch gating levels: 0 – 4. When the fetch gating level is set to $T$ ($1 \leq T \leq 4$), the fetch unit disables fetching once every $T$ cycles (0 being the default no fetch gating configuration). We employ five different issue widths: 2–6. When the issue-width is altered, the additional functional units and the appropriate register file ports are disabled so as to reduce leakage power. The instruction window can be scaled to four different sizes: 16, 32, 48, and 64 instructions. The adaptation is achieved by dividing the instruction window into four banks of equal sizes, each containing 16 instructions. Each bank can be enabled or disabled independently [21]. We have to wait for all the instructions from a bank to be committed before it can be disabled. Thus instruction window resizing has more overhead compared to fetch gating and issue width scaling. We assume a 4-way set-associative 64KB L1 data cache. The data cache can be resized through selective cache way disabling [10]. We can thus achieve 16KB to

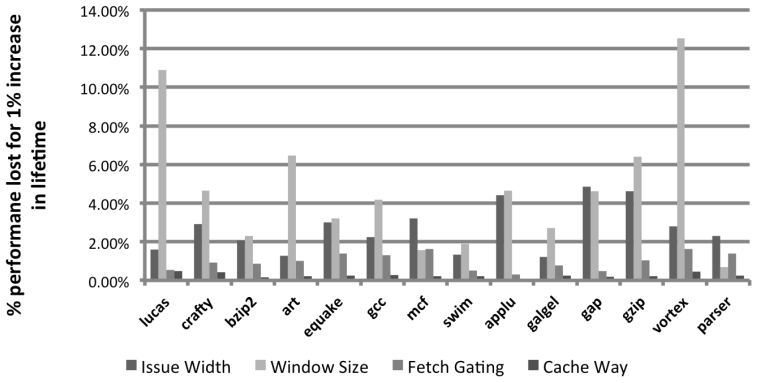



FIGURE 6.3: Performance-reliability tradeoff.

64KB L1 data cache size. To ensure the correctness of the program, the blocks from the disabled cache ways have to be flushed before they are disabled. Note that we only adapt the data cache and not the instruction cache. This is because fetch gating can achieve similar effect as instruction cache resizing. We assume that the architecture has specific instructions that can change the configurational parameters at runtime.

The best adaptation mechanism is the one that can satisfy the reliability or the thermal targets with minimal impact on performance. For each benchmark program from SPEC 2000, we first identify the most compute intensive phase that leads to either increased steady-state temperature or worst reliability under the baseline non-adaptive configuration (see Section 6.3). Next we adapt each parameter individually and quantify its impact on improving the reliability (or reducing the steady-state temperature) of the identified phase.

Figure 6.3 (Figure 6.4) shows the percentage of performance lost in order to increase the lifetime reliability MTTF (reduce the steady-state temperature) by 1% compared to the default configuration. As each parameter has a range of values, we find the mean performance lost for 1% increase in MTTF (or decrease in temperature) compared to the default configuration. It is obvious from the figures that instruction window resizing contributes to serious performance degradation while attempting to improve either reliability or temperature. This

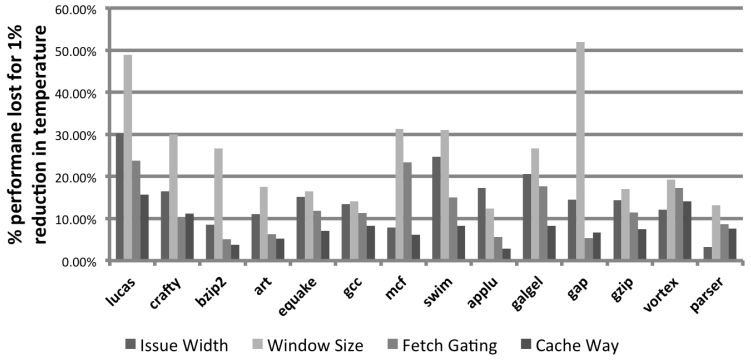



FIGURE 6.4: Performance-temperature tradeoff.

is because, scaling window size has only localized impact on power consumption and considerable performance overhead per transition. Therefore, we decide to eliminate window resizing from further consideration.

In terms of reliability, cache way disabling is the clear winner with minimal impact on performance. There are two reasons behind this behavior. First, for applications with smaller memory footprint, small data cache size suffices and reduces power consumption by disabling the unused cache ways. For applications with larger memory footprint, smaller cache size increases the number of cache misses and thus reduces the switching activity in the core due to the delayed delivery of data from memory. This leads to reduced power density in the back end of the core and hence increased reliability. Even though moderate hardware modifications are required for selective cache way disabling [10], the benefits are considerable from both thermal and reliability management perspective. However, employing only cache way disabling is not sufficient to meet the target lifetime reliability. From Figure 1, it is evident a combination of mechanisms should be employed to achieve the desired MTTF. Thus, our final set of parameters adapted at runtime are issue-width scaling, fetch gating, selective data cache way disabling in conjunction with DVFS.

## 6.2 Dynamic Reliability Management

We now present our dynamic reliability and thermal management framework based on architectural adaptation. The reliability and/or thermal management module is periodically invoked once every adaptation interval: $10^7$ cycles or 2.8ms at 3.6 GHz. As our focus is on temperature induced lifetime reliability issues and the temperature changes occur very slowly, we set the adaptation interval in the order of milliseconds. At every adaptation interval, we first check if there is any significant variation in the workload characteristics, that is, whether there is a new application or the same application has moved into a new phase [31]. If the workload characteristics change, then we may need to adapt the architectural parameters. This is achieved through two major components: (1) the monitoring module, and (2) the configuration search module.

*Monitoring module:* The monitoring module employs a combination of measurements and modeling to estimate the MTTF and the temperature corresponding to the current workload. We assume that the processor is fitted with circuit-level multi-use sensors similar to the ones presented in ElastIC architecture [114]. These sensors can characterize performance, lifetime degradation, temperature, and power consumption at finer granularity. Once the data from the sensors and the physical parameters (such as supply voltage, current, and activity factor) are collected, the MTTF and the temperature are estimated. The MTTF estimation relies on the RAMP [109] model and is entirely implemented in software. The MTTF computation in the RAMP model involves complex operations and frequent exponentiation, which are avoided through pre-computation and fast exponentiation. Thus the overhead is estimating MTTF and temperature is negligible compared to the adaptation interval.

*Configuration Search Module:* The goal of this module is to select the configuration with maximum performance that satisfies the MTTF and/or thermal constraints. The configuration search module is also implemented entirely in

software and consists of two major components: A) naive Bayesian classifier and B) performance prediction module.

### 6.2.1 Naive Bayesian Classifier

Our objective in configuration search is to quickly filter out the unreliable and/or thermally unsafe configurations. We employ a naive Bayesian classifier for this purpose. Classification problems are characterized by the need to classify an input pattern into one of the output categories. Among the various classifiers (naive Bayesian, decision trees and neural network) available, we chose naive Bayesian classifier [126] because its simplicity allows each input pattern to contribute towards the final classification decision. It offers several additional advantages such as fast training time, minimal computation time, and the ability to add new attribute without re-training.

For our configuration filtering problem, we need to select both the workload characteristics and the adaptive architectural configurations as input parameters. Thus each input pattern consists of seven parameters: (a) issue width, (b) fetch gating level, (c) operating frequency, (d) number of integer instructions issued per cycle, (e) number of floating point instructions issued per cycle, (f) number of memory instructions issued per cycle, and (g) number of branch instructions issued per cycle. The output is yes or no classification indicating whether the workload, configuration pair satisfy the reliability and/or the thermal constraint. Note that the inputs to the classifier are the number of instructions issued rather than number of instructions committed because the number of instructions issued influence the temperature and hence the MTTF of the microprocessor, whereas the number of instructions committed determines the performance of the microprocessor. This estimation of number of instructions issued per cycle is obtained through the performance prediction model discussed in Section 6.2.2. Among the architectural parameters, we decide to leave out the

number of cache ways as an input to the classifier. This is because the effect of cache resizing can be captured sufficiently with the performance prediction model.

We train the classifier off-line either during system installation and/or when the system conditions (e.g., ambient temperature) change. The training set is generated by running a set of micro-benchmarks under various configurations and checking if the MTTF and/or thermal constraints are satisfied. The micro-benchmarks contain loops with varying mix of integer, floating point, memory, and branch instructions. The instruction mix are generated in a pseudo random fashion to account for the variability in the workloads that may execute on the processor. To overcome the problem of random sampling, we employ Latin hypercube sampling to enumerate 100 representative configurations from the configuration space. We train the classifier with 50 micro-benchmarks each running on the 100 selected configurations.

After training, we test our classifier using a number of SPEC 2000 benchmarks. We simulate each benchmark at 100 configurations points and determine if the execution violates the MTTF constraint. We compare our simulation outcome with the corresponding output from the classifier. Classification errors can be categorized into false positive and false negative. A classifier commits false positive error when it erroneously classifies a reliable configuration point as unreliable. False negative errors are committed when the classifier erroneously classifies an unreliable configuration point as reliable. We observe that our classifier is very accurate with only 6.4% false negatives and 8.5% false positives, on an average, across all the benchmarks. Note that the only impact of a false positive error is reduced performance as the configuration will not be selected, while false negative errors may violate the reliability constraint. When a selected configuration fails to meet the reliability target during execution, it will be detected in the monitoring module. The module will then invoke emergency

fail safe mechanisms such as clock gating and/or power gating to bring the situation under control. We extend our classifier to incorporate both the MTTF and the temperature constraints to develop our dynamic thermal and reliability management (DTRM) technique. We obtain the training set by running the micro-benchmarks under various configurations and checking if both the MTTF and the temperature constraints are satisfied. We train this combined classifier and observe 7.9% false negative and 9.2% false positive.

### 6.2.2 Performance Prediction Model

The performance prediction model is required for two reasons. First, we need to predict the performance of the reliable and thermally safe configurations for the current workload so as to choose the optimal one. Secondly, the classifier requires the workload characteristics (instruction mix issued per cycle) for a configuration to be classified.

The inputs to the performance prediction model are the number of integer, floating point, memory and branch instructions committed in the previous adaptation interval as well as the total number of instructions committed $N_{useful}$, which could be obtained from hardware performance counters present in modern microprocessors.

Our performance prediction model is inspired by the interval based models proposed in [65, 83]. The interval based model suggests that there exists a sustained background performance level that is punctuated by transient miss-events such as branch mis-prediction and cache misses. The cycles per instruction (CPI) can be expressed as

$$CPI = CPI_{steady} + CPI_{miss} \tag{6.1}$$

$$CPI_{miss} = CPI_{bmiss} + CPI_{icmiss} + CPI_{dcmiss} \tag{6.2}$$

where $CPI_{steady}$ is the sustained background performance in the absence of miss-events and $CPI_{bmiss}, CPI_{icmiss}$ and $CPI_{dcmiss}$ denote the performance loss incurred due to branch mis-predictions, instruction cache miss and data cache miss, respectively. $CPI_{miss}$ can be computed by counting the number of corresponding miss-events and miss penalties as follows,

$$CPI_{miss} = \frac{N_{icmiss} \times P_{icmiss} + N_{bmiss} \times P_{bmiss} + N_{dcmiss} \times P_{dcmiss}}{N_{useful}} \tag{6.3}$$

where $N_{icmiss}, N_{bmiss}$ and $N_{dcmiss}$ are the number of instruction cache miss, data cache miss, and branch mispredictions over our adaptation interval. Data cache miss can be further divided into L1 data cache miss $CPI_{d1cmiss}$ and L2 cache miss $CPI_{d2cmiss}$. The penalty values ($P_{icmiss}, P_{bmiss}$ and $P_{dcmiss}$) are computed using the first order superscalar model [65]. We observe that the performance impact of miss events $CPI_{bmiss}, CPI_{icmiss}$, $CPI_{d2cmiss}$ are fairly constant across configurations except for L1 data cache ($CPI_{d1cmiss}$), which we adapt. This is because changing issue width and fetch gating has minimal impact on the number of miss events.

*Memory Exploration Module:* As we adapt L1 data cache dynamically, $CPI_{d1cmiss}$ varies across the cache configurations. When a new program phase $P^{'}$ is encountered, the memory exploration module is triggered. As there are only four data cache configurations, we execute phase $P'$ with all the four cache configurations, one per adaptation interval. We sample and memoize the L1 data cache miss rate $CPI_{d1cmiss}$ for the various cache configurations. We use this information if phase $P^{'}$ is encountered again.

*Prediction:* For the current configuration $C$,

$$CPI_{steady}(C) = CPI(C) - CPI_{miss}(C) \tag{6.4}$$

$$CPI_{miss}(C) = CPI^{'}_{miss}(C) + CPI_{d1cmiss}(C) \tag{6.5}$$

where $CPI^{'}_{miss}(C)$ represents all the miss events except L1 data cache miss. In general, at a new configuration $C'$

$$CPI^{'}_{miss}(C^{'}) = CPI^{'}_{miss}(C) \tag{6.6}$$

Now the CPI for the configuration $C^{'}$ can be expressed as,

$$CPI(C^{'}) = CPI_{steady}(C^{'}) + CPI^{'}_{miss}(C) + CPI_{d1cmiss}(C^{'}) \tag{6.7}$$

$CPI_{d1cmiss}$ is provided by the memory exploration module.

We now need to determine $CPI_{steady}(C^{'})$. At fetch gating level T, the number of instructions delivered per cycle to the pipeline is $\frac{T}{T+1} \times FW$, where FW is the fetch width. In the steady state, the number of instructions issued per cycle must be the same as the number of instructions fetched per cycle. Thus $IPC_{steady}(C^{'})$ at configuration $C^{'} = \langle IW, FG \rangle$ can be expressed as

$$IPC^{ideal}_{steady}(IW, FG) = min(IW, \frac{T}{T+1} \times FW) \tag{6.8}$$

We refer $IPC_{steady}(C^{'})$ as $IPC^{ideal}_{steady}(C^{'})$ because the equation assumes that all the instructions are of unit latency. To account for variable functional unit latency, we compute a ratio $\mu$ between ideal steady state IPC and observed steady state IPC for the current configuration C.

$$\mu = \frac{IPC_{steady}(C)}{IPC^{ideal}_{steady}(C)} = \frac{1}{CPI_{steady}(C) \times IPC^{ideal}_{steady}(C)} \tag{6.9}$$

As the latency of the functional units are not adapted, the value $\mu$ remains constant across the configuration.

$$IPC_{steady}(C^{'}) = \mu \times IPC^{ideal}_{steady}(C^{'}) \tag{6.10}$$

$$CPI(C^{'}) = \frac{1}{IPC_{steady}(C^{'})} + CPI^{'}_{miss}(C) + CPI_{d1cmiss}(C^{'}) \tag{6.11}$$

Our Bayesian classifier accepts the number of instructions issued corresponding to different classes as input.

$$IPC^{X}_{issue}(C^{'}) = IPC_{issue}(C^{'}) \times \frac{N^{X}_{useful}}{N_{useful}} \tag{6.12}$$

where $IPC^{X}_{issue}(C^{'})$ ($N^{X}_{useful}$) is the number of instructions issued per cycle (total number of instructions committed) of type $X$ (integer, floating point, memory, or branch).

### 6.2.3 Search Space Pruning

Our configuration design space consists of four axes (frequency, issue width, fetch gating and cache ways). From Equation 6.8, it is evident that $IPC_{steady}$ is limited by both issue width and fetch gating. Increasing either of them alone will not facilitate increase in performance. Thus the four dimensional design space can be reduced to three dimensional design space (frequency, cache ways, and $IPC_{steady}$). For each point in this space, the classifier determines whether that particular configuration meets the constraints. The search process can be further optimized by doing linear search for frequency, cache ways and binary search along the $IPC_{steady}$ axis. This optimization is based on the fact that at a particular frequency level $F^{'}$ and steady IPC $IPC^{'}$, if $IPC^{'}$ does not meet the MTTF constraints, then all the configurations with $IPC_{steady}$ values higher than $IPC^{'}$ will not meet the reliability or temperature constraint either. This is because higher performance leads to higher temperature, which has negative impact on the lifetime reliability. For $F$ frequency levels, $I$ levels of $IPC_{steady}$ values and $C$ cache configurations, the maximum number of configurations explored are $O(F \times C \times ln(I))$. In our adaptive framework, we have eight frequency values,

six IPC steady values and four cache ways, resulting in a total of 192 configuration points. The configuration search module takes approximately 10K cycles (2.8 $\mu$s at 3.6 Ghz) to determine the optimal point in the worst case.

## 6.3 Experimental Evaluation

We use SimpleScalar [11] simulator with Wattch [20] power models for our experiments. Our baseline non-adaptive processor is modeled as 6-way issue, 64-entry instruction window, 64 KB L1 data and instruction cache, 2MB L2 unified cache and 4K entry bimod branch predictor. Our adaptive architecture has four possible issue widths (2–6), five possible fetch gating (1–4, no fetch gating), four possible L1 data cache sizes (64KB – 16KB). We vary the processor frequency from 3.6 GHz to 2.5 GHz.

We assume 10 $\mu$s penalty to change the frequency settings [106]. We use Hotspot-5.0 [107] for thermal simulation with a floor plan similar to Alpha 21364. To include the effects of temperature on leakage power, we use the leakage power density value provided in [110]. RAMP [109] is employed to evaluate the lifetime reliability. We set the reliability budget (MTTF) in our experiments as 30 years [109]. For DTRM technique, the maximum temperature allowed is 82$^o$C. We use 14 benchmarks from SPEC 2000.

We compare our architectural adaptation based DRM technique, called **Adaptive**, with the following approaches: (a) **DVFS:** state-of-the-art hardware based DRM technique employing only DVFS. We use a PI-controller based scheme[33], and (b) **Freq-FG:** a technique combining two different mechanisms for reliability management, PI-controlled DVFS and fetch gating [106]. For mild stress, a constant fetch gating level of 3 is engaged. As the stress becomes severe, controller based DVFS is employed.

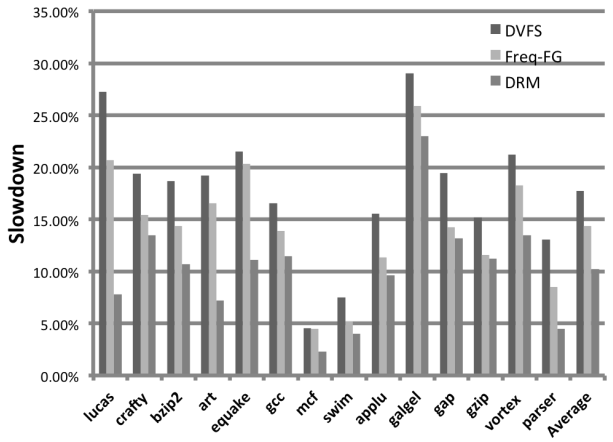



FIGURE 6.5: Comparison of different DRM techniques

Figure 6.5 plots the slowdown in performance compared to the baseline non-adaptive architecture at maximum frequency (without any thermal or reliability constraints). Our adaptive technique outperforms others, i.e., it achieves lower performance degradation. On an average, *Adaptive* has 10.22% slowdown, while *DVFS* and *Freq-FG* have 17.72% and 14.33% slowdown, respectively. Thus *Adaptive* reduces performance degradation by 42.30% compared to *DVFS* and 28.68% compared to *Freq-FG*.

Figure 6.6 plots the time varying trends in IPC, frequency, architectural parameters, and performance (BIPS) for *bzip2*. These plots provide insight into why *Adaptive* performs better. A higher value of an architectural parameter implies better performance. We do not adapt the architectural parameters in *Base* and *DVFS*. *Adaptive* manages to operate at higher frequency (and thus have better performance) because it scales micro-architecture structures to reduce power consumption. We also observe more transitions in frequency in other techniques compared to *Adaptive*, resulting in thermal cycling and consequently worse reliability.

As there is no existing techniques for integrated temperature and reliability management, we compare our DTRM technique with the DRM technique. The set point for DRM technique is only MTTF=30 years but DTRM technique has

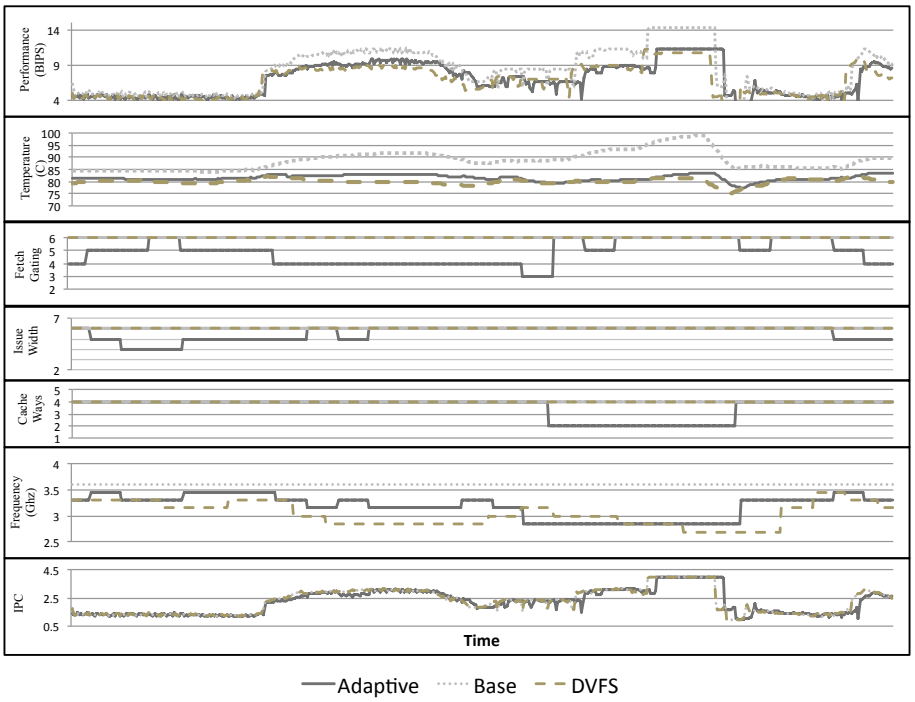



FIGURE 6.6: Time varying trends for bzip2.

the additional set point for temperature ($82^oC$). DTRM technique is an easy extension of our adaptive DRM technique where the classifier is trained to choose the configurations that meet both the temperature and the reliability target. We observed that, the DRM technique, on an average, has to sacrifice 14.47% performance to meet both the temperature and the reliability targets.

## 6.4 Summary

We propose a dynamic reliability management technique that adapts micro-architectural parameters in conjunction with DVFS. Our adaptive method achieves the reliability target while reducing performance overhead by 42.30% compared to DVFS alone and 28.68% compared to DVFS with fetch gating. We also extend our technique to incorporate temperature constraints along with reliability constraints.

# Chapter 7

# Energy-Aware Synthesis of Application Specific MPSoCs

In this chapter, we focus on exploiting functional heterogeneity for minimizing the energy consumption. This is achieved by synthesizing customized MPSoCs to suit the needs of a given multimedia application. Unlike the power management techniques proposed in previous chapters, the technique discussed in this chapter is static in nature. In modern era, MPSoCs have significantly proliferated in portable devices, where these MPSoCs have to satisfy stringent requirements of the target application(s) and/or the target device. Therefore, application specific MPSoCs are deployed in portable devices [41] where an MPSoC is (extremely) customized for a given application under an objective function and various constraints. The benefits of processor customization has been extensively studied in existing literature [52]. An MPSoC can be customized in several different ways for energy reduction. Following are the four design techniques that are currently being widely used:

- Dynamic Voltage and Frequency Scaling (DVFS) allows processors to operate at multiple discrete voltage-frequency (v-f) levels. DVFS is particularly

suitable for multimedia applications where the slack of non-critical tasks is exploited by the use of a lower v-f level to reduce the energy consumption without sacrificing the performance [104].

- Customization of processors aims to match the processing elements of an MPSoC to the computational requirements of the tasks at hand. Processor customization involves addition/removal of functional units, hardware accelerators, custom register files, etc. Custom processors are typically realized through the use of Application Specific Instruction set Processors (ASIPs) [113], where custom instructions are added to access the custom hardware. These custom instructions, when carefully designed, can reduce instruction fetches and register file accesses and improve the energy efficiency of a processor [76]. The designers should take into consideration the increase in on-chip area due to the addition of custom instructions which can increase static power, and hence offset their benefits.

- The cache of a processor contributes significantly to its power consumption [48, 73], in particular static power because it consumes significant amount of on-chip area. Customization of cache according to memory access pattern of a task can significantly reduce energy consumption [125].

- Task mapping allows a designer to map tasks of an application to the processors. Task mapping is done so as to balance the workload across all the processors in an MPSoC, improving their utilization and thus reducing energy consumption of the MPSoC [15].

Given the above design parameters, customization of an MPSoC for a target application becomes an optimization problem where the MPSoC's design space (resulting from the options available for the design parameters) is explored for an optimal solution. While there exist several works in literature that have focused on a subset of the aforementioned design parameters (for example, [113]

considered processor customization and task mapping; [48] considered cache customization), these optimization techniques are designed to work efficiently only with the considered set of design parameters. A mere combination of these individual optimization techniques to cover all the aforementioned design parameters is not the most effective solution. In fact, the authors of [113] illustrate that optimization with simultaneous processor customization and task mapping resulted in 16% better solutions compared to when processor customization and task mapping was performed independently one after the other (in any order). It should be noted that probability of getting suboptimal solutions increases when more design parameters are considered in the design space.

**Motivational Example.** We analyze three typical multimedia applications (JPEG encoder, MP3 encoder and H.264 encoder) to observe the sub-optimality in using independent optimization techniques for DVFS, processor customization, cache customization and task mapping. For each application, we optimized the MPSoC for minimum energy consumption under performance and area constraints, where multiple v-f levels per task, multiple custom instructions per task, multiple cache configurations per processor and general task mapping were used as the design parameters. Further details of the experimental setup are provided in Section 7.3. Figure 7.1 plots the minimum energy design point obtained by

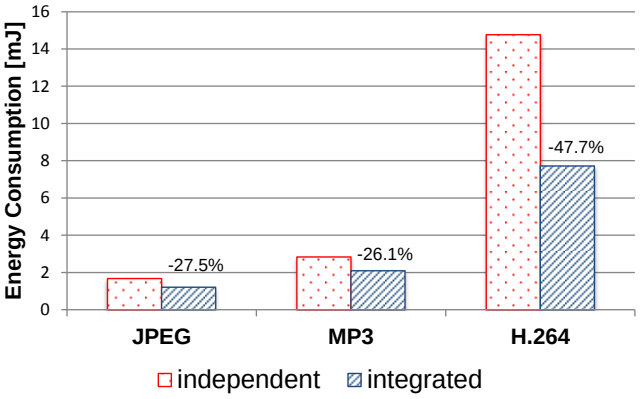



FIGURE 7.1: Comparison of 'independent' and 'integrated' optimization techniques.

the "independent" and "integrated" optimization techniques. In the "independent" technique, an optimal solution is sought for each design parameters one after the other. For example, first an optimal task mapping is selected, then optimal v-f levels for all the tasks are identified given the task mapping, and so on. It is important to note that the sequence of independent optimization techniques affects the optimality of the solution, and thus we exhaustively attempted all possible orders of individual optimizations (for $n$ design parameters, independent optimizations can be performed in $n!$ ways). Therefore, the solution of "independent" optimization technique is the best possible solution from the use of independent individual optimization techniques. The "integrated" optimization technique explores all the design parameters in an integrated and synergistic fashion so as to take into account the complex interplay of DVFS, processor customization, cache customization and task mapping. For example, use of custom instructions for a task modifies its code size and memory access pattern, which in turn affects the customization of the cache for the processor on which this task will be mapped. Thus, the interplay of design parameters must be considered to find a globally optimal solution.

It is evident from Figure 7.1 that a "integrated" optimization technique has a far better potential of reaching the globally optimal solution than the "independent" technique. More importantly, the quality of the solutions from the "independent" technique are significantly inferior even when all the possible ways of combining optimal solutions from individual techniques are exhausted. For example, as shown in Figure 7.1 the amount of energy saved using "integrated" technique is atleast 26.15%. The advantage of synergistic use of DVFS, processor customization, cache customization and task mapping comes at a price. The complexity of the optimization problem, which depends on the number and types of the design parameters, and the number of options considered for those parameters, increases manifold. In fact, the optimization problem with DVFS, processor customization, cache customization and task mapping is an NP-Hard

problem [15]. For a glimpse of the optimization problem's complexity, consider an application with only four tasks, four custom instructions per task, four v-f levels and four cache configurations. Then, the total number of design points is more than a billion. Therefore, a carefully crafted optimization technique that takes into account the interplay of DVFS, processor customization, cache customization and task mapping is required to quickly find globally optimal or near-optimal solutions. In this chapter, We propose a comprehensive framework for exploration of a complex design space consisting of four design parameters: DVFS, processor customization, cache customization and task mapping. As part of the framework, we propose two analytical estimators that use a minimal number of cycle-accurate simulations, and hence speed up design space exploration. Additionally, we propose an optimal algorithm and a heuristic to search the complex, exponential design space for optimal or near-optimal solutions. Finally, we demonstrate the effectiveness of our framework compared to an optimization technique consisting of existing techniques using real multimedia applications.

## 7.1 Problem Formulation

**MPSoC architecure.** We target application specific MPSoCs that consist of customizable processors, which can be realized with the use of ASIPs. As shown in Figure 7.2, each processor has a private cache and local memory, and communicates with other processors via dedicated communication buffers (for example, FIFO queues). Each processor can be customized by both extending its baseline instruction set architecture (with the addition of custom instructions) and customizing its cache (size, line size, etc.). Additionally, each processor can operate at several discrete voltage-frequency (v-f) levels. Thus, the heterogeneity in the MPSoC is manifested in terms of DVFS, processor customization and cache configurations.

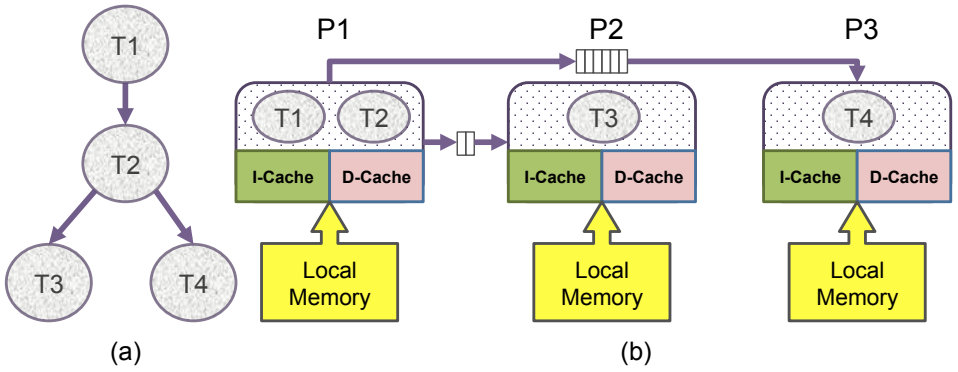



FIGURE 7.2: (a)Task graph (b)MPSoC architecture.

**Application model.** The target application domain comprises of multimedia applications, which contain compute-intensive sub-kernels or tasks that are executed repeatedly. We represent these applications as directed acyclic graphs[1], where vertices represent tasks and edges represent communication between the tasks. The tasks are mapped to the processors, and then buffers are instantiated only between those processors whose mapped tasks need to communicate data. Benoit et al. [15] categorizes mapping of a task graph on an MPSoC with fixed number of processors into: one-to-one mapping, where only a single task is mapped to a processor; interval based mapping, where only adjoining tasks are mapped to a processor; and, general mapping, where no restrictions are placed at all. The type of task mapping determines the placement of the communication buffers between the processors in an MPSoC. We use general mapping because it offers greater flexibility and has the potential to reach a better solution (explained later). Once the tasks are mapped, the MPSoC executes those tasks in the form of a virtual pipeline because multimedia applications inherently benefit from a pipelined execution [102].

Figure 7.3 illustrates mapping of a task graph to a two processor MPSoC using interval and general mappings. In interval mapping, tasks $T1$ and $T2$ are mapped to the first processor while $T3$ is mapped to the second processor. The execution of the processors is similar to a virtual pipeline with two stages. During an

[1]Cyclic graphs are converted to acyclic graphs by graph unfolding [124].

iteration of the pipeline, all the tasks mapped on a processor are executed once. The period of the virtual pipeline is equal to the maximum latency from all of its stages in the steady-state, as marked in Figure 7.3. In general mapping, tasks $T1$ and $T3$ are mapped to the first processor and $T2$ is mapped to the second processor to better balance the workload. In this case, the period is determined by $P2$ which is smaller compared to the period from interval mapping. The price is paid in terms of a longer "initialization" period; however, this is done only once at the start of the application. Note that the "initialization" schedule (for example, execution of $T1$ twice before the execution of $T3$) and "steady-state" schedule (for example, execution of $T3$ followed by $T1$) for any general mapping can be produced using software pipelining [15].

**Problem Statement.** In the MPSoC architecture and application model described above, each processor has a number of cache configurations available for it. Each task can be accelerated with a set of custom instructions, and thus each task has multiple implementations corresponding to different sets of custom instructions that can be used for it. Each set of custom instructions for a task has an additional area cost. Additionally, each task can be executed at one of the available v-f levels. The latency and energy consumption of a task then depends on the cache configuration of the processor on which it is mapped, and the set of custom instructions and v-f level selected for it. The areas of the baseline processor, additional custom instructions (from all the tasks mapped on the processor) and the cache configuration determine the total area of the custom processor. The area of the MPSoC is then the summation of the area of all the processors and the communication buffers. Likewise, energy consumption of the MPSoC is the addition of the energy consumption of the processors (including custom instructions, their caches and local memories) and the communication buffers. Putting it all together, the optimization problem can be formally stated as follows: *Given an application task graph, several discrete v-f levels for each task, different sets of custom instructions for each task, different cache configurations*

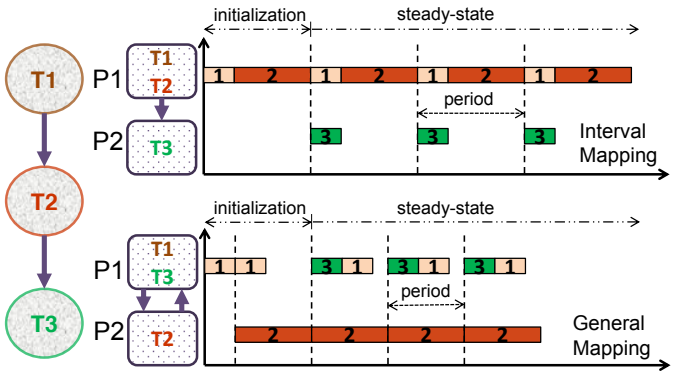



FIGURE 7.3: Different task mappings on an MPSoC.

*for each processor, a steady state period constraint, and an area constraint, the goal is to minimize the total energy consumption of the MPSoC under the provided constraints.* To solve this optimization problem, one needs to search the resulting design space for: (1) the optimal number of processors and mapping of the tasks on them, (2) optimal cache configuration for each of the individual processors, and (3) optimal set of custom instructions and v-f level for each of the tasks. It is important to note that our optimization problem cannot be solved naively because of its exponential complexity that results from all the possible combinations of v-f levels, sets of custom instructions, cache configurations and task mappings.

## 7.2 Proposed Framework

We propose a framework, shown in Figure 7.4, to solve the optimization problem described in the last section. At high level, our framework integrates three components. The profiler component uses a cycle-accurate simulator to produce profiling information for all the application tasks. Next, the profiling information is exploited by the estimation component to estimate the steady-state latency

and energy consumption of the application tasks. Finally, the design space exploration component searches for an optimal or near optimal design point. The following paragraphs explain these components in more detail.

### 7.2.1 Profiler

The input to the profiler consists of the following:

- A multimedia application and its task graph, represented as $\{T_1, T_2, ..., T_N\}$.
- Baseline processor and input data representative of the worst-case.
- $R$ v-f levels for each task, represented as $\{V_1, ..., V_R\}$.
- $O_i$ sets of custom instructions for task $T_i$, represented as $\{CI_{i1}, ..., CI_{iO}\}$. For a task $T_i$, $CI_{i1}$ refers to the use of only baseline processor without any custom instructions (zero additional area).
- $Q$ cache configurations for the processors, represented as $\{C_1, ..., C_Q\}$.

The profiler uses a cycle-accurate simulator to profile all the possible implementations of a task, where an implementation refers to a combination of a set of custom instructions and a cache configuration with the highest v-f level in the

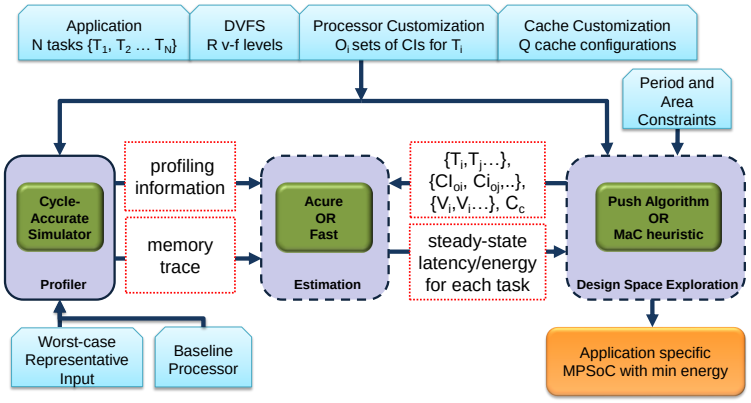



FIGURE 7.4: Framework Overview.

baseline processor. For example, for task $T_i$, $O_i \times Q$ simulations, all at the highest v-f level, are run to capture its latency, power consumption and memory trace for all combinations of sets of custom instructions and cache configurations. During these simulations, input data representative of the worst-case (provided by the designer) is used so that the MPSoC can deliver the required performance at all times when deployed. Estimation of the steady-state latency and power consumption for a task at a v-f level other than the highest level is done in the estimation component of our framework. It is important to note that simulation of all possible task mappings with different v-f levels, sets of custom instructions and cache configurations is not practically feasible due to exponential nature of the design space. Therefore, our profiler uses a minimal number of simulations so as to keep simulation time low while gathering enough information for the estimation component.

### 7.2.2 Latency and Energy Estimation

Two estimators are proposed in this section to estimate steady-state latency and energy consumption of a number of tasks mapped on a baseline processor with their corresponding sets of custom instructions and v-f levels, and a cache configuration.

#### 7.2.2.1 Accurate (Acure) Estimator

**Single task.** For a task with a given set of custom instructions and a cache configuration, we estimate its first iteration's latency ($L_v$) and energy consumption

($E_v$) at a v-f level $v$ using:

$$\begin{aligned} L_v &= \frac{L_h \times F_h}{F_v} \\ P_v &= \frac{P_h \times (V_v)^2 \times F_v}{(V_h)^2 \times F_h} \\ E_v &= P_v \times L_v \end{aligned}$$

where $L_h$ and $P_h$ are the latency and power consumption at the highest v-f level, captured in the profiler component. Given the first iteration's latency and energy consumption at a certain v-f level, the steady-state latency and energy consumption at the same v-f level depends primarily upon the cache configuration. During repeated execution of a task, some of the cache misses from the first iteration may become hits in subsequent iterations due to reuse. We define "local miss" and "global miss" to distinguish between those cache misses. Let $M$ be the sequence of memory requests accessed in an iteration of a task. Let $m$ be a memory request in $M$ and let $sm$ be the cache set that $m$ maps to. If $M$ is simulated in isolation starting with an empty cache and a reference to $m$ results in a cache miss, then $m$ is classified as: a) *Local miss* if there are less than $N$ unique references to cache set $sm$ in $M$ before $m$, where $N$ is the associativity of the cache and b) *Global miss* if $m$ is not a local miss. If $m$ is a global miss or hit, it is not affected by the cache state at the start of an iteration of $M$, that is, it behaves the same way in every iteration. However, if $m$ is a local miss, then it may hit or miss in subsequent iterations depending on the cache state at the start of an iteration of $M$. Intuitively, local misses are the first $n$ cold misses to each cache set that may benefit from reuse later. For an example, assume a direct mapped cache with four sets, $c[0...3]$. Additionally, assume that the memory request pattern of a task is $\{m_0, m_1, m_2, m_3, m_5, m_6, m_7\}$, where $\{m_0\}$ maps to $c[0]$, $\{m_1, m_5\}$ map to $c[1]$, $\{m_2, m_6\}$ map to $c[2]$ and $\{m_3, m_7\}$ map to $c[3]$. Let $LM^1$ be the set representing memory requests that resulted in local misses during the first iteration of the task, and $CS^1$ be the set representing cache state

at the end of the first iteration. Then, $LM^1 = \{m_0, m_1, m_2, m_3\}$ ($\{m_5, m_6, m_7\}$ are global misses) and $CS^1 = \{m_0, m_5, m_6, m_7\}$. In the steady-state, $m_0$ from $LM^1$ will always be a hit while $\{m_1, m_2, m_3\}$ will always result in local misses. Table 7.1 illustrates local misses, global misses and cache states across different iterations for the running example.

| Iter. | Cache State (CS) | Local Misses (LM) | Global Misses |
|---|---|---|---|
| 1 | $\{m_0, m_5, m_6, m_7\}$ | $\{m_0, m_1, m_2, m_3\}$ | $\{m_5, m_6, m_7\}$ |
| 2 | $\{m_0, m_5, m_6, m_7\}$ | $\boldsymbol{\{m_1, m_2, m_3\}}$ | $\{m_5, m_6, m_7\}$ |
| 3 | $\{m_0, m_5, m_6, m_7\}$ | $\boldsymbol{\{m_1, m_2, m_3\}}$ | $\{m_5, m_6, m_7\}$ |

TABLE 7.1: Cache state across iterations of a task.

In summary, the steady-state latency will be less than or equal to the first iteration latency because the number of local misses might reduce. The reduction in the number of local misses is $LM_r = \left|CS^1 \cap LM^1\right|$. The steady-state latency ($L^{ss}$) and energy consumption ($E^{ss}$) is then estimated using the following equations:

$$\begin{aligned} L^{ss} &= L^1 - (LM_r \times ML) \\ E^{ss} &= E^1 - (LM_r \times ME) \end{aligned}$$

where ML and ME refer to lower-level memory latency and energy per access. $L^1$ and $E^1$ refer to the first iteration's latency and energy consumption of a task, including its communication latency and energy respectively. Since the global misses remain constant across iterations and have already been captured in $L^1$ and $E^1$, they do not affect steady-state latency and energy consumption. For estimation, both $CS^1$ and $LM^1$ are computed by processing the memory trace captured in the profiler component. Note that the steady-state latency and energy consumption of a task can also be computed by simulating it for multiple iterations in the profiler. However, for long running tasks, the simulation time

for multiple iterations might be significant. Our estimation technique had errors of less that 1% compared to cycle-accurate simulations of multiple iterations (see Section 7.4), and hence we did not simulate multiple iterations of a task in the profiler.

**Multiple tasks.** Now, we extend our estimation technique for multiple tasks. We assume that all the tasks are non-preemptible which is a valid assumption [103] for multimedia applications because each task has to process its input data before outputting it to the next task. When more than one task is mapped to a processor, then each task can pollute the cache state of other tasks. For the sake of simplicity, we explain our estimation technique with two tasks $T_1$ and $T_2$; however it can easily be extended to any number of tasks. Let $CS_1^1$ and $CS_2^1$ be the cache states at the end of the first iteration of tasks $T_1$ and $T_2$ respectively, and $LM_1^1$ and $LM_2^1$ be the sets containing local misses during the first iteration. In steady state, the number of misses reduces for a particular task when its locally missed memory requests survive through the execution of the other task. For a particular cache set $sm$, we define the operator $\bigodot$ as $m' \bigodot m'' = m''$, if $m''$ is not null or else $m' \bigodot m'' = m'$. This means that the memory request $m''$ (when $m'' \neq null$) has replaced $m'$ in the cache set $sm$. Then, the reduction in the number of local misses for $T_1$ and $T_2$ is:

$$\begin{aligned} LM_{r,T_1} &= \left|(CS_1^1 \bigodot CS_2^1) \cap LM_1^1\right| \\ LM_{r,T_2} &= \left|(CS_2^1 \bigodot CS_1^1) \cap LM_2^1\right| \end{aligned}$$

Therefore, the steady-state latency and energy consumption of the two tasks are:

$$\begin{aligned} L^{ss}_{T_1,T_2} &= L^1_{T_1} + L^1_{T_2} - ((LM_{r,T_1} + LM_{r,T_2}) \times ML) \\ E^{ss}_{T_1,T_2} &= E^1_{T_1} + E^1_{T_2} - ((LM_{r,T_1} + LM_{r,T_2}) \times ME) \end{aligned}$$

If two communicating tasks are mapped to the same processor, then they do not need to communicate through a communication buffer. We capture the amount of data transferred (in words) and the latency per word during the first iteration of a task in the profiler component. Given this information, we estimate the communication latency of

a task by multiplying the latency per word with the amount of data transferred. A similar approach is used for estimation of the communication energy. Once the communication latencies and energies of the two tasks are available, we subtract them from $L^{ss}_{T_1,T_2}$ and $E^{ss}_{T_1,T_2}$ to account for the saving in communication latency and energy from their mapping on the same processor.

It is clear that our estimation technique allows to calculate the steady-state latency and energy consumption of any number and order of tasks from latency, energy consumption and memory trace of first iterations of the individual tasks. Therefore, we do not simulate all the possible mappings of tasks in the profiler component, which reduces simulation time significantly.

#### 7.2.2.2 Fast Estimator

The computational complexity of estimating steady-state latency and energy consumption in *Acure* estimator depends upon the number of tasks and the size of their memory traces. When the number of complex tasks mapped on a processor increases, *Acure* estimator might become slow for rapid design space exploration. Therefore, in *Fast* estimator, we trade-off the time spent in processing of memory traces (to compute $LM^1_i$ and $CS^1_i$ for a task $T_i$) with the estimation accuracy.

**Single task.** Like *Acure* estimator, first of all, the latency and energy consumption of first iteration is estimated at the given v-f level. Afterwards, rather than analyzing the memory trace, we use the first iteration's latency and energy consumption as the steady-state latency and energy consumption of a task.

**Multiple tasks.** The steady-state latency and energy consumption of two tasks, $T_1$ and $T_2$, is computed by adding the steady-state latency and energy consumption of the individual tasks. The communication latency and energy are accounted for in a similar fashion to the *Acure* estimator.

Note that the accuracy of *Fast* estimator depends upon the cache behavior. If the reduction in local misses across different iterations of a single task or across multiple tasks is significant, then the error in estimation will be high.

**Algorithm 1:** Push Algorithm

```
1  tasks = {T_1, T_2...T_N};
2  eProcs = {} ;                                          // existing processors
3  map[] = {} ;                                  // map[P] contains tasks mapped on P
4  currMetrics = {currA = 0, currP = 0, currE = 0};
5  bestSol = {};
6  PUSH(tasks, eProcs, map, A_c, P_c)
7    if areaPruning(tasks, A_c) then return;
8    if periodPruning(tasks, P_c) then return;
9    if energyPruning(tasks, bestSol) then return;
10   if tasks ≠ null then
11       T_i ← task i from tasks;
12       // map to an existing processor
13       for each P in eProcs do
14           for o = 1 to O_i do // custom instructions
15               for v = 1 to R do // v-f levels
16                   map[P] ← T_i with CI_io and V_v;
17                   currMetrics = metrics(eProcs, map);
18                   if currP ≤ P_c and currA ≤ A_c then
19                       PUSH(tasks, eProcs, map, A_c, P_c);
20                   else
21                       restore currMetrics previous value;
22                       remove T_i from map[P];
23       // map to a new processor
24       for c = 1 to Q do // cache configurations
25           for o = 1 to O_i do // custom instructions
26               for v = 1 to R do // v-f levels
27                   nP = new processor with C_c;
28                   eProcs ← nP;
29                   map[nP] ← T_i with CI_io and V_v;
30                   currMetrics = metrics(eProcs, map);
31                   if currP ≤ P_c and currA ≤ A_c then
32                       PUSH(tasks, eProcs, map, A_c, P_c);
33                   else
34                       restore currMetrics previous value;
35                       remove T_i from map[P];
36                       remove nP from eProcs;
37   if tasks ≠ null then
38       return failure;
39   else
40       update bestSol if required; return;
```

### 7.2.3 Design Space Exploration

#### 7.2.3.1 Prune and Search (Push) Algorithm

The *Push* algorithm uses two basic operations "prune" and "search" to quickly push itself through the complex design space towards the optimal design point. The "prune" operation prunes certain parts of the design space based on constraints, while the "search" operation finds a partial solution in a subset of the design space. These partial solutions are combined successively to reach the globally optimal design point. Theoretically the worst-case complexity of the Push algorithm is exponential because it searches for the optimal design point (based on the branch and bound algorithm); practically, it is able to prune a large part of the complex design space by exploiting the constraints.

Algorithm 1 shows the pseudo code of the *Push* algorithm. For ease of understanding, consider that the design space is represented as a tree, which is shown in Figure 7.5. The parameters of the design space are summarized in the table. For the sake of simplicity, we do not show all the nodes in the design space tree. Note that $L_{1111}$ represents the latency of the task $T_1$ with custom instruction set $CI_1$, v-f level $V_1$ and cache configuration $C_1$ (a similar notation is used for energy as well). The annotations on edges illustrate the options of the design parameters. Each level $i$ of the tree corresponds to a call of the *Push* procedure, where the algorithm has a partial solution for tasks $T_1, T_2, ..., T_{i-1}$ (their corresponding sets of custom instructions and v-f levels, stored in *map[]*), and the processors (with their corresponding cache configurations, stored in *eProcs*) that have already been mapped with those tasks. Let period, area and energy consumption of the partial solution be *currP*, *currA* and *currE* respectively (stored in *currMetrics*). With this partial solution at level $i$, the algorithm prunes the subtrees based on constraints (lines 7-9) which are explained later. Note that the *areaPruning*, *periodPruning* and *energyPruning* functions return true when the subtrees are pruned. If the pruning is unsuccessful, then the algorithm maps task $T_i$ either to one of the existing processors (lines 13-22) or a new processor (lines 24-36) ensuring the area and period constraints are met, and then moves on to the next task by calling the Push procedure. Here, the algorithm uses the *metrics* function (lines 17, 30) to calculate the area, period and energy consumption of the new mapping using either the *Acure* or *Fast* estimator from Section 7.2.2. This process is repeated until all the tasks have been mapped or no more

| T | CI | V | C | CI Area | L | E |
|---|---|---|---|---|---|---|
| T1 | 1 | 1 | 1 | $A_{11}$=0 | $L_{1111}$=2 | $E_{1111}$=2 |
| | 2 | | 2 | $A_{12}$=5 | $L_{1212}$=1 | $E_{1212}$=1 |
| T2 | 1 | 1 | 1 | $A_{21}$=0 | $L_{2111}$=3 | $E_{2111}$=4 |
| | | | 2 | | $L_{2112}$=4 | $E_{2112}$=5 |

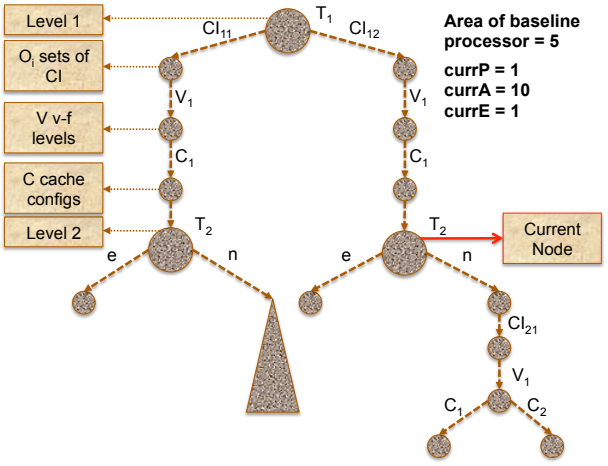



FIGURE 7.5: Illustration of Push algorithm.

tasks can be mapped given the area and period constraints (lines 37-38). Mapping of all the tasks means a new solution is found, which is used to update the best solution seen so far (stored in *bestSol*, line 40) if the new solution's energy consumption is better than the best solution.

For example, at level 1 in Figure 7.5, the current node indicates that $CI_{12}$ set of custom instructions and $V_1$ v-f level have been selected for task $T_1$ which is mapped to a processor with $C_1$ cache configuration. The algorithm reaches the current node only after traversing the entire left subtree for the task $T_1$. The metrics for partial solution at level 1 are in *currP*, *currA* and *currE*, while *bestE* is the energy consumption of the best solution seen so far. From the current node, task $T_2$ can be mapped either to the existing processor (left subtree, edge annotated as e) or a new processor (right subtree, edge annotated as n). The algorithm can prune the subtrees based on the following observations:

**Area constraint.** If all the remaining tasks are mapped to existing processors without any custom instructions (use of baseline processor only), then the total area will still be equal to *currA* because no additional area will be used for the unmapped tasks. If

$currA$ violates the area constraint $A_c$, then it is safe to prune the subtree. In the running example, if $A_c = 9$, then the entire subtree will be pruned (because $currA = 10$).

**Period constraint.** A lower bound on the period can be estimated by mapping remaining tasks to separate processors and using their lowest latency implementations (i.e $L_i^{min}$), that is, $max(currP, L_i^{min}, ..., L_N^{min})$). In the running example, $currP = 1$ and $L_2^{min} = L_{2111} = 3$, so lower bound on period equals max(1, 3) = 3. If $P_c = 2$, then the entire subtree will be pruned.

**Lowest possible energy consumption (LPE).** For all the unmapped tasks, $LPE$ is estimated as the summation of their minimum energy consumptions, less their communication energies. That is, $LPE$ refers to the scenario where minimum energy implementations of all the tasks are used with no energy spent in data communication. If $(LPE + currE)$ is greater than the energy consumption of the best solution seen so far, then the entire subtree can be pruned as the partial solution is already worse than the best solution. In the running example, if the best solution's energy is 4, then the entire subtree will be pruned because $LPE + currE = 5$.

#### 7.2.3.2 Map and Customize (MaC) Heuristic

To better handle the exponential complexity of the design space, we propose a two stage algorithm consisting of the "map" stage and "customize" stage. In the "map" stage, candidate task mappings are produced considering a homogeneous MPSoC. In the "customize" stage, already produced task mappings are used to customize the MPSoC with the selection of custom instructions, v-f levels and cache configurations. One can think of the "map" stage as application-level balancer and the "customize" stage as system-level balancer, which work in synergy to find a near-optimal solution.

**Map stage.** In this stage, a homogeneous MPSoC with variable number of processors is considered. The input to this stage consists of $tasks = \{T_1, T_2...T_N\}$, and their code sizes and latencies on a baseline processor with smallest cache configuration, lowest v-f level and without any custom instructions. The goal is to generate a set of task mappings that will possibly lead to a globally optimal solution.

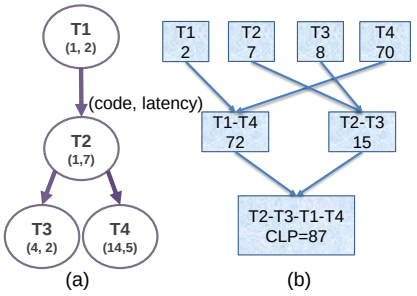
Ideally, we would like to combine tasks that will complement each other in terms of both the latency and the energy consumption. Let us categorize tasks based upon their latencies as *short* and *long* tasks. Likewise, we categorize tasks based upon their code size as *small* and *big* tasks. Our intuition is that a *small* task well complements a *big* task in terms of the cache configuration, while a *short* task well complements a *long* task in terms of latency. Thus, we propose to combine *small-short* tasks with *big-long*, and *small-long* with *big-short* tasks, because they will result in a complementary effect in their combined latency and energy consumption. If $code_i$ and $L_i$ is the code size and latency of a task $T_i$ respectively, then we define code-latency product for a set of tasks $\{T_i, T_j, ...\}$ as

$$CLP(i, j, ...) \quad = \quad \sum_{x=\{i,j,...\}} code_x \times L_x$$

If the tasks are sorted in ascending order according to $CLP$ metric, then the *smallest-shortest* task will set the lower bound while the *biggest-longest* task will set the upper bound. All the other tasks will be dispersed in between these bounds. We use the $CLP$ metric to obtain a task sequence $\Gamma(N)$ where tasks with complementary characteristics are adjacent to each other. Given $N$ tasks, we consider them as $N$ subsequences and compute the $CLP$ metric for each of them, followed by sorting them in ascending order according to $CLP$. Then, we combine the $i$-th subsequence with $(N + 1 - i)$-th subsequence, that is, combining subsequences with complementary characteristics. After the first run, a total of $\lceil \frac{N}{2} \rceil$ subsequences are obtained. We repeat the above process till only one subsequence is left, which is the final sequence of tasks with complementary characteristics. An example of task sequencing for four tasks is shown in Figure 7.6. The annotation in each node of the task graph on the left-hand side is $(code, latency)$, while the right-hand side illustrates the number of task subsequences and their corresponding $CLP$ metrics for each run. The final sequence $\Gamma(N) = \{T_2, T_3, T_1, T_4\}$ where $\{T_2, T_3\}$ subsequence represents a *small-long* and *big-short* combination, while $\{T_1, T_4\}$ represents a *small-short* and *big-long* combination.

After obtaining the task sequence using the $CLP$ metric, we proceed to enumerate different mappings of the task sequence considering variable number of processors as follows:

$$map_i = \forall T_j \mapsto \{P_1, P_2, ..., P_i\} : 1 \leq j \leq N, \quad i \leq N$$




| Procs | Period | Mappings |
|---|---|---|
| 1 | 16 | $\{T_2,T_3,T_1,T_4\}$ |
| 2 | 9 | $\{T_3,T_1,T_4\}\{T_2\}$ |
| 3 | 7 | $\{T_2\}\{T_3\}\{T_1,T_4\}$<br>$\{T_2\}\{T_1,T_3\}\{T_4\}$<br>$\{T_2\}\{T_3\}\{T_1,T_4\}$ |
| 4 | 7 | $\{T_2\}\{T_3\}\{T_1\}\{T_4\}$ |

(c)

FIGURE 7.6: Illustration of map stage: (a)Task graph (b)Task sequencing (c)Different task mappings.

In essence, $map_i$ represents mapping of all the tasks onto $i$ number of processors. We model the enumeration of task mappings as a chains-on-chains problem [91], where the aim is to map $j$ tasks on $i$ processor such that the mapping is load balanced, that is, the period is minimized. Our intuition is that a balanced mapping at this stage will possibly lead to a better customization in the later stage. Although there exist several polynomial-time algorithms for solving the chains-on-chains problem [91], we use a dynamic programming based solution from [15]. Figure 7.6(c) illustrates different task mappings for the final task sequence of Figure 7.6(b) with their optimal periods.

Task Mapping: $\{T_2\}\{T_3\}\{T_1,T_4\}$ $A_c = 20$ $P_c = 5$

| Run | P1 (period = 7) | | P2 (period = 2) | | P3 (period = 7) | | sArea (20) |
|---|---|---|---|---|---|---|---|
| | A[P1] | PushM | A[P2] | PushM | A[P3] | PushM | |
| 1 | 8.75 | -- | 2.5 | 1.5 | 8.75 | -- | 18.5 |
| 2 | 9.25 | -- | -- | 1.5 | 9.25 | 9.0 | 9.5 |
| 3 | 9.5 | 9.4 | -- | 1.5 | -- | 9.0 | 0.1 |

FIGURE 7.7: Illustration of customize stage.

**Customize stage.** The algorithm for customization of the MPSoC for different task mappings is shown in Algorithm 2. For a task mapping (stored in $map_i$), some area from the total available area (stored in *sArea*) is allocated to each processor (stored in *A[P]*) proportional to its period (lines 9-10). This is based on the intuition that a processor with higher period may have to use complex custom instructions and bigger cache configuration to reduce its period. Given the allocated area for a processor, we employ a modified version of the *Push* algorithm, *PushM* (line 12), to find the optimal set of custom instructions and v-f levels for all the tasks of the given processor, and its optimal

cache configuration. The *PushM* algorithm uses lines 13–22 of the *Push* algorithm, that is, an optimal solution is searched for the given (existing) processor only, ignoring the addition of new processors. The custom processor returned by the *PushM* algorithm is added to the best solution for the current task mapping (stored in *bestSol[i]*), while the area of the custom processor is subtracted from the total available area (line 16). This process is repeated until all the processors have satisfied the period constraint (line 11-16) or all the processors currently in *vProcs* could not satisfy the period constraint (lines 17-18). Finally, the algorithm returns the task mapping and customized MPSoC with minimum energy consumption from all of the input task mappings (line 22).

A working example of the algorithm is shown in Figure 7.7 for one of the task mappings from Figure 7.6. The first column reports the run of the algorithm while the rest of the columns report the area allocated to each processor and the total available area. For example, in the first run, $P1$ and $P2$ are allocated an area of 8.75 and 2.5 respectively from total available area of 20. During the first run, the *PushM* algorithm succeeds for $P2$ and fails for $P1$ and $P3$. Thus, the area of the custom processor for $P2$ (1.5) is subtracted from the total available area, which is redistributed among $P1$ (9.25) and $P3$ (9.25) for the next runs. In the second and third runs, the *PushM* algorithm successfully customizes $P3$ and $P1$ under the allocated area and period constraint.

## 7.3 Experimental Methodology

We used a commercial environment from Tensilica [57] to realize application specific MPSoCs. We used Xtensa LX2 processors and accompanying toolset RD-2011.2 which includes Xtensa ISS cycle-accurate simulator, XTMP multiprocessor simulation environment, and XPRES compiler. For each application task, we used XPRES compiler to generate different sets of custom instructions, which consist of any combination of FLIX instructions, fused, vector and specialized operations. At least, five sets of custom instructions were generated per application task. We used five different instruction cache configurations by changing cache sizes from 1 KB to 16 KB. Although we only tested our framework with instruction cache configurations, a designer can easily apply it to data cache configurations. For each processor, we used five different frequency levels ranging from 533 MHz to 1.5 Ghz with their corresponding voltage levels. The Xt-Xenergy tool

**Algorithm 2:** Customize MPSoC

```
1  maps = {map_1, map_2, ..., map_N};
2  bestSol[] = {} ;                                   // bestSol[i] for map_i
3  while maps ≠ {} do
4      vProcs[] ← all the processors from map_i;
5      tasks[] ← tasks mapped to processors in vProcs;
6      sArea = A_c;
7      while vProcs ≠ {} do
8          // allocate area proportional to period
9          for each P in vProcs do
10             A[P] ← proportion of sArea using P's period;
11         for each P in vProcs do
12             r = PushM(tasks[P], vProcs[P], {}, A[P], P_c);
13             if r ≠ failure then
14                 update bestSol[i][P] ;                 // solution for P
15                 remove P from vProcs;
16                 sArea −= area returned by PushM();
17         if all P in vProcs failed then
18             break;
19     if vProcs ≠ {} then
20         bestSol[i] ← failure;
21     remove map_i from maps;
22 return minimum energy solution from bestSol;
```

from Tensilica is used to compute the energy consumption of a processor at the highest v-f level, including its caches and local memory for a given 90nm technology. The area of the processor and its caches and local memories is also obtained from Tensilica toolset. For communication buffers, we estimated their area and energy consumption using CACTI [40].

For evaluation, we used both real multimedia applications and synthetic applications. We partitioned the multimedia applications into their tasks as mentioned in [25] for JPEG encoder (5 tasks) and MP3 encoder (5 tasks), and in [61] for H.264 encoder (7 tasks). For synthetic applications, we generated task graphs using TGFF [32], and used kernels from Mibench [51] and StreamIt [116] as tasks in those task graphs. We chose ten kernels from Mibench and StreamIt where a reasonable trade-off in performance and area was observed for different custom instructions and cache configurations. We created three synthetic applications: SA1 with 10 tasks, SA2 with 15 tasks and SA3 with 20 tasks to evaluate the scalability of the proposed algorithm and heuristic. Given

the above setup, the design space of each application contained *at least a billion design points*. All the experiments were conducted on an Intel Xeon 2.53 Ghz processor with 16 GB memory.

## 7.4 Results

Table 7.2 summarizes the error observed in computation of the steady-state latency and energy using the *Acure* and *Fast* estimators, compared to cycle-accurate simulation. The table reports the *maximum error* observed, from amongst all the applications, when different number of tasks are mapped on a processor with different sets of custom instructions and cache configurations. The errors observed in *Acure* estimator are very low and remain fairly constant across the number of tasks. On the other hand, the errors in the *Fast* estimator increase with the number of tasks, reaching to a maximum of 15.71%. This is because the cache behavior is significantly disrupted when a greater number of tasks are mapped to the same processor. Thus, the *Acure* estimator will better guide the design space exploration algorithms than the *Fast* estimator. In our framework,

| Mapped Tasks | Latency Error [%] | | Energy Error [%] | |
|---|---|---|---|---|
| | Acure | Fast | Acure | Fast |
| 1 | 0.54 | 1.72 | 0.69 | 2.92 |
| 2 | 0.61 | 1.99 | 0.82 | 3.10 |
| 4 | 0.75 | 4.82 | 0.97 | 6.65 |
| 8 | 0.86 | 8.31 | 1.04 | 9.98 |
| 16 | 0.91 | 11.20 | 1.22 | 13.52 |
| 20 | 1.07 | 13.64 | 1.29 | 15.71 |

TABLE 7.2: Maximum error in the Acure and Fast estimators.

the *Acure* and *Fast* estimators can either be combined with the *Push* algorithm or *MaC* heuristic, which results in four possible optimization techniques. Additionally, we also use the "independent" optimization technique, where optimal solutions are sought for in individual optimization problems, and all the possible ways of combining individual optimal solutions are exhausted. Since the "independent" technique can be constructed from existing techniques, we use it as the state-of-the-art for comparison purposes. Note that the use of *Acure* estimator with the *Push* algorithm will yield the most optimal solution from amongst all the five optimization techniques. Since our design spaces

contain at least billion points, it is not practical to apply all the possible period and area constraints. We used Latin Hypercube Sampling to generate 50 uniformly distributed tuples for each application, where each tuple represents a combination of area and period constraints.

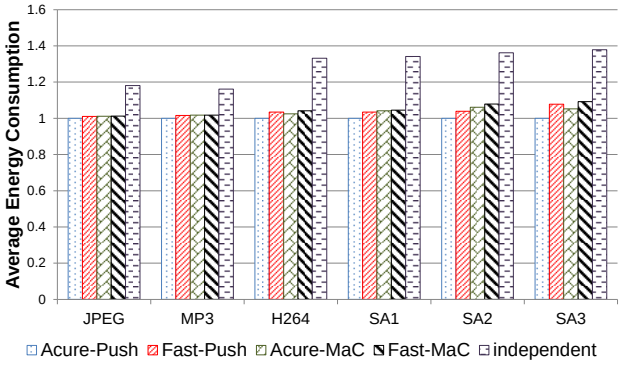



FIGURE 7.8: Comparison of different optimization techniques, normalized to Acure-Push.

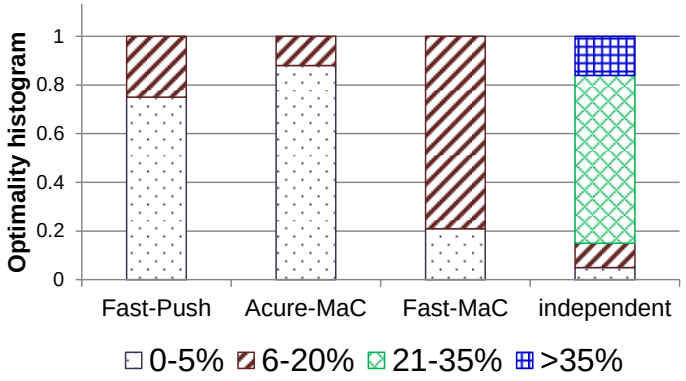



FIGURE 7.9: Error distribution in different optimization techniques for SA3 application.

Figure 7.8 plots the average energy consumptions of the solutions obtained from various optimization techniques, normalized to the energy consumption of the solution from *Acure-Push* technique under 50 different constraints. From amongst all the applications, on average, *Fast-Push*, *Acure-MaC* and *Fast-MaC* found 8%, 6% and 9% sub-optimal solutions respectively. It is noteworthy that the *Acure-MaC* outperformed *Fast-Push* for H.264 and SA3 applications even though the *Push* algorithm is optimal. This is due to higher estimation errors in the *Fast* estimator compared to the *Acure* estimator for H.264 and SA3 applications, which misguided the *Push* algorithm. Out of all the five optimization techniques, the "independent" technique performs the worst; on average, it

resulted in up to 37.9% sub-optimal solutions. More importantly, our most sub-optimal heuristic, *Fast-MaC*, improved the optimality of the solutions by up to 76.25% on average when compared to the "independent" technique. The maximum error in solutions from various optimization techniques yielded similar findings. The "independent" technique resulted in solutions that were up to 57.1% sub-optimal compared to a maximum of 16%, 13% and 19% sub-optimal solutions from *Fast-Push*, *Acure-MaC* and *Fast-MaC* respectively. To better highlight the sub-optimality of the optimization techniques, Figure 7.9 plots the distribution of energy difference between the solutions obtained from those technqiues with the one obtained from Acure-Push for SA3 application. It is evident that 85% of the solutions from the "independent" technique have more than 20% increase in energy with respect to Acure-Push, while all the solutions from Fast-Push, Acure-MaC and Fast-MaC have less than 20% increase in energy. This signifies the fact that a carefully crafted optimization technique which explores all the design pa rameters in a synergistic and integrated fashion is required for effective exploration of large and complex design spaces.

| Apps. | # Tasks | Acure Push | Fast Push | Acure MaC | Fast MaC | Ind. |
|---|---|---|---|---|---|---|
| JPEG | 5 | 910 | 770 | 4 | 3 | 75 |
| MP3 | 5 | 1181 | 890 | 12 | 9 | 75 |
| H.264 | 7 | 1492 | 1028 | 84 | 17 | 492 |
| SA1 | 10 | 3662 | 2621 | 299 | 87 | 991 |
| SA2 | 15 | 5021 | 3267 | 450 | 230 | 1982 |
| SA3 | 20 | 7732 | 4919 | 785 | 540 | 3811 |

TABLE 7.3: Exploration time (in secs) of optimization techniques.

Table 7.3 reports the exploration time of all the five optimization techniques in seconds. For each application, these exploration times are calculated by taking an average of the total time spent in finding solutions for all the 50 constraints. It is evident that the optimization techniques with the *Fast* estimator are faster than the *Acure* estimator. More importantly, the *MaC* heuristic is significantly, at least 9 times (SA3 application), faster than the *Push* algorithm. It is also noteworthy that the "independent" technique's exploration time is at least 7 times more than our fastest heuristic *Fast-MaC* even with 76.25% sub-optimal solutions. Thus, our *MaC* heuristic based optimization techniques provide superior solution with lower exploration times than the "independent" technique.

## 7.5 Summary

In this chapter, we have proposed a framework to synthesize an energy-aware application specific MPSoC. We synergistically explore the complex interplay of DVFS, custom instructions, cache configurations and task mapping. Our framework uses two analytical estimators, the *Push* algorithm for optimal solutions and *MaC* heuristic for near-optimal solutions. The experimental results show that the *MaC* heuristic is at least 14 times faster than the *Push* algorithm with average errors of up to 9%. Also, our *MaC* heuristic reduces energy consumption by up to 76.25% on average with 7 times lower exploration time compared to the "independent" optimization technique.

# Chapter 8

# Conclusions

This chapter concludes the thesis by summarizing the major contributions. This thesis focuses on developing efficient power management schemes for heterogeneous multi-cores. Most of the previous works focussed on developing power management mechanisms for homogeneous multi-cores. Our contributions differ by addressing the challenges introduced by heterogeneity that are not present in homogeneous multi-cores.

Our first major contribution is in terms of dynamic power management techniques for heterogeneous multi-cores. Among the dynamic techniques, we first present accurate models that can predict the power-performance behavior across different core types in heterogeneous systems. Second, we discuss a control theory based power management scheme that incorporates multiple PID controllers. Then, we propose price theory based power management scheme with desirable features like scalability, priority-driven and priority consciousness. All the dynamic power management schemes are evaluated in a real heterogeneous platform – ARM big.LITTLE. Lastly, we propose a dynamic reliability management technique with thermal/power constraints and performance as an optimization goals. Our second major contribution is in terms of design of heterogeneous application-specific MPSoC with minimal energy consumption as a first class design constraint.

# Chapter 9

# Future Work

In this chapter, we summarize the future work that can be applied to the contributions in this thesis in the chronological order. Emerging applications like computer vision, data mining, search, media processing, etc., have a unique property of tolerating erroneous solutions for a decreased computation time. For example, video encoders can discard computations to return lossy encoded video frames that can still be tolerated by the users. We want to explore these kinds of applications that allow approximate computing to improve the energy efficiency in heterogeneous multi-cores. In [105], the authors have introduced a concept of approximate computing, where multiple iterations are skipped to tradeoff performance and accuracy of the results. We believe that incorporating approximate computing in our dynamic power management techniques (*HPM* and *PPM*) will be a notable contribution.

Most of the modern mobile devices can operate at different voltage-frequency levels to extend their battery life. In [42], the authors proposed a power-performance model for a superscalar out-of-order processor with DVFS capability. Therefore, we hope to build a similar model that not only estimates performance/power across different core types but at different voltage-frequency operating points.

In this thesis, we have focused on single-threaded serial applications on multi-cores. As more cores will be integrated in future many core architecture, it is important that any power management technique should be able to handle multi-threaded applications. The major challenge in multi-threaded application is thread clustering based on the

amount of data sharing. Therefore, we hope that this thesis will inspire the future research in developing power management techniques for multi-threaded applications on heterogeneous multi-cores.

# Bibliography


[1] ARM infocenter. http://infocenter.arm.com/.

[2] GCC Processor pipeline description, http://gcc.gnu.org/onlinedocs/gccint/processor-pipeline-description.html.

[3] SPEC CPU Benchmarks. http://www.spec.org/benchmarks.html.

[4] SPEC CPU Benchmarks. http://www.spec.org/benchmarks.html.

[5] Nvidia. the benefits of multiple cpu cores in mobile devices, 2010. http://www.nvidia.com/content/PDF/tegra_white_papers/Benefits-of-Multi-core-CPUs-in-Mobile-Devices_Ver1.2.pdf.

[6] The death of microprocessors, 2010. http://www.embedded.com/design/embedded/4025001/The-death-of-microprocessors.

[7] ARM Ltd., 2011. http://www.arm.com/products/tools/development-boards/versatile-express/index.php.

[8] Linaro Ubuntu release for Vexpress, November 2012. http://releases.linaro.org/12.10/ubuntu/vexpress/.

[9] Mohammad Abdullah Al Faruque, Janmartin Jahn, Thomas Ebi, and Jörg Henkel. Runtime thermal management using software agents for multi-and many-core architectures. *Design & Test of Computers, IEEE*, 27(6):58–68, 2010.

[10] David H Albonesi. Selective cache ways: On-demand cache resource allocation. In *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on*, pages 248–259. IEEE, 1999.

[11] Todd Austin, Eric Larson, and Dan Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002.

[12] Saisanthosh Balakrishnan, Ravi Rajwar, Mike Upton, and Konrad Lai. The impact of performance asymmetry in emerging multicore architectures. In *ACM SIGARCH Computer Architecture News*, volume 33, pages 506–517. IEEE Computer Society, 2005.

[13] Michela Becchi and Patrick Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Computing frontiers*, pages 29–40. ACM, 2006.

[14] Luca Benini, Davide Bertozzi, Alessio Guerri, and Michela Milano. Allocation, scheduling and voltage scaling on energy aware MPSoCs. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 44–58. Springer, 2006.

[15] Anne Benoit and Yves Robert. Mapping pipeline skeletons onto heterogeneous platforms. *Journal of Parallel and Distributed Computing*, 68(6):790–808, 2008.

[16] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.

[17] Partha Biswas, Sudarshan Banerjee, Nikil Dutt, Paolo Ienne, and Laura Pozzi. Performance and energy benefits of instruction set extensions in an FPGA soft core. In *VLSI Design, 2006. Held jointly with 5th International Conference on Embedded Systems and Design., 19th International Conference on*, pages 6–pp. IEEE, 2006.

[18] Paolo Bonzini, Dilek Harmanci, and Laura Pozzi. A study of energy saving in customizable processors. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 304–312. Springer, 2007.

[19] M. Breughe, S. Eyerman, and L. Eeckhout. A mechanistic performance model for superscalar in-order processors. In *ISPASS*, pages 14–24, 2012.

[20] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. *ACM SIGARCH Computer Architecture News*, 28(2):83–94, 2000.

[21] Alper Buyuktosunoglu, David Albonesi, Stanley Schuster, David Brooks, Pradip Bose, and Peter Cook. A circuit level implementation of an adaptive issue queue for power-aware microprocessors. In *Proceedings of the 11th Great Lakes symposium on VLSI*, pages 73–78. ACM, 2001.

[22] Jeffrey S Chase, Darrell C Anderson, Prachi N Thakar, Amin M Vahdat, and Ronald P Doyle. Managing energy and server resources in hosting centers. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 103–116. ACM, 2001.

[23] Gang Chen, Kai Huang, Christian Buckl, and Alois Knoll. Energy optimization with worst-case deadline guarantee for pipelined multiprocessor systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 45–50. EDA Consortium, 2013.

[24] Jian Chen and Lizy K John. Efficient program scheduling for heterogeneous multi-core processors. In *Proceedings of the 46th Annual Design Automation Conference*, pages 927–930. ACM, 2009.

[25] Liang Chen, Nicolas Boichat, and Tulika Mitra. Customized MPSoC synthesis for task sequence. In *Application Specific Processors (SASP), 2011 IEEE 9th Symposium on*, pages 16–21. IEEE, 2011.

[26] Ryan Cochran, Can Hankendi, Ayse K Coskun, and Sherief Reda. Pack & Cap: adaptive DVFS and thread packing under power caps. In *Proceedings of the 44th annual IEEE/ACM international symposium on microarchitecture*, pages 175–185. ACM, 2011.

[27] Jason Cong and Bo Yuan. Energy-efficient scheduling on heterogeneous multi-core architectures. In *Low Power Electronics and Design*, pages 345–350. ACM, 2012.

[28] NVidia Corporation. Bring high-end graphics to handheld devices., 2011. http://www.nvidia.com/content/PDF/tegra_white_papers/Bringing_High-End_Graphics_to_Handheld_Devices.pdf.

[29] Samsung Corporation. Samsung Exynos., 2011. http://www.samsung.com/exynos/.

[30] Robert H Dennard, Fritz H Gaensslen, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9(5):256–268, 1974.

[31] Ashutosh S Dhodapkar and James E Smith. Managing multi-configuration hardware via dynamic working set analysis. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, pages 233–244. IEEE, 2002.

[32] Robert P Dick, David L Rhodes, and Wayne Wolf. TGFF: task graphs for free. In *Proceedings of the 6th international workshop on Hardware/software codesign*, pages 97–101. IEEE Computer Society, 1998.

[33] James Donald and Margaret Martonosi. Techniques for multicore thermal management: Classification and new exploration. volume 34, pages 78–88. ACM, 2006.

[34] Thomas Ebi, M Faruque, and Jörg Henkel. Tape: Thermal-aware agent-based power econom multi/many-core architectures. In *Computer-Aided Design-Digest of Technical Papers, 2009. ICCAD 2009. IEEE/ACM International Conference on*, pages 302–309. IEEE, 2009.

[35] Thomas Ebi, Janmartin Jahn, and Jörg Henkel. Agent-based thermal management for multi-core architectures. In *Organic Computing—A Paradigm Shift for Complex Systems*, pages 587–588. Springer, 2011.

[36] Thomas Ebi, David Kramer, Wolfgang Karl, and Jörg Henkel. Economic learning for thermal-aware power budgeting in many-core architectures. In *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2011 Proceedings of the 9th International Conference on*, pages 189–196. IEEE, 2011.

[37] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376. IEEE, 2011.

[38] Alimonda et al. A feedback-based approach to DVFS in data-flow applications. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 2009.

[39] Carta et al. A control theoretic approach to energy-efficient pipelined computation in MPSoCs. *ACM Trans. Embedded Comput. Syst.*, 2007.

[40] Tarjan et al. CACTI 4.0. *HP laboratories, Technical report*, 2006.

[41] Wolf et al. Multiprocessor system-on-chip (MPSoC) technology. *IEEE TCAD*, 2008.

[42] Stijn Eyerman and Lieven Eeckhout. A counter architecture for online dvfs profitability estimation. *Computers, IEEE Transactions on*, 59(11):1576–1583, 2010.

[43] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E Smith. A mechanistic performance model for superscalar out-of-order processors. *TOCS*, 27(2):3, 2009.

[44] Stijn Eyerman, Kenneth Hoste, and Lieven Eeckhout. Mechanistic-empirical processor performance modeling for constructing CPI stacks on real hardware. ISPASS, pages 216–226, 2011.

[45] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. Maestro: Orchestrating lifetime reliability in chip multiprocessors. In *High Performance Embedded Architectures and Compilers*, pages 186–200. Springer, 2010.

[46] Milton Friedman. Quantity theory of money. *J. Eatwell et al*, pages 1–40, 1989.

[47] Yang Ge, Qinru Qiu, and Qing Wu. A multi-agent framework for thermal aware task migration in many-core systems. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 20(10):1758–1771, 2012.

[48] Ann Gordon-Ross, Frank Vahid, and Nikil Dutt. Automatic tuning of two-level caches to embedded applications. In *Proceedings of the conference on Design, automation and test in Europe-Volume 1*, page 10208. IEEE Computer Society, 2004.

[49] Peter Greenhalgh. Big. LITTLE processing with ARM Cortex-a15 & Cortex-a7: Improving energy efficiency in high-performance mobile platforms. *white paper, ARM September*, 2011.

[50] Marisabel Guevara, Benjamin Lubin, and Benjamin C Lee. Navigating heterogeneous processors with market mechanisms. In *HPCA*, pages 95–106, 2013.

[51] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14. IEEE, 2001.

[52] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding sources of inefficiency in general-purpose chips. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 37–47. ACM, 2010.

[53] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. Simpoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism*, 7(4):1–28, 2005.

[54] H. Hoffmann, J. Eastep, M.D. Santambrogio, J.E. Miller, and A. Agarwal. Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments. In *Proceedings of the 7th international conference on Autonomic computing*, pages 79–88. ACM, 2010.

[55] Henry Hoffmann, Jonathan Eastep, Marco D Santambrogio, Jason E Miller, and Anant Agarwal. Application heartbeats for software performance and health. In *ACM Sigplan Notices*, volume 45, pages 347–348. ACM, 2010.

[56] Qualcomm Inc. Qualcomm Snapdragon Processor., 2011. http://www.qualcomm.com/chipsets/snapdragon.

[57] Tensilica Inc. Diamond Standard 108Mini Controller: A Small, Low-Power, Cacheless RISC CPU, 2010. http://www.tensilica.com/uploads/pdf/108Mini.pdf.

[58] C. Isci, A. Buyuktosunoglu, C.Y. Cher, P. Bose, and M. Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pages 347–358. IEEE, 2006.

[59] Haris Javaid, Xin He, Aleksandar Ignjatovic, and Sri Parameswaran. Optimal synthesis of latency and throughput constrained pipelined MPSoCs targeting streaming applications. In *Hardware/Software Codesign and System Synthesis (CODES+*

*ISSS), 2010 IEEE/ACM/IFIP International Conference on*, pages 75–84. IEEE, 2010.

[60] Haris Javaid, Aleksander Ignjatovic, and Sri Parameswaran. Rapid design space exploration of application specific heterogeneous pipelined multiprocessor systems. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 29(11):1777–1789, 2010.

[61] Haris Javaid, Muhammad Shafique, Sri Parameswaran, and Jörg Henkel. Low-power adaptive pipelined MPSoCs for multimedia: an H.264 video encoder case study. In *Proceedings of the 48th Design Automation Conference*, pages 1032–1037. ACM, 2011.

[62] PJ Joseph, Kapil Vaswani, and Matthew J Thazhuthaveetil. A predictive performance model for superscalar processors. In *International Symposium on Microarchitecture*, pages 161–170, 2006.

[63] Seungrok Jung, Jungsoo Kim, Sangkwon Na, and Chong-Min Kyung. Energy-aware instruction-set customization for real-time embedded multiprocessor systems. In *Proceedings of the 14th ACM/IEEE international symposium on Low power electronics and design*, pages 335–338. ACM, 2009.

[64] Tejas S Karkhanis and James E Smith. A first-order superscalar processor model. In *Computer Architecture*, pages 338–349, 2004.

[65] Tejas S Karkhanis and James E Smith. A first-order superscalar processor model. In *ACM SIGARCH Computer Architecture News*, volume 32, page 338. IEEE Computer Society, 2004.

[66] Tejas S Karkhanis and James E Smith. Automated design of application specific superscalar processors: an analytical approach. In *SIGARCH*, volume 35, pages 402–411, 2007.

[67] Eric Karl, David Blaauw, Dennis Sylvester, and Trevor Mudge. Reliability modeling and management in dynamic microprocessor-based systems. In *Proceedings of the 43rd annual Design Automation Conference*, pages 1057–1060. ACM, 2006.

[68] D. Koufaty, D. Reddy, and S. Hahn. Bias scheduling in heterogeneous multi-core architectures. In *Proceedings of the 5th European conference on Computer systems*, pages 125–138. ACM, 2010.

[69] David Koufaty, Dheeraj Reddy, and Scott Hahn. Bias scheduling in heterogeneous multi-core architectures. In *Computer systems*, pages 125–138. ACM, 2010.

[70] Rakesh Kumar, Keith I Farkas, Norman P Jouppi, Parthasarathy Ranganathan, and Dean M Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *MICRO*, pages 81–92, 2003.

[71] Steven Landsburg. *Price theory and applications*. South-Western Pub, 2010.

[72] Benjamin C Lee and David M Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *SIGOPS Operating Systems Review*, volume 40, pages 185–194, 2006.

[73] Benjamin C Lee and David M Brooks. Illustrative design space studies with microarchitectural regression models. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 340–351. IEEE, 2007.

[74] T. Li, D. Baumberger, D.A. Koufaty, and S. Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, page 53. ACM, 2007.

[75] Tong Li, Paul Brett, Rob Knauerhase, David Koufaty, Dheeraj Reddy, and Scott Hahn. Operating system support for overlapping-ISA heterogeneous multi-core architectures. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12. IEEE, 2010.

[76] Hai Lin and Yunsi Fei. Exploring custom instruction synthesis for application-specific instruction set processors with multiple design objectives. In *Low-Power Electronics and Design (ISLPED), 2010 ACM/IEEE International Symposium on*, pages 141–146, 2010.

[77] Yongpan Liu, Robert P Dick, Li Shang, and Huazhong Yang. Accurate temperature-dependent integrated circuit leakage power estimation is easy. In

*Proceedings of the conference on Design, automation and test in Europe*, pages 1526–1531. EDA Consortium, 2007.

[78] Jun Lu and Qinru Qiu. Scheduling and mapping of periodic tasks on multi-core embedded systems with energy harvesting. In *Green Computing Conference and Workshops (IGCC), 2011 International*, pages 1–6. IEEE, 2011.

[79] Benjamin Lubin, Jeffrey O Kephart, Rajarshi Das, and David C Parkes. Expressive power-based resource allocation for data centers. In *Proc. of the 21st International Joint Conference on Artificial Intelligence*, pages 1451–1456, 2009.

[80] K. Ma, X. Li, M. Chen, and X. Wang. Scalable power control for many-core architectures running multi-threaded applications. *ACM SIGARCH Computer Architecture News*, 39(3):449–460, 2011.

[81] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *USENIX*, pages 279–294, 1996.

[82] A.K. Mishra, S. Srikantaiah, M. Kandemir, and C.R. Das. CPM in CMPs: Coordinated power management in chip-multiprocessors. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–12. IEEE, 2010.

[83] T Mitra and R Jayaseelan. Dynamic thermal management via architectural adaptation. In *Design Automation Conference, 2009. DAC'09. 46th ACM/IEEE*, pages 484–489. IEEE, 2009.

[84] Matlab Nonlinear Models. http://www.mathworks.com/help/stats/nonlinear-regression.html.

[85] Gordon E Moore et al. Cramming more components onto integrated circuits, 1965.

[86] Trevor Mudge. Power: A first-class architectural design constraint. *Computer*, 34(4):52–58, 2001.

[87] Thannirmalai Somu Muthukaruppan, Haris Javaid, Tulika Mitra, and Sri Parameswaran. Energy-aware synthesis of application specific MPSoCs. In *Computer Design (ICCD), 2013 IEEE 30th International Conference on*. IEEE, 2013.

[88] Thannirmalai Somu Muthukaruppan and Tulika Mitra. Lifetime reliability aware architectural adaptation. In *VLSI Design and 2013 12th International Conference on Embedded Systems (VLSID), 2013 26th International Conference on*, pages 227–232. IEEE, 2013.

[89] Thannirmalai Somu Muthukaruppan, Anuj Pathania, and Tulika Mitra. Price theory based power management for heterogeneous multi-cores. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 161–176. ACM, 2014.

[90] Thannirmalai Somu Muthukaruppan, Mihai Pricopi, Vanchinathan Venkataramani, Tulika Mitra, and Sanjay Vishin. Hierarchical power management for asymmetric multi-core in dark silicon era. In *Proceedings of the 50th Annual Design Automation Conference*, page 174. ACM, 2013.

[91] Ali Pınar and Cevdet Aykanat. Fast optimal load balancing algorithms for 1D partitioning. *Journal of Parallel and Distributed Computing*, 64(8):974–996, 2004.

[92] Mihai Pricopi, Thannirmalai Somu Muthukaruppan, Vanchinathan Venkataramani, Tulika Mitra, and Sanjay Vishin. Power-performance modeling on asymmetric multi-cores. In *Proceedings of the 2013 international conference on Compilers, architectures and synthesis for embedded systems*. ACM, 2013.

[93] R. Raghavendra, P. Ranganathan, V. Talwar, Z. Wang, and X. Zhu. No power struggles: Coordinated multi-level power management for the data center. In *ACM SIGOPS Operating Systems Review*, volume 42, pages 48–59. ACM, 2008.

[94] Krishna K Rangan, Gu-Yeon Wei, and David Brooks. Thread motion: fine-grained power management for multi-core systems. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 302–313. ACM, 2009.

[95] Marisha Rawlins and Ann Gordon-Ross. An application classification guided cache tuning heuristic for multi-core architectures. In *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, pages 23–28. IEEE, 2012.

[96] Tajana Simunic Rosing, Kresimir Mihic, and Giovanni De Micheli. Power and reliability management of SoCs. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 15(4):391–403, 2007.

[97] Efraim Rotem, Alon Naveh, Doron Rajwan, Avinash Ananthakrishnan, and Eliezer Weissmann. Power-management architecture of the Intel microarchitecture code-named Sandy Bridge. *Micro, IEEE*, 32(2):20–27, 2012.

[98] Arjun Roy, Stephen M Rumble, Ryan Stutsman, Philip Levis, David Mazières, and Nickolai Zeldovich. Energy management in mobile devices with the cinder operating system. In *Proceedings of the sixth conference on Computer systems*, pages 139–152. ACM, 2011.

[99] Martino Ruggiero, Andrea Acquaviva, Davide Bertozzi, and Luca Benini. Application-specific power-aware workload allocation for voltage scalable MPSoC platforms. In *Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE International Conference on*, pages 87–93. IEEE, 2005.

[100] J.C. Saez, M. Prieto, A. Fedorova, and S. Blagodurov. A comprehensive scheduler for asymmetric multicore systems. In *Proceedings of the 5th European conference on Computer systems*, pages 139–152. ACM, 2010.

[101] Andreas Schranzhofer, Jian-Jian Chen, and Lothar Thiele. Dynamic power-aware mapping of applications onto heterogeneous MPSoC platforms. *Industrial Informatics, IEEE Transactions on*, 6(4):692–707, 2010.

[102] Seng Lin Shee, Andrea Erdos, and Sri Parameswaran. Heterogeneous multiprocessor implementations for JPEG:: a case study. In *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, pages 217–222. ACM, 2006.

[103] Seng Lin Shee and Sri Parameswaran. Design methodology for pipelined heterogeneous multiprocessor system. In *Proceedings of the 44th annual Design Automation Conference*, pages 811–816. ACM, 2007.

[104] Youngsoo Shin, Kiyoung Choi, and Takayasu Sakurai. Power optimization of real-time embedded systems on variable speed processors. In *Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design*, pages 365–368. IEEE Press, 2000.

[105] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. Managing performance vs. accuracy trade-offs with loop perforation. 2011.

[106] Kevin Skadron. Hybrid architectural dynamic thermal management. In *Proceedings of the conference on Design, automation and test in Europe-Volume 1*, page 10010. IEEE Computer Society, 2004.

[107] Kevin Skadron, Mircea R Stan, Wei Huang, Sivakumar Velusamy, Karthik Sankaranarayanan, and David Tarjan. Temperature-aware microarchitecture. 31(2):2–13, 2003.

[108] Jayanth Srinivasan, Sarita V Adve, Pradip Bose, and Jude A Rivers. The case for lifetime reliability-aware microprocessors. In *ACM SIGARCH Computer Architecture News*, volume 32, page 276. IEEE Computer Society, 2004.

[109] Jayanth Srinivasan, Sarita V Adve, Pradip Bose, and Jude A Rivers. The case for lifetime reliability-aware microprocessors. 32(2):276, 2004.

[110] Jayanth Srinivasan, Sarita V Adve, Pradip Bose, and Jude A Rivers. The impact of technology scaling on lifetime reliability. In *Dependable Systems and Networks, 2004 International Conference on*, pages 177–186. IEEE, 2004.

[111] Jayanth Srinivasan, Sarita V Adve, Pradip Bose, and Jude A Rivers. Exploiting structural duplication for lifetime reliability enhancement. In *ACM SIGARCH Computer Architecture News*, volume 33, pages 520–531. IEEE Computer Society, 2005.

[112] Haihua Su, Frank Liu, Anirudh Devgan, Emrah Acar, and Sani Nassif. Full chip leakage estimation considering power supply and temperature variations. In *Proceedings of the 2003 international symposium on Low power electronics and design*, pages 78–83. ACM, 2003.

[113] Fei Sun, Niraj K Jha, Srivaths Ravi, and Anand Raghunathan. Synthesis of application-specific heterogeneous multiprocessor architectures using extensible processors. In *VLSI Design, 2005. 18th International Conference on*, pages 551–556. IEEE, 2005.

[114] Dennis Sylvester, David Blaauw, and Eric Karl. Elastic: An adaptive self-healing architecture for unpredictable silicon. *Design & Test of Computers, IEEE*, 23(6):484–490, 2006.

[115] Radu Teodorescu and Josep Torrellas. Variation-aware application scheduling and power management for chip multiprocessors. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 363–374. IEEE Computer Society, 2008.

[116] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In *Compiler Construction*, pages 179–196. Springer, 2002.

[117] Paul Turner. Sched: Entity Load-tracking Re-work., 2011. https://lkml.org/lkml/2012/2/1/763.

[118] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel Emer. Scheduling heterogeneous multi-cores through performance impact estimation (PIE). In *Proceedings of the 39th International Symposium on Computer Architecture*, pages 213–224. IEEE Press, 2012.

[119] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel Emer. Scheduling heterogeneous multi-cores through performance impact estimation (PIE). ISCA, pages 213–224, 2012.

[120] Sravanthi Kota Venkata, Ikkjin Ahn, Donghwan Jeon, Anshuman Gupta, Christopher Louie, Saturnino Garcia, Serge Belongie, and Michael Bedford Taylor. SD-VBS: The San Diego vision benchmark suite. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 55–64. IEEE, 2009.

[121] Sravanthi Kota Venkata, Ikkjin Ahn, Donghwan Jeon, Anshuman Gupta, Christopher Louie, Saturnino Garcia, Serge Belongie, and Michael Bedford Taylor. SD-VBS: The San Diego vision benchmark suite. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 55–64. IEEE, 2009.

[122] X. Wang, K. Ma, and Y. Wang. Adaptive power control with online model estimation for chip multiprocessors. *Parallel and Distributed Systems, IEEE Transactions on*, 22(10):1681–1696, 2011.

[123] Jonathan A Winter, David H Albonesi, and Christine A Shoemaker. Scalable thread scheduling and global power management for heterogeneous many-core architectures. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 29–40. ACM, 2010.

[124] Tao Yang and Cong Fu. Heuristic algorithms for scheduling iterative task computations on distributed memory machines. *Parallel and Distributed Systems, IEEE Transactions on*, 8(6):608–622, 1997.

[125] Chuanjun Zhang and Frank Vahid. Cache configuration exploration on prototyping platforms. In *Rapid Systems Prototyping, 2003. Proceedings. 14th IEEE International Workshop on*, pages 164–170. IEEE, 2003.

[126] Wenyi Zhao, Rama Chellappa, P Jonathon Phillips, and Azriel Rosenfeld. Face recognition: A literature survey. *Acm Computing Surveys (CSUR)*, 35(4):399–458, 2003.