Improving Heterogeneous System Efficiency: Architecture, Scheduling, and Machine Learning

Daniel A Nemirovsky

Department of Computer Architecture

Universitat Politècnica de Catalunya

A thesis submitted for the degree of

Doctor of Philosophy in Computer Architecture

October, 2017

Director: Dr. Adrian Cristal

Codirector: Prof. Mateo Valero

For Kim

Abstract

Computer architects are beginning to embrace heterogeneous systems as an effective method to utilize increases in transistor densities for executing a diverse range of workloads under varying performance and energy constraints. As heterogeneous systems become more ubiquitous, architects will need to develop novel CPU scheduling techniques capable of exploiting the diversity of computational resources. In recognizing hardware diversity, state-of-the-art heterogeneous schedulers are able to produce significant performance improvements over their predecessors and enable more flexible system designs. Nearly all of these, however, are unable to efficiently identify the mapping schemes which will result in the highest system performance.

Accurately estimating the performance of applications on different heterogeneous resources can provide a significant advantage to heterogeneous schedulers for identifying a performance maximizing mapping scheme to improve system performance. Recent advances in machine learning techniques including artificial neural networks have led to the development of powerful and practical prediction models for a variety of fields. As of yet, however, no significant leaps have been taken towards employing machine learning for heterogeneous scheduling in order to maximize system throughput.

The core issue we approach is how to understand and utilize the rise of heterogeneous architectures, benefits of heterogeneous scheduling, and the promise of machine learning techniques with respect to maximizing system performance. We present studies that promote a future computing model capable of supporting massive hardware diversity, discuss the constraints faced by heterogeneous designers, explore the advantages and shortcomings of conventional heterogeneous schedulers, and pioneer applying machine learning to optimize mapping and system throughput. The goal of this thesis is to highlight the importance of efficiently exploiting heterogeneity and to validate the opportunities that machine learning can offer for various areas in computer architecture.

Keywords: Computer Architecture, Multi Core, Heterogeneous Systems, Scheduling, Machine Learning, Deep Learning

Acknowledgements

Working to achieve a PhD is a unique adventure to experience. It is unlike anything I have ever experienced. An era that has been altogether marvelous in its own right, both its soaring peaks and excruciating valleys, surely to be warmly rekindled as a collection of moments frozen in memory that one is fond to call their own. It is a remarkable journey that navigates one to explore and understand, to create and withstand. Above all else, it is a trial of one's very own perseverance and dedication. An impassable mountain were it not for the mentorship, camaraderie, and tireless support of one's advisors, colleagues, friends, and family.

So here, on these two humble pages, I shall try and undoubtedly fail to express my sincerest gratitude to all those who have been a part of this expedition at one time or another.

I would first like to thank my advisors Dr. Adrian Cristal and Professor Mateo Valero for accepting me into their research group and guiding my work. Mateo is a tremendously capable professional, highly intelligent and motivational. His leadership has fostered an outstanding academic atmosphere that I was fortunate to be a part of. I greatly appreciate the opportunity that you have given me.

There are few individuals that I have had the pleasure of working alongside that I consider geniuses in their respective fields but Dr. Adrian Cristal is certainly one. He has provided a guiding hand in all aspects of my research and has continually inspired me with his ability to clarify any technical barriers with the utmost ease.

I would also like to thank Dr. Osman Unsal who was the co-director of our research group and has been a wonderful unofficial advisor. Technically adept, Osman is responsible for shaping a great deal of my research and has always provided me with excellent advice for turning my abstract ideas into practical approaches.

I would like to also extend my gratitude to the members of my defense committee and reviewers of my thesis for your time and detailed as well as constructive critiques. They include: Dr. Adria Armejach, Dr. Vasilis Karakostas, Dr. Oscar Palomar, Dr. Gulay Yalcin, Dr. Antonio Cortes, and Dr. Petar Radojković.

The outstanding and friendly staff at the Barcelona Supercomputing Center and UPC's Departament d'Arquitectura de Computadores, especially Joana Munuera, deserve much recognition for their support over the years from dealing with all of us eccentric engineers.

I was fortunate to have brilliant colleagues which turned into great friends including Nikola Marković, Nehir Sonmez, Oriol Arcas, Ivan Ratković, Milan Stanić, Tugberk Arkose, Mladen Slijepcević, Behzad Salami, Damian Roca, Francesco Ciaccia, Josue Esparza, Javier Arias, and Vladimir Gajinov.

I would like to also thank my friends and family for their support over these years, especially my loving parents Laura and Mario and sister Michelle for their boundless dedication and encouragement. And to my other and better half, Kim, you're enduring patience and love I could not imagine living without.

Contents

1 Introduction

The field of computer architecture has been evolving over the last decade to embrace multi and many core (i.e., chip multi-processors or CMP) designs and more than ever, resource heterogeneity in terms of CPU and accelerator diversity found within systems on a chip (SoC). CMPs have become so readily utilized that they can be found in severs, desktops, laptops, mobile devices, and automated devices making part of the Internet of Things (IoT). Each of these devices may be used for running different workloads under specific circumstances, and within certain constraints. With processor fabrication technology currently at 7nm [25] and expected to reach 5nm by 2020, the advances in transistor densities have provided substantial opportunities for architects to combine even more general purpose cores, GPUs, accelerators, and cache memory onto a single system on a chip (SoC). Exploring heterogeneous solutions, however, requires studying microarchitecture as well as novel tools and approaches for optimization within the context of performance, energy consumption, and physical size or environment requirements.

Diversity is also found within the workloads that a CMP may need to execute. Applications may behave in very different manners from one another depending upon their program structure, computational, memory, I/O requirements, and parallelism. The rich heterogeneity of applications and system requirements is nicely complemented by the heterogeneity found in state-of-the-art CMPs such as ARM's big.LITTLE [38] and Nvidia's Tegra3 [74]. Extrapolating the trend of exploiting transistor density increases using heterogeneity, systems of the future may include hundreds or thousands of accelerators specialized for executing common libraries, kernels, or even whole programs.

In order to effectively support current and future heterogeneous architectures, single and multi-threaded applications must be properly managed by a heterogeneous scheduler. An effective scheduler should be aware of a system's diverse computational resources and how threads perform and interfere with one another on different cores. Heterogeneous scheduling uses its knowledge to periodically determine which threads to assign to specific cores. This process, otherwise known as mapping, is critical to perform efficiently in order to exploit hardware diversity and maximize system performance. An efficient heterogeneous scheduler can provide significant performance and energy benefits to heterogeneous CMPs compared with typical schedulers targeted for homogeneous systems. As a result, heterogeneous scheduling can provide architects with added flexibility in their design choices including the number of and types of cores as well as the cache configuration.

The design opportunities provided by heterogeneous schedulers are significant and provide insightful studies. There is, however, still room for improvement over state-of-the-art heterogeneous schedulers regarding the mapping scheme. Current schedulers, which make use of a priority [54] or round-robin [72] based heuristic to determine the mapping scheme, are not capable of predicting a mapping scheme to maximize system performance. To do so, a scheduler must know or at least be able to predict what the system performance would be for different mapping options and then select the one that results in the highest value. The correct approach and set of tools to use in order to predict system performance for different mapping options could require identifying behavioral statistics for an executing thread and finding relationships between these statistics and the thread's resulting performance on the different core types.

Managing and mapping threads is a problem that shares similarities with recommendation and navigation systems both of which have benefitted using machine learning (ML) techniques [62, 60, 16]. Though heterogeneous scheduling is a popular area of research, there has yet been no work exploring the use of ML for mapping optimization. In fact, using ML techniques to optimize computer systems has been as a whole seldom explored (see Chapter 8.3). The work in this thesis serves to demonstrate the practicality and improvements that can result by applying ML towards optimizing computer systems.

Artificial neural networks (ANNs) in particular are beginning to be utilized in a wide variety of fields due to their great promise in learning relationships between input data and numerical or categorical outputs [42]. The relationships are often hard to identify and program for manually, but using ANNs can result in excellent prediction accuracies. ANN based predictors have been shown to be useful in accurately predicting target categories and values for a wider variety of fields such as predicting network traffic [12] and stock market prices [39]. Moreover, feedforward ANNs such as those used in this work are lightweight in both computation and memory overheads and can be easily accelerated using GPUs or specialized hardware [103].

1.1 Problem statement and objectives

Given the contextual descriptions mentioned above, the problem statement of this thesis is how can we understand and utilize the rise of heterogeneous architectures, the benefits of heterogeneous scheduling, and the promise of machine learning techniques with respect to maximizing system performance. The purpose of this thesis is to highlight the importance of effectively exploiting heterogeneous systems, demonstrate the usefulness that ML can provide to heterogeneous scheduling, and in doing so, highlight the potential that ML can offer for various areas in computer architecture. To our knowledge, this work is the first to apply artificial neural network machine/deep learning techniques to heterogeneous scheduling.

The rest of this chapter describes an overview of the scope of the thesis, key challenges, contributions, and thesis outline.

1.2 Scope of the thesis

This thesis touches upon the fields of machine/deep learning, heterogeneous architectures, CPU scheduling, and multitasking. Described in this section is the scope of which this thesis goes into depth in each of these areas. More details about each of these topics is provided in section 2.1.

1.2.1 Machine learning

This work examines ML centered around artificial neural networks (ANNs) and deep ANNs (DNNs). These models are used for predicting performances of threads or of a whole system and are composed of layers of artificial neurons that are trained to develop relationships between the input parameters and the output target. The architectures of the specific ANNs used in this work is described in detail in Part II entitled "Applying ML to Heterogeneous Scheduling."

1.2.2 CMPs

Heterogeneous architectures can include various general purpose as well as accelerator computational cores within a system on a chip (SoC). There are two main types of heterogeneity that are explored within the scope of this thesis. The first is based on the hardware resource diversity of all hardware elements found within a SoC such as a CPU, a graphical processing unit (GPU), digital signal processor (DSP), and other accelerators. This can be considered more of a macro type of heterogeneity since the diversity is found between the different resources each of which serves a different purpose.

In contrast, the second type of heterogeneity that is explored in depth in this thesis is what can be referred to as micro heterogeneity or in other words, diversity between resources dedicated to a similar purpose. This can include diversity between different cores in a CPU related to microarchitectural differences. Conventional CMPs either rely on homogeneous cores (known as symmetric CMPs or SCMPs) or on cores which share an Instruction Set Architecture (ISA) but differ in terms of performance or functionality (referred to as asymmetric CMPs or ACMPs). ACMPs have been shown to be able to outperform SCMPs for a fixed area or power budget [81], [83] and are attractive candidates to exploit diverse workloads and performance requirements. In this thesis, we explore the reaches of heterogeneous diversity but our contributions are primarily applied to ACMP architectures since they are currently the most popular conventional implementations of heterogeneous systems.

1.2.3 Scheduling

The growth of heterogeneous architectures has led to a reexamination of scheduling mechanisms to exploit the resource diversity. Schedulers are typically responsible for CPU resource allocation for executing processes aimed at maximizing overall CPU utilization. This work focuses on exploring different practical techniques to optimize the scheduling mechanisms which are responsible for (i) selecting threads to execute and (ii) mapping those threads onto the hardware cores. The selection process is important in order to guarantee fairness in terms of execution time provided to all available threads, and the mapping process is critical for exploiting the system resources given different workload characteristics. The scheduler is called periodically (i.e., after every scheduling quantum which is typically within the range of 1ms-4ms) to reassess the selection and mapping schemes.

1.2.4 Workloads

Scheduling is critical in CMP systems when performing multitasking and executing parallel applications. Different applications and individual threads behave differently on the distinct hardware cores and may affect the performance of other neighboring threads (for example by causing interference in the shared resources). In this thesis, we evaluate various computational and memory intensive single and multi-threaded benchmarks. We explore how their diverse behaviors affect the abilities of our performance predictors and heterogeneous schedulers.

1.3 Key challenges

Each of the areas considered within the scope of this thesis consists of particular constraints and challenges.

For heterogeneous architectures, this involves identifying the performance, energy, and size requirements as well as what types of hardware resource diversity can be practically implemented. It is important to examine the advantages and disadvantages that result from composing CMPs using different core types, memory structures, and accelerators for executing diverse workloads under specific constraints. Crafting an analytical framework is a powerful method that can help with analyzing different heterogeneous implementations.

For scheduling, the challenges include identifying the mechanisms and shortcomings of current techniques, heuristics and tools that could improve these mechanisms, and the feasibility of proposed implementations. Judging the practicality of the proposed implementations combines balancing the performance benefits that arise from the proposal with the overhead costs imposed which can include quantifying the scheduling or algorithm computation cost, context swaps, and the runtime statistics needed to be gathered.

A similar set of challenges can be found when applying ML techniques to the mapping mechanism of a scheduler. Though numerous ML models could be useful in predicting system performance, each model may result in different accuracies, implementation practicality, and overhead costs. After a certain level of complexity, the gains in performance from using ANNs may be outweighed by the overhead costs owing to the amount of calculations needed to be performed and/or the parameters needed to be gathered and stored.

Finally there is the challenge of determining the experimental framework including simulators and benchmark suites with which to conduct our studies and validate our proposals. The simulator should be hardware validated, accepted by the community, and be able to accurately model numerous heterogeneous cores and scheduling mechanisms. The benchmarks should be representative of the environments that are targeted by the proposals and include computational, memory, and I/O intensive single and multi-threaded workloads. It is necessary for the chosen simulator to present performance, energy, and physical area results of the different experiments. The scalability of the proposals is also a challenge pertinent to design practicality and should be tested using a simulator that can accommodate increasing the number and diversity of cores and benchmarks.

1.4 Thesis contributions

Our main contributions can be divided into two parts. Part I consists of a motivating contribution highlighting a long term vision for massively heterogeneous systems and two preliminary studies which explore the benefits that heterogeneous schedulers can provide for energy savings and architecture flexibility. Part II is composed of studies which justify and validate applying ML models to heterogeneous scheduling, resulting in significant performance improvements and implications for computer architecture as a whole. Specifically, we describe a pioneering proposal to decouple a scheduler's thread selection and mapping function, and apply machine/deep learning to predict thread and system performance at the quantum granularity. Below we provide a brief overview of each of the main contributions in this work.

1.4.1 Reimagining heterogeneous computing

A semantic gap seems to be emerging between the advances in hardware and software in terms of resource and program diversity and the use of high abstraction levels. This is due to the physical and compatibility constraints that hardware developers face, which are far more flexible at the software level. However, given the current state of the industry, architects need to consider radical paradigm-shifting computing-model proposals. Such proposals will not necessarily offer clear roadmaps or practical short-term solutions, but they could contain the hints and alternate ideas needed to rethink the long-term vision that computer architects hope to achieve.

In this motivating work, we address an emerging hardware-software semantic gap by considering an unconventional computing model that merges current heterogeneous state-of-the-art hardware with the concept of abstraction that has been very useful and ubiquitous in software. To do so, we use the concept of abstraction to reinterpret the notion of the ISA. Most ISAs in use today remain (for compatibility reasons) based on designs developed several decades ago to solve that era's physical and software constraints. The current CMP model's scalability is inherently constrained because of the current ISA's functional granularity, which introduces high memory footprint and energy consumption. As a remedy, we propose a functional ISA (F-ISA) that increases the functional abstraction level of the machine instructions. Consequently, this extra level of functional abstraction enables a dramatic increase in the heterogeneous diversity of a processor's computational units, resulting in greater specialized execution particular to the needs of the software algorithms. We hope that this alternative computational model can significantly improve and fine-tune system performance relative to latency, memory footprint, and power.

1.4.2 Preliminary studies

Drawing inspiration from how heterogeneous system may look like in the future from the subsection mentioned above, the objective of the preliminary studies is to examine the opportunities and limitations present in current CMP and scheduling designs. We believe this discussion presents a realistic starting point in order to acquire several key insights instrumental for advancing towards the future of heterogeneous architectures. Each of the following two contributions provides separate investigations into analyzing CMP design tradeoffs, and improving performance and energy efficiency.

Extending the flexibility of ACMPs for mobile devices through alternative cache organizations

The focus of this work is to highlight how architects can optimize the energy efficiency and size of ACMP systems by using novel memory configurations. This approach offers a method to tailor an ACMP system to provide suitable performance, energy efficiency, and area for very demanding mobile devices. We propose three alternative cache configurations and examine their effects on system performance and processor size when executing applications concurrently. Our results show that adopting an alternative cache hierarchy in conjunction with a scheduler targeting asymmetrical systems can lead to substantial energy savings of over 17%, power reductions of over 5%, and over 19% reductions in physical size while still outperforming execution times achieved with conventional operating system schedulers on a CMP with larger caches by over 10%.

Performance and energy efficient hardware-based scheduler for Symmetric/Asymmetric CMPs

Since more and more applications become multi-threaded we expect to find a growing number of threads executing on a machine. Consequently, the operating system will require increasingly larger amounts of CPU time to schedule these threads efficiently. Instead of perpetuating the trend of performing more complex thread scheduling in the operating system, we propose a hardware implementation of the Thread Lock Section-aware Scheduling (TLSS) scheduling mechanism [71]. This lightweight mechanism helps to identify multi-threaded application bottlenecks such as thread synchronization sections and complements state-of-the-art heterogeneous schedulers. It is, to our knowledge, the first hardware based lock section-aware scheduling that is energy attentive and can be applied to both asymmetric and symmetric CMPs. It achieves average performance gains of 10.9% compared to the state-of-the-art Linux OS Scheduler when applied on an SCMP. At the same time, it is 81% more EDP (energy-delay product) efficient when applied on an ACMP and compared to the Linux OS Scheduler on an SCMP, where ACMP and SCMP take relatively the same chip area.

1.4.3 Applying ML to heterogeneous scheduling

Accurately estimating the performance of applications on different heterogeneous resources can provide a significant advantage to heterogeneous schedulers seeking to improve system performance. Having demonstrated the opportunities and limits available using current CMP and scheduling designs, the contributions of this part of the thesis examine applying machine learning techniques to conventional heterogeneous schedulers in order to predict the system performance for different mapping schemes.

A machine learning approach for performance prediction and heterogeneous CPU scheduling

In this study we propose a unique throughput maximizing heterogeneous CPU scheduling model that uses ML to predict the performance of multiple threads on diverse system resources at the scheduling quantum granularity. We demonstrate how lightweight ANNs can provide highly accurate performance predictions for a diverse set of applications thereby helping to improve heterogeneous scheduling efficiency. We show that online training is capable of increasing prediction accuracy but deepening the complexity of the ANNs can result in diminishing returns. Notably, our approach yields 25% to 31% throughput improvements over conventional heterogeneous schedulers for CPU and memory intensive applications on an ACMP.

A deep learning mapper (DLM) for heterogeneous scheduling

This work pioneers applying a deep learning mapper to a scheduling model that decouples thread selection and mapping routines. We use a conventional scheduler to select threads for execution and a deep learning mapper to map the threads onto a heterogeneous hardware. The validation of our preliminary study shows how a simple deep learning based mapper can effectively improve system performance for state-of-the-art schedulers by 8%-30% for CPU and memory intensive applications.

1.5 Thesis outline

The subsequent sections of this thesis are structured as follows. Chapter 2 presents the necessary background on heterogeneous systems, application behaviors, CMP scheduling, and machine learning models. It elaborates on the motivating factors of this thesis and concludes by providing an inspirational vision of what a reimagined massively heterogeneous computing model for the future can look like. Chapter 3 describes the experimental frameworks and metrics that were used in testing and evaluating the contributions presented in this thesis. This includes descriptions of the system simulator, processor models, and benchmark suites.

The thesis is then divided into two parts composed of preliminary investigations followed by a study on applying ML to heterogeneous scheduling:

Part I presents two preliminary studies on the increased architectural flexibility and energy efficiency benefits provided by heterogeneous scheduling in Chapters 4 and 5.

Part II presents two pioneering studies on applying ML to heterogeneous scheduling. Chapter 6 defines a scheduling model based upon a thread behavior predictor, ANN based thread performance predictors for each core type, and a scheduling heuristic. Chapter 7 presents an extension of the machine learning based heterogeneous scheduler which expands the scope of the predictors to cover thread interference effects and extends their applicability to be compatible with nearly all conventional schedulers as an add-on module.

Chapter 8 discusses the related work regarding all of the studies presented in this thesis and Chapter 9 details our conclusions. A list of publications and references is provided at the end.

2 Motivation

This chapter presents the necessary background on homogeneous and heterogeneous CMPs, application behaviors, multi-core scheduling, and ML techniques. We identify the potential of heterogeneous CMPs, scheduling mapping shortcomings, and ML tools as the motivating factors for this thesis. The chapter concludes by providing a compelling vision of what a reimagined massively heterogeneous computing model for the future can look like and why designing efficient heterogeneous scheduling is at the core of scaling heterogeneity.

2.1 Background

2.1.1 Chip Multi-processors (CMPs)

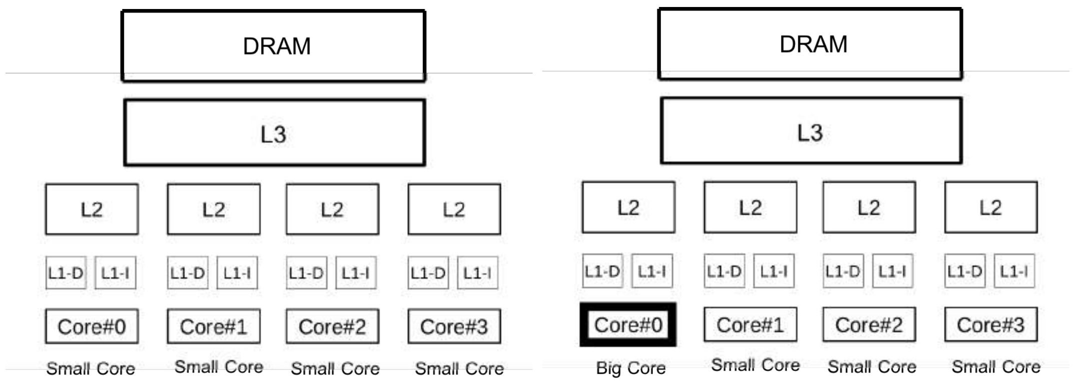
Computer architects have been greatly benefiting from the steady progress of chip fabrication techniques which have vastly expanded the number of available transistors to make use of in their designs. Previously, architects made use of the added transistors by designing larger and more complex single core CPUs until the trend fell out of favor due to power constraints and limits to parallel execution among others. Current trends exploit these extra transistor densities by implementing several identical (i.e., homogeneous) computational cores on a single chip, thereby improving the performance when running several applications concurrently or parallelized workloads. These types of homogeneous CMPs are known as symmetric or SCMPs.

As long as only a single core type is used, SCMPs can includes either complex and powerful cores, such as those found in the Intel Xeon [52] or AMD Opteron [17] processors, or simple and lower-power cores as in Sun's Niagara [59]. A drawback of SCMPs, however, is that all cores will have to be implemented with the same complexity as the most powerful core required to meet certain constraints. This is a disadvantage since not all applications nor environments will have similar constraints. Hence, the hardware is essentially limited by the weakest link of all applications which requires higher core performance. This disadvantage is apparent in conventional SCMP designs which are becoming more limited by power dissipation and efficiency concerns. The power dissipation issues, which are also highlighted in the study termed Dark Silicon [33], have served to highlight the bounds of current design practices. Thermal dissipation constraints limits the total amount of transistors that may be powered concurrently at nominal voltage, thereby leaving large areas of a transistor rich chip to be off at any given time. The need to develop faster, smaller, and less power hungry CPUs has motivated research related to heterogeneous or asymmetric CMPs (ACMPs).

Several studies [81, 83] have been conducted that highlight the ability of ACMPs to outperform SCMPs for a fixed area or power budget. A wide range companies have also conducted research on ACMP systems including Intel [64, 106], and HP [81]. Commercial ACMP implementations such as ARM's big.LITTLE [38] and Nvidia's Tegra3 [74] have also been popularly used.

Compared to SCMPs, ACMPs employ cores that execute the same instruction set architecture (ISA) but differ in terms of microarchitecture. These differences can include issue width, whether the core supports out-of-order (OoO) or in-order execution, brach predictors, cache configuration, and also voltage and clock frequency. An advantage of using cores that share the ISA as opposed to specialized accelerators is that the application code does not need to be compiled separately for each core type that has a different ISA. Figure 2.1 illustrates the differences between an SCMP and an ACMP design.

Determining which combination of different types of cores to include in a CMP design is a non-trivial task and depends heavily upon the performance, power, and size requirements. These requirements are based upon the workloads, environments, and devices that a CMP is targeted towards. Relying only on homogeneous



(a) SCMP with 4 small cores. (b) ACMP with 1 large and 3 small cores.

Figure 2.1: An example of SCMP and ACMP CPU designs. Note that the cores must be all of the same ISA (e.g., ARM, x86) but may be more of different complexities for the ACMP system. The designs both have private L1 and L2 caches and a shared L3 cache.

CMP approaches is a simple approach but may result in sub-optimal performance compared with a heterogeneous CMP. Part of the motivation and proposed solutions of this work is what criteria is needed to evaluate potential CMP designs and how to determine the optimal CMP configuration (either SCMP or ACMP) given certain performance requirements. Though employing hardware specialization can be useful for determining which core type yields the most effective performance/energy trade-off for a specific type of program behavior, it is up to the scheduler to determine a thread mapping scheme with which to effectively exploit system diversity.

2.1.2 Caches

CMPs include a cache hierarchy which consists of a set of private and shared cache structures to keep data that will be reused close to the cores that need them. A typical cache hierarchy configuration for a CMP may include separate but private instruction and data level 1 caches (L1), combined but private L2 caches, and a last level cache (LLC) shared amongst all the cores. In addition to private or shared, caches can also take on a distributed nature similar to non-uniform cache accesses (NUCA). A distributed cache is logically a single shared structure but physically composed of separate cache structures located in different areas of the chip. Load and store accesses to this cache may end up resulting in different latencies depending on how far the data physically resides from the core requesting it.

The L1 cache typically include tens of kilobytes (KB), the L2 contains up to a few megabytes (MB), and the L3 or LLC with at least an order of magnitude more cache (e.g., tens of MB). Architectural trends have also been to use the increases in transistor densities to enlarge the sizes of the caches on chip. The levels of the caches, whether private, shared, or distributed, size of each cache, associativity of the caches, and replacement policies are various of the properties that may differ greatly between CMP designs and might affect the system's and application's performance.

In order to ensure that no two threads are simultaneously modifying and using the same data, a coherency mechanism is utilized by the caches. Various protocols which change the state (e.g., modified, shared) of a data line or directory based schemes can be used to enforce coherency between different cores' private cache structures. Since the coherency overheads rise as the amount of shared data accesses from different cores increase, selecting which threads to run and where (i.e., which physical core to map the threads to) can directly impact performance.

Cache footprint

This section considers the cache footprint in terms of energy consumption, power budget, and physical size for an example case of an ACMP with one large and three small cores with private L1 (32KB) and L2 (256KB) cache per core and a single L3 (8MB) as the shared LLC (this ACMP is detailed comprehensively in Section 3.2). Modifying or reorganizing the cache hierarchy in order to mitigate the cache footprint may be a promising endeavor depending upon the performance/energy requirements of a CMP design.

The cache hierarchy of the example ACMP system when running both SPEC and SPLASH-2 benchmark suites consumes on average about 30% of the total energy and power budget of the processor. The last level L3 cache (LLC) alone consumes a significant chunk of the processor's energy, power, and size budget. Using measurements taken with McPAT [89] when running the 4 core ACMP, the LLC is responsible for on average about 10% of the total execution energy. However, in experimental simulations of a four core ACMP, the LLC has been shown to be responsible for around 21% of the processor's subthreshold leakage power (note that the total processor leakage power was upwards of 25% of the total peak power). In terms of physical size that it takes up on the die, the four cores, which include the L1 and L2 cache structures, take up about 67% of the total chip size while the LLC takes up a substantial 32%, and the interconnection network (NoC) a mere 1%.

2.1.3 Program structures and behaviors

A CMP system is regularly used to run programs which were designed for different purposes. Additionally, each of these applications may utilize multiple software threads to divide their work. Voice over IP programs, web browsers, media players and editors, and even daemons which seek to perform background optimizations and updates are just a small example of the diverse range of applications that a CMP will be required to execute concurrently.

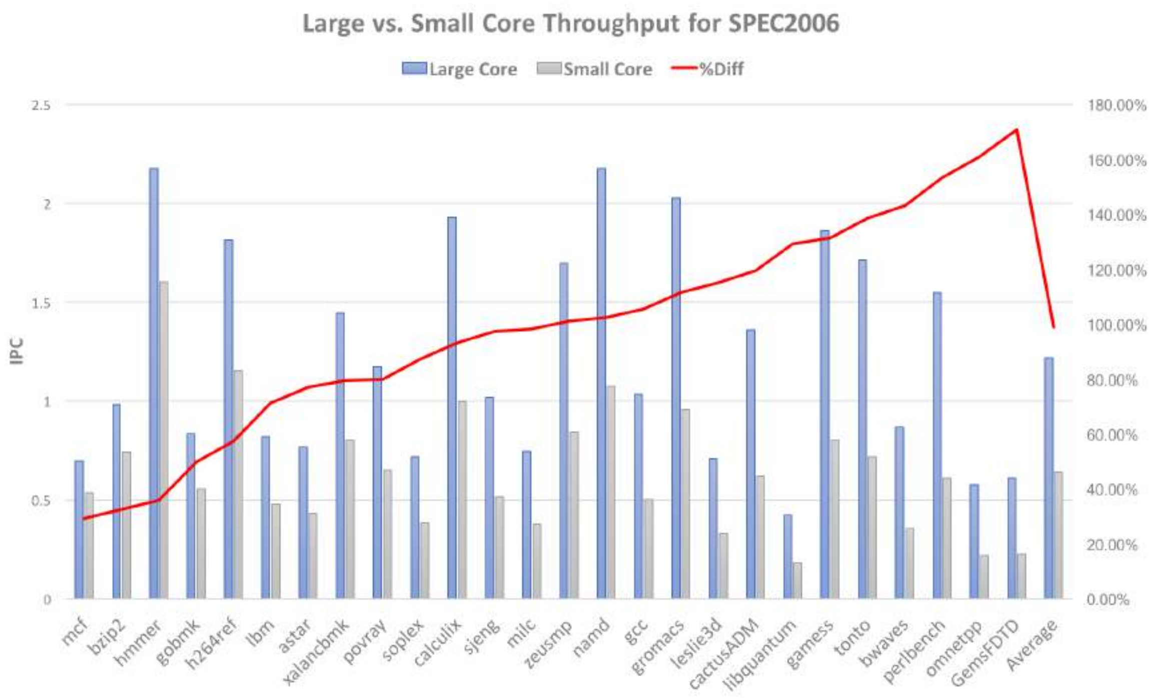
Program structure refers to the algorithms, variables/objects, methods/functions, and dataflow used in the code itself. In contrast, a program's or thread's behavior is characterized by periodic patterns related to the instructions per cycle (IPC), instruction mix, branch prediction, cache and memory accesses and hit/miss rates, and other statistics related to its performance on a particular hardware. Variations in program structure can result in different behaviors in terms of computational, memory, and I/O intensity, all of which may vary significantly. Consequentially, program behavior is directly related to its structure and the physical hardware which it is executed on. Even applications targeted to solve similar problems may be coded so differently from one another that they may behave differently when executed on identical hardware. The memory access patterns are of particular interest in this case since applications may interfere with one another either by polluting shared memory structures such as the last level cache, or by accessing shared data structures which can be the case for multi-threaded programs.

Figure 2.2 illustrates the performance differences that result from executing SPEC2006 and SPLASH-2 applications on a large core compared to a small core (for core details see Section 6.2.1). On average, the applications achieve nearly 2-2.3x better performance (instructions per cycle or IPC) when executing on the large core compared with the small core. Variations in IPC differences can also be observed between applications. For some applications, these IPC differences can be either very minor (mcf 29%, barnes 6%, and radix 1%) or very sizable (gemsFDTD 171%, omnetpp 161%, and water.nsq 200%). These variations can be partially explained by the code's structure and algorithms, including loops, data dependencies, I/O and system calls, and memory access patterns among others.

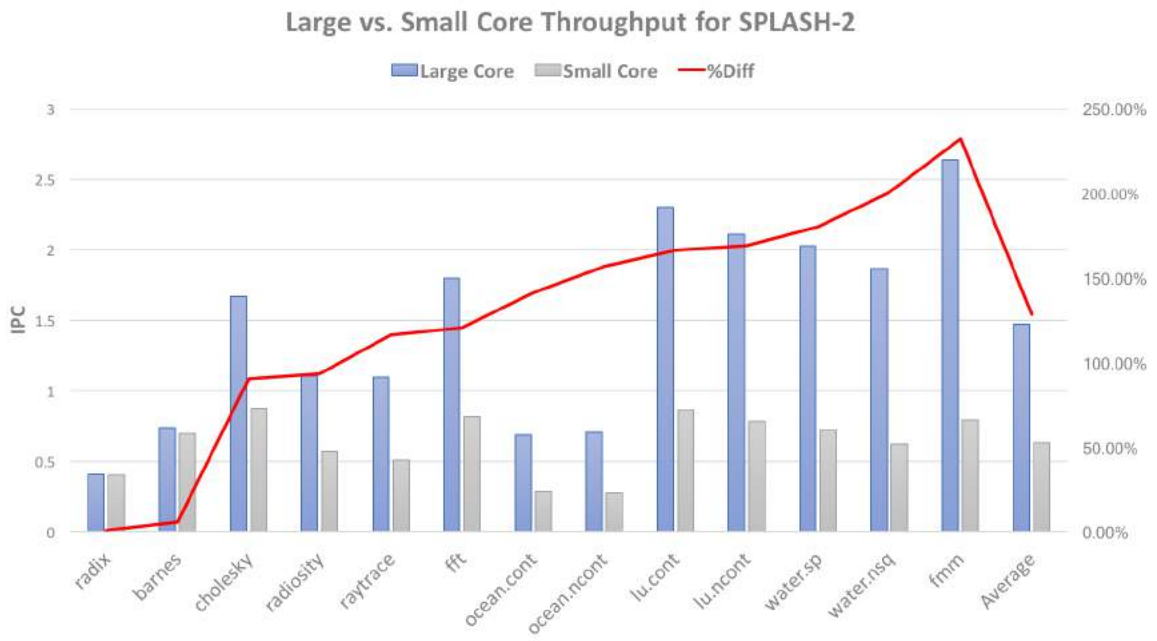
These discrepancies between the performance gains of each benchmark when running on the large core vs. the small core clearly demonstrates the potential for system throughput optimization if an effective scheduling mapping scheme is chosen. For instance, if a combination of these benchmarks is run in parallel on an ACMP, then a scheduler with knowledge of performance behaviors of each application for all core types may be able to find an optimal mapping scheme that results in significantly better system performance than a random mapping scheme.

When investigating whether a program is computational, memory, or I/O intensive, it is often useful to break down their execution in terms of a CPI (cycles per instruction) stack. This stack separates the amount of total execution time an application spent on each type of instruction. Depending on the CPI stack, it is therefore possible to diagnose the application as being computational, memory, or I/O intensive.

Understanding program structures and behavioral characteristics is crucial when executing diverse programs on heterogeneous systems since each program may behave differently on distinct core types. For example, applications containing large quantities of instruction or memory level parallelism (ILP and MLP) could be more suited for execution on out-of-order (OoO) cores. On the other hand, applications with high amounts of thread level parallelism (TLP) or explicit instruction level parallelism may be ideal candidates for execution on lightweight in order cores. By identifying the behaviors of applications on the different cores, it is possible to better exploit the heterogeneous hardware. For instance, consider a two core system composed of a large OoO core and small in order core. In this case, system performance could be optimized when the small in order core is used to run computationally intensive workloads while the larger OoO core is used to execute memory intensive workloads, that would otherwise suffer from substantial performance degradation due to memory request stalls if run on an in order core.



(a) Core to core IPC differences for SPEC2006 benchmarks.



(b) Core to core IPC differences for SPLASH-2 benchmarks.

Figure 2.2: The performance differences that result from executing each benchmark on a large vs. small core. Higher IPC numbers represent faster performance.

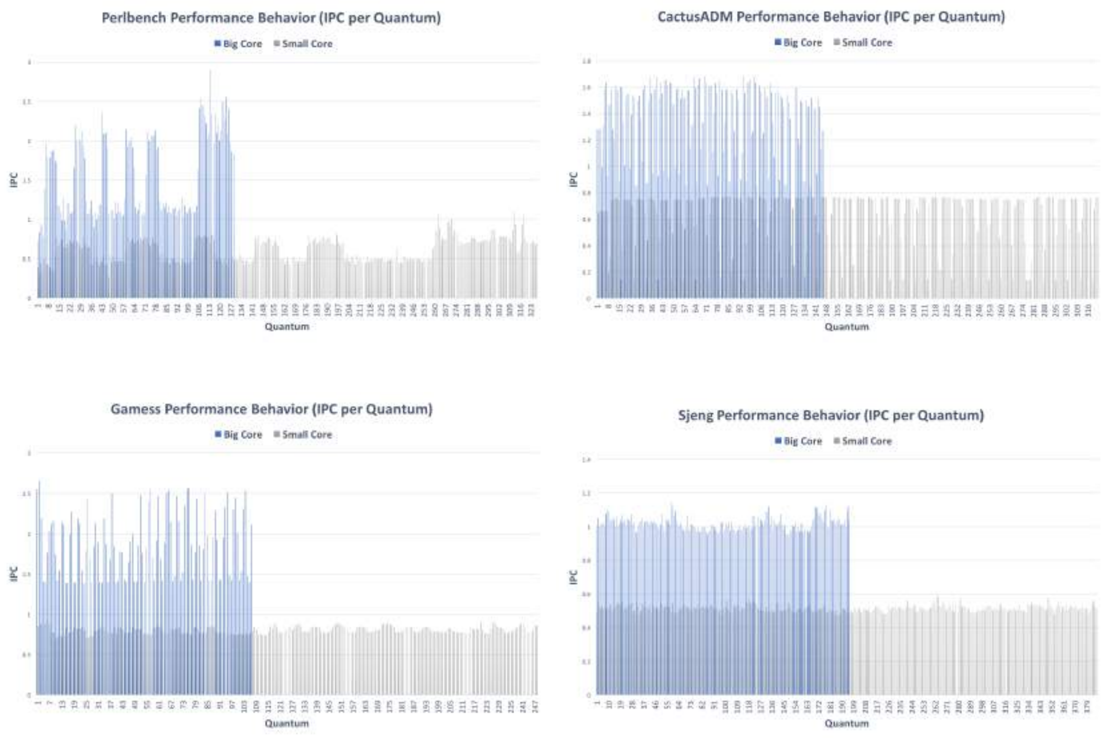


Figure 2.3: The IPC per quantum behavior of four SPEC benchmarks when running on the big core compared to the small core.

While not all programs exhibit the same behavior, studies [29, 94] have shown that the behavioral periodicity in different applications is typically consistent. In fact, the behavioral periodicity has been shown to be roughly on the order of several millions of instructions and is present in various different and even non correlated metrics stemming from looping structures inside of applications. Heterogeneous architectures such as ACMPs provide excellent environments for exploiting the behavioral diversity of concurrently executing programs.

Figure 2.3 helps to visualize this behavioral periodicity. It shows the IPC variability of the perlbench, cactusADM, gamess, and sjeng SPEC benchmarks throughout their simulated execution on an Intel Nehalem using a 1ms execution quantum. Though there are clearly periodic behavioral phases that span tens and sometimes hundreds of quanta, it is also possible to observe that for finer granularities, the IPC variation from quantum to quantum is quite minimal, and more so on the small core.

CMPs typically employ a shared memory model which may result in cooperative or interference effects when running applications and threads in parallel that access shared resources using a common address space. Parallel programs, in specific, may consist of several threads which need to synchronize or share locked data structures, commonly resulting in data races due to data dependencies. Accessing and modifying shared data from different cores may cause cache lines to be transferred between private caches which penalizes application performance and total system throughput. These effects are lower for work stealing workloads, however, since they allow idle cores to steal work from busy cores. As described in [48], inter-thread synchronization bottlenecks such as contended critical sections may cause thread imbalances at runtime leading to adverse performance effects. However, parallelized applications can be effectively exploited by CMPs since they can execute the different parallel sections of code using the various hardware resources available. Distinct thread and memory management/sharing approaches can also result in significant effects on system performance and utilization.

2.1.4 CMP scheduling

In order to properly manage threads and system resources, CMPs rely upon a scheduler that can be implemented either in software, for example in the operating system (OS), or hardware.

CMP schedulers rely chiefly upon two mechanisms to fulfill their policy objectives: 1) thread selection and 2) thread to core mapping (from here on referred to simply as mapping).

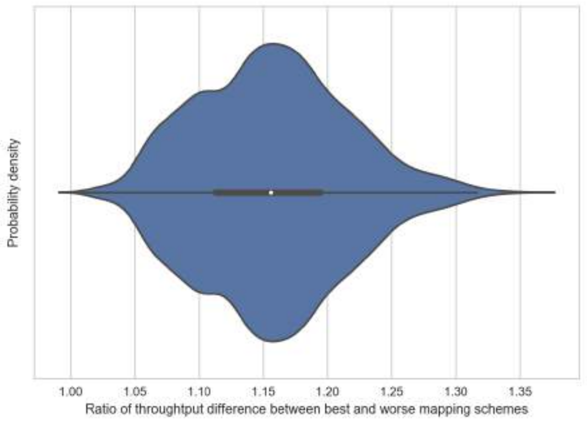
The thread selection mechanism is responsible for selecting a subset of threads to run from a larger pool of available threads. It does so by using heuristics which order the threads using priorities or scores related to how critical the threads are (e.g., time constrained or system level tasks may be given a higher priority than background tasks which search for application updates) or how much execution time or progress the threads have made so far. To identify all available threads, the scheduler keeps track of the current state of all the different threads (e.g., whether a thread is ready for execution, paused, or stalled on a disk access). Thread selection generally ensures that no threads are continually starved of system resources thereby guaranteeing a certain level of fairness and load balancing.

The scheduler may be triggered periodically via a specified scheduling quantum to preempt the current execution of the running thread(s) or also be called asynchronously as in the case of a thread stall. For the rest of this work, we will utilize the terms scheduling quantum and execution quantum interchangably.

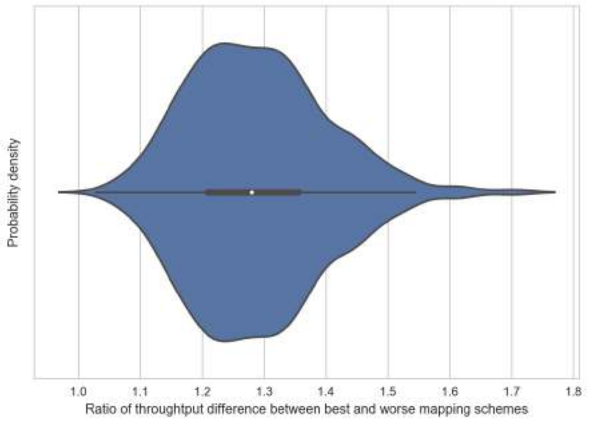
Different CPU schedulers may have different thread to core optimization objectives. Some may try to maximize system throughput (IPC), others may want to finish the shortest thread or longest thread in the shortest time, and others may try to optimize for energy efficiency. Deciding on the correct scheduling policy to apply depends upon the target systems, programs, and applicability to different environments.

Mapping

Finding the optimal thread to core mapping on a heterogeneous system is no simple endeavor. This is especially the case when executing diverse workloads since the performance of each application is likely to vary from quantum to quantum



(a) SPEC2006.



(b) SPLASH-2.

Figure 2.4: The importance of how different mapping schemes for a given set of applications can result in significantly different system throughputs. The y-axis represents the probability of a given ratio of throughput differences (x-axis) between the best and worst mapping schemes for all possible combinations of four applications from SPEC2006 and SPLASH-2 when running on a 1-large and 3-small core system. X-axis values should be read as the factor of system throughput improvement of selecting the best mapping scheme compared to the worst mapping scheme for a given four application combination. The white dot in the middle of the x-axis represents the mean and the black dots represent observed values which occur more frequently near the mean.

Benchmark	Min	Max	Mean
SPEC2006	1%	36%	16%
SPLASH-2	3%	71%	29%

Table 2.1: The minimum, maximum, and average differences (in percentage terms) between the best and worst mapping schemes for all possible combinations of SPEC2006 and SPLASH-2 benchmarks on a 1-large and 3-small core system.

and from core to core. The specific performance differences that result from executing an application on different core types can be measured by distance or as a ratio. These differences can vary from application to application (inter-application) as well as from phase to phase within an application (intra-application).

The inter-application variations in core to core IPC differences shown in Figure 2.2 also exist at the quantum level within each application (i.e., intra-application) as shown in Figure 2.3. It is important to highlight that these differences can vary as the application enters and exits different phases of execution. The more inter and intra application variations of core to core IPC differences there are, the harder it is for a scheduler to identify the optimal mapping scheme, but the more opportunities for improvement there are. A scheduler that needs to produce a new mapping scheme every quantum can further take advantage of these fine granularity performance discrepancies between applications and threads. Exploiting these variances may lead to identifying a mapping scheme every quantum which maximizes system performance.

To showcase how identifying these core to core IPC differences can translate into mapping benefits, consider the case where four applications (e.g., A, B, C, and D) are selected to run on an ACMP with one large core and three small cores. Four mapping schemes which assign one application to the large core and the other three to the small cores can be A-BCD, B-CDA, C-DAB, D-ABC. Each mapping scheme will produce a different resulting system IPC. The overall benefits of an effective mapper will be based upon the difference between the best and worst mapping schemes. For instance if A-BCD is the best mapping scheme resulting in a system IPC of 4 and C-DAB is the worst with a system IPC of 2, then the difference in percentage terms would be 100% (i.e., $(4-2)/2$).

To demonstrate this in practical terms, we found the differences between the best and worst mapping schemes for all possible combinations of four applications from both the SPEC and SPLASH suites. Figure 2.4 exposes the more detailed distribution these results for both SPEC and SPLASH. Both subfigures plot the probability density distribution of the best vs worst mapping schemes which highlights the importance of how different mapping schemes for a given set of applications can result in significantly different system throughputs. While both distributions are fairly similar, SPEC does appear to suffer from slightly more variation and less overall throughput differences although it is still very significant. Table 2.1 presents the average, minimum, and maximum differences (in percentage terms) between the best and worst mapping schemes for all possible combinations. The minimum differences range from 1%-3%, average differences from 16%-29%, and maximum differences from 36%-71% for all possible mapping schemes for SPEC and SPLASH respectively. These results expose the theoretical benefits that may be gained from an effective scheduler at the application level granularity. Practical schedulers, however, work at the quantum level granularity and may additionally take advantage of intra-application core to core performance differences which could expose greater opportunities for mapping optimization.

In order to identify an optimal mapping scheme, a heterogeneous scheduler should be able to estimate the system performance that each individual mapping scheme would produce. Conventional schedulers, however, typically do not make use of the mechanisms needed to exploit this potential. As we shall see, machine learning can be an effective tool for schedulers to utilize in order to help estimate system performance. In the studies presented in Part II we use the powerful prediction capabilities of artificial neural networks (ANNs) to greatly increase the benefits that CPU schedulers can provide for maximizing heterogeneous multi core performance.

CMP schedulers in practice

The vast majority of conventional CMP schedulers are designed for SCMP system configurations and therefore cannot take advantage of differences between computational cores. The Completely Fair Scheduler (CFS) [54], which was integrated with the Linux 2.6 kernel, is an example of a homogeneous scheduler. The state-of-the-art CFS selection scheme combines priorities with execution time metrics in order to select the threads to run next; however, the mapping scheme is relatively simplistic. When mapping, the CFS evenly distributes the threads onto the cores such that all cores have approximately the same number of threads to run. These threads are effectively pinned to the core because they are only swapped with threads on their assigned core and not with those of another core (i.e., threads do not move from the core they were initially assigned to).

In order to fully utilize heterogeneous systems, however, schedulers should be aware of and make use of a system's diverse resources. Several heterogeneous schedulers targeting ACMP systems have been proposed that show significant improvements over conventional SCMP schedulers. Some of these make use of online or offline profiling as well as sampling or estimation techniques to determine the optimum thread to core mapping (in relation to performance and/or power) whenever a specific event is detected or scheduling time quantum is completed [66], [20], [55] among others.

The fairness-aware scheduler by V. Craeynest et al. [56] is a heterogeneous scheduler which works similarly to the CFS but instead of mapping all threads evenly on all cores and pinning them there, it maps the highest priority thread (i.e., the one that has made the fewest progress) to the most powerful core. For example, in a four core system with one powerful core and three smaller energy efficient cores, this scheduler will send the thread with the highest priority to the large core and the next three highest priority threads to the other 3 small cores. This is similar to the Global Task Scheduler [51] used by ARM in its big.LITTLE heterogeneous systems which modifies the Linux kernel to send ready threads who have spent the most time waiting for a core to the large cores and the subsequent threads to the small cores. Thread lock section-aware scheduling [71] expands upon the fairness-aware scheduler by prioritizing threads who have recently switched back from system level to user level code to run on the powerful cores. This is done in an attempt to catch and accelerate the critical sections of a thread.

Another scheduler targeted at heterogeneous systems is the hardware round-robin scheduler by Markovic et al. [72]. Instead of using priorities for thread selection, this approach chooses which threads to run next in a round-robin manner (thereby guaranteeing fairness) and then maps the selected threads to the cores. Using the same 4 core system as described above, this scheduler will rotate the threads in a manner similar to a first in first out queue, from small core to small core to small core to big core and then back into the thread waiting pool until all threads have had a chance to execute.

Schedulers produce overheads which may mitigate efficiency gains due to the cost of managing and selecting threads, calculating optimal thread to core mapping, and context switch penalties. It is therefore imperative for effective schedulers to balance finding an optimal mapping without triggering unnecessary context swaps and being as lightweight as possible.

Context switches

Implementing dynamic scheduling requires switching of the context of the cores including swapping the architectural state (register file), flushing the pipeline, and reloading the working-set into the private caches. A context switch incurs a fixed cost for storing and restoring the architecture state (at most a few kilobytes) [88] with most of the overheads being due to reloading the caches. Figure 2.5 presents the context switch overheads costs when executing different working set sizes on an Intel i7-4600U core[18]. The overheads grow gradually with the size of the working set until an excessively large working set causes additional page faults which increase the overheads significantly. The study [55] has shown the migration overhead to be less than 1.5% across different types of single-threaded workloads, ranging from memory to compute intensive, for a 4 MB shared LLC using a 1ms quantum.

2.1.5 Machine learning

Part of the attraction of machine/deep learning is the flexibility that its algorithms provide to be useful in a variety of distinct scenarios and contexts. For instance, advances in computer vision and natural language processing using convolutional neural network techniques [62, 60, 16] have led to high levels of prediction accuracy enabling the creation of remarkably capable autonomous vehicles and virtual assistants. Generally speaking, machine/deep learning techniques can be used to build

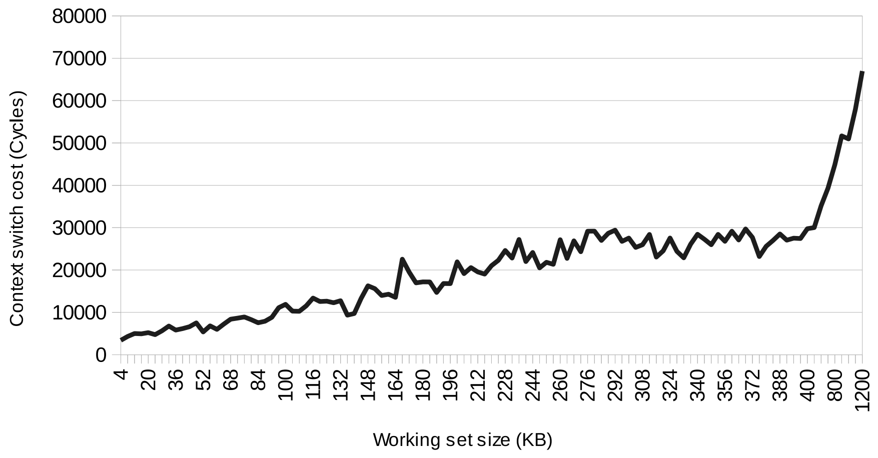


Figure 2.5: Context switch cost on a dual core Intel(R) Core(TM) i7-4600U CMP [18]. The overhead includes costs related to swapping architectural state, flushing the cores' pipelines, and reloading data into the private caches.

highly accurate predictors, pattern recognition systems, and also systems for data generation. Though ML techniques have been gaining traction over the last few years, its application toward improving hardware performance remains in its earliest stages. As of yet, there has been no seminal work applying machine learning for predicting thread performance on heterogeneous cores and maximizing system throughput.

Artificial neural networks (ANNs) in particular, are beginning to be utilized in a wide variety of fields due to their great promise in learning relationships between input data and both numerical or categorical outputs. The relationships learned by the ANNs are often hard to identify and program manually but can provide excellent prediction accuracies. Moreover, ANNs and deep ANNs (DNNs) are readily utilized for prediction purposes and can be relatively lightweight and flexible to implement. They consist of a set of input parameters (considered to be the first layer) connected to a hidden layer of artificial neurons which are then connected to other hidden layers before connecting to one or more output neurons. The inputs to the hidden and to the output neurons are each assigned a numerical weight that is multiplied with its corresponding input parameter and then added together with the result of the neuron's other incoming connections. The sum is then fed into an activation function (usually a rectified linear, sigmoid, or similar). The output of these neurons is then fed as input to the next layer of neurons or to the the output neuron(s). The result of the output neuron(s) is the prediction that the ANN makes which may be either a numerical value estimate or a classification estimate using one-hot bit encoding.

An ANN can learn to produce accurate predictions by adjusting its weights using a supervised learning method and training data. Other unsupervised methods of learning such as clustering also exist but fall outside the scope of this work. Supervised learning is performed via a learning algorithm such as backpropagation that adjusts the weights in order to find an optimal minima which reduces the prediction error based on an estimated output, the target output, and an error function. The more hidden layers and hidden units there are, the more intricate relationships that may be learned, however having too deep of an ANN can also lead to overfitting. Overfitting is when the model achieves very low error for data it has seen but produces high error when predicting for previously unseen data. Generalizing the ANN to be able to consistently predict at similar accuracies for seen (i.e., training) and unseen (i.e., testing) data is a central issue in ANN design and many different types of approaches are used.

One such common practice in machine learning is to divide the data into training (70%), validation (15%), and test (15%) sets in order to not overfit the model. Evaluating how a model performs on the validation and test sets allows to modify the ANN to improve its generalization when predicting for unseen data. Other approaches include using regularization techniques to regulate the importance of parameters within the learning algorithm and more intricate methods such as dropout which trains the ANNs in phases excluding certain hidden units to ensure all neurons are able to contribute to a prediction. A way in which to evaluate the error of a model and how many data samples are needed to provide a consistent accuracy is in the use of learning curves. The learning curves show how the training and validation errors of the ANNs changes as the amount of training data increases. After a certain amount of training data, the curves level off so that increasing the amount of training data will not impact the ANN's accuracy.

Another approach for providing prediction flexibility and generalization to new data is the use of online learning. While ANNs can be trained statically before being ever used, they can also keep learning dynamically by periodically training as new data samples are provided. This method improves the accuracy and generalizability of the ANNs when predicting for applications that are run more than once. Learning dynamically is an area of active research since there are tradeoffs between how much to adapt the ANN for newer data and how that will impact the ANNs generalizability to past data as well as heavier computational costs. Online learning can also be extremely important in scenarios where the applications and behaviors of the workloads being run change over time.

Advances in learning algorithms have enabled faster training times and allowed for the practical use of intricate ANN architectures. The training calculations themselves may also be executed in parallel for the neurons within the same layer. The latency of these calculations can be further mitigated through the use of hardware support including GPUs, FPGAs, or specialized neural network accelerators. Deep learning methods expand on more simplistic machine learning techniques by adding depth and complexities to existing models. Using DNNs or ensembles of different machine learning models is a typical example of deep learning.

We believe the predictive power of ML techniques such as artificial neural networks (ANNs) will be of significant use to computer architects for optimizing system performance for example by estimating system performance, branch predictions, and cache/memory accesses.

2.2 Reimagining Heterogeneous Computing

This section proposes a motivating long-term vision to support the expansion of massively heterogeneous architectures which support and complement the diversity and functionality found in software applications, tasks, and kernels.

In an innovative step forward, the hardware community has started to tackle the power wall and the memory wall by diversifying the computational cores. Whereas initial chip multiprocessors (CMPs) integrated several identical computation cores per chip, we now see an increasing tendency to explore the reaches of integrating an expansive range of diverse computational cores.

On the software front, the standardization of programming practices has promoted the development of vast amounts of commonly utilized libraries, tools, and frameworks. This has marked an unprecedented growth in the level of abstraction available to the average programmer. Emphasis on code optimization and portability has also led to the development of programming models and runtime systems that let programmers define and extract substantial amounts of parallelism in their code, such as OpenCL [102] Cilk Plus [86] and OpenMP [24].

Used effectively, these programming models allow significant gains in application performance and resource utilization. Yet, while these software models have been useful and complementary to advances on the hardware front, their considerable runtime overheads, implementation complexity, and limited adoption have hindered their appeal and applicability. Conversely, parallel applications have been shown to share similar characteristics, which increase their appeal for standardized libraries and hardware practices.

For now, a semantic gap seems to be emerging between the advances in hardware and software. The scope and potential of new technologies have started to lead to heterogeneous many-core systems, but hardware architects have not similarly embraced, as of yet, the use of high abstraction levels. This is because of the physical and compatibility constraints that hardware developers face, which are far more flexible at the software level. However, given the current state of the industry, architects need to consider radical paradigm-shifting computing-model proposals. Such proposals will not necessarily offer clear roadmaps or short-term pragmatic solutions, but they could contain the hints and alternate ideas needed to rethink the long-term vision that computer architects hope to achieve. In this section, we address this hardware-software semantic gap by considering an unconventional computing model that merges current heterogeneous state-of-the-art hardware with the concept of abstraction that has been so useful and ubiquitous in software, but not present in hardware. To do so, we use the concept of abstraction to reinterpret the notion of the ISA.

Most ISAs in use today remain (for compatibility reasons) based on designs developed several decades ago to solve that era's physical and software constraints. The current CMP model's scalability is inherently constrained because of the current ISA's functional granularity, which generally results in a large memory footprint and high energy consumption. As a remedy, we propose a functional ISA (F-ISA) that increases the functional abstraction level of the machine instructions. Consequently, this extra level of functional abstraction enables a dramatic increase in the heterogeneous diversity of a processor's computational units, resulting in greater specialized execution particular to the needs of the software algorithms. We hope that this alternative computational model can significantly improve and fine-tune system performance relative to latency, memory footprint, and power.

2.2.1 Conceptual discussion

Computation as we know it today is grounded on the interaction and communication between two key elements: data (to be used, manipulated, and/or produced) and functions (specifying what is done with the data). We identify three data-function computational models: (i) data to function, (ii) function to data, and (iii) data and function.

Data to function

This model is embodied in a system that comprises separate memory (instruction and data) and computational structures in which, via one or more machine-level instructions, data is sent from the memory (such as DRAM) to be executed in a functional unit (such as an integer adder). This method is the standard approach used by current CPUs, and both Von Neumann and Harvard architectures fall within its scope. This model's limiting factor is that a functional unit's complexity is determined by the ISA's functional abstraction level. An add instruction corresponds to an adder functional unit, a branch instruction to a branch unit, and so on. Because nearly all conventional ISAs rely on low-level functional instructions, the physical characteristics of the data and functional units are intrinsically bound by the exposed level of functionality.

The limited diversity and complexity of the functional units mean that they can understand only primitive data objects (e.g., integers, doubles, and branches). Consequently, this produces a homogenizing effect on the executable data objects and furthers the need for repetitive executions to perform higher-complexity functions (such as list sort and matrix multiplication). If the data is homogenized into primitive data objects, then more accesses to memory are required to load the necessary data into the cores to perform the desired (higher abstraction level) functions. These factors can significantly contribute to a system's overall memory footprint and power usage, a problem that is further exacerbated by the fact that conventional CPUs consolidate the available functional units into a handful of computational cores. Compared with a futuristic heterogeneous core configuration, in which each core contains specialized functional units suited to specific data structures, memory contention is greater when all cores are equally capable of executing the same functions on the same data. This is due to having larger number of simultaneous accesses to primitive data objects which are typically stored in shared memory structures.

Function to data

In this model, functional machine-level instructions are sent to a unified (that is, homogeneous) memory structure that contains both data and functional units, such that the execution happens directly within the memory structure. A key limiting factor of this model is that in order to maintain compatibility and reliability, all memory structures and substructures must contain access to the same set of functional units. A variation of this model is used occasionally to complement standard CPU processing. Processing in memory technologies [98] and other vector functional units directly built in memory are examples of function to data cases.

Data and function

This is a hybrid approach that is similar to the function-to-data model but allows for separate and distributed (that is, heterogeneous) memory structures. It is therefore possible to separate different data objects (such as matrices from lists) into separate memory structures that contain the functional units specific to those data objects (for example, the matrix multiplication unit versus the list sorting unit). Although this approach is elegantly promising, its principal drawback is that the size and composition of the physical memory structures should correspond to the data structure itself. Any slight modification or extension to an existing data object could result in significant performance penalties. This is because changes in the data object can affect not only the functional part of the execution but also the organization and quantity of data that the object uses. Moreover, if this radical approach is implemented directly, it will create considerable code portability and compatibility complications. Although there seem to be interesting developments in implementing this model using specific accelerators and FPGAs, research in this area is only just starting to scratch the surface of its true potential.

Discussion

To expand the hardware's heterogeneous diversity, we must rely on one of the three computational models discussed. The feasibility of the function-to-data scheme is lacking, because the need to replicate every type of functional unit for every memory structure is unrealistic and unreasonable. Variations of this model, however, offer clear opportunities for some fields of application, most notably embedded systems and graphical processing.

On the other hand, we can reconsider using the standard data-to-function model. But as we noted, if we were to use this approach by replicating the number of cores on a chip, we would hit an impasse. If we could extract enough parallelism from the workloads to keep the system constantly fed and balanced, our progress would be halted by the memory wall and dark silicon, whereas if we underfed the system, we would be inefficiently using the available transistor richness.

The data-and-function model could perhaps offer the most benefit in terms of specialization based on a particular workload or programming model, but its structural inflexibility and code portability and compatibility concerns must be overcome. However, we could attenuate these concerns by raising the ISA's functional abstraction level. This could allow for a substantially greater diversity and quantity of functional units, while also providing flexibility in how the code is executed.

2.2.2 Functional ISA

Current ISAs, which most conventional CPUs are designed to execute, provide low levels of functionality that limit the physical functional units' scope and diversity. Far from considering them ineffective or wanting to replace such instrumental CPUs, we seek to foster code portability and compatibility using a flexible model and applying it gradually. Thus, we developed a higher functional-level intermediate ISA based on a conceptual combination of hardware principles and software abstraction techniques. Similar to typical ISAs, our F-ISA consists of instructions that define a functional method and a data element. However, this F-ISA follows a top-down approach, starting from a software perspective, to determine the functionality and data context of each particular F-ISA instruction. A program is typically developed using one or more abstraction levels (that is, using libraries, classes, and subclasses), which, when unfolded, for instance by a compiler, expose lower and lower abstraction levels until only pseudo machine code (such as LLVM [61]) is left.

The theoretical objective of the F-ISA is to capture and follow each of these abstraction levels for as many function or method instances as possible. Practically speaking, F-ISA's appeal is its applicability in commonly used libraries and frameworks that have functions, objects, and methods found across different programs. An example of such a ubiquitously used library is the Java standard library, which includes object structures and methods corresponding to vectors, hash tables, and lists. Mobile applications are valuable, and more recent examples in which common data structures, functions, and methods (belonging to iOS or Android application programming interfaces) are frequently used across different applications that still rely on conventional general purpose CPUs for execution.

Compared to the composition of a typical ISA, the structure of F-ISA is flexible and open-ended so that its final composition can be adapted on a per-application basis. The end representation of a program as F-ISA code is analogous to a call graph, in which each node represents one particular functional instruction. Every instruction consists of three elements: an instruction identifier, a function identifier, and a data context address.

The instruction identifier is a unique address associated with the memory location where the F- ISA instruction can be found, similar to a typical instruction's PC (program counter) address, and is needed to preserve code sequentiality.

The function identifier is similar to a traditional opcode and denotes the specific executable function that the instruction will request (for example, MATMUL would be an identifier for matrix multiplication). Because each unique function or method in a program is assigned a particular function identifier, many more function identifiers will be needed than there are typical opcodes.

The data context address provides the address of the context where all the relevant data needed to properly execute the function can be found. To create the data context, the data objects can be assigned particular addresses at compile time (although they could also be allocated dynamically), and then these addresses are organized into a data context that is assigned to a corresponding F-ISA instruction.

We can consider the actual size of an F-ISA instruction to be the sum of the function identifier and the data context address. For example, an 80-bit F-ISA instruction would allow for a 16-bit function identifier and the direct 64-bit address for the data context.

2.2.3 Example case

To illustrate how a program could be represented as F-ISA instructions, consider Figure 2.6. Once compiled, the F-ISA representation of this program, which has the instruction-level syntax <*function identifier, data context address*>, can assume the form shown in Figure 2.7. Note that *&ABC* corresponds to the address of the data context that contains the addresses of matrices *A*, *B*, and *C*. Similarly, the address *&A* contains the data context address of matrix *A*, and so on for the other instructions.

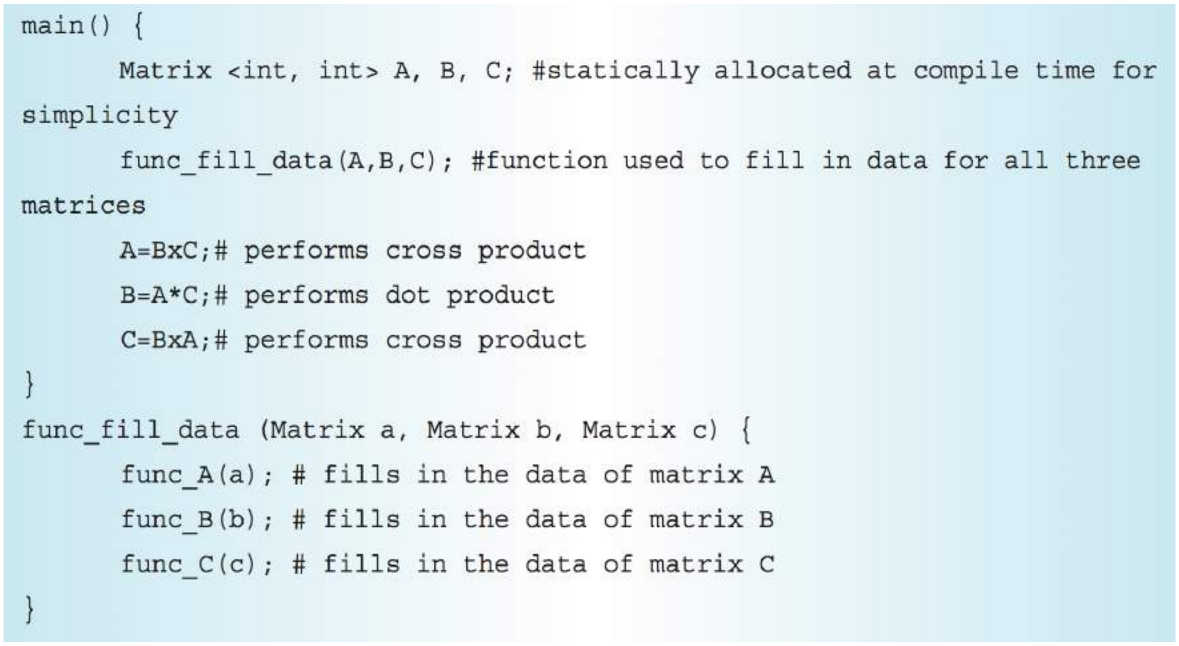
Each instruction defines a function that can either directly correspond to a physical functional unit present in the hardware (for example, an add instruction to an adder) or be further unfolded into the instructions representing its subfunctions. If the first method is available, the instruction is dispatched to the functional unit for execution, which assumes that it will be able to most efficiently perform the function based on the information provided in the instruction. The second method allows for an F-ISA instruction to be unfolded into another set of F-ISA instructions. This is shown in the case of the instruction [<*func fill data, &ABC*>, which can be broken down further into three separate F-ISA instructions: <*func A, &A*>, <*func B, &B*>, and <*func C, &C*>.

This unfolding process can be continued for each instruction until the first method (that is, when the level of functionality of the instruction matches that of a hardware functional unit) can be applied. If the function *main()* marks the highest functionality level that an F- ISA instruction can specify, the lowest is equivalent to the functional level expressed in conventional ISAs. For practical reasons, an F-ISA instruction can also be expressed as a routine comprising a set of machine-level instructions (that is, typical ISA instructions). This feature allows for the preservation of conventional ISA processors because any F-ISA instruction can be sent to a typical computational core as a machine-level routine. Additionally, an F-ISA instruction's ability to be expressed either as a collection of machine-level instructions or unfolded into lower-level functional instructions allows for gradual implementation and compatibility.

Given that the same program should be able to run on different physical systems, which could each consist of different computational cores and functional units, a specialized hardware or software runtime will be needed to determine whether to unfold or dispatch the F-ISA instructions to the appropriate and available functional units. Although a detailed description of such a runtime falls outside the scope of this work, we have been researching and designing possible hardware implementations. Figure 2.8 shows the steps a theoretical F-ISA runtime unit would take when determining how to process each F-ISA instruction. To most effectively take advantage of the program just presented, the hardware should have a matrix accelerator that includes functional units specifically designed to efficiently execute matrix-based operations, such as dot and cross products.

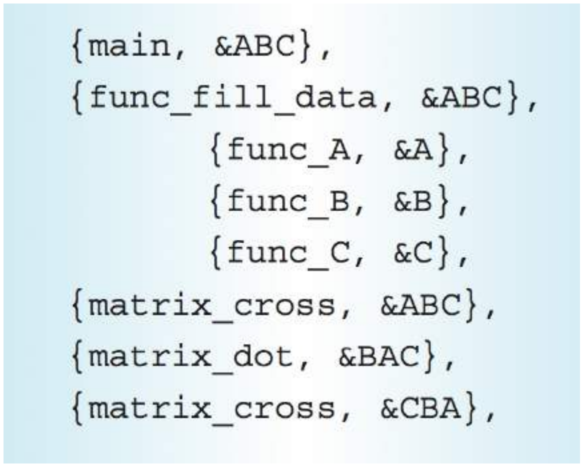
Furthermore, because most, if not all, matrix object data should be stored close to the accelerator (assuming that the accelerator contains local memory or is close to physically distributed memory), there would be lower memory bandwidth contention due to reduced data movement and cache conflict and capacity misses within the processor. Therefore, combining the use of F-ISA instructions with specialized functional units or accelerators could enable significant latency, power, and memory footprint performance benefits.

At this point, we do not expect to present a comprehensive and faultless conceptual model. Because this is a new and unconventional proposal, various challenges must be met to strengthen the model's practical realization. Some key topics that we need to elaborate further include runtime and compiler support, memory management (including data allocation and mapping), data dependencies, program



```
main() {
       Matrix <int, int> A, B, C; #statically allocated at compile time for
simplicity
       func_fill_data(A,B,C); #function used to fill in data for all three
matrices
       A=BxC;# performs cross product
       B=A*C;# performs dot product
       C=BxA;# performs cross product
}
func_fill_data (Matrix a, Matrix b, Matrix c) {
       func_A(a); # fills in the data of matrix A
       func_B(b); # fills in the data of matrix B
       func_C(c); # fills in the data of matrix C
}
```

Figure 2.6: Pseudocode of a program comprising cross/dot product functions and data initialization functions for three matrices.



```
{main, &ABC},
{func_fill_data, &ABC},
       {func_A, &A},
       {func_B, &B},
       {func_C, &C},
{matrix_cross, &ABC},
{matrix_dot, &BAC},
{matrix_cross, &CBA},
```

Figure 2.7: A representation of the matrix multiply program when compiled into F-ISA code. Each F-ISA instruction has the syntax <*function identifier, data context address*>.

sequentiality, and I/O and OS exception and interrupt handling. Although certain existing techniques, including dynamic compilation, dynamic memory allocation, and dataflow runtime models, could help alleviate some of these concerns, we will

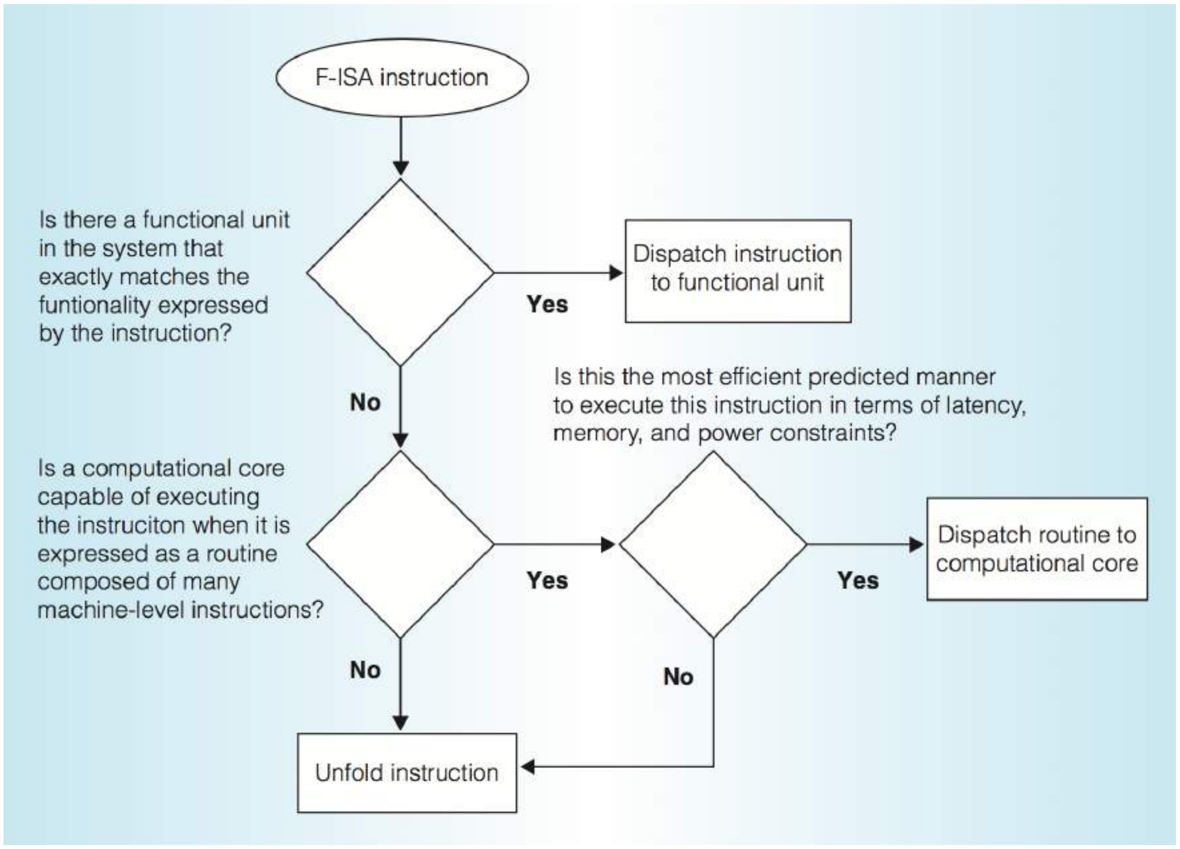

F-ISA instruction
Is there a functional unit in the system that exactly matches the funtionality expressed by the instruction?
Yes
Dispatch instruction to functional unit
No
Is this the most efficient predicted manner to execute this instruction in terms of latency, memory, and power constraints?
Is a computational core capable of executing the instruciton when it is expressed as a routine composed of many machine-level instructions?
Yes
Yes
Dispatch routine to computational core
No
No
Unfold instruction

Figure 2.8: The decision-making process of an F-ISA runtime dispatcher during the execution of an F-ISA instruction. The runtime is tasked with deciding whether to dispatch a function directly to a computational core for execution or to unfold it into its subfunctions and repeat the process. The runtime will dispatch functions for execution only on cores that can execute the function in the most effective manner based on the state of the system; otherwise, the function will be unfolded.

undoubtedly have to develop new schemes to strengthen the proposed model.

Of foremost importance to realize this aim is the need to address how the runtime will determine whether to unfold a F-ISA instruction or send it to directly execute on a specialized core and which core to send it to if more than one is available. The mechanism needed is tightly related to the area of CPU scheduling in conventional CMPs but requires significant advances in scheduling for heterogeneous systems which as of today remain relatively limited in scope, sophistication, and practicality.

2.2.4 Conclusion

The vast increases in transistor densities on chips have offered a challenge to computer hardware architects. Keeping in mind the power limitations of uniprocessor designs, architects have responded by integrating several optimized computational cores within one processor. The push into embedded computing has added further latency and power constraints, resulting in an increasing interest in heterogeneous many-core processors. However, the current trend of exploiting heterogeneity is inherently limited in scope by the low functionality level provided by the ISAs of existing processor architectures. This results in a narrow set of possible functional units that can be used and also serves to homogenize the data, which, for good reasons, had been meticulously distinguished at the software level.

This work offers a new and unconventional computing model that raises the level of functional abstraction of the hardware instructions to enable greater flexibility and diversity for implementations of hardware functional units and accelerators. This method can enable significant advances in relation to object data mapping and execution, resulting in latency, memory footprint, and power/performance gains. The hope of our long-term vision is to support a powerful heterogeneous many-core processor that complements and embodies the diversity and functionality found in applications.

For this vision to be realized, however, significant advances in heterogeneous scheduling techniques and scalability will be needed. With the industry just starting to adopt heterogeneous multi-cores, it is imperative to focus on building efficient and scalable heterogeneous scheduling methods which will serve as a foundation for future heterogeneous architectures. With this futuristic computing model in mind, the rest of this work focuses on exploring the relatively new field of heterogeneous scheduling as applied to the current state-of-the-art heterogeneous CMPs.

3 Methodology

Before proceeding with the main studies of this thesis, it is important to describe the experimental methodology utilized for validating our proposals. Properly implementing and evaluating the proposals requires a simulation framework with configurable CMP architectures and schedulers that can be used to run a set of diverse benchmarks and provide statistical data such as detailed performance and energy results.

This chapter describes in detail the simulator, system architectures, benchmarks, schedulers, and machine learning tools utilized in the studies presented in this thesis. Furthermore, a specific methodology section highlighting the exact simulation configurations is provided in each study in Parts I and II.

3.1 Simulator

In deciding which simulation framework to utilize, it is important to consider a simulator which is widely used in the computer architecture research community, produces fast simulations within reasonable error bounds, is highly configurable in terms of architectures and schedulers, and includes hardware validated CMP representations capable of simulating the execution of various threads in parallel. Sniper [105] is a standout candidate capable of dealing with each of these issues and is the simulation framework used in this thesis. Sniper is a popular parallel x86-64 multicore simulator which is based on an interval core simulation model. It provides fast simulation (tens millions of instructions per second or MIPS) while

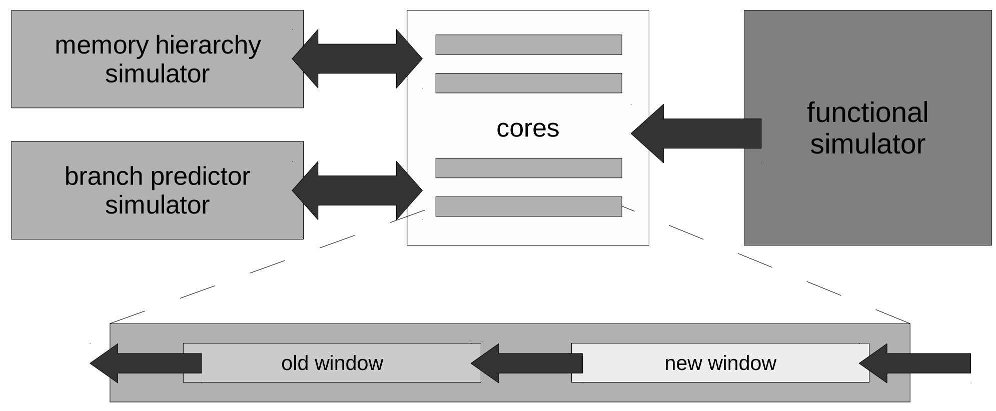


Figure 3.1: Sniper simulator interval model

still averaging within 25% performance results errors validated against multi-socket Intel Core2 and Nehalem systems. As a result, Sniper is capable of more detailed and accurate CMP studies when compared to simplistic one-IPC simulators and also useful in scenarios evaluating a system running large parallelized workloads which would take too long using cycle-accurate simulators.

In contrast to slow and highly complex cycle accurate simulators, the interval model [19] utilizes an analytical model to abstract core performance. It is capable of generating the timing and performance statistics of each individual core without needing to reproduce every instruction passing through a core's pipeline stages. In an interval model, the simulator fast-fowards simulation to certain miss events (e.g., serialization instructions, branch mispredictions, and cache and TLB misses) and estimates the number of instructions and cycles executed since the previous miss event. The period between the two miss events is called an interval. Miss events and corresponding latencies are generated by functional simulators corresponding to a core's branch predictor, memory hierarchy, cache coherence, and interconnection network. An analytical model then uses these events and latencies to calculate the timing for each interval. Driving simulation using separate functional simulators and an analytical model enables being able to model complex performance behaviors (including thread interference effects) when executing concurrent threads on a CMP.

An illustration of how the interval model simulates the timing for a CMP's cores is shown in Figure 3.1. It maintains an instruction window (one per simulated core) corresponding to the size of the reorder buffer (ROB) in an out-of-order (OoO) core. Modeling OoO execution, the instruction window can handle miss events that occur during outstanding memory accesses. New instructions are inserted into the tail of the instruction window while a core's progress depends upon the instruction at the head. To account for OoO execution, only non overlapping miss latency penalties may be added to the total simulated time. If no miss events occur, then the instructions are dispatched at a rate dependent upon instruction execution latencies and inter-instruction dependencies. To express the amount of execution cycles spent in the different components (e.g., caches, branch predictor, etc.) of a system, Sniper is capable of reproducing the CPI stacks visually. This helps to identify how much system efficiency benefits improving each component will result in. Figure 3.3 provides an illustration of CPI stacks which is elaborated on below in Section 3.3.2

Sniper is also highly flexible in terms of simulation configurability including the ability to evaluate different homogeneous and heterogeneous CMPs based on customizable x86 architectures. The Intel Nehalem is the most state-of-the-art processor architecture included in Sniper. A CMP composed of numerous Nehalem cores can be chosen to be simulated with each core capable of being customized to emulate, for example, different instruction windows, dispatch width, branch predictor, ALUs, ld/st queues, out of order execution, and cache structures.

The configurability additionally extends to being able to modify and choose to use different schedulers during each simulation. Sniper includes a set of software implemented CMP schedulers which can be easily modified to emulate the Linux Completely Fair Scheduler (CFS) and Fairness-aware scheduler. To model power and area, Sniper is integrated with the Multicore Power, Area, and Timing (McPAT) framework [89]. McPAT is configured in this work to measure results based on a 45nm technology. For the experiments presented this thesis, we modify the system architecture, core microarchitecture, cache hierarchy, and scheduler Sniper simulation configurations. The Sniper simulator is free to use for academic research and maybe be found online [101].

Table 3.1: Detailed configurations of the large and small cores.

Small core	4-wide, 5-stage out-of-order, 16-entry ROB, 6-entry LD/ST queue, 2.66GHz
Large core	4-wide, 12-stage out-of-order, 128-entry ROB, 48-entry LD/ST queue, 2.66GHz
IL1 caches	private 32KB write-through, 4-cycle, 8-way
DL1 caches	private 32KB write-through, 4-cycle, 8-way
L2 cache	private unified 256KB write-back, 8-cycle, 8-way
L3 cache	shared 8MB, write-back, 30 cycle, 16-way
Line replacement	Least recently used (LRU)
Cache coherence	MESI protocol, on-chip distributed directory, L2-to-L2 cache transfers allowed, 8K entries/bank, one bank per core
NoC	12.8 GB/s per direction and per connected chip pair
Memory	modeling all queues and delays, latency 120 cycles, controller bandwidth 7.6 GB/s
Area	$190mm^2$ for an ACMP with 1 large core and 3 small cores using 45nm transistor technology.

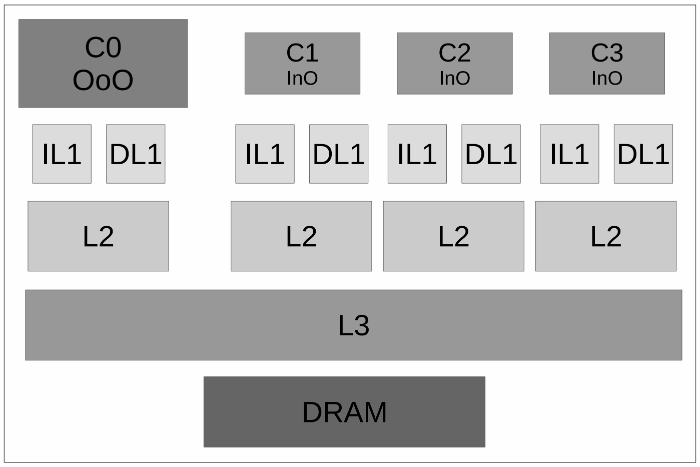


Figure 3.2: Heterogeneous CMP architecture principally used in this thesis.

3.2 CMP architecture

The concepts and proposals presented in this thesis are targeted towards different heterogeneous CMP systems. The principal heterogeneous CMP we evaluate consists of a four core ACMP with one large core and three small cores shown in Figure 3.2. The cache structures for both the large and small core are the same for most of the studies presented in this thesis unless specifically mentioned otherwise. The main differences between the large and small core are the instruction window sizes, ld/st queues, and branch predictors which help to extend or restrict OoO execution. These differences result in approximately 3x performance of the large compared to small cores but at a cost of extra power and size requirements. Complementing a large powerful core with small more energy efficient cores is reflective of current state-of-the-art ACMP implementations describe previously in Section 2.1.1. Table 3.1 provides more detailed information of the ACMP configuration which will be used regularly (with occasional slight modifications) in the following chapters of this thesis. Based on simulations of this ACMP implemented using 45nm transistor technology, the total area equates to 190mm^2 with the last level cache (LLC) taking up about 60mm^2 or nearly one third of the chip.

3.3 Benchmarks

Two different comprehensive benchmark suites are utilized in the experiments presented in this thesis. The first is SPEC2006 [47] which includes benchmarks that are both computational and memory intensive with large working sets and memory footprints. The SPEC benchmarks are single-threaded applications and for our experiments, we execute several of the applications concurrently. The second benchmark suite used is SPLASH-2 [91] which is composed of a set of multi-threaded applications and kernels. Each SPLASH-2 benchmark is typically run by executing various parallel threads on the CMP concurrently. Using both SPEC2006 and SPLASH-2 enables the use of both application and thread level parallelism for validating proposals to maximize CMP resource utilization. Both benchmark suites are compiled using gcc 4.8 with the -O3 optimization level flag. Their instruction traces are gathered using Pin [15] (which is integrated in Sniper) while executing on an Intel(R) Core(TM) i7-4600U CMP processor [18] running an unmodified Ubuntu 14.04.3 LTS (GNU/Linux 3.13.0-61-generic x86_64) operating system using the reference input datasets.

3.3.1 SPEC2006

SPEC2006 is a widely used benchmark suite within the computer architecture research community to evaluate the performance and energy results of different computer system innovations. The benchmarks are divided into two subsets (SPEC2006FP and SPEC2006INT) dependent upon whether they perform large amounts of floating point calculations. If floating point operations account for more than 30% of all dynamically executed instructions, then the application is categorized into the FP subset, and INT if this is not the case. The FP subset includes seventeen benchmarks written in C, C++, and FORTRAN while the INT subset consists of twelve benchmarks written in C and C++ . In the experiments presented in this thesis, three applications (*dealII, wrf, sphinx3*) did not compile for our platform, therefore, only twenty six of the twenty nine total SPEC benchmarks have been used.

Table 3.2 lists these twenty six benchmarks and their dynamic instruction count and instruction mix. Several interesting observations can be made from the statistics given in the table. Firstly, the instruction counts for most of the benchmarks run into the trillions and are representative of the computational intensities found in contemporary applications. Secondly, the percentage of branches executed by the benchmarks differs between the INT and FP applications. In contrast to most INT benchmarks which consist of around 20% branch instructions (with the exception of *456.hmmer* and *464.h264ref* having only 7% branches), the FP benchmarks typically consist of about 5% branches or lower (with the exception of *450.soplex* and *453.povray* resulting in approximately 15% branches). Lastly, the benchmarks typically consist of a large average dynamic basic block size which may contain substantial instruction level parallelism (ILP) that could be exploited using OoO.

3.3.2 SPLASH-2

SPLASH-2 (1995) is older than SPEC2006 but consists of a mixture of eight multi-threaded applications and four kernels which are designed to solve graphics, engineering and scientific problems. Table 3.3 provides an overview of the SPLASH-2 benchmarks utilized in the studies of this thesis with *volrend* being the only benchmark not used due to compilation issues.

Fig. 3.3 shows the execution time breakdown of the SPLASH-2 benchmarks. Since these benchmarks are multi-threaded, they spend significant portions of execution time on synchronization and memory accesses. The synchronization bottlenecks are due to critical sections or barriers that cause significant thread latencies while waiting for a lock. The low percentage of branches found in applications such as cholesky, fft, and ocean provide for sizable basic block and ILP opportunities. Applications with high instruction level parallelism (ILP) and thread level parallelism (TLP) are prime candidates for ACMP architectures blending OoO and in-order style cores. Multi-threaded applications such as those in SPLASH-2 are suitable benchmarks for validating heterogeneous schedulers needing to manage numerous concurrently executing threads.

Benchmarks	Inst. Count (Billion)	Branches (%)	Loads (%)	Stores (%)
400.perlbench	2,378	20.96	27.99	16.45
401.bzip2	2,472	15.97	36.93	12.98
403.gcc	1,064	21.96	26.52	16
410.bwaves	1,178	0.68	56.14	8.08
416.gamess	5,189	7.45	45.87	12.98
429.mcf	327	21.17	37.99	10.55
433.milc	937	1.51	40.15	11.79
434.zeusmp	1,566	4.05	36.22	11.98
435.gromacs	1,958	3.14	37.35	17.31
436.cactusADM	1,376	0.22	52.62	13.49
437.leslie3d	1,213	3.06	52.3	9.83
444.namd	2,483	4.28	35.43	8.83
445.gobmk	1,603	19.51	29.72	15.25
450.soplex	703	16.07	39	7.75
453.povray	1,220	13.23	35.4	16.1
454.calculix	3,041	4.2	40	10
456.hmmer	1,589	7.1	47.5	17.8
458.sjeng	2,400	21.5	27.4	14.65
459.GemsFDTD	1,450	2.5	54.15	9.7
462.libquantum	3,950	15	34.6	10.8
464.h264ref	4,230	7.5	41.63	13.14
465.tonto	3,024	5.05	45	12.8
470.lbm	1,800	0.87	38.3	11.8
471.omnetpp	782	21	35.1	20.19
473.astar	1,153	16	41.12	13.9
483.xalancbmk	1,247	25.9	34.1	10.19

Table 3.2: An overview of the instruction counts and mix for the utilized SPEC2006 benchmarks.

Benchmark	Input problem size	Description
barnes	65,536 particles	models interaction of 3D bodies
choleskey	tk29.O	factoring of a sparse matrix
fft	4,194,304 data points	six step FFT
fmm	65,536 particles	models interaction of 2D bodies
lu.count	1024x1024 matrix, 64x64 blocks	factoring dense matrix
lu.ncout	1024x1024 matrix, 64x64 blocks	factoring dense matrix
ocean.count	514 x 514 grid	models ocean movements
ocean.ncount	514 x 514 grid	models ocean movements
radiosity	large room	computes distribution of light
radix	8,388,608 integers	integer radix sort kernel
raytrace	car	3-D rendering
water.nsq	4096 molecules	models water molecules
water.sp	4096 molecules	models water molecules

Table 3.3: A list of the SPLASH-2 benchmarks used in our experiments, workload sizes, and descriptions.

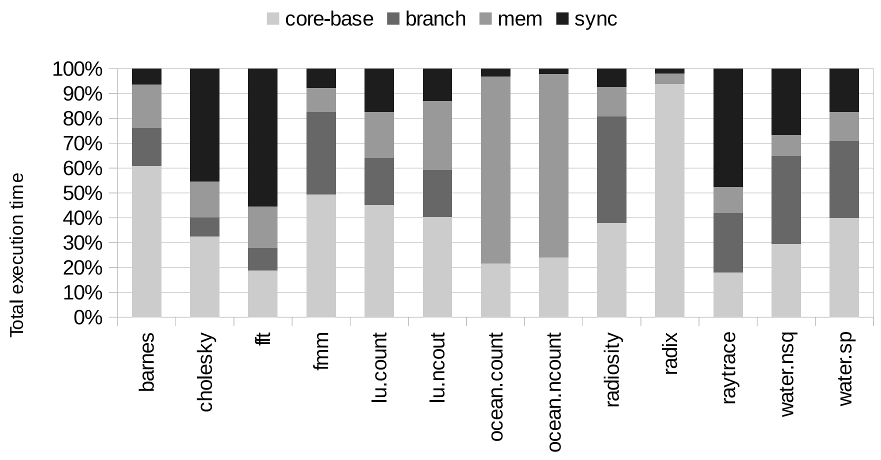


Figure 3.3: Execution time breakdown of SPLASH-2 benchmarks on a four core ACMP. This is akin to CPI stacks showing distribution of computational effort spent on different CPU operations. Illustrates the total time spent in computation (core-base), branches (branch), memory accesses (mem) and synchronization (sync) events. Synchronizations events include barriers, locks and pauses.

3.4 Schedulers

Comparative performance

The studies presented in this thesis make use of four different schedulers (CFS, Fairness-aware, HRRS, and TLSS) described previously in Section 2.1.4. Depending on the study and implementations proposed, each scheduler uses either 4ms or 1ms scheduling quantum which is within typical ranges and manage anywhere from four to twenty six concurrently executing threads. To account for context switch overheads due to architectural state swapping, we apply a 1000 cycle penalty which is consistent with the value utilized in the studies [72, 71]. The additional latency costs from needing to reload data into the caches after every context switch is captured by the simulation. The cache latency cost can be orders of magnitude higher than the architecture state swapping overheads and depends upon the workload size and memory access behaviors.

For the baseline scheduler, we sought to use a conventional scheduler readily used in industry, personal, and research systems. For this reason, we chose to use a scheduler based on the Linux Completely Fair Scheduler (CFS) [54]. This scheduler maps all threads to the cores based on a round robin scheme in order to balance the workloads between the cores. The threads remain assigned (i.e., pinned) to their designated core until completion, unless another core becomes idle, and are basically never swapped between cores. The scheduler is therefore unable to identify optimal mapping schemes that best take advantage of the heterogeneous resources. To select which threads to execute next on a particular core, the scheduler chooses the thread that has had the least amount of execution time from the pool of threads corresponding to that particular core.

The second scheduler used is based on the Fairness-aware scheduler [56]. Referred to in the studies as either the Fair or Fairness-aware scheduler, it works in a similar fashion to the CFS but with one critical difference. Instead of distributing the threads evenly between the cores and pinning them to these cores, the Fair scheduler distinguishes the differences between core types and selects the threads that have made the least amount of progress (in terms of execution time or instruction executed). It then assigns the thread with the least progress of this subset to be assigned to the most powerful core. For instance, given eight threads that need to be executed on the ACMP described in Section 3.2, the Fair scheduler will select four threads to execute next quantum (which corresponds to the number of available cores), and assigns the thread with the least progress made to run on the large core and the other three to the small cores.

The third scheduler used is a hardware implementation of the round robin scheduler [72]. This scheduler (referred to as HRRS or round robin in the studies) uses a round robin scheme to select and map the threads onto the ACMP cores. Every quantum, a new thread is assigned to the large core (if there is more than one thread executing) thereby providing fair execution time for all threads on the large core.

Finally, the fourth scheduler used is the thread lock-section aware scheduler (TLSS) [71]. This scheduler prioritizes assigning to the large core threads which return from system to user mode. If no threads experience this transition, then TLSS defaults to using the method of the Fair scheduler.

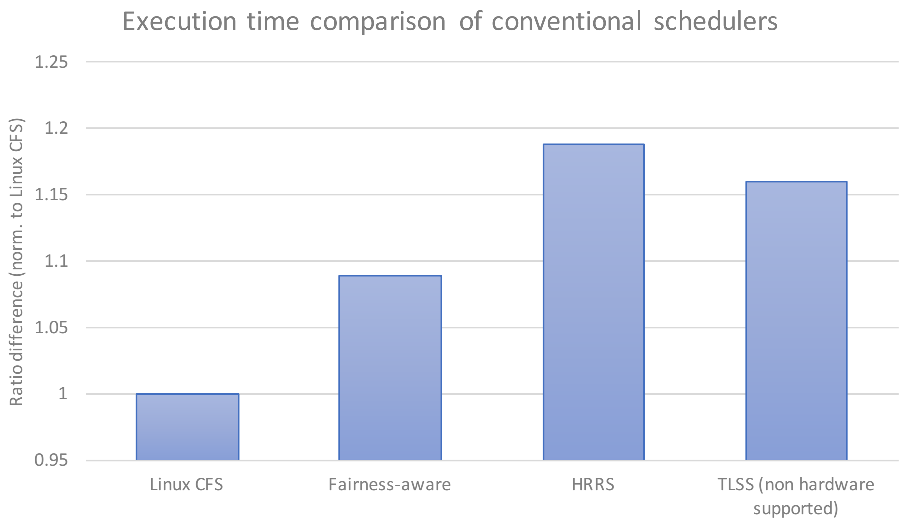


Figure 3.4: A performance comparison of the Linux CFS, Fairness-aware, Hardware Round-Robin (HRRS), and TLSS schedulers normalized to the Linux CFS when running the SPEC2006 and SPLASH-2 benchmark suites on a 1 large and 3 small core ACMP. Higher speedup numbers are better.

Figure 3.4 presents a comparison between the four different schedulers when simulated on ACMPs baed on the system described in 3.2 running both SPEC2006 and SPLASH-2 benchmark suites. There are clear performance enhancing opportunities of using the heterogeneous schedulers over the conventional Linux OS pinned scheduling technique with speedup benefits between the range of 9% (Fairness-aware), 19% (HRRS), and 16% for the non hardware supported TLSS scheduler implementations. These benefits can also lead to energy savings since they require less time using power hungry system resources.

3.5 Artificial neural networks (ANNs)

The proposals in Part II of this thesis make use of ANN-based performance predictors whose output is used by the scheduler to identify an optimal mapping scheme. The predictors are used to either estimate what the resulting IPC will be of a thread on a particular core type or to predict the total system IPC for a particular thread to core mapping combination.

Before describing the different ANN architectural properties investigated, it is important to mention how the data used to train and predict the ANNs are gathered. These data can be either statically or dynamically collected and consist of different statistics describing the execution of the threads on the different cores. Performance counters provided by Sniper, but also feasible in physical hardware, are used to gather the statistics (e.g., IPC, instruction mix, and cache accesses) after ever scheduling quantum. The set of statistics for each quantum form one data sample. The exact statistics chosen are described in added detail in the relevant studies.

Every hyper-parameter of each ANN (i.e., the number of hidden units, layers, training and error functions, and activation functions) used in this thesis has been carefully studied and tuned to balance prediction accuracy and overheads. The ANN implementations within the scope of this thesis range from 1-5 of hidden layers and 6-25 hidden units per layer. The activation functions are either sigmoidal or rectified linear while a stochastic gradient descent algorithm is used for training.

The latency overheads vary per ANN implementation and are quite minimal. For example, an ANN with 9 inputs, 1 hidden layer with 6 units, and 1 output unit requires 150 floating point and 80 memory operations for each prediction it makes. Conservatively assuming it takes 20 cycles to complete each of the 230 operations, this results in 4,600 cycles. The scheduler is called every 1M cycles (assuming the CPU clock is set at 1GHz and the scheduling quantum occurs every 1ms) which means that the computational overheads of this example ANN predictor are approximately 0.5% per prediction. Implementing the ANNs also requires overheads in terms of modifications to the OS and/or hardware support (depending on the choice of implementation). The ANNs models proposed in the studies in this work require minimal changes to the OS scheduler, the most important being the the thread to core mapper.

To implement the ANNs, the studies make use of either the Matlab machine learning toolkit [67] or Python and the machine learning library scikit-learn [78]. Both of these toolkits provide ample support for implementing, training, optimizing, and online prediction using different ANN configurations. For several of the studies in this work, the scheduling modules in the Sniper simulator are modified to make use of the ANN implementations.

3.6 Measuring Error and Results

Several different metrics are used to evaluate the proposal described in this thesis. Execution time (in seconds) or instructions per cycle (IPC) is used to quantify system performance. System IPC is the result of averaging the sum of the IPC results from the different cores during each quantum. This assumes that all cores run at the same clock frequency, but may be easily adjusted for cases where the cores run at different frequencies. Speedup is measured by dividing the baseline execution time by the new proposal's execution time. The power budgets of a given architecture are measured in Watts (W) and multiplying this by the execution time provides the energy consumption results given in Joules (J). Multiplying the energy results with execution times produces the Energy-Delay-Product (EDP) which is a popular metric for measuring architectural efficiency. Chip size is measured in millimeters squared (mm^2) and is useful for comparing architectures that must fit within certain size constraints. Sniper provides easy access to the simulation statistics necessary to calculate these metrics.

For measuring the accuracy of the ANNs, different metrics are used. These metrics evaluate the accuracies and correlations between the predicted IPC values (for a particular mapping scheme) compared to the real observed IPC values. The percentage error metric shown in Equation 3.1 expresses the error rates in terms of misprediction percentages, or in other words, how far off as a percentage was the predicted IPC from the observed IPC. Where in our case y is the predicted IPC and t is the target (i.e., observed) IPC value for quantum i and n is the total number of quanta (i.e., samples). The mean squared error metric which is shown in Equation 3.2 characterizes the error as the average of the sum of the squared differences between the predicted and target (observed) IPC values. Lastly, an R^2 coefficient is calculated in Equation 3.3 that described the error as a correlation between the predicted and target IPC values based on a regression sum of squares (u) and the residual sum of squares (v).

$$\begin{aligned} error_i &= \frac{|y_i - t_i|}{t_i} \\ \mu_{error} &= \frac{1}{n} \times \sum_{i=1}^{n} error_i \end{aligned} \tag{3.1}$$

$$MSE = \frac{1}{n} \times \sum_{i=1}^{n} (y_i - t_i)^2 \tag{3.2}$$

$$\begin{aligned} R^2 &= 1 - \frac{u}{v} \\ u &= \sum (y_{true} - y_{pred})^2 \\ v &= \sum (y_{true} - \bar{y}_{true})^2 \end{aligned} \tag{3.3}$$

Part I

Preliminary Studies on the Potential of Heterogeneous Architectures

The objective of this part of the thesis is to examine the opportunities and limits present in current CMP and scheduling designs. We believe this discussion presents a realistic starting point from which to acquire several key insights instrumental for advancing towards the future of heterogeneous architectures that was outlined in Section 2.2. Each of the following two chapters provides separate investigations into analyzing CMP design tradeoffs and improving performance and energy efficiency.

Chapter 4 highlights how architects can optimize the energy efficiency and size of ACMP systems by using novel memory architectures in conjunction with a heterogeneous scheduler. This approach offers a method to tailor an ACMP system to provide suitable performance, energy efficiency, and size for very demanding mobile devices. We propose three alternative cache configurations and examine their effects on system performance and processor size when executing applications concurrently. Our results show that adopting an alternative cache hierarchy together with a scheduler targeting asymmetrical systems can lead to substantial energy savings of over 17%, power reductions of over 5%, and over 19% reductions in physical size while still outperforming execution times achieved with conventional operating system schedulers on a CMP with larger caches by over 10%.

Chapter 5 presents a hardware implementation of the Thread Lock Section-aware Scheduling (TLSS) scheduling mechanism described in [71]. The TLSS algorithm helps to identify multi-threaded application bottlenecks such as thread synchronization sections and complements the Fairness-aware Scheduler method. The work we present is to our knowledge the first hardware supported implementation of TLSS that is energy attentive and can be applied to both asymmetric and symmetric CMPs. It achieves an average performance gains of 10.9% compared to the state-of-the-art Linux OS Scheduler when applied on an Symmetrical Chip Multi-Processor (SCMP). At the same time, it is 81% more EDP (energy-delay product) efficient when applied on an Asymmetrical Chip Multi-Processor (ACMP) and compared to the Linux OS Scheduler on an SCMP, where ACMP and SCMP take relatively the same chip area.

Taken together, these two chapters study conventional CMP and scheduler designs and investigate techniques that highlight the architectural opportunities and restrictions present in current designs.

4 Extending the flexibility of ACMPs for mobile devices using alternative cache configurations

The focus of the work in this chapter is to highlight how heterogeneous schedulers can provide CMP architectural flexibility in terms of cache configurations. Alternative cache configurations can help to optimize the energy efficiency and size of ACMP systems. This approach offers a method to tailor an ACMP system to provide suitable performance, energy efficiency, and size for very demanding mobile devices. We propose three alternative cache configurations and examine their effects on system performance and processor size when executing applications concurrently. Our results show that adopting the most novel of these three configurations in conjunction with a scheduler targeting asymmetrical systems can lead to substantial energy savings of over 17%, power reductions of over 5%, and over 19% reductions in physical size while still outperforming execution times achieved with conventional operating system schedulers on a CMP with larger caches by over 10%.

The contributions in this chapter include:

- Extending the flexibility of ACMPs for mobile devices using an alternative cache configuration technique. We introduce three alternative cache configurations (Larger, Asymmetric, and Distributed) for an ACMP featuring one large core and three small cores. While all three configurations achieve benefits, the Distributed approach is shown to be the most novel and beneficial for ACMP systems that have frequent context swaps.

- Experimental results drawing from both the SPEC2006 and SPLASH-2 benchmark suites show that alternative cache schemes utilized in conjunction with a simple ACMP scheduler achieve notable energy (over 17%), power (over 5%), physical size (over 19%), and performance (over 18%) benefits over an ACMP containing a larger cache, and greatly outperforms an alternative frequency reduction technique.

4.1 Motivation

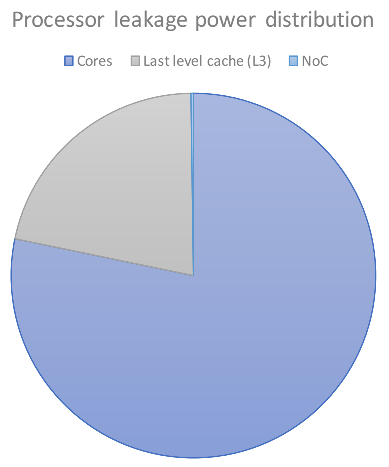
The diversity of devices and environments reflect the differing design priorities and choices architects make depending on the target market and usage expectations. For example, though a fitness tracking wristband and a tablet are both mobile devices, they differ in form factor, utilization, and expectations. While a fitness tracker can be expected to last for several days of constant usage running only a small selection of applications, a tablet may be expected to last a day or two but while executing a wider selection of applications much faster than a smaller device but still not as quick as a desktop or server. The main ideas motivating the study of this chapter reflect the need to decide how to balance the design decisions when constrained with specific performance, energy consumption, and area design parameters.

In this study, we focus on how a conventional ACMP system may be improved for both energy efficiency and size constraints using a dedicated scheduler in conjunction with reexamining the cache hierarchy. Doing so enables us to represent the potential beneficial options a processor architect may choose from in needing to adapt an ACMP processor to different mobile device and usage expectations.

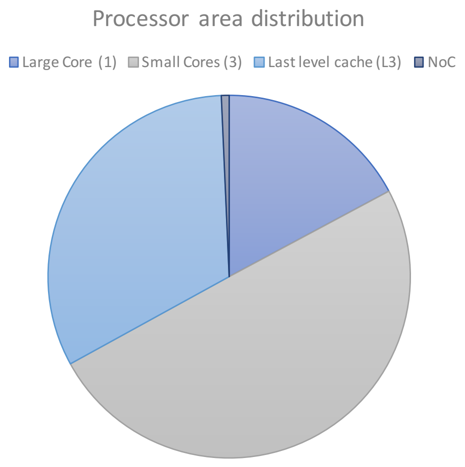
As described in Section 2.1.4, there are clear performance enhancing opportunities of using the hardware round robin scheduler (HRRS) over the conventional Linux OS CFS pinned scheduler technique for ACMPs. For SPEC2006 on our ACMP configuration, using the HRRS method results in execution speedup gains of nearly 16% and energy savings of almost 10% while requiring about 8% more power (due to hardware support overhead). Similar numbers are achieved when running the SPLASH-2 benchmark as it results in about 13% speedup gains and 7% energy savings while requiring 6% more power. The consistent benefits of the HRRS implementation over the Linux scheduler on the ACMP provides added flexibility in the architectural design choices within the processor. Specifically, our work seeks to improve the energy efficiency of ACMPs by balancing the execution speedups achieved using heterogeneous schedulers such as HRRS with gains in power and energy efficiency through the use of alternative cache hierarchy configurations. These alternative organizations also help to decrease the footprint of the caches which are the largest elements on the physical chip. This chapter explores the performance benefits achieved with heterogeneous schedulers and the energy and size footprint of the conventional ACMP cache hierarchy.

4.1.1 Cache footprint

The cache hierarchy of the ACMP system when running both SPEC and SPLASH-2 consumes on average about 30% of the total energy and power budget of the processor. The last level L3 cache (LLC) alone consumes a significant chunk of the processor's energy, power, and size budget. Using measurements taken with McPAT [89] when running the 4 core ACMP, the LLC is responsible for on average about 10% of the total execution energy. Leakage power, however, is costly since it is power that is dissipated even when the components are not active and hence is an important metric to consider when intending to design energy efficient chips. For this ACMP, the leakage power represents upwards of 25% of the total peak power. As shown in Figure 4.1a, the LLC is responsible for 22% and the four cores make up 78% of the processor's subthreshold leakage power. Figure 4.1b highlights how the four cores, which include the L1 and L2 cache structures, take up about 67% of the total chip area while the LLC takes up a substantial 32%, and the interconnection network (NoC) a mere 1%. For our simulations based on a 45nm transistor technology, the total area of the processor corresponds to 190mm^2 with the LLC taking up about 60mm^2. Along with other alternatives such as frequency reduction, altering the cache configurations is a viable path towards maintaining the speedup gained using HRRS while reducing power and size requirements and increasing energy savings without modifying the internal microarchitecture of the computational cores.



(a) Leakage power distribution of the ACMP.



(b) Area distribution of the ACMP.

Figure 4.1: The subthreshold leakage power and chip area distributions of a 1 large and 3 small core ACMP including results for the L3 last level cache (LLC) and network on chip (NoC).

4.2 Alternative cache configurations

In order to balance the performance gains achieved by ACMP schedulers with size, energy, and power efficiency benefits for the overall ACMP system, we focus on reorganizing the existing cache hierarchy shown to have substantial size and power footprints. The alternate configurations we have proposed and evaluated have been configured to eliminate the large last level cache (LLC) and reorganized in order to mitigate the resulting increase in total DRAM accesses.

The loss of a shared LLC results in smaller processor area but longer average memory access latencies since more memory accesses will end up reaching DRAM. Additionally, context swaps may incur heftier penalties since the working set sizes of the active threads may not be fully contained or easily transferred between L2 caches. For example, a recently swapped thread may request the data that is still stored in its previous L2 and when it arrives, it may evict the data from the current L2 which could be requested by another thread. In the case of a shared LLC, this evicted data would still reside in the LLC, but with no shared LLC, any requests for this evicted data will have to reach the DRAM. Conversely, the amount of time it takes for a memory request to reach DRAM will slightly decrease by the amount of time it used to take to perform a tag access on the original L3 cache.

Each of the four cores in every ACMP cache configuration use separate and private L1 data cache and L1 instruction caches. For L2 and L1 cache details, including tag/data access latencies, and coherency protocol refer to table 3.1.

1. **Baseline** : The baseline configuration consists of a four core ACMP (one large core and three small cores defined in Section 4.3.1) where each core has a private L1 (32KB) and L2 (256KB) data cache and a shared L3 last level cache (LLC) of 8MB.

2. **Larger** : The simplest approach, this alternative configuration alleviates the loss of the L3 cache by increasing the size of each L2 cache from 256KB to 512KB. Though doubling the overall size of each L2 increases the physical size of each core, we have chosen to use this approach to demonstrate the effects of the most simplistic and intuitive approach at increasing L2 cache hits without resulting in significant energy/power penalties.

3. **Asymmetric** : This heterogeneous cache configuration keeps the size of the small cores' L2 caches steady at 256KB but increases the size of the large core's L2 cache from 256KB to 1MB. Though one of the characteristics of the large core is that it is capable of sustaining more outstanding misses than the small cores, the extra quantity of cache allows for more data from potentially other threads to be locally available which can enable recently swapped threads to avoid some cache warmup and run faster. This increase in the large L2 cache size and subsequent L2 large to small cache ratio was chosen as a reflection of the large core to small core size and performance capabilities (i.e., the large core is configured to be generally between three to four times more powerful than the small cores).;

4. **Distributed** : This distributed cache proposal intends to emulate a larger shared cache in order to alleviate the adverse memory latency effects caused by eliminating the shared LLC. Similar to the first alternative approach, this proposal doubles each of the cores' L2 caches to 512KB but instead of each L2 being private to each core, they are distributed such that every core can access every L2 cache, albeit with non uniform cache access latencies. Unique to this cache organization, this scheme forbids data to be replicated across L2 caches. Using this method, an L1 miss from core 0 is sent to core 0's L2, if it misses again, then it is sent to core 1's L2, and if it misses there then it is sent to core 2's L2, then to core 3's L2 upon which if it still misses it is sent to DRAM. Data is not moved between physically separate L2 caches and no coherency mechanism for the L2 needs to be utilized. Every access to the different L2's will result in different access latencies due to the varying distance from core to cache as well as the extra latency penalties for each L2 miss. (e.g., the time for core 0 to access core 1's L2 will be higher than accessing its own L2 but lower than accessing core 3's L2). This is similar to approaches taken in conventional non-uniform cache architectures (NUCA) [58] but the L2 caches are accessed in order. A memory management unit makes sure that two identical memory accesses from separate cores are not replicated in both of their L2's. A new data line is stored in the L2 structure of the core that first requested the line. The distributed and non-duplicate nature of this approach

eliminates the need for cache coherency management and also helps to reduce the costs of cache warm up and pollution effects after every context swap.

4.3 Methodology

The experimental setup of this work consists of the four cache hierarchy configurations (one baseline and three alternate proposals) to be tested using a fixed processor configuration running applications from two benchmark suites on a parallel multi-core simulator. In addition, these results are compared with an alternative scheme of running the baseline configuration at a lower frequency in order to reduce energy consumption.

4.3.1 Processor configuration

The processor that is used for all experimental runs in this work is a quad-core ACMP consisting of one large core and three identical small cores. Both types of cores are based on the Intel Nehalem x86 architecture running at 2.66GHz (2.1GHz is used for the lower frequency configuration). Each core type has a 4 wide dispatch width, but whereas the large core has 128 instruction window size, 8 cycle branch misprediction penalty, and 48 entry load/store queue, the small core has a 16 instruction window size, 14 cycle branch misprediction penalty, and a 6 entry load/store queue.

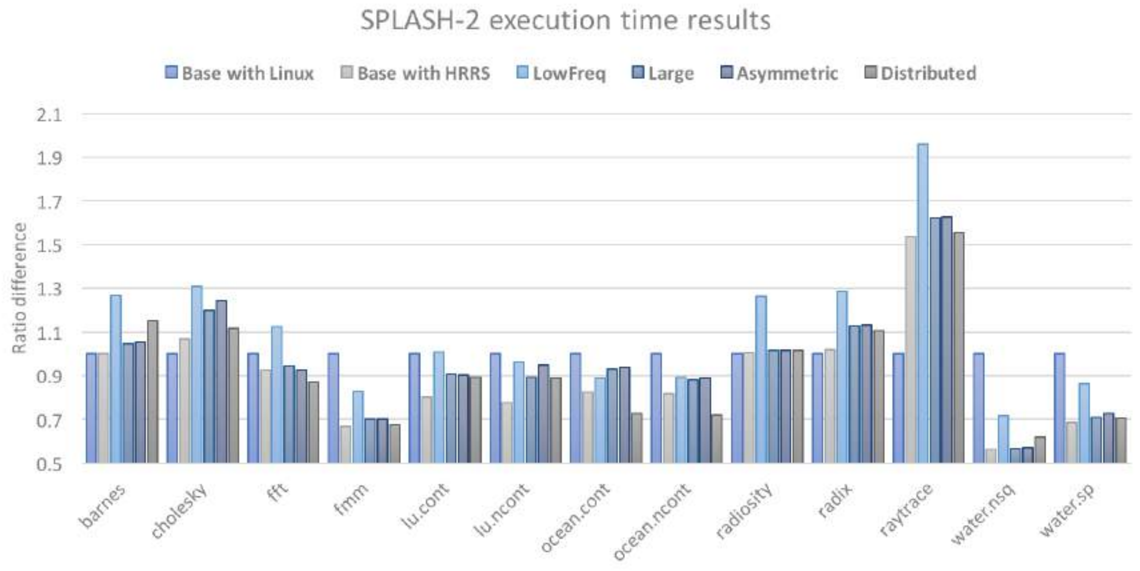
4.3.2 Benchmark execution

We have utilized the SPEC2006 benchmarks to run multiple instances of the single threaded applications concurrently on the system. We run each workload on the four simulated cores. The SPLASH-2 benchmark suite is configured to run with 4 parallel threads and is executed separately. Therefore, at any given time, there will be a maximum of four threads running from one application. All applications or threads are run from start to finish. In the case of having several applications or threads running at the same time, the threads that finish first are restarted such that the number of threads running at any one time on the system remains constant. Once the longest thread/application has been completed, the simulation is ended.

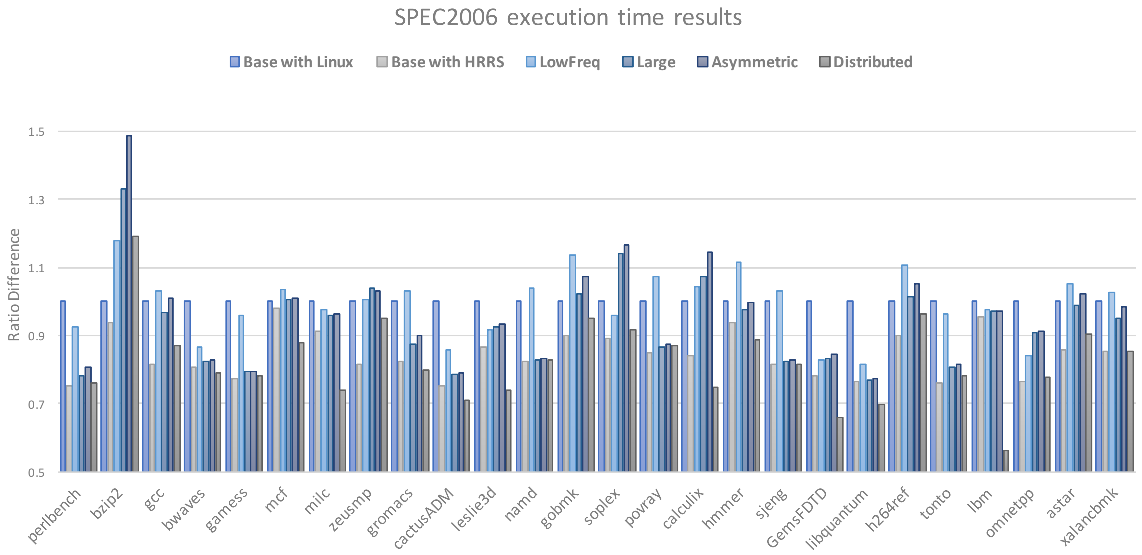
4.4 Experiments and evaluation

Figure 4.2 illustrates the execution time results obtained by running the different configurations on each application of both benchmark suites. The context swap overheads caused by how the different schedulers remap the threads onto the cores is included in the simulated results. The results are normalized to the ACMP system with the baseline cache configuration which utilizes an emulated Linux OS CFS scheduling policy (referred to as *Base with Linux* in the figures). As expected, the execution time of the alternative cache configurations were slightly slower than the baseline configuration due to the loss of a large shared LLC and extra DRAM accesses. However, though execution speeds may take slightly longer using these techniques, they are outweighed by significant gains in energy and space savings. Workload size, instruction length, and memory access patterns of the different applications result in different overheads due to frequent context swaps (the HRRS policy triggers context swaps every quantum) which explains deviations in application performance running on different cache configurations. For instance, while the frequency reduction scheme consistently underperforms in nearly all benchmarks, the *Distributed* cache configuration is able to attenuate context swap overheads in applications that suffer from heavier context swap penalties (e.g., *ocean* in SPLASH-2 and *lbm* and *calculix* in SPEC).

Figures 4.3a and 4.3b illustrate the average for execution time, power, and energy consumption of all SPEC2006 and SPLASH-2 benchmarks for each different cache configuration. The results are also normalized to the ACMP system with the baseline cache configuration utilizing the Linux OS CFS scheduling policy (*Base with Linux* in the figures). The bars in the charts representing the alternative cache configurations all utilize the HRRS scheduling policy. Figure 4.4 highlights the difference in processor sizes resulting from the component modifications of the separate cache configurations normalized to the baseline configuration (which includes a shared LLC).

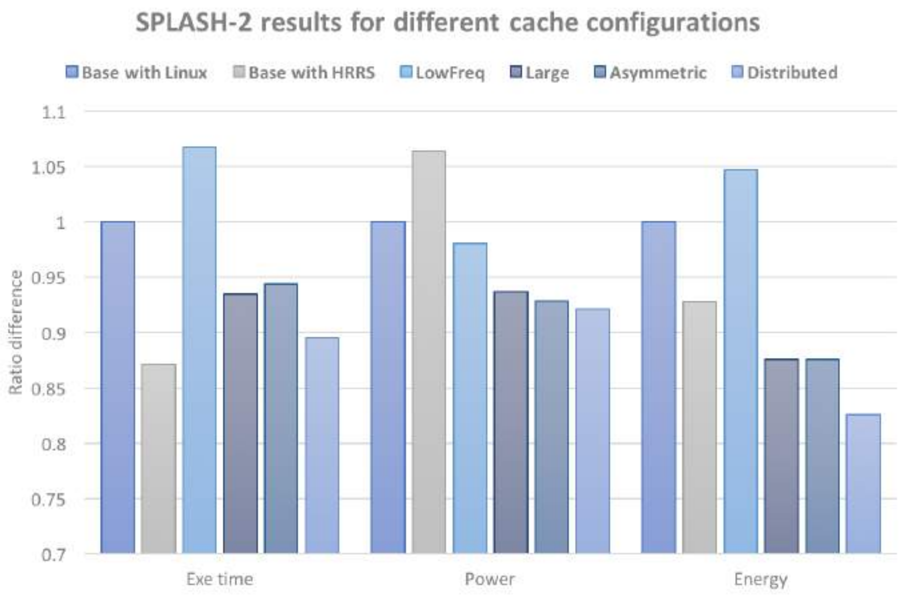


(a) SPLASH-2 results.

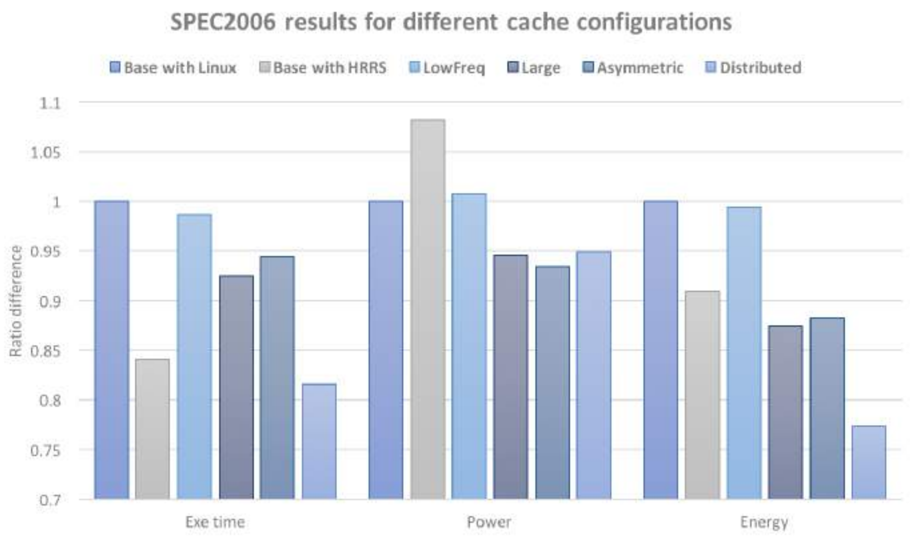


(b) SPEC2006 results.

Figure 4.2: Execution time performance results obtained by running all configurations on SPLASH-2 and SPEC2006. All configurations except *Base with Linux* use HRRS scheduling policy. Lower numbers are better.



(a) SPLASH-2 results.



(b) SPEC2006 results.

Figure 4.3: Average results for execution time, power, and energy consumption of all cache configurations on SPLASH-2. All configurations except *Base with Linux* use HRRS scheduling policy. Lower numbers are better.

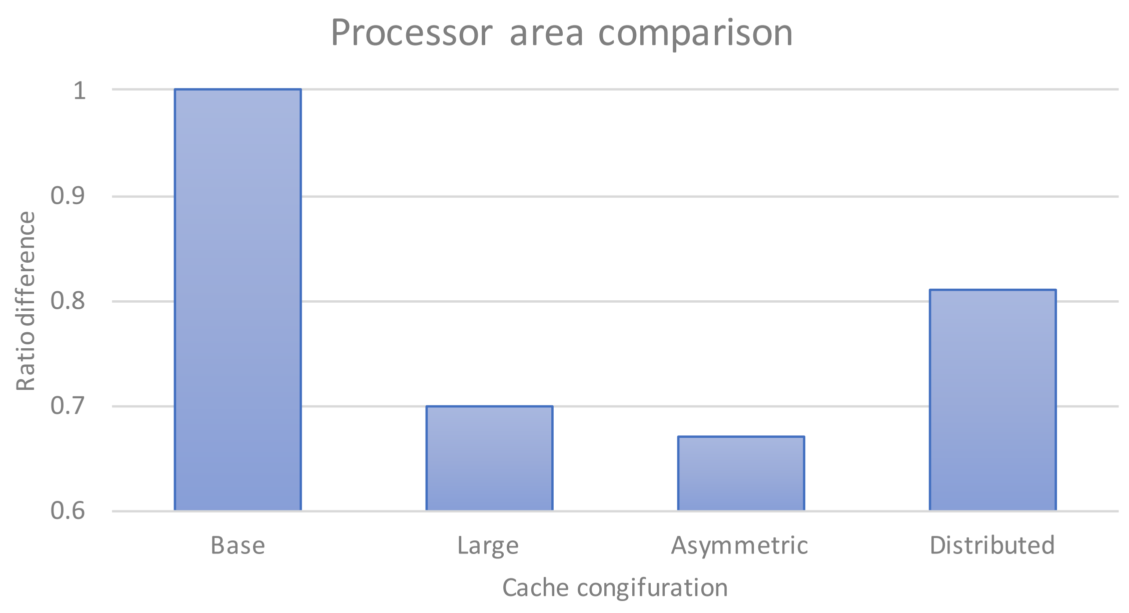


Figure 4.4: The resulting processor sizes of each alternative cache scheme normalized to the baseline configuration which includes a LLC. Note that a processor that is 70% the size of the baseline processor can be also read as a 30% reduction in processor size.

The *Large* cache configuration is able to alleviate extra misses to DRAM by being able to contain more data in the cores' L2 cache and achieves modest average speedup (7.5% for SPEC2006, 6.5% for SPLASH-2), power (5.5% for SPEC2006, 6.3% for SPLASH-2), and energy (12.5% for SPEC2006, 12.5% for SPLASH-2) gains compared to the baseline configuration using the Linux scheduler. Even though the size of the L2's were doubled, the elimination of the L3 results in a 30% reduction in processor size relative to the baseline ACMP configuration.

The *Asymmetric* cache configuration is an interesting alternative of how to allocate a limited cache budget to different core types. Since it has a larger amount of L2 cache compared with the small cores, the discrepancy between the performance of both core types becomes even greater and since all threads can benefit from its added effectiveness since they evenly share execution time on the large core. This configuration results in adequate average speedup (5.6% for SPEC2006, 5.6% for SPLASH-2), power (6.6% for SPEC2006, 7.2% for SPLASH-2), and energy (11.8% for SPEC2006, 12.4% for SPLASH-2) gains. In terms of size, this cache configuration, which is overall smaller than the *Large* cache configuration, results in a reduced processor size of 33%.

The *Distributed* cache configuration produces the most promising results. Since data may not be replicated between L2 caches but all cores may access each others' L2, this setup avoids many of the cache warmup and pollution overheads incurred in the other configurations during the frequent context swaps. The distributed cache also acts as a pseudo-shared L2 LLC such that is four times larger than any individual L2 cache thus increasing the probability of a memory access hit. Moreover, since the HRRS policy essentially provides each thread equal time on all cores, the penalty for non uniform L2 access is shared by all cores which results in what could be considered a shared L2 with shorter access time than the L3 found in the baseline configuration. The gains are significant showing average speedup (18.4% for SPEC2006, 10.4% for SPLASH-2), power (5.1% for SPEC2006, 7.8% for SPLASH-2), and energy (22.6% for SPEC2006, 17.4% for SPLASH-2) benefits. The differences in power savings from this scheme compared to the *Large* configuration are mainly due to the existence of a directory cache coherency mechanism in the *Large* configuration which is not needed in the *Distributed* scheme. The distributed configuration results in a total processor size reduction of 19% which is significant but is not as impressive as the other two configurations due to the need for extra resources to provide the extra inter-L2 communication and data access management needed to implement this scheme.

Conversely, the trivial approach of minimizing power and energy by utilizing a reduced frequency with the baseline cache configuration running the HRRS policy produces uninspiring results in speedup (1.4% for SPEC2006, -6.8% for SPLASH-2), power (-0.7% for SPEC2006, 1.9% for SPLASH-2), and energy consumption (1.6% for SPEC2006, -4.7% for SPLASH-2). In addition, the processor size is the same in this case as the baseline configuration processor.

While all three alternative cache configurations show promise in reducing size, power, and energy while maintaining speedup, the results from the *Distributed* configuration standout. Compared to the baseline configuration running the HRRS it is able to maintain to within 3% the speedup gains achieved by the HRRS running on the baseline configuration while requiring nearly 13% less power and consuming 11% less energy. Overall, the results show that adopting an alternative cache hierarchy in conjunction with a scheduler targeting asymmetrical systems such as HRRS can achieve energy savings of over 17%, power reductions of over 5%, and speedups of over 10% over a ACMP with more cache using a Linux scheduler found in conventional operating systems. In addition, a processor implemented with an alternative cache configuration can provide physical size reductions of 19% up to 33% compared with a conventional ACMP. The benefits of implementing an alternative cache configuration significantly increases the flexibility of an ACMP system and makes the option much more tempting for architects seeking to fit fast and energy efficient processors into ever smaller and more agile mobile devices.

4.5 Future work

In order to scale to larger many core systems, a cache configuration such as those proposed should be implemented in groups of cores. That is to say that a 16 core system with 4 large cores and 12 small should be broken up into 4 groups of 4 cores (1 large and 3 small). However, if these alternative cache configurations are to be scaled, evaluations on large many core systems including hundreds of cores should also be conducted. Exploring combinations of heterogeneous and distributed cache configurations for ACMPs appears to be a promising research avenue to pursue. More radical alternative cache hierarchy proposals combining asymmetric and distributed cache configurations will need to be studied. Furthermore, different ACMP scheduling strategies can be examined and incorporated into the simulations. Additional performance benefits gained from scheduling mechanisms offer more flexibility in designing alternative cache configurations that can enhance the system's energy efficiency and performance tradeoff.

4.6 Conclusion

In this study we have proposed three alternative cache configurations and shown how they outperform typical frequency reduction strategies in achieving better energy efficiency. Additionally, these configurations also result in substantial reductions in the physical size of the processor. By utilizing a simple hardware based round-robin scheduler in conjunction with an alternative cache configuration such as a distributed scheme, it is possible to achieve substantial energy savings of over 17%, power reductions of over 5%, and 19% physical processor size reductions while still outperforming the execution times achieved with conventional operating system schedulers on an ACMP with larger caches by over 10%. These benefits show that considering alternative cache configurations may be a feasible option architects can consider when choosing which processor designs to implement within demanding performance, energy, and size budgets such as in mobile devices.

5 Performance and energy efficient hardware-based scheduler for symmetric/asymmetric CMPs

The study in this chapter demonstrates the significant performance and energy advantages that can be attained by improving CMP scheduling. We validate this claim using a hardware based lock section-aware scheduler (TLSS) that is energy attentive and can be applied to both asymmetric and symmetric CMPs. It achieves average performance gains of 10.9% compared to the Linux OS CFS when applied on an SCMP and 81% more EDP (energy-delay product) efficient when applied on an ACMP and compared to a similarly sized SCMP. The contribution of this study is the hardware implementation of the TLSS approach (outlined as a software approach in the work [71]) and its application to both an SCMP and ACMP.

This chapter provides the following **contributions**:

- We discuss a hardware implementation of the TLSS scheduling policy [71] which is aware of lock sections and kernel to user code transitions. It is influenced by the Fairness-aware Scheduling and bottleneck identification techniques, thereby reducing thread serialization and improving parallel thread performance.

- We evaluate and analyze the performance, power, and energy consumption of the TLSS policy on an SCMP. It lowers total performance of the application by 10.9% (geometric mean) compared to the Linux OS Scheduler on an SCMP.

- We offer power and energy efficiency evaluation of the TLSS policy applied on an ACMP and compare it to the area equivalent SCMP system configurations as well as to a Fairness-aware Scheduler on the ACMP system, where TLSS is by 81% and 7.3% more EDP (Energy x Delay)-efficient respectively.

5.1 Thread lock section-aware scheduling (TLSS)

In the following subsections we briefly describe the TLSS policy, propose its possible application to an SCMP and discuss its hardware implementation.

5.1.1 TLSS algorithm

The intuition behind the TLSS method is to predict which threads are currently entering a critical section of code to accelerate it by sending it to execute on a large core. This helps to improve an application's overall performance by mitigating bottleneck effects for threads waiting on a lock. It uses knowledge about whether a thread has recently transitioned from kernel to user model (those that haven't been triggered by invoking the scheduler during the periodic scheduling quantum) to estimate that the thread is entering a critical section. To illustrate the inner workings of the TLSS approach, we will assume an x86 ACMP hardware containing one large out-of-order (OoO) core and three smaller and identical in-order cores. The operating system is provided an abstracted homogeneous hardware view comprised of four identical logical cores. The OS scheduler maps threads to the logical cores which enables the OS scheduling policies and implementation to be left unmodified.

While the OS scheduler maps threads to the logical cores at every software-quantum or other interrupts, the TLSS in turn maps the threads running on the logical cores to the physical cores as shown in Fig. 5.1 every hardware-quantum. In essence, TLSS can be viewed as mapping the logical cores that the OS sees and schedules threads onto, to the physical cores of the underlying hardware which actually execute the threads. Furthermore, the TLSS algorithm must produce a new scheduling scheme every hardware-quantum set by the hardware implementation

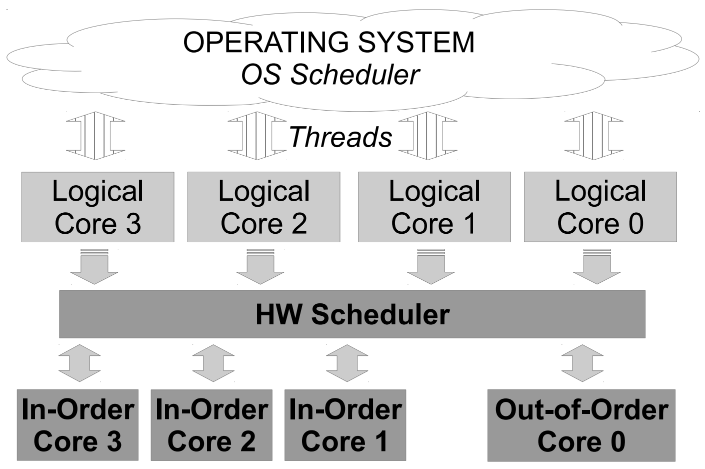


Figure 5.1: TLSS scheduling - All logical cores are the same while the large physical core is represented by Core 0 and the small physical cores are shown as Cores 1, 2 and 3.

(as opposed to the software-quantum which invokes the OS scheduler which happens less frequently - 1ms vs. 4ms). In order to minimize the amount of overhead in implementing the scheduling policy, the TLSS algorithm determines the next scheduling scheme to apply before the beginning of the next hardware-quantum.

The defining characteristic of the TLSS algorithm is its use of determining whether a core made a recent transition from executing kernel code to user code (an indication that an interrupt has occurred) to make scheduling decisions. Some of these transitions activate the OS scheduler which will swap the currently executing thread on a logical core for another thread in its ready queue. When this happens, the hardware context of the thread being swapped out must be saved and replaced by the context of the new thread chosen by the OS to be executed. Futex calls may be mainly in the userspace but will require kernel mode when dealing with contended locks and for requesting processes to be woken up and put on the wait queue. The futex calls requiring accessing the wait queue and needing to enter kernel mode are those focused on in this study. Therefore, by catching and utilizing the kernel-level to user-level execution transitions in the cores, we are able to localize some but not all of the critical sections of a thread without requiring any extensions to the ISA. Transitions due to clock interrupts are not considered for flagging kernel to user transitions and are ignored by the algorithm.

In order to determine these execution level transitions, the TLSS monitors the state of the cores' context control registers (the lower two-bits of the code segment descriptor that determine the current privilege level of the code executing) during the hardware-quantum and when a transition is detected a “transition-flag” (which requires reusing or adding one bit in hardware) is set. To select the scheduling scheme to apply for the start of the next hardware-quantum, the TLSS checks to see which of the “transition-flags” belonging to the four cores are set. If none are set, the TLSS proceeds to schedule based on a random selection scheme akin to the Fairness-aware Scheduler. If only the large core has its "transition-flag” set, then no swap is made. If only one small core has its "transition-flag” set, the logical core running on that small physical core will be swapped with the logical core running on the large physical core at the start of the next hardware-quantum even if the large core also had its “transition-flag” set. Lastly, if more than one of the small cores has its “transition-flag” set, one of their corresponding logical cores will be chosen at random to be swapped with the logical core running on the large physical core at the start of the next hardware-quantum. As a side-note, a core running system code at the moment of determining the next scheduling scheme cannot be selected to be swapped.

While Fairness-aware scheduling strives at achieving fairness by running each logical core thread on each physical core type for an equal amount of time, the TLSS scheduling instead tries to enhance these scheduling benefits by discovering and running the critical sections of code on the larger cores.

TLSS scheduling policy as described above can be easily applied to an SCMP. It would only require that the TLSS policy designates one core as a "large core", different then the other cores in the system although it is not. This may be useful (as detailed in section 5.2.3) due to the shared data amongst threads being located in the cache of the large core, thus minimizing cache misses.

5.1.2 Hardware implementation discussion

From the perspective of the physical hardware, the TLSS scheduling policy guarantees only two things. First, a thread will not occupy a large physical core for more than one hardware-quantum unless it is the only runnable thread at the end of the hardware-quantum. Second, if a thread reaches a critical section and must wait on a lock (operating system futex), the TLSS policy will attempt to promote the thread, upon acquiring the lock, to be scheduled on the large physical core to continue its execution. However, TLSS does not necessarily execute all the critical sections of a thread on a large core.

The difference between TLSS with state-of-the-art bottleneck acceleration found in BIS[48] and UBA[49] is that the TLSS does not require any ISA extensions that impact on the re-usability of code. Conversely, the ease of the TLSS implementation comes at the cost of not being able to catch all of the critical sections compared to the BIS and UBA policies, since it can only identify a transition from system to user-level code and does not have the precise knowledge that the thread entered into a critical section. Unlike to BIS and UBA, the TLSS scheduling technique does not facilitate hardware overheads in order to be able to store and restore the architecture state in the cores. TLSS is intended for x86 systems and hence utilizes the x86 hardware context switching mechanism, called Hardware Task Switching in the CPU manuals [43]. To use it, TLSS needs to tell the CPU where to save the existing CPU state, and where to load the new CPU state. The CPU state is always stored in a special data structure called a TSS (Task State Segment). To trigger a context switch and tell the CPU where to load its new state from, the far version of CALL and JMP instructions are used. The offset given is ignored, and the segment is used to refer to a "TSS Descriptor" in the Global Descriptor Table (GDT). The TSS descriptor is used to specify the base address and limit of the TSS to be used to load the new CPU state from. The CPU has a register called the "TR" (or Task Register) which tells which TSS will receive the old CPU state. When the TR register is loaded with an "LDTR" instruction the CPU looks at the GDT entry (specified with LDTR) and loads the visible part of TR with the GDT entry, and the hidden part with the base and limit of the GDT entry. When the CPU state is saved the hidden part of TR is used.

The hardware context switching mechanism can be used to change all of the CPU's state except for the FPU/MMX and SSE state. If the FPU/MMX and SSE state also needs to be changed during a context switch there are a few options. The data could be explicitly saved, or the CPU can generate an exception the first time an FPU/MMX or SSE instruction is used. With the second option, the exception handlers would save the old FPU/MMX/SSE state and reload the new state. We have used the second option since it may prevent this data from being changed when it is not necessary.

TLSS has hardware additions which include a "transition-bit" on every core and a separate unit with a vector that holds all of the "transition-bits", one counter and one decoder to facilitate round-robin mechanism. The size of these depends on the number of the cores in the system (e.g.,. for a four core system a 2-bit counter is needed). However, in contrast to the low additional overhead needed by TLSS, UBA requires the Lagging Thread Identification (LTI), the Bottleneck Identification (BI) and the Acceleration Coordination (AC) [49] while BIS requires a Bottleneck Table (BT) where each entry corresponds to a bottleneck, an Acceleration Index Table (AIT) augmented to the each small core, and a Scheduling Buffer (SB) added to each large core [48].

5.1.3 Hardware versus Software implementation

TLSS uses a similar approach as the Fairness-aware Scheduling method in that the operating system level scheduling is untouched and it maintains a consistent view of the underlying hardware. The hardware is able to provide the abstraction of a symmetric hardware to software while dynamically rescheduling threads among the cores in an asymmetric multi-core system [74]. Both of these approaches (TLSS and Fairness) may also be implemented at the OS level by extending the OS scheduler but the advantage of a hardware approach is that it provides finer granularity of the scheduling quanta and requires no changes to the OS code while minimizing scheduling overhead [55].

5.2 Evaluation

In this section we evaluate the TLSS approach application on an SCMP. We also offer detailed performance, power and energy comparison of the TLSS scheduler on an ACMP over Fairness-aware Scheduling [56] on an ACMP and Linux OS scheduler on an SCMP where ACMP and SCMP systems take approximately same chip area.

In our work we run each benchmark from the SPLASH-2 suite on the four simulated cores with each core capable of executing one hardware thread context at a time. We run one iteration for each application except for radix where we run ten consecutive iterations since radix execution time is one order of magnitude less compared to other applications of the SPLASH-2.

A context switch incurs a fixed cost for storing and restoring the architecture state [88] for which we presume a fixed 1,000 cycle penalty. We simulate the warming of the cache hierarchy during context switches for the workload migration. The study [55] has shown the migration overhead to be less than 1.5% across different types of single-threaded workloads, ranging form memory-intensive to compute-intensive, for a 4 MB shared LLC using a 1ms hardware-quantum.

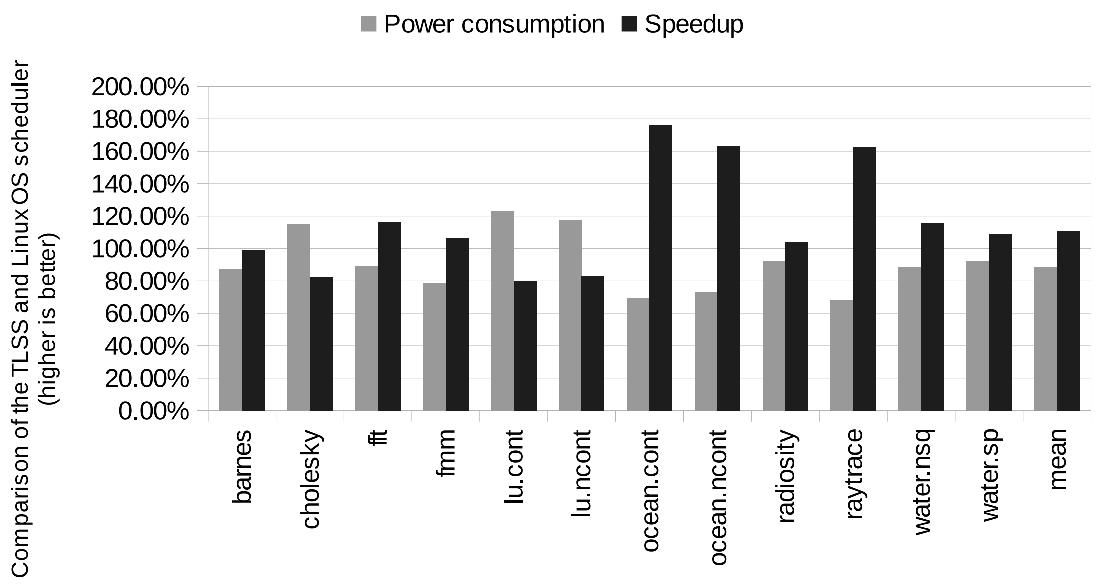


Figure 5.2: Performance and power consumption comparison of the TLSS and the Linux OS scheduler on an SCMP, consisted of four large cores, for the SPLASH-2 benchmark suite.

5.2.1 Performance per Watt Evaluation

Fig. 5.2 shows the performance benefit and power consumption of the TLSS over the Linux OS Scheduler on an SCMP with four large cores. The current practice used by contemporary operating system schedulers, for example as implemented starting in the Linux 2.6 kernel, is having threads pinned to the cores in a round-robin fashion starting from the large core to the small cores and repeating till all threads are assigned. The threads are then selected to be executed in a round-robin fashion on the respective core that they are pinned to.

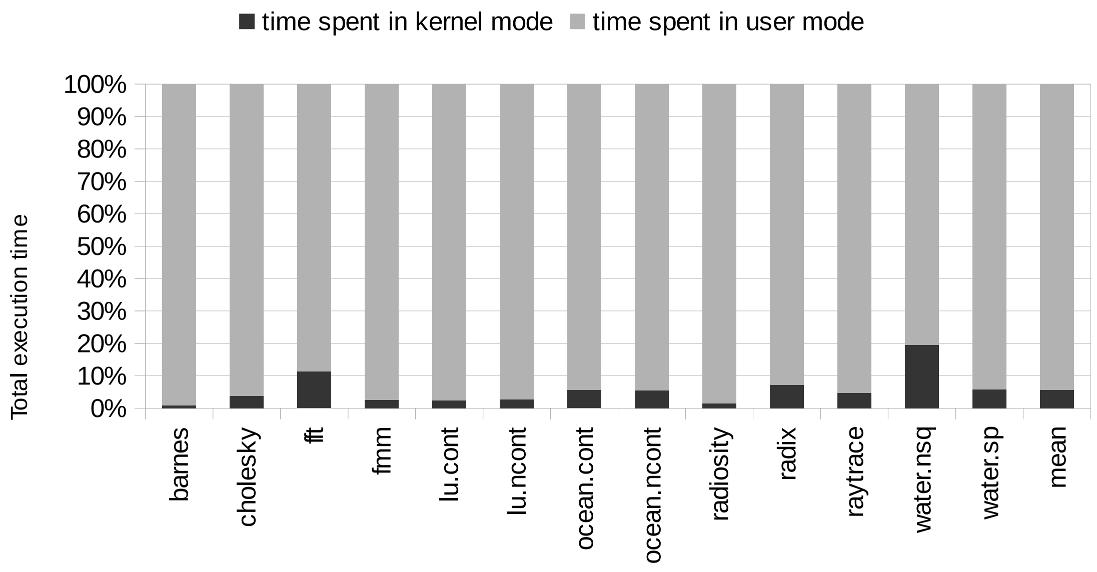


Figure 5.3: Distribution of the total execution time of the application.

The performance benefits arise form the fact that 1) the time spent executing user and kernel code averages to 94.32% and 5.68% respectively as represented in Fig. 5.3, 2) on average, 89.62% of the system calls in the non-sequential (i.e., when more than one thread is running concurrently) sections of the SPLASH-2 applications are caused by the synchronization (futex) calls and 3) the percentage of threads continuing execution on the core marked as large after exiting futex system calls is 64.22% for TLSS scheduler. The TLSS policy localizes more of the synchronization sections onto the core marked as large, therefore optimizing a program's execution by improving overall average performance of the applications. Fig. 5.4 represents the distribution of the total number of last level cache (LLC) accesses between the core marked as large and the cores marked as small for the

Linux OS and TLSS scheduler on an SCMP system. While the total number of the LLC accesses increases up by only up to 1.5% for the TLSS scheduler compared to the Linux OS scheduler due to the more context switches, the total number of the LLC accesses from the core marked as large is significantly larger for the TLSS than for the Linux OS scheduler. This is due to the fact that more synchronization sections are localized on the core marked as large.

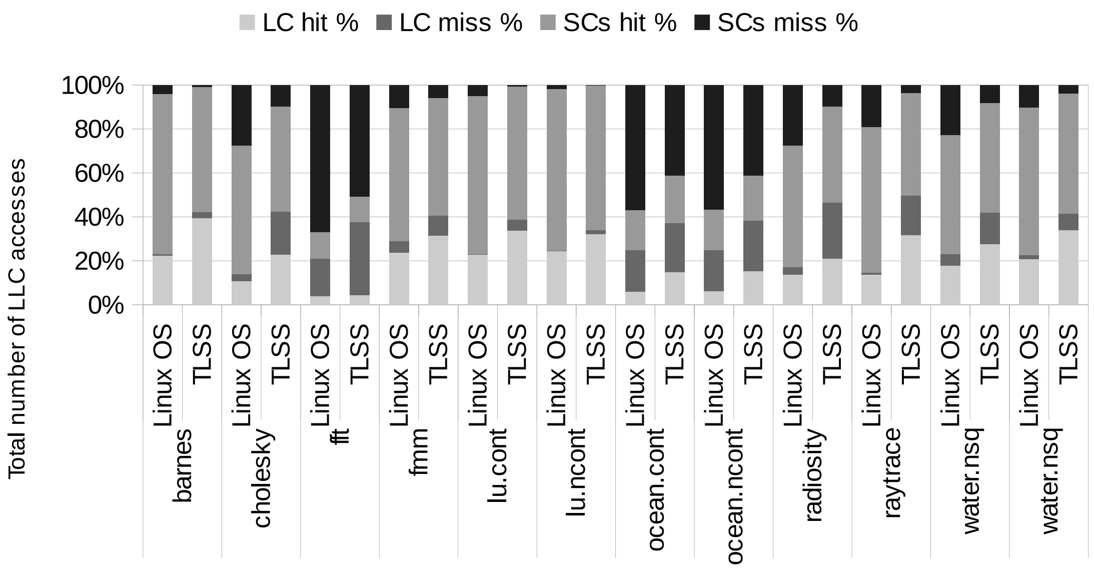


Figure 5.4: The LLC cache accesses breakdown for the core marked as large and cores marked as small cores of the Linux OS and TLSS scheduler for the SPLASH-2 benchmark on an SCMP. LC stands for large core and SC stands for small core.

While TLSS is able to maximize performance substantially on an ACMP system over Linux OS scheduler [71], it is also outperforming Linux OS scheduler on average by 10.9% on an SCMP system due to the better localization of the synchronization sections. Since threads in *cholesky* and *lu* applications are compute intensive, the overhead cost due to the frequent context switches triggered by the TLSS algorithm on the hardware threads (running on an SCMP system with four identical OoO cores) leads to performance degradation compared to the Linux OS scheduler which would keep the threads pinned to the cores.

TLSS minimizes the energy consumption in the cores marked as small but increases it on the core marked as large, which leads to only 10.55% more overall energy consumption compared to the Linux OS scheduler in an SCMP system. However, since TLSS also is able to speedup execution time for most workloads, our results show a 6.3% geometric average (and up to 22.5% for *ocean.count*) performance per watt improvement of the TLSS over the Linux OS Scheduler. Fig. 5.5 shows the performance per watt benefit expressed as IPC/Watt of the TLSS over the Linux OS Scheduler on an SCMP.

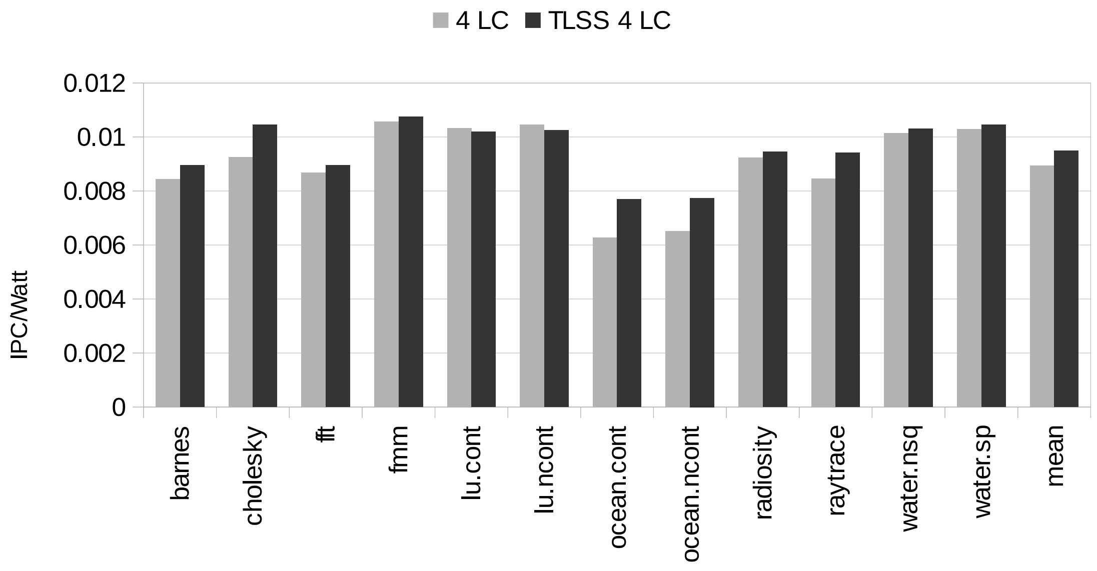


Figure 5.5: Performance per watt expressed as IPC/Watt comparison of the TLSS and the Linux OS scheduler on an SCMP consisted of four large cores for the SPLASH-2 benchmark suite.

5.2.2 Performance evaluation of TLSS on a many core ACMP

Results in Table 5.1 present the average cost of cache overheads in cycles for the system with shared and private last-level caches where workloads range from a few kilobytes to a few thousand kilobytes of data. This significant difference in the workload migration cost between shared and private LLC systems shows the TLSS

Table 5.1: Cost of workload migration (in cycles) during context switch for workloads ranging form a few kilobytes to a few thousand kilobytes.

	Shared LLC	**Private LLC**
Average	42985	301248
Min.	3406	15246
Max.	278112	3073444

mechanism would almost certainly suffer slowdowns for certain types of workloads when applied on system with private LLC. If each core in the system has a private LLC, the TLSS mechanism can not be adapted to overcome the drawbacks caused by the higher workload migration cost. On the other hand if a few cores share a portion of the LLC, TLSS can be modified so that it reschedules threads only among cores that share a portion of the LLC, while the OS scheduler keeps its property of keeping threads scheduled on the cores, or a group of cores, where they started execution.

Fig. 5.6 shows the speedups of using TLSS for 8/16/32 simulated cores with as many simulated threads per application, relative to the Linux OS scheduler. Each group of four simulated cores (1 large + 3 small) share a 4MB L3 cache. For simulated configurations of 8 (2 groups), 16 (4 groups) and 32 (8 groups) cores we get performance improvements of 24%, 30% and 34% respectively. On the other hand, BIS [48] proposal with 52 small cores (of similar configurations) having 3 large cores gives average performance benefits of 42%, while UBA [49] outperformes it by 8%. If we compare it to TLSS 32 cores (8 groups) configuration, BIS [48] appears to outperforms it by 8% and UBA [49] by 16% on average, with less large cores in the system with similar specifications. This comes as consequence of TLSS lightweight yet coarser grained approach which makes it unable to identify all bottlenecks in the multi-threaded application and send them only to be executed on the large cores.

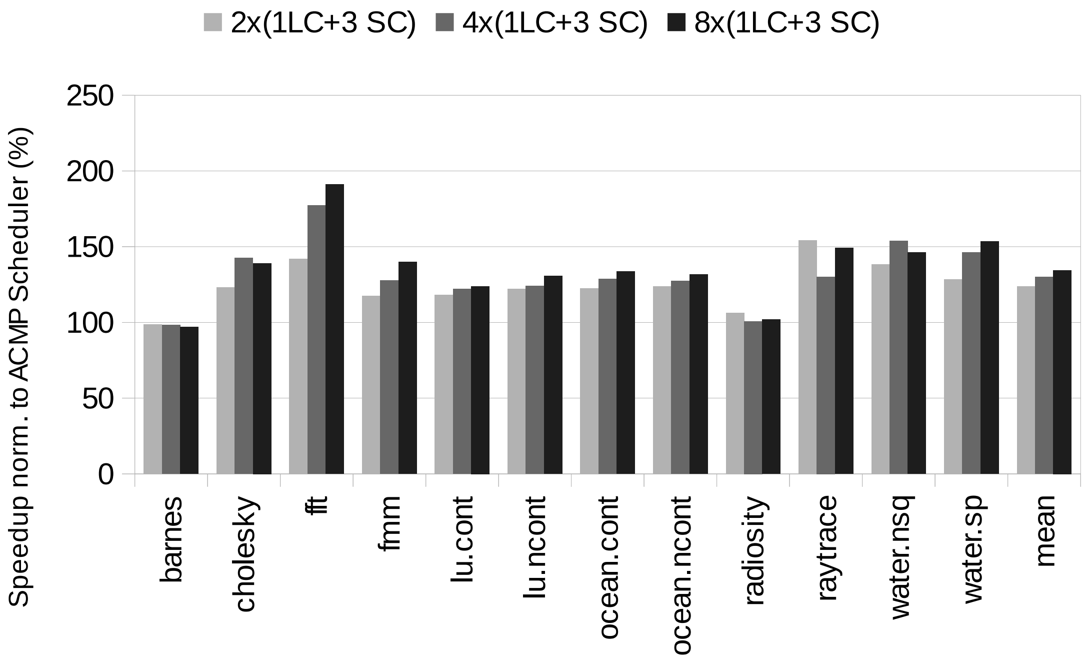


Figure 5.6: Speedup comparison of the TLSS and the Linux OS (ACMP) Scheduler for the SPLASH-2 benchmark suite in the private last-level L3 cache 8/16/32 cores system configurations where each group of one large and three small share a 4MB L3 cache.

5.2.3 Energy efficiency comparison of the different schedulers on an ACMP over an SCMP

We have used two commonly applied power metrics, in order to evaluate power and energy efficiency of different scheduling policies applied on the ACMP and SCMP systems that occupy approximately the same chip area. The first one represents total energy spent in the system during execution time. The second metric is the energy delay product $EDP = (D^2)P$ where P stands for average power of the system and D represents elapsed execution time. Total power multiplied by the delay (i.e., total execution time) gives the total energy consumed which if multiplied again by the delay gives the total energy delay product (EDP).

Since average power and energy consumption of the TLSS scheduling unit are negligible compared to the whole CMP system, several million times less, we did not consider them separately in this study. Fig. 5.7 and Fig. 5.8 represent normalized energy and EDP efficiency (less is better) of TLSS scheduler on an ACMP, Fairness-aware scheduler on an ACMP and Linux OS scheduler on an SCMP, where ACMP and SCMP occupy approximately the same chip area. TLSS policy is 13.5% more energy efficient and 7.3% more EDP efficient than Fairness-aware scheduling on an ACMP, while being 41% and 81% energy and EDP efficient respectively than Linux OS scheduler on an SCMP that occupies approximately the same chip area as the considered ACMP. Detailed execution time, average power and energy consumption results are presented in the Table 5.2.

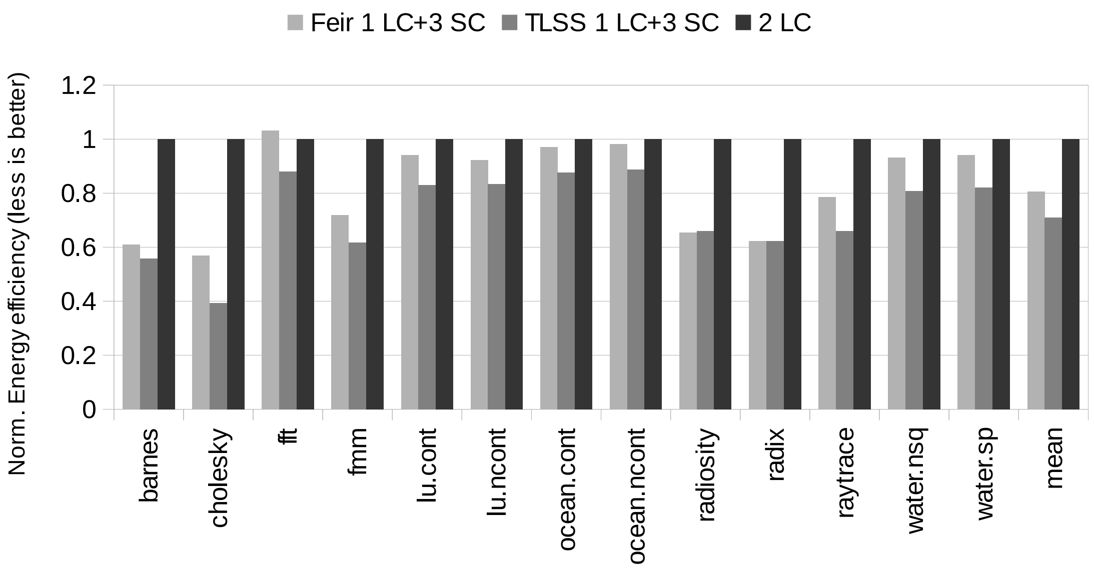


Figure 5.7: Normalized energy efficiency of the TLSS and Fairness-aware schedulers on an ACMP over Linux OS scheduler on an SCMP, where ACMP and SCMP occupy the approximately the same chip area.

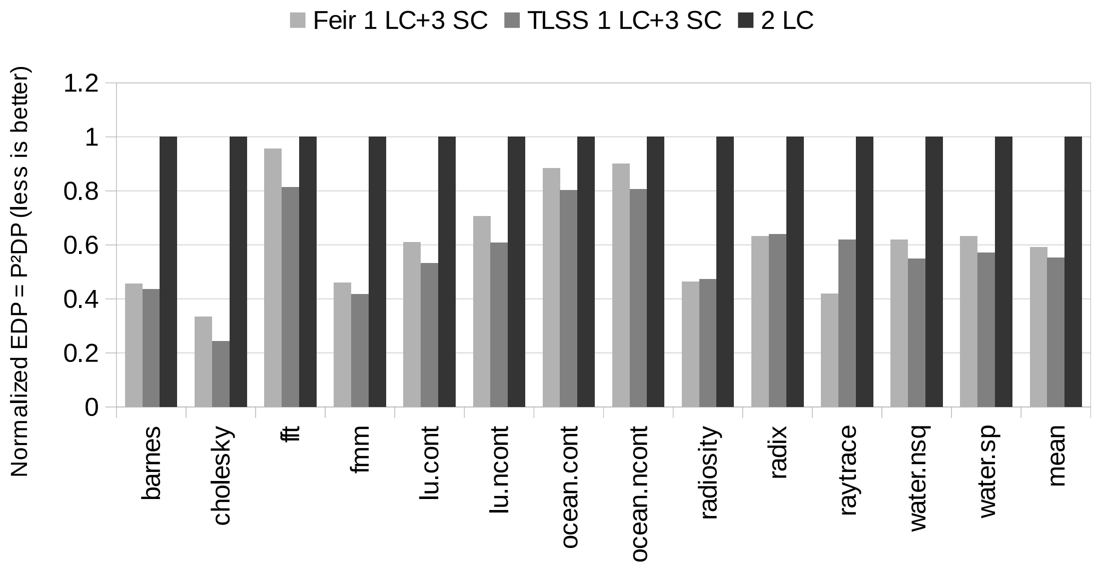


Figure 5.8: Normalized EDP, a commonly used power-delay product, of the TLSS and Fairness-aware schedulers on an ACMP over Linux OS scheduler on an SCMP, where ACMP and SCMP occupy the approximately the same chip area.

CMP	Fair 1 LC+3 SC ACMP			TLSS 1 LC+3 SC ACMP			Fair 2 LC SCMP			Fair 4 LC SCMP			TLSS 4 LC SCMP		
Application	T [s]	P [W]	E [J]	T [s]	P [W]	E [J]	T [s]	P [W]	E [J]	T [s]	P [W]	E [J]	T [s]	P [W]	E [J]
barnes	0.803	37.67	30.24	0.702	39.37	27.64	0.980	50.51	49.52	0.476	85.31	40.59	0.481	97.99	47.17
cholesky	0.260	67.77	17.62	0.170	71.45	12.16	0.267	115.8	30.95	0.095	244.96	23.27	0.115	212.52	24.42
fft	0.672	60.18	40.44	0.575	60.03	34.5	0.602	65	39.15	0.534	80.07	42.78	0.459	90.07	41.33
fmm	0.554	76.64	42.46	0.451	80.85	36.43	0.493	119.74	59.03	0.251	197.27	49.44	0.235	251.58	59.09
lu.cont	0.351	72.35	25.39	0.313	71.52	22.41	0.242	111.48	26.98	0.133	197.48	26.23	0.166	160.76	26.74
lu.ncont	0.372	68.7	24.72	0.330	67.63	22.34	0.288	92.88	26.78	0.136	187.56	25.55	0.164	159.99	26.24
ocean.cont	1.773	35.68	63.25	1.590	35.88	57.04	1.662	39.18	65.11	1.696	45.39	76.99	0.964	65.33	62.95
ocean.ncont	1.858	37.11	68.96	1.700	36.71	62.42	1.735	40.48	70.26	1.735	47.24	81.97	1.065	64.91	69.13
radiosity	1.374	44.68	61.38	1.371	45.18	61.96	1.488	63.08	93.85	0.748	108.39	81.04	0.719	117.57	84.49
radix	0.477	30.82	14.7	0.471	31.2	14.7	0.781	30.3	23.6	0.388	55.98	21.7	0.412	53.85	22.2
raytrace	0.850	27.81	23.64	0.406	48.84	19.83	0.577	52.1	30.06	0.401	76.24	30.59	0.247	111.76	27.61
water.nsq	1.015	61.42	62.33	0.862	62.74	54.06	0.724	92.37	66.91	0.420	156.55	65.75	0.364	176.46	64.2
water.sp	0.275	67.81	18.65	0.231	70.29	16.27	0.196	101.05	19.8	0.109	176.75	19.22	0.100	191.31	19.08

Table 5.2: Simulation measured execution Time [s], average Power [W] and consumed Energy [J] for the asymmetric and symmetric CMPs with different Schedulers.

These improvements in energy efficiency and EDP are attributable to the speedup gains produced by the TLSS method. An ACMP is composed of one large and three small cores while an SMP system has two large cores, where these two configurations take approximately the same chip area. The speedup is the result of an ACMP system having more cores, therefore being able to utilize parallelism better than an SCMP system with fewer cores. Additionally, the power reduction arises from the fact that small cores consume a lot less power then large cores. The performance benefits of the TLSS over the Fairness-aware scheduler also contribute to lower energy consumption when running the same system configuration. It should be noted that SPLASH-2 is a multithreaded benchmark suite, and that TLSS is envisioned to be useful for these cases. It may be less practical for applications whose threads do not compete for locks and should be further explored in future work.

5.3 Conclusion

In this chapter we have discussed an implementation of the Thread Lock Section-aware Scheduling policy [71]. TLSS scheduling is heavily influenced by Fairness-aware Scheduling as well as bottleneck identification techniques. It seeks to provide performance/power benefits from running parallel workloads on ACMPs without the need for substantial hardware extensions, sampling, or runtime overheads. By incorporating minimal hardware additions, our TLSS policy promotes the critical sections of code to be executed on the larger rather than smaller cores within an ACMP. We have proposed a possible application of the TLSS mechanism on SCMPs by designating one of the cores as a large core. Our results show the 10.9% in performance gains due to localizing crucial sections on the large core, while consuming the same amount of energy, compared to a state-of-the-art Linux OS Scheduler running SPLASH-2 benchmarks on an SCMP. As a consequence, the TLSS has an average 6.3% performance per watt gains on an SCMP. Furthermore, comparing the TLSS policy on an ACMP with the Linux OS scheduler on an SCMP, where ACMP and SCMP occupy approximately the same chip area, shows an improvement of 41% and 81% in energy and EDP efficiency respectively.

The energy and performance improvements discussed in this study help to demonstrate the utility that can arise from even simple prediction techniques such as critical section identification when applied to CMPs. However, the rudimentary prediction of the critical sections by TLSS leaves much to be desired without having to resort to the hefty overheads required of more precise techniques such as UBA and BIS. Next we will explore how more sophisticated estimation techniques which can predict system performance for different mapping schemes can be implemented using machine learning models and result in impressive throughput benefits.

Part II

Applying ML to Heterogeneous Scheduling

Having demonstrated the opportunities and limits available using current CMP and scheduling designs in Part I, this part of the thesis examines applying machine learning techniques to conventional heterogeneous schedulers in order to characterize and estimate the performance of threads on different core types.

Accurately estimating the performance of applications on different heterogeneous resources can provide a significant advantage to heterogeneous schedulers seeking to improve system performance. Recent advances in machine learning techniques including artificial neural network models have led to the development of powerful and practical prediction models for a variety of fields. As of yet, however, no significant leaps have been taken towards effectively employing machine learning for heterogeneous scheduling.

In the following two chapters, the case is made of the advantages that applying machine/deep learning (ML/DL) techniques can provide for computer architecture. In particular, we apply artificial neural networks (ANNs) to help estimate system performance and improve the mapping of conventional heterogeneous schedulers leading to significant overall system performance gains.

Chapter 6 is entitled "A Machine Learning Approach for Performance Prediction and Heterogeneous CPU Scheduling." It presents a study proposing a unique throughput maximizing heterogeneous CPU scheduling model that uses machine learning to predict the performance of multiple threads on diverse system resources at the scheduling quantum granularity. We demonstrate how lightweight ANNs can provide highly accurate performance predictions for a diverse set of applications thereby helping to improve heterogeneous scheduling efficiency. We show that online training is capable of increasing prediction accuracy but deepening the complexity of the ANNs can result in diminishing returns. Notably, our approach yields 25% to 31% throughput improvements over conventional heterogeneous schedulers such as HRRS and Linux CFS for CPU and memory intensive applications.

Chapter 7 entitled "A Deep Learning Mapper (DLM) for Heterogeneous Scheduling," describes a scalable scheduling model that decouples thread selection and mapping routines. We use a conventional scheduler to select threads for execution and a deep learning mapper to map the threads onto a heterogeneous hardware. The validation of our preliminary study shows how a simple deep learning based mapper can effectively improve system performance for state-of-the-art schedulers by 8%-30% for CPU and memory intensive applications.

The justification and validation for the innovations proposed in this Part can be summarized as follows:

- **Claim**: Heterogeneous scheduling is a prime candidate in computer architecture where applying ML/DL could result in significant performance improvements. Using lightweight ANN performance predictors to improve a scheduler's mapping is a practical approach to provide performance improvements. The objective of these studies is to justify and validate this claim.

- **Why scheduling**: Heterogeneous scheduling is a popular area of research since it is an important method to efficiently exploit the hardware diversity in new architectures. Managing threads is a problem that shares similarities with recommendation systems and navigation systems both of which have benefitted using ML [9, 28].

- **Mapping for heterogeneous CMPs**: System performance (total instructions per cycle) can vary significantly as a function of how the threads are mapped onto a heterogeneous multicore. Therefore, an intelligent scheduler is one which identifies a performance maximizing mapping scheme.

- **Why ANNs**: Artificial neural network based predictors are one such manner which has been shown to be useful in accurately predicting target categories and values for a wider variety of fields such as predicting network traffic and stock market prices. Moreover, feedforward ANNs such as those used in this work are lightweight in both computation and memory requirements and can be easily accelerated using GPUs or specialized hardware.

- **Validation**: The preliminary results compared to [72] (using the same experimental setup) show significant performance benefits of over 30%. The accurate IPC predictions of the ANNs allow our schedulers to identify an optimal mapping scheme which other schedulers cannot. We show that increasing the complexity of the ANN may result in diminishing returns but may be able to learn interference relationships as well. We believe these initial results are significant for proof of concept studies that pioneers using ML/ML for heterogeneous scheduling.

It is our hope that the novelty of these studies expose some of the exciting opportunities available by applying machine and deep learning in the field of computer architecture.

6 A Machine Learning Approach for Performance Prediction and Heterogeneous CPU Scheduling

In this study we use lightweight ANN performance predictors to improve a scheduler's mapping, compare it with other state-of-the-art schedulers, and demonstrate that significant improvements can be achieved by applying ML to heterogeneous multicore scheduling.

The preliminary results compared to conventional schedulers show significant performance benefits between 25%-31%. The accurate IPC predictions of the ANNs allow our scheduler to identify an optimal mapping scheme which other schedulers cannot. We show that increasing the complexity of the ANN results in diminishing returns (only 3-6% higher) compared with the more lightweight Base and Online models which are easier to implement and have less overhead. These initial results help to validate the proof of concept described in this study that to our knowledge is the first to apply machine learning to heterogeneous scheduling.

The contributions of this chapter include:

- A heterogeneous scheduling model making use of a next quantum thread behavior predictor, machine learning based performance predictors, and a system throughput maximization scheduling policy.
- The design, implementation, and evaluation of two lightweight and one deep ANN based performance predictors for two different core types which produce an accurate estimated instruction per cycle (IPC) value per execution quantum for different threads on an ACMP system.

6.1 ML based heterogeneous scheduling

In this section we present our heterogeneous CPU scheduling model (targeted to an ACMP system) using ANN based performance predictors. The structure of the scheduler is shown in Figure 6.1 and consists of four parts. First is the statistical information about each thread's state and which core type it has been executing on. The second is a next quantum behavior predictor (NQP) that predicts what will be a thread's behavior during the next scheduling quantum (described in further detail in Section 6.1.2). Thirdly is a set of ANN based performance predictors which use a thread's behavior statistics to estimate its performance for a given core type. The fourth and final part is a scheduling policy that uses the estimated performance of all threads on all core types in combination with knowledge of the available system resources to determine and initiate a mapping scheme for the next execution quantum which maximizes system throughput.

The scheduler contains a list of threads where each entry details the thread ID, state (e.g., running, ready, or stalled), and which core it previously executed on. There are also two additional fields storing the performance estimates of the thread if it were to be run on the large or small core during the next quantum. One of these fields will be determined by the NQP while the other will be provided by an ANN performance predictor. For example, after a thread executes on the large core, the scheduler stores the observed IPC in the large core performance estimate field and then uses the value generated by the small core ANN predictor, which is propagated by the NQP, to fill in the small core performance estimate field. As a result, threads only need to use one of the two ANN performance predictors that correspond to the core type they did not run on last quantum.

Waiting or stalled threads do not need to go through the NQP or performance predictors since their estimated IPC values will remain the same since last executing. At the end of each execution quantum, new IPC estimates are generated for all running threads and the scheduler will apply its scheduling policy to determine the optimal thread to core mapping which maximizes total system throughput.

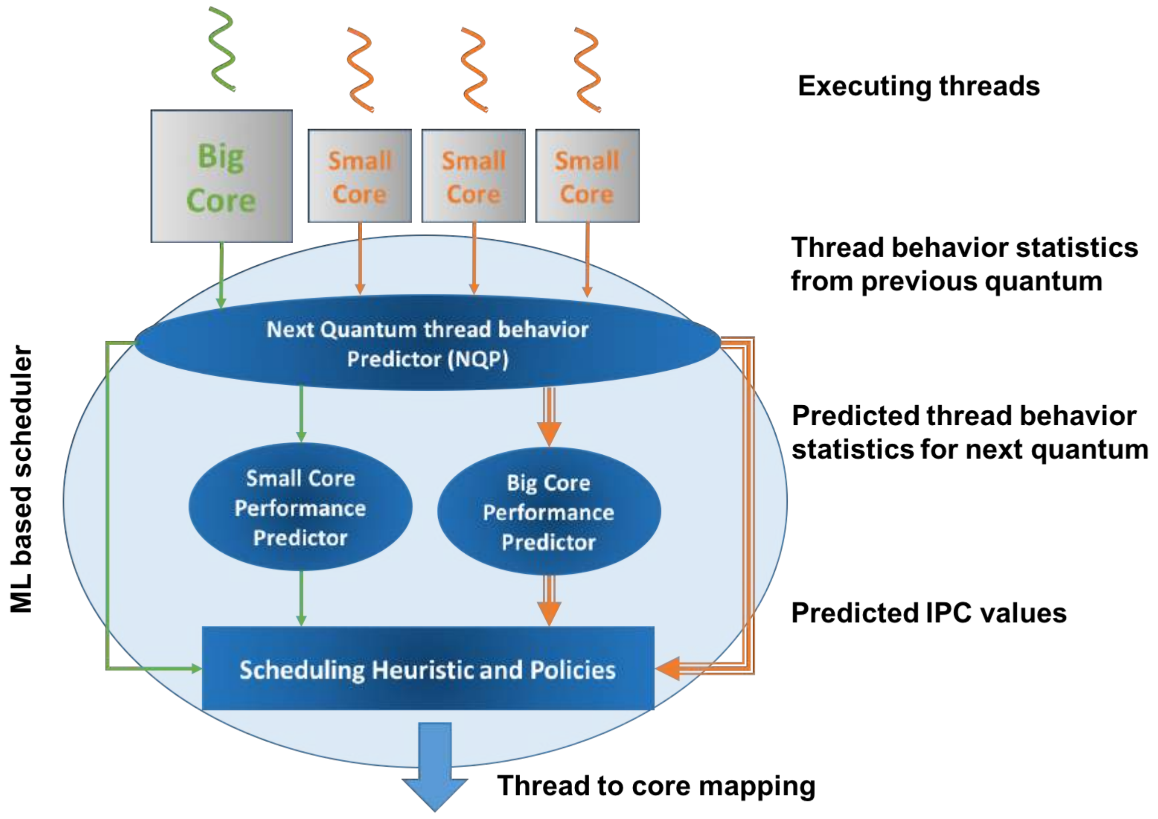

Executing threads
Big Core
Small Core
Small Core
Small Core
Thread behavior statistics from previous quantum
ML based scheduler
Next Quantum thread behavior Predictor (NQP)
Predicted thread behavior statistics for next quantum
Small Core Performance Predictor
Big Core Performance Predictor
Predicted IPC values
Scheduling Heuristic and Policies
Thread to core mapping

Figure 6.1: The proposed ML based heterogeneous scheduling model. After each scheduling quantum, statistics collected from the threads pass through the NQP and the ANN based performance predictors. A scheduling policy then uses the estimated IPC performance of all threads for both the large and small cores to identify an optimal mapping to maximize system throughput for the next quantum.

6.1.1 Parameter engineering

To provide contextual awareness to a CPU scheduler, certain thread and hardware statistics should be periodically collected. These may include values indicating a thread's states, execution time, number of instruction executed, types of instructions executed, number of memory operations, cache accesses/misses, and available cores and their types (if it is a heterogeneous system). The amount of statistics needed to be collected depends upon the complexity and optimization scheme of the scheduler. Being able to predict the performance of a thread on a particular core depends on characterizing the type of instructions that will be executed next quantum as well as estimating their effects on the given hardware resources. It is therefore important to choose an appropriate set of thread statistics that will be used as input parameters to our predictors.

However, in order to use statistical information as input to performance predictors for different core types, the statistics should be as generic as possible. This is to say that the statistical input should not be dependent upon a particular core implementation which may, for example, have different branch predictors and private cache structures than other core types in the system. Normalizing the statistics into ratios (e.g., instruction mix ratios) ensures ANN input to be consistent in cases when training is done with data collected from the small core but predicting is done using input statistics gathered from the large core. This is the procedure used for the small core ANN performance predictors which is discussed in Section 6.1.3.

Without using normalization to get the instruction mix ratio, we would be left with inconsistent statistical input to the ANNs since the number of actual executed instructions of each type depend heavily on the microarchitecture of the cores (e.g., an out-of-order core may execute more instructions than an in-order core even though the instruction mix ratios may be the same). Different forms of generalization can also be used in cases when the core types have different ISAs or cache configurations. Generalizing statistics enables our approach to be useful in systems with a variety of different architectures.

Different statistical values can be used as inputs to the ANNs. These can include branch predictions, instruction mixes, TLB and cache accesses/misses, and DRAM accesses. In determining the final set of statistics to use, we sought to balance suitable ANN predictor accuracy while minimizing the collection and arithmetic overheads. After exploring how different statistics (on an x86 system) affect ANN prediction accuracies we settled upon the nine values given in Table 6.1.

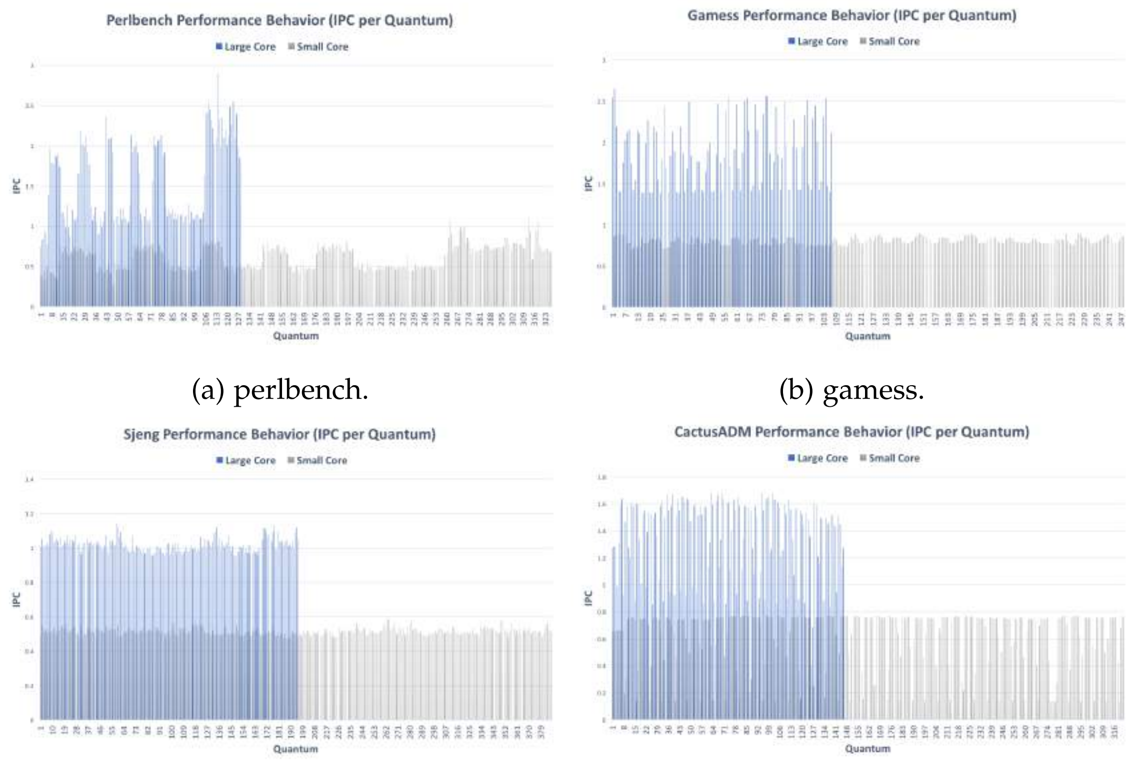
	Thread statistics
1	DL1
2	L2
3	L3
4	Loads
5	Stores
6	FP add/sub
7	FP mul/div
8	Branches
9	Generic arithmetic

Table 6.1: A list of the 9 statistics collected per thread (based on an x86 ISA), normalized into ratios, and then used as input parameters to the ANNs in order to predict the performance for that thread on a given core type.

Many conventional CPUs come with hardware support for collecting similar statistics and in future work we will seek to further improve the set of statistics needed in order to mitigate collection and processing overheads while maintaining or improving the accuracy of our performance prediction model.

6.1.2 Next quantum thread-behavior predictor (NQP)

Several novel approaches [109, 29] have been proposed which predict program behavior based upon various statically or dynamically collected program statistics. However, to lower overheads and for simplicity, in this study, we use a next quantum thread behavior predictor (NQP) that will always predict the next behavior to be equal to the previous quantum behavior. That is to say that if a thread just finished executing during the previous quantum on the small core, it will predict that the thread will perform the same (equal IPC) during the next quantum on the same core (e.g., large core) The NQP does not rely on states, instead it sends the behavioral statistics (e.g., instruction mix ratios, cache misses) gathered during a thread's previous execution quantum as the input parameters to the ANNs.



(a) perlbench.

(b) gamess.

(c) sjeng.

(d) cactusADM.

Figure 6.2: The IPC per quantum behavior of four SPEC benchmarks when running on the large core compared to the small core.

Given that program behavior periodicity has been shown to be on the order of several million of instructions and that we utilize a conventional scheduling quantum of 1ms, we believe this "previous-value" based NQP is adequate for our purpose of demonstrating the potential of performance predictors even though it can certainly be optimized.

Figure 6.2 helps to visualize this behavioral periodicity. It shows the IPC variability of the perlbench, gamess, sjeng, and cactusADM SPEC2006 benchmarks throughout their simulated execution on an Intel Nehalem x86 using a 1ms execution quantum. Though there are clearly periodic behavioral phases that span tens and sometimes hundreds of quanta, it is also possible to observe that for finer granularities, the IPC variation from quantum to quantum is quite minimal, and more so on the small core. The figure also highlights the intra and inter application core to core IPC differences which can fluctuate between phases.

The metric used to measure the accuracy of both the NQP and the ANN predictors is based on the prediction error as a percentage of the observed (target) IPC value. The formulas used are given in equation 3.1 and reproduced below:

$$error_i = \frac{|y_i - t_i|}{t_i}$$

$$\mu_{error} = \frac{1}{n} \times \sum_{i=1}^{n} error_i$$

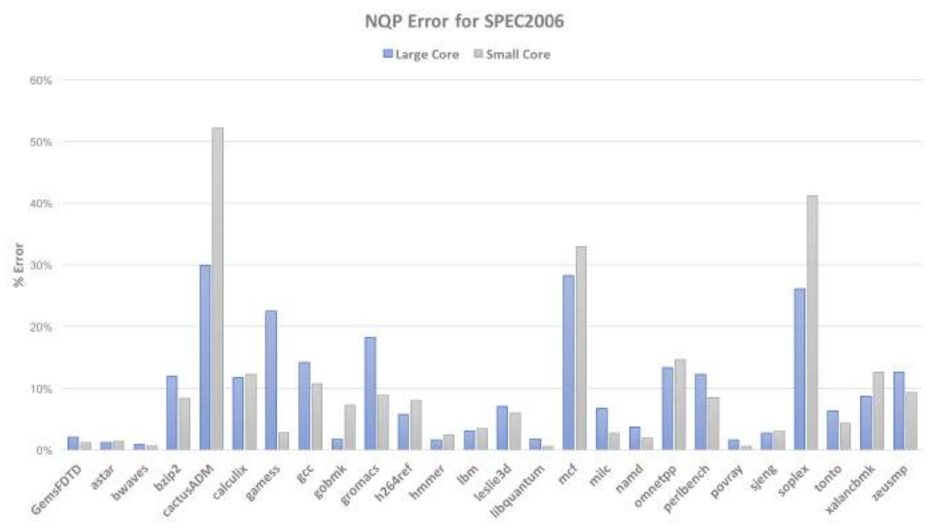
Where y is the predicted IPC and t is the target (i.e., observed) IPC value for quantum i and n is the total number of quanta (i.e., samples).

Figure 6.3 presents NQP accuracy results for SPEC2006 and SPLASH-2 benchmarks. These benchmarks were simulated executing on an Intel Nehalem x86 based heterogeneous configuration (described in Section 6.2.1) using a 1ms execution quantum. The error is calculated using equation 3.1 after measuring the IPC differences of threads after running on the same core for sequential quanta. The NQP results in average errors (including both benchmark suites) of 8% and 7% for the large and small cores respectively. However, the results vary between individual benchmarks with some outliers (e.g., *cactusADM*, *soplex*, *cholesky*, and *ocean*) exhibiting larger errors. Higher errors demonstrate that the application behaviors tend to change more frequently quantum to quantum and hence are harder for the NQP to estimate accurately. These error variations can have a significant impact on the accuracy of the ANN predictors and the efficiency of the resulting mapping scheme.

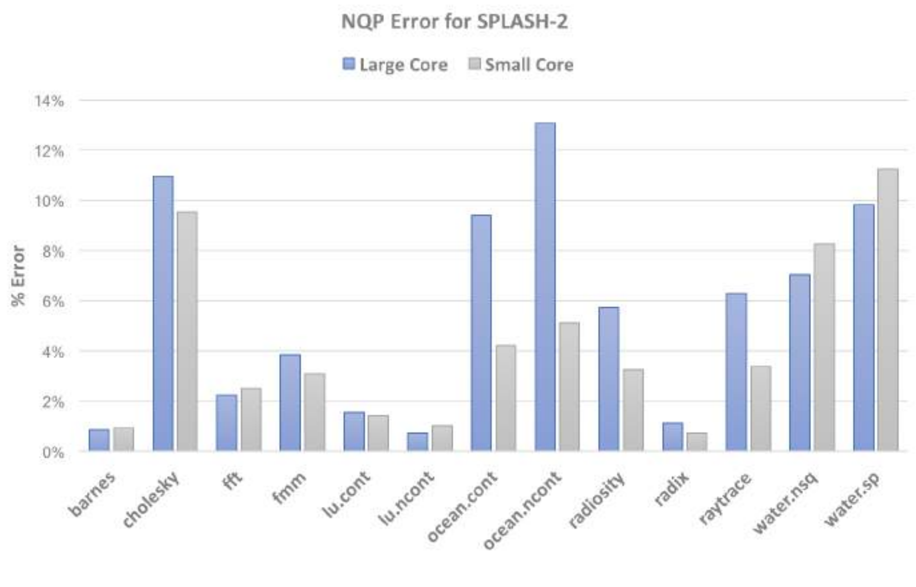
Since the NQP predicts what the 9 statistics determining the behavior of a thread will be for the next quantum based (using the same 9 statistics collected during the previous execution quantum), it will then pass them to the ANN in order for them to predict the thread performance for the next quantum on the core type it *did not* previously execute on last quantum.

6.1.3 ANN performance predictors

One of the core contributions of this study centers around the implementation and use of ANN based performance predictors. We implement separate performance

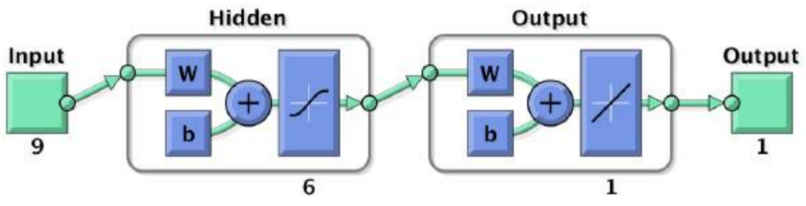


(a) SPEC2006.

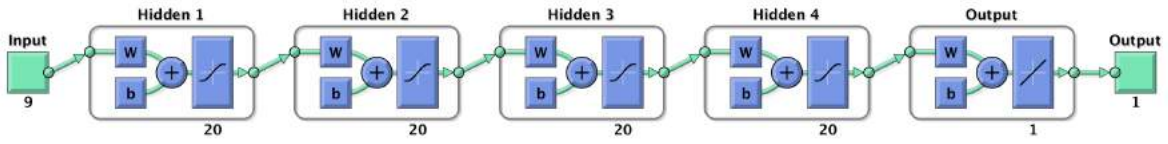


(b) SPLASH-2.

Figure 6.3: The percentage error of the Next Quantum thread behavior Predictor (NQP) for the different core types for SPEC2006 and SPLASH-2. Lower numbers are better.



(a) ANN used by the Base and Online performance predictors.



(b) ANN used by the Deep performance predictors.

Figure 6.4: A representation of the different ANN architectures used by the performance predictors

predictors, one per core type, which take as input a set of statistics gathered from an individual thread and output an IPC value that the thread is estimated to achieve on a particular core type during the next execution quantum.

We implement and evaluate three different ANN models which are given below. Two separate implementations of each ANN model are trained, one to predict for the large core and a separate ANN to predict for the small core. In future work, we will consider using different ANN architectures for the large and small cores. The ANNs are implemented using the Matlab ML toolkit [67]. The three ANN performance predictor types are:

1. **Base**: A lightweight ANN (shown in Figure 6.4a) composed of 9 input parameters, 1 hidden layer with 6 hidden units, and 1 output unit. All input units are interconnected to every hidden unit. The hidden unit's activation function is the hyperbolic tangent sigmoid transfer function while the output unit used is a rectified linear unit in order to predict a numerical value for the IPC. The base predictor is used to evaluate the accuracy of the predictor as if it would only have seen and been trained on a subset of all the applications that are to be run on a system. To accomplish this, we train both the large and small core predictors with data collected from individual executions of all SPLASH-2 benchmarks on both the

large and the small cores. We then evaluate the accuracy of the predictor on both the SPLASH-2 and SPEC2006 benchmark suites.

2. **Online**: A replica of the Base ANN (see Figure 6.4a) but whose purpose is to demonstrate the increased accuracy attainable once online learning has been utilized to keep training the predictor dynamically after all applications from both SPLASH-2 and SPEC2006 are executed at least once. Online training helps to generalize the predictions for diverse workloads and is also useful for improving the prediction accuracy for applications that are executed more than once.

3. **Deep**: A deep learning version of the performance predictors composed of the same 9 inputs but with 4 hidden layers of 20 hidden units and a single rectified linear output unit (shown in Figure 6.4b). This type of predictor is used to showcase the outstanding accuracy that these types of ANN based performance predictors are capable of. Like the Online predictor type, it also emulates using online learning for all the benchmarks in both SPLASH-2 and SPEC2006. Though this type of predictor is not the most practical in terms of overheads needed for weight storage and online learning, it does help to highlight whether increasing the accuracy of the predictors would add significant benefits to the scheduler's performance, or whether the bottleneck is somewhere else such as the accuracy of the NQP.

Training and prediction

Each training data sample consists of 9 input parameters and 1 target IPC value which is periodically collected each quantum during execution of an individual application on both the large or small core separately. Training data generated from applications running on the large core will be used to train the large core ANN predictor and vice versa. The error minimization function used for training the ANN is the mean square error (mse) which is readily used in ML models. Overfitting the ANNs to the training data can lead to large errors when predicting for unseen data. This is a minor concern for the Online and Deep ANNs since the predictor will keep learning as it sees new data and applications are generally run more than once. To test for overfitting on the Base ANN, the total training data set was randomly divided into training (70% of the total data samples), validation

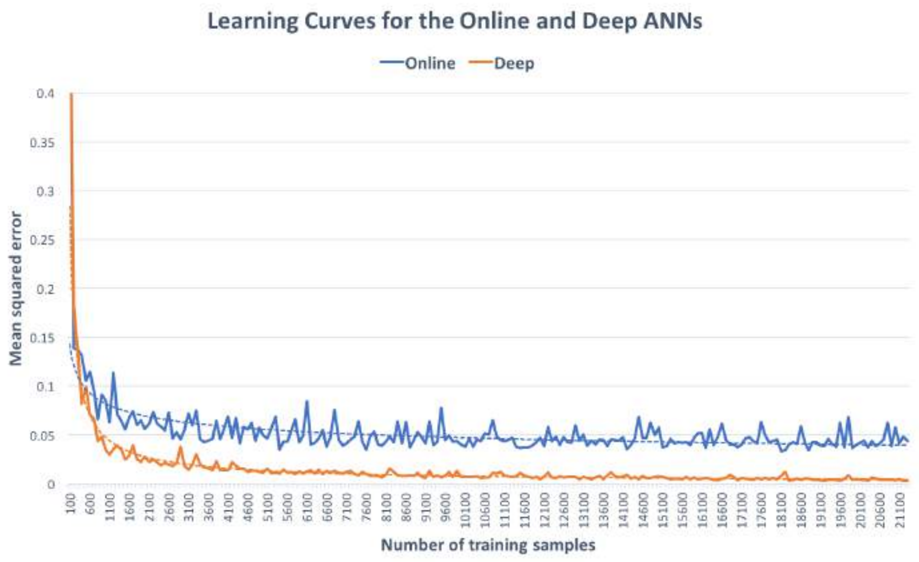


Figure 6.5: The learning curves for the Online and Deep ANN performance predictors. The prediction error decreases as the number of training samples from SPEC2006 and SPLASH-2 benchmarks increase. The training set consists of samples collected every quantum from simulated execution of both benchmark suites. The order of these samples is then randomized and provided as training samples to the ANNs.

(15%), and test (15%) subsets as is common practice in ML. The training subset is used to train the ANN while the validation set is used to measure the ANN's generalization and the test set is employed to measure the final ANN accuracy. The learning curves given in Figure 6.5 show how the training errors of the Online and Deep ANNs decreases as the amount of training data increases. It is representative of the amount of diverse training samples (i.e., execution quanta) the ANNs need to start predicting consistently. After a certain amount of training data, the curves level off so that increasing the amount of training data will not significantly impact the ANN's accuracy. The noise in the curves illustrates how sometimes the training may stop at a local minima or plateau instead of reaching the optimal minima where the error is lower. The shallower Online ANN is more susceptible to getting stuck in local minima and even at optimal minima still results in higher errors than the Deep ANN whose additional hidden units and layers allow it to learn more complex relationships.

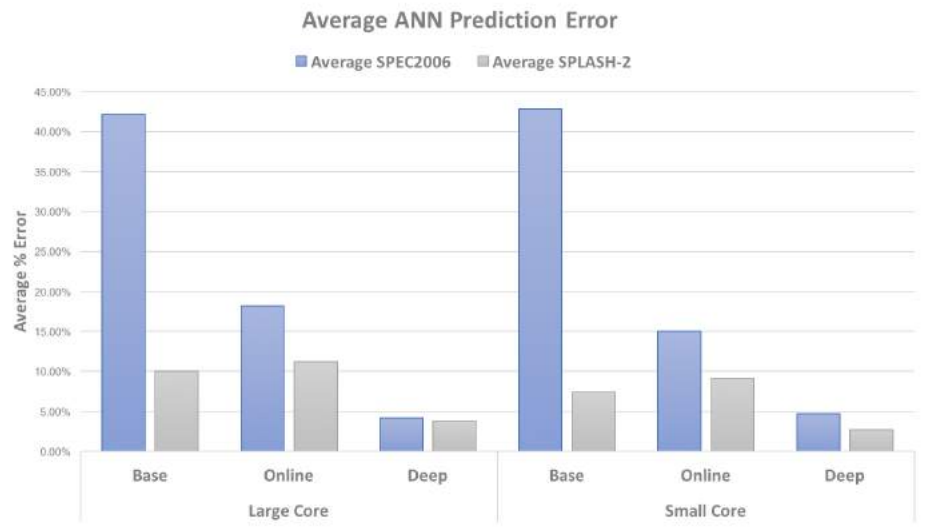


Figure 6.6: The average IPC prediction errors for both SPEC2006 and SPLASH-2 suites for both core types. These values represent the true accuracy of the ANN predictors assuming error free input data (i.e., no noise due to NQP). Lower numbers are better.

The ANNs are useful for predicting the performance of a thread on the core type it has not executed on during the previous quantum. In terms of implementation, however, this translates into needing to train the ANNs using data collected from one core type but predicting using data collected from the other core type. To predict how a thread currently running on a large core will perform on the small core during the next quantum, the nine statistics collected from its execution on the large core is used as the input for the small core ANN predictor. Similarly, the large core ANN predictor uses the statistics collected from a thread's previous execution on a small core. The need to use data from one core type to predict for another core type means that the resulting ANN predictions that the scheduler will use are implicitly inclusive of the NQP errors. This is because the NQP already suffers errors when predicting thread behavior for the next quantum (as seen in Figure 6.3), so the inputs to the ANNs which come from the NQP are inherently inclusive of some level of noise.

Accuracies

The accuracy results in terms of average percentage error for the performance predictors are given in Figure 6.6. These accuracies are generated by using data from the large core to test the accuracy of the large core predictor and vice versa. It illustrates the true accuracy of the ANN given no errors in the inputs which is the case when using the model in real time since the input from the small core will be used to predict for the performance on the large core and vice versa. As shown, the worst performing predictor type for the SPEC2006 benchmarks is the Base as expected since we have only trained it with data from the SPLASH-2 benchmarks. Its total average error for SPEC2006 is around 42%, 43% for the large core and small core respectively. Its accuracy is much better for SPLASH-2 (expected since it is trained with these benchmarks) attaining 10%, 7% average error for the large core and small core respectively.

The Online predictor type greatly reduces the error especially for the SPEC2006 benchmarks but slightly reduces the accuracy on the SPLASH-2 benchmarks. This is due to the ANN generalizing its weights for the behaviors of all benchmarks and is therefore less likely to overfit just for those from the SPLASH-2 suite compared to the Base. The average errors for the Online predictors are 18%, 15% (SPEC2006) and 11%, 9% (SPLASH-2) for the large and small core respectively.

The Deep predictor type performs the best due to its complex architecture and use of online learning. It should be noted that this network is the most susceptible to overfitting but can also learn to predict better for newly run applications dynamically once they run a second time after online training. Its error are 4%, 5% (SPEC2006) and 4%, 3% (SPLASH-2) for the large and small core respectively.

These results show that the Online predictor is capable of improving the accuracy of the Base predictor for SPEC2006 by over 4x while maintaining similar accuracies for SPLASH-2. The Deep predictor improves upon the Base by nearly 10x for SPEC2006 and over 2x for SPLASH-2 while also lowering the standard deviations for both. Evidently, greater performance prediction accuracy can be achieved by including more diverse training data and using more complex ANNs.

Overheads

The overheads of the Online predictor, which we deem to be the most practical approach, consists of approximately 150 floating point (FP) and 80 memory operations for each prediction it makes. Conservatively assuming it takes 20 cycles (though up to 4 FP per cycle may be done in advanced x86 designs [6]) to complete each of the 230 operations, this results in 4,600 cycles of overhead each quantum. The scheduler is called every 1M cycles (CPU clock is set at 1GHz and the scheduling quantum occurs every 1ms) which means that the computational overheads of the Online predictor are approximately 0.5% per prediction. Assuming four predictions (one for each running thread) are to be sequentially calculated every quantum, the total overheads are 2%. Online training may produce more computational overheads but can be triggered less frequently or when the system is less busy or there are idle cores.

6.1.4 Mapping

After passing through the NQP and ANN performance predictors, the threads' IPC estimates for both core types are stored in the corresponding entries for these statistics in the scheduler's list of threads. At this point, the scheduler determines the possible mapping schemes for the next quantum. Each mapping scheme assigns specific threads to particular cores. The computational complexity for mapping is then $O(\binom{n}{k})$, which is related to the number of combinations that place different threads on the large core(s). In this case, n is the number of threads and k is the number of large cores with the assumption that all other threads are mapped onto small cores. In the case of a 1-large 3-small core system executing four threads, four different mapping scheme combinations exist which result in a different thread being assigned to the large core. For each possible mapping scheme, the scheduler estimates the total system IPC by calculating the sum of the predicted IPC of each thread for each core. The scheduler then ranks the different mapping schemes based on the system IPC estimates and selects the highest for the next quantum.

6.2 Evaluation

This section presents the experimental setup and results of our simulations validating our proposal. The significance of the results should be viewed in terms of how well they demonstrate the potential that applying lightweight ML techniques, such as ANNs, can have on improving system throughput.

6.2.1 Methodology

The processor that is used for all experimental runs in this study is a quad-core heterogeneous asymmetric multi-core processor consisting of 1 large core and 3 identical small cores. Both types of cores are based on the Intel Nehalem x86 architecture running at 2.66GHz. Each core type has a 4 wide dispatch width, but whereas the large core has 128 instruction window size, 8 cycle branch misprediction penalty (based on the Pentium M predictor), and 48 entry load/store queue, the small core has a 16 instruction window size, 14 cycle branch misprediction penalty (employing a one-bit history predictor), and a 6 entry load/store queue.

The 1-large 3-small multi-core system configuration is based on the experimental framework used in previous work [72, 56] that we evaluate our proposal against. These works also make use of Sniper, which unfortunately does not provide for a wide selection of different architectures such as ARM but does support hardware validated x86 core types. We believe that using the experimental setup as employed in previous work allows for the fairest comparison.

In running the simulations presented in this evaluation section, we created 12 combinations of 4 different SPEC2006 applications given in table 6.2. These different combinations were chosen to fairly include benchmarks which exhibit low, medium, and high amounts of IPC prediction errors on our performance predictors (shown categorized in the last column of the table). All benchmarks are run from start to finish and the benchmarks that finish first are restarted such that the number of benchmarks running at any one time on the system remains constant. Once all the benchmarks finish at least once, the simulation is ended. This is done to be consistent with simulating in the context of a real system which generally has

Combo	Benchmarks	ANN error
1	astar / lbm / tonto / h264ref	High
2	gcc / cactusADM / gromacs / bwaves	Med
3	hmmer / mcf / gcc / bwaves	Med
4	leslie3d / gcc / bwaves / mcf	Med
5	omnet / astar / bwaves / gemsFDTD	High
6	perlbench / calculix / gamess / bzip2	Low
7	calculix / mcf / soplex / gcc	Med
8	gobmk / xalancbmk / namd / zeusmp	Low
9	astar / zeusmp / cactusADM / gemsFDTD	High
10	namd / gromacs / calculix / cactusADM	Med
11	povray / gromacs / libquantum / bzip2	High
12	sjeng / leslie3d / gobmk / milc	Med

Table 6.2: Simulated Benchmark Combinations. ANN error signifies which combination exhibited more ANN errors when they were trained and tested with the benchmarks within that combination.

enough ready threads to stay continuously busy and to demonstrate the ability of our scheduler to maximize system throughput.

Schedulers

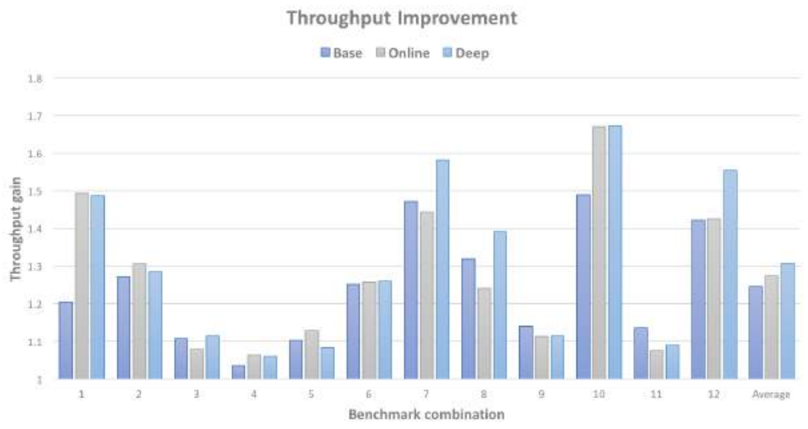
We have chosen to use the Hardware Round Robin Scheduler [72] as the baseline to validate our proposals since it has been shown to provide performance improvements for single and multi-threaded workloads over both the Linux scheduler and Fairness-aware scheduler [77, 56]. As a reminder, the round robin scheduler functions by swapping the thread executing on the large core with one of the threads executing on a small core every quantum.

The three ML schedulers evaluated only differ in the ANN performance predictors they use and are named as such (i.e., the Base scheduler utilizes Base ANN models for predicting for the large and small cores, the Online scheduler uses the Online ANN model, and the Deep uses the Deep ANN model).

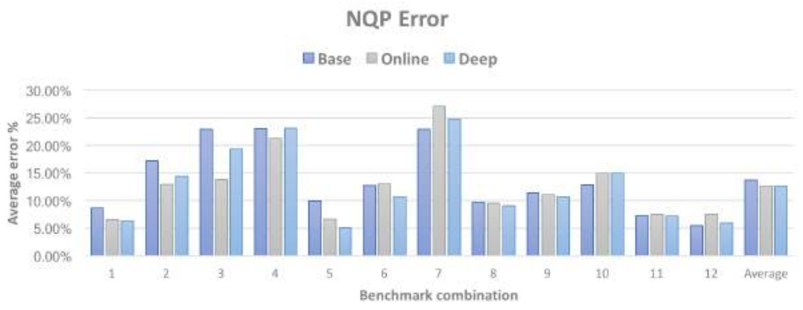
To account for context swap overheads, we apply a 1000 cycle penalty consistent with the metric value utilized in the other studies.

6.2.2 Results

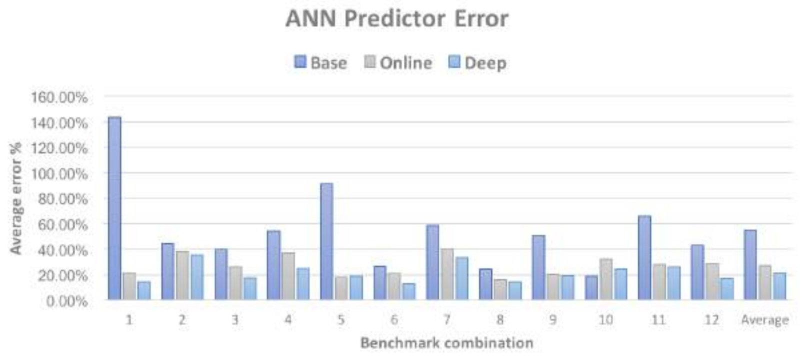
This section presents the predictor errors and system throughput of our 3 schedulers compared to the round robin scheduler after running the 12 combinations of different SPEC2006 benchmarks.



(a) The system throughput increase of the different schedulers normalized to the round robin scheduler. Higher numbers are better.



(b) The average NQP prediction error for all benchmark combinations for the different schedulers. Lower numbers are better.



(c) The average ANN prediction error for all benchmark combinations for the different schedulers. Lower numbers are better.

Figure 6.7: Simulation results of our ML based heterogeneous schedulers.

Figure 6.7a shows the system throughput improvement of our schedulers over the round robin approach. Since our scheduling policy optimizes for maximum system throughput, these results are central for evaluating and validating our ML based heterogeneous scheduling approach. The three proposed schedulers achieve impressive average throughput improvements over the round robin scheduler of 25%, 28%, and 31% for the Base, Online, and Deep schedulers respectively. These improvements reflect the importance and significant differences that can arise from using distinct mapping schemes. The results show that the ANN schedulers are able to identify optimal mapping schemes that the round robin scheduler cannot.

The variations between the three schedulers for each individual benchmark combination are noticeably related to the differences in their ANN and NQP errors. Some of the throughput differences between the schedulers for some of the combinations can also be attributed to the fact that the different ANNs learn different relationships between the input parameters and output performance. Some of these learned relationships may be very beneficial for some applications but less so for others which behave irregularly. In addition, secondary effects such as context swap penalties and thread interference can vary depending on the mapping and thread behaviors.

Figure 6.7b shows the average performance (IPC) prediction error of the NQP of the different schedulers. The error vary significantly between different benchmark combinations (from 5% to 27%) and even slightly between the different schedulers (up to 10%). The former is expected since the benchmarks behave differently from one another and go through different phases of execution. However, the differences in NQP error between the three schedulers when executing the same benchmark combination is not necessarily intuitive. We believe that these discrepancies are due to differences in mapping outcomes between the schedulers which results in the benchmarks progressing differently and causing some variance of when and how quickly they switch between phases. Differences in NQP error can produce a noticeable impact on the accuracies of the ANNs. However, though the measured average error rate of the NQP can reach up to 14%, our proposed schedulers nonetheless result in high accuracy and performance gains. This demonstrates that even accounting for these phase and periodic variations in application behavior, it is still a reasonable approach for our proposal to predict the behavior for the next quantum based upon the previous quantum.

Finally, Figure 6.7c shows the average performance (IPC) prediction error of the three different ANNs. As expected, the Base scheduler, which averages 55% error, performs worse than both the Online and Deep predictors and achieves very poor accuracy results for a couple of the combinations. These large errors illustrate that the Base ANN did not learn to generalize adequately to account for the behaviors present in these combinations. Continuing to learn dynamically as new applications are run can greatly improve the accuracy as illustrated by the Online scheduler results which average 27% error. Further increasing the complexity of the ANNs can also result in accuracy improvements as shown by the Deep results which average 21% error. The errors are higher than those shown earlier (6.1.3) for the individual benchmark runs because these errors also inherently include those of the NQP. This is the case because the input to the ANNs comes directly from the NQP. For example, combination 10 had lower NQP errors on the Base scheduler which is reflected in the lower ANN error compared with the Online and Deep ANNs.

In sum, these promising results validate the instrumental value that using ANNs performance predictors can provide for effective heterogeneous scheduling.

6.3 Future work and conclusion

We seek to expand the scope of our work in the future by conducting studies related to the input parameters, ANN hyperparameters (e.g., number of hidden layers and units, training algorithms and regularization methods, etc.), improving the NQP, and scalability. In particular we would like to develop further parameter normalization methods to allow for more compatibility between very diverse architectures and adapt our model to include thread interference effects.

In this study, we have pioneered applying machine learning to a throughput maximizing heterogeneous CPU scheduler capable of predicting the performance of multiple threads on diverse system resources at the scheduling quantum granularity. We have shown how a lightweight ANN can provide highly accurate performance predictions for a diverse set of applications even while only being trained on a small subset. In addition, we described how the predictor accuracy can be greatly improved upon through the use of online learning and deep learning. Our approach yields significant results with average throughput improvements between 25% to 31% over conventional heterogeneous schedulers for CPU and memory intensive applications.

Though very promising, this approach does not take into consideration thread interference effects, nor does it have an efficient scheduling mechanism for dealing with the scenario when more threads than cores need to be selected and mapped therefore requiring further modifications before being scalable. Next, we shall see how a deep ANN (DNN) capable of predicting total system performance inclusive of thread interference effects can be easily applied to a decoupled scheduling model to handle large number of active threads.

7 A Deep Learning Mapper (DLM) for Heterogeneous Scheduling

The objective of the study in this chapter is the proof of concept of the opportunities that arise by applying DL to computer architecture designs. The novelty of this work centers on decoupling the selection and mapping mechanisms of a heterogeneous scheduler and fundamentally, the implementation of a deep learning mapper (DLM) which uses a DNN to predict system performance. The selector remains responsible for ensuring fairness and selecting the threads to execute next scheduling quantum while the mapper is charged with identifying an optimal mapping of selected threads onto available cores. Compared to the previous study which did not include a thread selection mechanism, this approach also uses an ANN to predict the performance for the whole ACMP and not just per thread and per core. The results of our proposal are promising, the DLM is capable of improving the performance of existing conventional schedulers such as HRRS, Fairness-aware, and Linux CFS by 8%-30% for computational and memory intensive applications.

The contributions of this chapter include:

- A heterogeneous scheduling model which decouples thread selection and mapping to the computational resources.
- An implementation of a deep learning mapper (DLM) that uses a deep neural network.

- An experimental validation presenting throughput results comparing 3 state-of-the-art schedulers with and without making use of the DLM targeted towards an ACMP system.

The rest of this chapter is structured as follows. Section 7.1 presents our proposed scheduling model with a description of a practical implementation. Validation of our implementation with experimental results is found in section 7.2. Lastly, we discuss future work and conclusion in section 7.3.

7.1 Scheduling model

In this section we present our scheduling model (shown in Figure 7.1) with decoupled thread selection and mapping mechanisms. This scheduling model uses (i) a conventional scheduler (CS) to select a subset of available threads to execute next quantum (using its prioritization scheme) and (ii) the deep learning mapper (DLM) to map the selected threads onto the diverse system resources (using a throughput maximization scheme). The scheduling quantum (the periodicity to run the scheduler) chosen is 4ms for the CS and 1ms for the DLM which reflect typical quantum granularities of CS approaches. This difference allows the DLM to take advantage of the finer grained variations in program behaviors and optimize the mapping on the heterogeneous system while still maintaining CS objectives. Furthermore, the context swap penalties is generally lower for the DLM since it only swaps threads which are already running and have data loaded in their private caches while the CS may select to run any thread that may not have any of its data in the caches.

In addition to selecting the threads to run next, the CS is responsible for thread management, including modifying their statuses, dealing with thread stalls, and mapping for the first quantum of new threads or when the number of available threads is less than the number of available cores. When active, the DLM essentially provides a homogeneous abstraction of the underlying heterogeneous hardware to the CS since it only needs to select threads to run and not whether to execute on a large or small core.

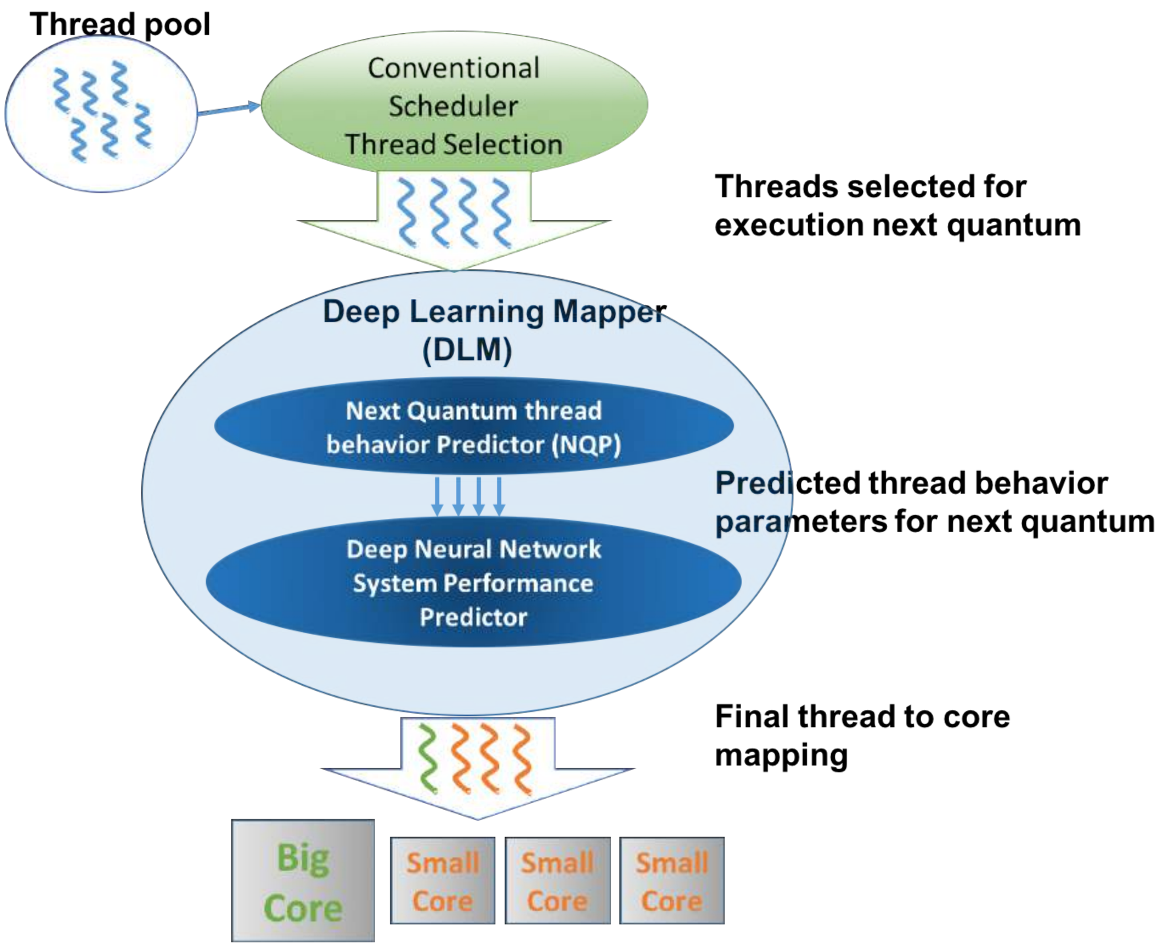

Thread pool
Conventional Scheduler Thread Selection
Threads selected for execution next quantum
Deep Learning Mapper (DLM)
Next Quantum thread behavior Predictor (NQP)
Predicted thread behavior parameters for next quantum
Deep Neural Network System Performance Predictor
Final thread to core mapping
Big Core
Small Core
Small Core
Small Core

Figure 7.1: The scheduling model. A conventional scheduler is used to select the threads to run next quantum and the DLM then uses the NQP and DNN predictor to find the optimal mapping to maximize system performance.

7.1.1 Deep learning mapper (DLM)

The DLM is responsible for finding a mapping of the selected threads onto the hardware cores which optimizes system throughput. This objective helps to demonstrate the significant potential that using DNN based performance predictors can have for a continuously busy system. The DLM is composed of:

- Statistical information about each selected thread pertaining to its behavior. These are collected during the thread's previous execution quantum.
- A next quantum behavior predictor (NQP) that predicts what will be a thread's behavior during the next execution quantum.
- A DNN based performance predictor that uses the behavior statistics of all threads selected for execution to estimate system performance for different mapping combinations.

Upon being passed to the DLM, the selected threads enter the next quantum predictor (NQP) which outputs behavioral statistics for each thread. These statistics are then fed into the DNN system performance predictor which predicts the system IPC for different mapping combinations. Since the objective of the mapper is to map threads onto the hardware cores to maximize system throughput, the mapping combination with the highest predicted system performance is chosen for the next quantum.

The NQP utilized in this study is based upon the NQP presented in the previous chapter. For a review of the NQP and parameter engineering analysis, refer to Sections 6.1.2 and 6.1.1.

In determining the final set of statistics, we sought to balance DNN predictor accuracy while minimizing the overheads due to gathering the statistics and the arithmetic operations needed to be performed. Based upon the heterogeneous system used in our study (detailed in section 7.2.1), we identified 12 different thread statistics that are useful in describing thread behaviors on the cores and are inclusive of thread interference effects. Compared to the previous study, the two floating point parameters are combined into a single parameter and we have included four additional parameters to better represent the cache access patterns of the threads.

The statistics are collected after a thread completes an execution quantum and are composed of the accesses and misses of the different structures of the cache hierarchy as well as the instruction mix executed. These 12 thread statistics (given as ratios) are given in Table 7.1.

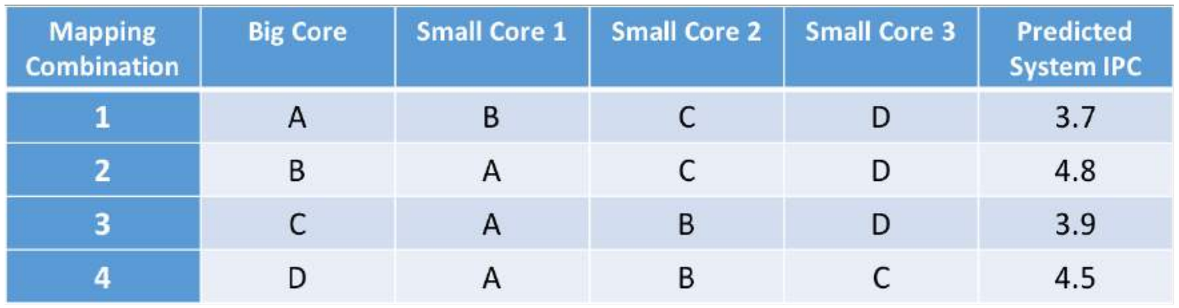
	Thread statistics
1	DL1
2	L2
3	L3
4	Loads
5	Stores
6	FP add/sub/mul/div
7	Branches
8	Generic arithmetic
9	IL1 divided by DL1 loads
10	L2 divided by DL1 misses
11	L3 divided by DL1 misses
12	L3 divided by L2 misses

Table 7.1: A list of the 12 statistics collected per thread (based on an x86 ISA), normalized into ratios, and then used as input parameters to the ANN. Since the ACMP has four cores, the ANN uses the parameters from four threads as input in order to predict the ACMP system performance (a total of 48 input parameters).

The 12 statistics are saved as part of a thread's context after each quantum it executes, overwriting the values from the previous quantum. Many conventional CPUs come with hardware support for collecting similar statistics and in future work we will seek to further explore the set of statistics needed in order to mitigate collection and processing overheads while maintaining or improving the accuracy of our performance predictor.

DNN system performance predictor

The key component behind the DLM is a DNN system performance predictor which takes as input a set of parameters from as many individual threads as there are hardware cores and then outputs an estimated system IPC value. The system we target is a heterogeneous CPU architecture composed of 4 cores with 2 different



Mapping Combination	Big Core	Small Core 1	Small Core 2	Small Core 3	Predicted System IPC
1	A	B	C	D	3.7
2	B	A	C	D	4.8
3	C	A	B	D	3.9
4	D	A	B	C	4.5

Figure 7.2: An example of how the DLM uses the DNN to predict for 4 different mapping combinations once it is passed the 4 threads selected by the CS (A, B, C, and D).

core types (1 large core and 3 small cores, described in section 7.2.1). The DNN predictor takes as input the 12 normalized parameters from the 4 threads (selected for execution by the CS) for a total of 48 input parameters. Thanks to the thread selection scheme utilized, the ANN only every needs to have no more than four threads to predict for and therefore its input can be statically set to 48 parameters.

The order in which the threads are inputted to the DNN correspond to which physical core they would be mapped to. For instance, we have set the first 12 thread parameters as corresponding to the thread mapped to the large core, the next 12 parameters correspond to the thread mapped to the first small core, the next 12 correspond to the thread mapped to the second small core, and the last 12 parameters correspond to the thread mapped to the third small core. This way, we are able to estimate what the system IPC would be for different mapping combinations.

An example of this is given in Figure 7.2. Here the CS has selected 4 threads (A, B, C, and D) from a larger pool of available threads to execute next quantum. There are 4 different combinations which we can map the 4 threads onto the hardware where each combination will have a different thread mapped onto the large core. The different mapping combinations represent the different ordering of the thread parameter inputs to the DNN. For instance, combination 1 will have the first 12 inputs correspond to thread A, the next 12 to thread B and so on. We could also consider all mapping permutations (i.e., different mappings between the small cores) but since the only shared structure is the L3, there should be negligible differences in performance and interference effects. In the example, the DNN predictions for the 4 different combinations are given in the last column. Combination 2 has the highest estimated system and will be chosen as the optimal mapping scheme for the upcoming quantum. It should be noted that unlike the study in the previous chapter where statistics from the small core were used as input for the large core predictor and vice versa, there is no indication to the ANN whether a thread has previously executed in the large or small core. This could be an interesting modification to make for future work by simply adding a binary parameter indicating if the thread was previously run on the large or small core.

We have implemented the DNN performance predictor using Python and the machine learning library scikit-learn [78]. After conducting a hyperparameter (i.e., the metrics that define the DNN architecture and training method) analysis for the DNN, which we sought to balance accuracy with implementation practicality, we settled on a DNN implementation consisting of 48 total inputs, 5 hidden layers of 25 hidden units each, and a single output unit that use a rectified linear activation function:

$$f(x) = max(0, x)$$

.

The DNN is capable of dynamic learning via online training using micro-batches of new training data collected as the system continues execution. Therefore, even if the predictor begins with very low accuracy, it will be able to learn dynamically such that its level of accuracy and generalization for different applications improves over time.

Each training data sample consists of 48 input parameters and 1 target system IPC value. These are collected after each scheduling quantum which has resulted in the execution of 4 threads on the 4 cores. The algorithm used for training is a stochastic gradient based optimizer with L2 regularization which is readily used in machine learning models to regulate overfitting by diminishing the importance of parameter weights. During training, the weights of the neural network are adjusted after each full iteration of a batch of training data, always aiming to minimize the mean square error (mse) between the predicted output and the target output. The number of training samples to use for online training is adjustable; for instance we could continue training after each single quantum or after 100 or more quanta, but it is important to balance overheads with the ability of the predictor to generalize for a wide scope of thread behavior. For our online DNN implementation, we have chosen to keep training our predictor every 20 execution quanta (i.e., a micro-batch of 20 samples). This requires needing to save only 20 quantum samples of data a time. The frequency of online training is related to the average number of quanta the benchmarks take to complete. A larger micro-batch could be used for longer applications or when the system is exceedingly busy in order to lower overheads. Micro batch training means that the weights are updated after the errors from all the samples in the micro-batch are calculated. One training epoch passes when the ANN updates its weights for all micro batches. Not all micro batches collected over the entirety of a systems execution history needs to be saved and utilized for training. This can cause large memory footprint and calculation overheads but would also train the ANN to predict for all applications ever run. Conversely, if only newer micro batches are used for training, then the ANN may forget what it has previously learned and just be accurate for newly run applications. Striking a balance between these approaches can be tricky but methods such as keeping only a subset of older micro batches and newer micro batches may be a useful middle ground. In this study we have retrained using a full history to showcase the potential of the DLM model. Improving online training scalability for the model will be an area of focus in future work.

Figure 7.3 plots the learning curves of the training and 10-fold cross-validation results of the online DNN predictor. It highlights how, as the quantity of training data grows, so too does the accuracy and generalizability of the predictor when executing all the applications from SPEC2006. The score is measured in terms of correlation between the predicted system performance and the observed system performance using an R^2 coefficient (see Equation 3.3) reproduced below:

$$R^2 = 1 - \frac{u}{v}$$
$$u = \sum (y_{true} - y_{pred})^2$$
$$v = \sum (y_{true} - \bar{y}_{true})^2$$

In particular, the figure shows that after about 15000 quanta, the correlation between the predicted performance and the observed performance on the data used

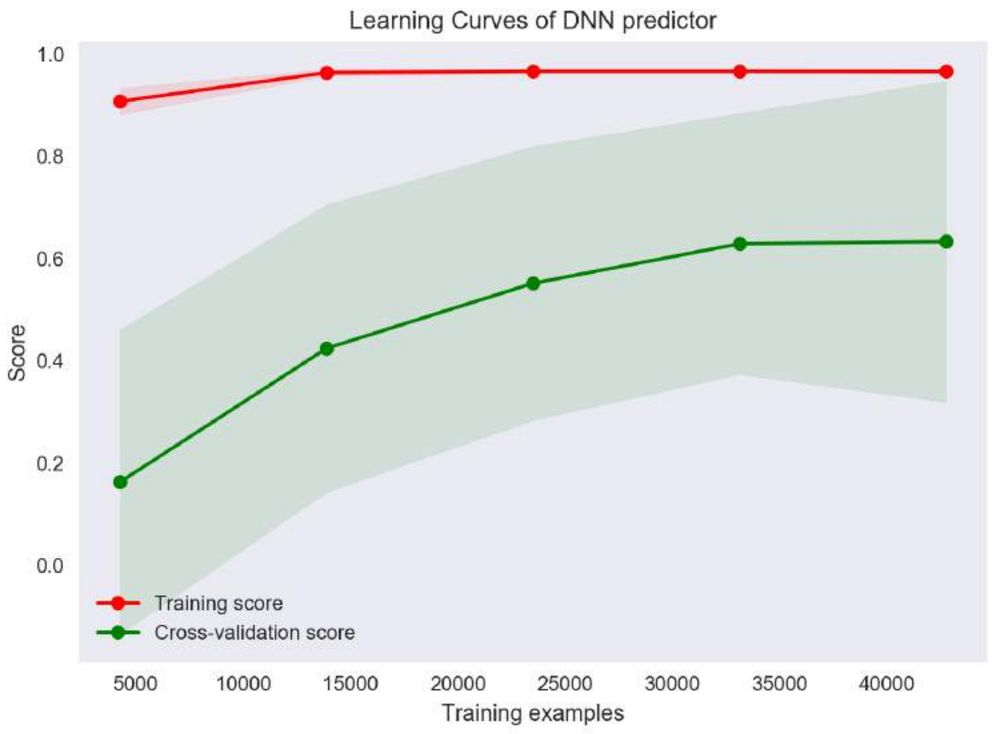


Figure 7.3: The learning curve of the online DNN predictor. As the amount of training data increases the predictor becomes more generalized (i.e., able to predict with similar accuracy levels for both training and test data) to account for different applications and behaviors. Higher y-axis numbers are better.

to train is very high (about 0.96) and after about 35000 quanta, the correlation stabilizes for the unseen validation data at about 0.64. The difference between the training and validation curves illustrates that the model has high variance which may indicate overfitting, but can be explored in future work by adding more regularization and fine tuning the input parameters, hyperparameters, and sample data. Since our model is capable of online learning, however, the prediction errors introduced by running new applications will gradually settle at lower levels after training dynamically.

7.1.2 Overheads

Schedulers typically add overheads due to the mapping calculations and resulting context swaps after each scheduling quantum. Since the DLM is triggered 4 times as often as the CS (1ms vs 4ms quantum), the DLM can also cause context swaps by swapping threads between cores before the next CS quantum. A minimum of 0 and maximum of 4 extra context swaps can be issued by the DLM before the next CS quantum. However, the DLM will only trigger a swap if the resulting mapping is beneficial to overall system performance. The overheads due to the NQP and performance predictor amount to less than 4000 floating point operations per predicted mapping combination and less than 16000 in total. However, not only can a large quantity of these calculations be done in parallel, but this overhead is still orders of magnitude less than it costs to swap contexts and load the caches. Online training also adds overheads but is only done after every 20 quanta (or the chosen frequency of micro-batch training) and can be hidden by running it in the background when a core is idle.

Storing the 64-bit weights of the DNN requires about 21KB of memory. The introduction of new statistical fields to save for each thread is also a minor overhead (96 bytes per thread) as is the memory needed to store the online training data (7,680 bytes for 20 samples of 4 threads' worth of parameters). Lowering these overheads is a topic for future work but are still reasonable for a viable implementation of the scheduling model.

7.2 Evaluation

This section presents the experimental setup and validation results of our scheduling model. The significance of the results should be viewed in terms of how well they demonstrate the potential that applying lightweight DL techniques such as DNNs can have on improving system throughput.

7.2.1 Methodology

Similar to the study presented in the previous chapter, the processor that is used for all experimental runs in this work is a quad-core heterogeneous asymmetric multi-core processor consisting of 1 large core and 3 identical small cores. The 1-large 3-small multi-core system configuration is based on the experimental framework used in previous work [72, 56] that we evaluate our proposal against.

We have used the popular SPEC2006 [47] benchmark suite to evaluate and train our scheduling model. The entirety of the benchmark suite is used with the exception of some applications which did not compile in our platform (*dealII, wrf, sphinx3*). All 26 benchmarks are run concurrently from start to finish and the simulation ends after all the benchmarks finish. This is done to emulate a busy system which must execute a diverse set of applications. This setup is also useful in demonstrating the ability of the DLM to improve system throughput.

We evaluate the performance for three different conventional schedulers (round-robin [72], Fairness-aware [56], and CFS [77]) with and without a DLM trained online. To account for context switch overheads due to architectural state swapping, we apply a 1000 cycle penalty which is consistent with the value utilized in the round robin study. The additional cache effects from the context switches are captured by the simulation.

Figure 7.4 compares the system throughput improvements achieved for all 3 schedulers when using a DLM after running SPEC2006. The results show an average percentage throughput increase of 8%, 20%, and 30% for the round-robin, fairness-aware, and CFS schedulers respectfully. These improvements are significant especially for a preliminary study with a simple deep neural network predictor. They also highlight how effective the DLM is at benefitting all 3 different

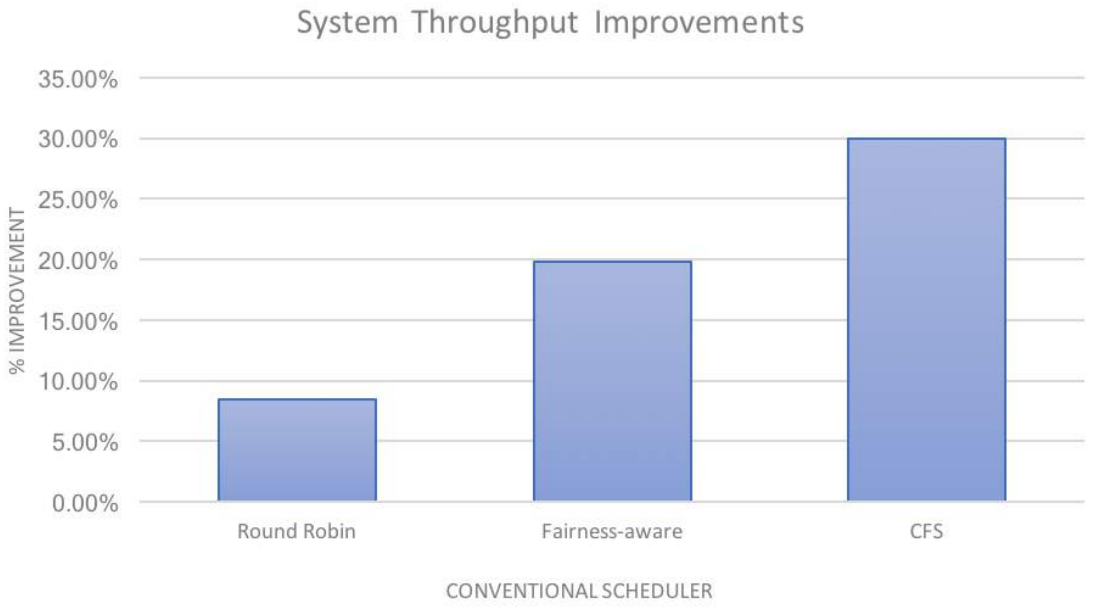


Figure 7.4: Average system throughput (IPC) improvements when using the DLM for all SPEC2006. Higher numbers are better.

state-of-the-art schedulers. The improvements demonstrate the ability of the DLM to find more optimal mappings than the schedulers can by themselves. It achieves this thanks to two main factors. The DNN predictor allows the DLM to make highly accurate predictions for different mapping combinations while the 1ms quantum provides the opportunity to detect and adjust the mapping for variations in thread behaviors. The differences in the throughput gains for the 3 schedulers are also consistent with how they perform relative to one another without the DLM. On a heterogeneous system, the round-robin scheduler has been shown to perform better than the fairness-aware scheduler, which in turn performs better than the CFS. We believe that the differences in improvements are largely due to the fact that NQP produces more noise as new threads are introduced into the cores (which is frequent for HRRS and Fairness-aware but less so for CFS) since this results in different interference and cache warm up effects. This consequentially introduces more error into the DNN predictor and causing the DLM to misidentify the best mapping option.

The average total percentage prediction error of the DLM (calculated using equation 3.1) for the experiments was 12%. Similar to the previous chapter, the results from our model includes errors from both the NQP and the DNN. This error rate is within reasonable margins and our results are notable when considering that the DLM still showed such significant throughput benefits.

7.3 Future Work and Conclusion

We seek to expand the scope of our work in the future by conducting an extensive exploration related to the thread statistics, alternative DL models, improving the NQP, and scalability issues. In particular we would like to develop parameter generalization methods to allow for more compatibility between very diverse architectures. Clustering and ensemble models could be used to further widen the scope of the DLM for dealing with irregular applications.

In this chapter we have presented a preliminary study which pioneers applying DL to heterogeneous scheduling. We outlined a scalable scheduling model that decouples thread selection and mapping routines. The thread selection mechanism of a conventional scheduler is used in conjunction with a deep learning mapper (DLM) to maintain fairness and increase system performance. The DLM uses a deep neural network to predict the system performance for different mapping options at the scheduling quantum granularity. This lightweight deep neural network can provide highly accurate predictions for a diverse set of applications while continuing to train dynamically. The validation of our approach shows that even a simple DL based mapper can significantly improve system performance for state-of-the-art schedulers by 8% to 30% for CPU and memory intensive applications.

We hope that the novelty of the two studies in this part of the thesis have exposed some of the exciting opportunities available by applying machine and deep learning techniques to the field of computer architecture.

8 Related Work

In this chapter we discuss the related work for the topics explored in this thesis covering heterogeneous architectures, scheduling, machine learning for systems, and programming/analytical models and unconventional architectures.

8.1 Heterogeneous architectures

There have been several developments in heterogeneous many-core systems, including MorphCore [104], ARM's big.LITTLE [38], and accelerated processing units [23]. Moncrieff et al. [22] and Menasce et al. [21] analytically examined the tradeoffs between utilizing fast and slow processors in heterogeneous processors. Their study showed that a system composed of few fast cores and many slow cores are effective in terms of cost and performance.

An ACMP system containing various cores of the same ISA but of different types was proposed by Kumar et al. [82]. Their process consists of deciding on the core that will perform in the most power efficient manner each time a new phase or program is detected using sampling techniques. Moncrieff et al. [22] and Menasce et al. [21] analytically examined the tradeoffs between utilizing fast and slow processors in heterogeneous processors. Their study showed that a system composed of few fast cores and many slow cores are effective in terms of cost and performance. In terms of microarchitectural differences, Chen et al. [46] implemented their ACMP with cores consisting of separate branch predictors, issue widths, and L1 cache sizes that used in conjunction with their scheduling method, achieved throughput and energy efficiency improvements. The scheduling approach used by Annavaram et al. [65] focuses on staying within a specified power envelope by measuring the energy per instruction of an ACMP running multithreaded applications and running parallel sections of code on the small cores and then migrating to the large cores for the sequential sections.

Grochowski et al.[31] studied the potential of ACMP architectures to save energy and improve throughput. The study [110] investigates the relationship between chip area and performance and determines a potential configuration. Private caches generally require coherence mechanisms to manage sharing of data which may impose overheads if the cores are mostly dealing with non shared data or experience many contention cycles if attempting to access a resource shared by multiple cores. However, as shown in the studies [57], [7], non-uniform cache architecture (NUCA) caches are able to mitigate both of these disadvantages and can be used to combine several separate cache modules into an emulated large shared cache.

8.2 Scheduling

Our studies have been influenced by a wide scope of research applied towards heterogeneous scheduling. The idea of the contemporary chip multiprocessor (CMP) originated in the early nineties with the research community recognizing the impact scheduling can have on system utilization and performance. An early scheduling algorithm for an asymmetric system called Single Architecture Heterogeneous Multiprocessor that does not support multi-programming is presented in [69].

Optimal scheduling of independent applications running on a preemptive heterogeneous CMP has been studied by Liu et al. [50]. They analyze past non-critical thread barrier stall times in order to estimate the future criticality of threads and dynamic voltage and frequency effects. Building upon the occupancy-based approach of Contreras and Martonosi [37], the work [1], samples current behavior to predict thread criticality and use the predictor to improve task stealing.

Scheduling based power management techniques have been examined in the work of Winter et al. [44] which employ several sampling based algorithms in order to analyze the optimal thread to core mapping. Sampling methods are also used in the work [81], which investigates performance maximization of multithreaded applications on an ACMP. A heterogeneous approach at allocating resources has been previously applied to CMP interconnection networks in order to leverage the non-uniformity in the demand of network resources [70]. A symbiotic job scheduler for a simultaneous multithreading processor which sought to schedule concurrent threads that do not severely impact the performance of one another on a single SMT processor is studied by Snavely et. al. [100].

Fairness-aware Scheduler [56] is a software implemented ACMP scheduling mechanism that enhances the pinned scheduler by triggering a software thread swap after a specified software quantum (typically 4ms). They target a two core type ACMP system and assign the thread with least execution progress to the more powerful core type each scheduling quantum. Another fair based scheduling approach for asymmetric CMPs was explored by Saez et. al. in [92]. Similar work by Becchi [66] consists of an ACMP that includes two distinct core sizes where thread to core assignment is managed by initiating a mandatory swap of threads between two different sized cores in order to measure the corresponding performance ratio. Based on this ratio, the threads are then scheduled to their core that will maximize the system performance. This work has given insight into ratio based ACMP scheduling techniques but is limited as the number of distinct core types used increase. Other work in this area has been done by Saez et al. [45] who use a utility factor, defined as the ratio of L1 miss latency compared to a baseline ACMP configuration (only small cores), with the aim of optimizing the performance of both single and multithreaded workloads. Likewise, Koufaty et al. [20] determine optimal thread to ACMP core mapping using a biasing method estimated by the quantity of external memory stalls and internal pipeline stalls. A formula based ACMP thread to core scheduling method is proposed by Srinivasan et al. [90] which is used to estimate and compare thread performance on individual cores. In contrast, a phase classification and regression analysis is utilized to optimize thread to core mapping in an ACMP by Khan et al. [75].

A separate study [4] aims to create a contention-aware scheduler that maximizes throughput by learning and mimicking the decisions of an oracle scheduler. Chronaki et al. [13, 14] propose a heterogeneous scheduler for a dataflow programming model which improves performance using a prioritization scheme and dynamic task dependency graph to assign newly created and critical tasks to fast cores. A statistical method using extreme value theory is used in [84] to determine the probabilities for optimal task assignment in massively multithreaded processors.

For scheduling within the domain of OpenMP parallel applications, the authors of [11] propose a NUMA-aware scheduler that determines which threads are sharing data on a common NUMA node. Based on monitoring memory related performance counters, the scheduler is able to efficiently map threads to NUMA nodes. A more generic NUMA-aware scheduler is presented in [87] which highlights how using schedulers unaware of the hardware architecture (referred to as UMA systems in the paper) can negatively impact performance. These novel proposals highlight the impact that efficient scheduling can have for increasing performance when there is hardware resources contention.

8.3 Machine learning for systems

There has been very few previous studies conducted on applying machine/deep learning to CPU scheduling. Much of the previous work using machine/deep learning for scheduling has been to classify applications, as well as to identify process attributes and a program's execution history. This is the approach of [73] which used decision trees to characterize whole programs and customize CPU time slices to reduce application turn around time by decreasing the amount of context swaps.

The work presented in [63] studies the accuracy of SVMs and linear regression in predicting the performance of threads on two different core types. However, they do so at the granularity of 1 second, use only a handful of benchmarks, and do not implement the predictor inside of a scheduler.

The studies by Frederic et. al. [80], [27] investigates using machine learning to automatically generate desired solutions for a set of problem instances and solve for new problems in a massively parallel manner.

In the study [41], CPU burst times of whole jobs for computational grids are estimated using a machine learning approach employing decision trees and k-nearest neighbors. A related work targeting grids by predicting execution time, memory and disk consumption for bioinformatics applications is done in [68]. An approach that utilized machine learning for selecting whether to execute a task on a CPU or GPU based on the size of the input data is done by Shulga et. al. [95]. Predicting L2 cache behavior is done using machine learning for the purpose of adapting a process scheduler for reducing shared L2 contention in [85]. Fedorova et. al. [36] proposes an algorithm that uses reinforcement learning to maximize normalized aggregate IPC. They demonstrate the need for balanced core assignment but do not provide an implementation.

In the work done by Bogdanski et. al. [10], choosing parameters for task scheduling and loadbalancing is done with machine learning. However, their prediction is whether it is beneficial to run a pilot program that will characterize a financial application. They also assume that the computational parameters of the workload stay uniform over certain periods of time. Nearly all of these approaches deal with either program or process level predictions and target homogeneous systems. It is important to note that one of the first pioneering studies of applying a primitive model of machine learning towards improving CPU performance was done by Jimenez et. al. [53] which employed perceptrons as a new method to predict branches.

8.4 Programming/analytical models and unconventional architectures

Several novel research studies have focused on evaluating microarchitectural details and providing an analysis or systematic approach at determining the most optimal modifications. A statistical simulation approach at modeling processors is presented in [32]. Design space exploration studies of embedded architectures is done in [34] and [79].

Characterizing and exploiting program behavior and phases has been the subject of extensive research. Duesterwald et. al. [29] and Sherwood et. al. [94] showed that programs exhibit significant behavioral variation and can be categorized into basic blocks and phases which can span several millions to billions of instructions before changing. Work done in [109] has taken advantage of the compilers ability to statically estimate an applications varying level of instruction level parallelism in order to estimate IPC using monotonic dataflow analysis and simple heuristics for guiding a fetch-throttling mechanism.

Future programming models and architectures will likely make use of different runtimes and abstraction layers. Proposals seeking to explore unconventional designs can gain insight from cross platform runtime systems such as the Java programming environment [5]. Java offers a rich framework of data structures and methods that are shared as libraries and ubiquitously used in academia and industry. Java and other highly abstract cross-platform languages rely on the use of virtual ISAs (such as Java Byte-code).

Existing virtual ISAs also express a very low level of functionality and must be translated into the physical processor's ISA in order to run [2, 99]. This procedure naturally causes overheads, but most have been greatly minimized thanks to a continual process of optimization. The virtual instruction set computing (VISC) proposal builds on an LLVM method and applies it toward heterogeneous architectures [3]. Although there are similarities with F-ISA, in particular, both are intended as a virtual intermediate representation and the VISC approach can be seen as complementary to F-ISA. The VISC design builds on vector parallelism as well as generalized macro dataflow graphs and, although it is novel, it sets an upper bound on the level of functional abstraction it can support from the computation cores. It also does not support function unfolding such as in F-ISA.

The neurocomputing Galatea project was an interesting proposal [107]. By using a low-level virtual machine language that includes threaded-code techniques and a certain amount of function unfolding capabilities, it can sustain code portability across different systems (in this case, neurocomputers). This technique can also be seen as complementary to F-ISA.

Although similar LLVM approaches could allow for function unfolding using threaded-code techniques, their low level of functional abstraction limits the diversity and complexity of computational cores that might be implemented, as compared with the F-ISA approach. In addition, the F-ISA proposal is intended for parallel applications running on typical workstations and mobile devices without the need to modify user code. Task dataflow programming models provide useful abstraction tools for developers to improve the parallelization and performance of their programs[35]. OmpSs and its earlier implementations [30, 26, 8] let programmers separate sections of code into tasks that are then passed to a runtime that uses dataflow methods, including data tokens and dependency graphs, to determine which tasks are ready for execution and which cores to dispatch them for execution. Whereas these programming models involve the developer inserting hints or pragmas into the application code, our approach aims at optimizing the hardware without requiring any modifications of the user code.

Nonconventional object processors have previously sought to exploit the modularity and parallelism inherent in object-oriented software techniques. The Intel iAPX 432 [111], Rekursiv [40], and Smalltalk on a RISC (SOAR) [108], are all examples of early, yet novel, object-oriented processors. More contemporary designs include the picoJava [76] and [93] processors based on a hardware implementation that can run Java bytecode. Many of these early systems, developed several decades ago, were confronted with technological limitations that increased the impact of software-related overhead latencies, for example, due to object type checking routines. Decoupled architecture techniques have also been implemented to great effect [96, 97]. These systems improve computer performance via separate instruction streams that can execute concurrently while still maintaining sequential program execution flow. These proposal show that there has been constant interest in the research community for pursuing unconventional system designs. Issues dealing with scalability, legacy support, and cross platform practicality are often the main barriers for new designs to be popularly adopted. However, widening CPU constraints as well as the development of new processor technologies and system efficiency techniques/tools, if properly utilized, can open new opportunities for unconventional designs to be widely adopted.

9 Conclusion

Computer architects have started embracing heterogeneous systems as an effective method to make use of increases in transistor densities for executing a diverse range of workloads under varying performance and energy constraints. While future of heterogeneous systems may include tens or hundreds of different accelerators, current heterogeneous designs are much more limited in scope. Typically combining powerful general purpose cores with energy efficient cores, conventional heterogeneous systems can still provide for substantial amounts of architectural flexibility with regards to computational and memory resource configurations. To effectively exploit conventional and future CMPs, advances in heterogeneous designs should be complemented by developments in CPU scheduling.

CMP schedulers typically rely on a thread selection mechanism to guarantee a level of fairness with regards to execution time for all workloads, and also produce a mapping scheme assigning workloads to specific hardware cores. Conventional CMP schedulers rely on tried and tested prioritization or scoring thread selection methods to provide fairness but are often unaware of hardware diversity and unable to produce optimal mapping schemes. Newer schedulers aimed at heterogeneous CMPs are able to recognize the differences between the cores and provide an added level of fairness not just in terms of total execution time but also in the amount of time spent running on each core type. As a result, state-of-the-art heterogeneous schedulers are able to produce significant performance improvements over their predecessors and enable more flexible CMP designs including cache configurations.

Identifying an optimal mapping scheme, however, is a difficult endeavor to pursue without being able to estimate how different applications will perform on the distinct core types. Most conventional schedulers are unable to do so and therefore cannot compare the throughput that different mapping schemes would result in. The proposals that do use performance estimation at the quantum granularity can often suffer from significant overheads due to their high complexity and lack of scalability.

Yet, managing and mapping threads is a problem that shares similarities with recommendation systems and navigation systems both of which have benefitted using machine learning techniques. Advances in machine learning (ML) prediction models have unlocked an exceptional opportunity of using these techniques for estimating system performance. Though heterogeneous scheduling is a popular area of research, there has yet been no seminal work exploring the use of ML for mapping optimization.

In this thesis we have explored the rise and utility of current heterogeneous architectures and presented a futuristic model which supports massive hardware diversity. Utilizing a functional ISA (F-ISA) increases the functional abstraction level of the machine instructions, thereby enabling a significant increase in the diversity of a processor's computational units. This results in greater specialized execution particular to the needs of the software algorithms which should be useful in improving system performance relative to latency, memory footprint, and power.

After conducting a design space exploration of conventional heterogeneous designs, we highlighted how CPU scheduling is at the crux of effective application and hardware resource management. We have shown how heterogeneous aware schedulers can provide for architectural design flexibility as well as energy and performance improvements. For instance, adopting a smaller or distributed cache hierarchy in conjunction with a heterogeneous scheduler was shown to lead to substantial energy and area savings of over 17% and 19% respectively.

The thread lock section-aware scheduler was presented to emphasize the opportunities that arise from improving heterogeneous scheduling for ACMP systems. A lightweight hardware implementation of this scheduler, which maps threads based on estimating whether a thread has entered a synchronization section, results in over 80% EDP improvements over a Linux scheduler and clearly demonstrates the benefits of using heterogeneous schedulers over homogeneous schedulers.

Understanding the scope and importance of heterogeneous architectures and scheduling techniques led us to our most impactful and pioneering studies of this thesis. In two separate studies, we demonstrated the novelty and usefulness of applying machine and deep learning techniques to computer architecture. The results of these studies have validated that using ANN/DNN predictors for mapping improves the performance of conventional heterogeneous schedulers for CPU and memory intensive applications by over 30% with minimal (2%) overheads. Combining ML/DL predictors with a decoupled thread selection and mapping scheduling model is a scalable and lightweight method that can help to support the heterogeneous architectures of the future.

10 Publications

The work of the thesis has resulted in the following publications.

10.1 Publications from the thesis

- Daniel Nemirovsky, Nikola Markovic, Osman Unsal, Adrian Cristal, and Mateo Valero, "Reimagining Heterogeneous Computing: a Functional Instruction Set Architecture (F-ISA) Computing Model.", IEEE Micro (2015).

- Nikola Markovic, Daniel Nemirovsky, Osman S. Unsal, Marteo Valero, and Adrian Cristal, "Performance and energy efficient hardware-based scheduler for Symmetric/Asymmetric CMPs.", In Computer Architecture and High Performance Computing (SBAC-PAD), 2015 27th International Symposium on, pp. 33-40. IEEE, 2015.

- Daniel Nemirovsky, Tugberk Arkose, Nikola Markovic, Osman Unsal, Adrian Cristal, and Mateo Valero, "A Machine Learning Approach for Performance Prediction and Heterogeneous CPU Scheduling.", In the International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2017 29th International Symposium on.

- Daniel Nemirovsky, Tugberk Arkose, Nikola Markovic, Osman Unsal, Adrian Cristal, and Mateo Valero, "A Deep Learning Mapper (DLM) for Heterogeneous Scheduling.", In Latin America High Performance Computing Conference (CARLA), 2017.

- Daniel Nemirovsky, Nikola Markovic, Osman Unsal, Adrian Cristal, and Mateo Valero, "Mathematical representation of the Hardware Round-Robin Scheduler analytical model for single-ISA heterogeneous architectures.", 2nd BSC Doctoral Symposium, Barcelona Supercomputing Center, Barcelona, 2015.

- Daniel Nemirovsky, Nikola Markovic, Osman Unsal, Adrian Cristal, and Mateo Valero, "Extending the Flexibility of ACMPs for Mobile Devices Using Alternative Cache Configurations." 10th MULTIPROG Workshop at the 12th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC) , Stockholm, Sweden, January 2017

- Daniel Nemirovsky, Tugberk Arkose, Nikola Markovic, Osman Unsal, Adrian Cristal, and Mateo Valero, "A Machine Learning Performance Prediction Model for Heterogeneous Systems.", 4th BSC Doctoral Symposium, Barcelona Supercomputing Center, Barcelona, 2017.

10.2 Publications not included in the thesis

- Damian Roca, Daniel Nemirovsky, Mario Nemirovsky, Marc Casas, Miquel Moreto, and Mateo Valero, "iQ: an efficient and flexible queue-based simulation framework." IEEE 25th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2017).

- Damian Roca, Daniel Nemirovsky, Mario Nemirovsky, Rodolfo Milito, and Mateo Valero, "Emergent Behaviors in the Internet of Things: The Ultimate Ultra-Large-Scale System." IEEE Micro 36, no. 6 (2016): 36-44.

- Markovic, Nikola, Daniel Nemirovsky, Osman Unsal, Mateo Valero, and Adrian Cristal, "Kernel-to-User-Mode Transition-Aware Hardware Scheduling.", IEEE Micro 35, no. 4 (2015): 37-47.

- Nikola Markovic, Daniel Nemirovsky, Osman Unsal, Mateo Valero, and Adrian Cristal, "Thread lock section-aware scheduling on asymmetric single-ISA multi-core.", IEEE Computer Architecture Letters 14.2 (2015): 160-163.
- Markovic, Nikola, Daniel Nemirovsky, Veljko Milutinovic, Osman Unsal, Mateo Valero, and Adrian Cristal, "Hardware Round-Robin Scheduler for Single-ISA Asymmetric Multi-core.", In European Conference on Parallel Processing, pp. 122-134. Springer Berlin Heidelberg, 2015.
- Nikola Markovic, Daniel Nemirovsky, Ruben González, Osman Unsal, Mateo Valero, and Adrián Cristal, "Object oriented execution model (OOM).", 2nd Workshop on New Directions in Computer Architecture (NDCA-2): held in conjunction with the 38th International Symposium on Computer Architecture (ISCA-38): 5th June: San Jose, California. INRIA, 2011.

List of Figures

List of Tables

Bibliography

[1] A. Bhattacharjee, and M. Martonosi (2009). Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. In *Proc. 36th Ann. Intl. Symp. on Comp. Arch. (ISCA)*, 290–301. 132

[2] Adve, V., Lattner, C., Brukman, M., Shukla, A. & Gaeke, B. (2003). LLVA: A Low-level Virtual Instruction Set Architecture. In *Proceedings of the 36th annual ACM/IEEE international symposium on Microarchitecture (MICRO-36)*, San Diego, California. 136

[3] Adve, V., Adve, S., Komuravelli, R., Sinclair, M.D. & Srivastava, P. (2012). Virtual Instruction Set Computing for Heterogeneous Systems. In *Proceedings of the 2012 4th USENIX Workshop on Hot Topics in Parallelism (HotPar'12)*, Berkeley, California. 136

[4] Anderson, G., Marwala, T. & Nelwamondo, F.V. (2013). Multicore scheduling based on learning from optimization models. *Int. J. Innovative Comput. Inform. Control. v9 i4*, 1511–1522. 133

[5] Arnold, K., Gosling, J., Holmes, D. & Holmes, D. (2000). *The Java programming language*, vol. 2. Addison-wesley Reading. 136

[6] Barker, K.J., Davis, K., Hoisie, A., Kerbyson, D.J., Lang, M., Pakin, S. & Sancho, J.C. (2008). A performance evaluation of the nehalem quad-core processor for scientific computing. *Parallel Processing Letters*, **18**, 453–469. 108

[7] Beckmann, B. & Wood, D. (2004). Managing wire delay in large chip-multiprocessor caches. In *Proc. of Int. Symp. Microarchit.*, 319–330. 132

[8] Bellens, P., Perez, J.M., Badia, R.M. & Labarta, J. (2006). Cellss: a programming model for the cell be architecture. In *SC 2006 Conference, Proceedings of the ACM/IEEE*, 5–5, IEEE. 137

[9] Bishop, C.M. (1995). *Neural networks for pattern recognition*. Oxford university press. 92

[10] Bogdanski, M., Lewis, P.R., Becker, T. & Yao, X. (2011). Improving scheduling techniques in heterogeneous systems with dynamic, on-line optimisations. In *Complex, Intelligent and Software Intensive Systems (CISIS), 2011 International Conference on*, 496–501, IEEE. 135

[11] C. Su, D. Li, D. Nikolopoulos, M. Grove, K. W. Cameron, and B. R. de Supinski (2011). Critical path-based thread placement for numa systems. In *Proc. 2nd international workshop on Performance Modeling, Benchmarking and Simulation of high performance computing systems (PMBS)*, 19–20. 134

[12] Cannady, J. (1998). Artificial neural networks for misuse detection. In *National information systems security conference*, 368–81. 3

[13] Chronaki, K., Rico, A., Badia, R.M., Ayguade, E., Labarta, J. & Valero, M. (2015). Criticality-aware dynamic task scheduling for heterogeneous architectures. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, 329–338, ACM. 133

[14] Chronaki, K., Rico, A., Casas, M., Moreto, M., Ayguade, E., Valero, M. *et al.* (2016). Task scheduling techniques for asymmetric multi-core systems. *IEEE Transactions on Parallel and Distributed Systems*. 133

[15] C.K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood (2005). Pin:building customized program analysis tools with dynamic instrumentation. In *Proc. ACM SIPLAN Conf. on Prog. Lang. Design and Impl.*, 190–200. 46

[16] Collobert, R. & Weston, J. (2008). A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, 160–167, ACM. 2, 27

[17] Conway, P. & Hughes, B. (2007). The amd opteron northbridge architecture. *IEEE Micro*, **27**. 14

[18] Corporation, I. (2013). Intel core i7-4600u processor. [Online] Available: `ark.intel.com/products/76616/Intel-Core-i7-4600U-Processor-4M-Cache-up-to-3_30-GHz`. 27, 28, 46, 148

[19] D. Genbrugge, S. Eyerman and L. Eeckhout (2010). Interval simulation: Raising the level of abstraction in architectural simulation. In *Proc. 16th Int. Symp. High Perform. Comput. Arch.*, 1–12. 42

[20] D. Koufaty, and D. Reddy, and S. Hahn (2010). Bias scheduling in heterogeneous multi-core architectures. In *Proc. 5th Eur. Conf. Comput. Syst.*, 125–138. 26, 133

[21] D. Menasce and V. Almeida (1990). Cost-performance analysis of heterogeneity in supercomputer architectures. In *Proc. of the 4th Int. Conf. on Supercomputing*, 169–177. 131

[22] D. Moncrieff, and R. E. Overill, and S. Wilson (1996). Heterogeneous computing machines and amdahl's law. *Parallel Computing*, **22**, 407 – 413. 131

[23] Daga, M., Aji, A.M. & Feng, W.c. (2011). On the efficacy of a fused cpu+gpu processor (or apu) for parallel computing. In *Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on*, 141–149, IEEE. 131

[24] Dagum, L. & Menon, R. (1998). Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, **5**, 46–55. 31

[25] DIGNAN, L. (2015). Ibm research builds functional 7nm processor. [Online] Available: http://www.zdnet.com/article/ibm-research-builds-functional-7nm-processor/. 1

[26] DONGARRA, J., TOURANCHEAU, B., PLANAS, J., BADIA, R.M., AYGUADÉ, E. & LABARTA, J. (2009). Hierarchical task-based programming with starss. *The International Journal of High Performance Computing Applications*, **23**, 284–299. 137

[27] DORRONSORO, B. & PINEL, F. (2017). Combining machine learning and genetic algorithms to solve the independent tasks scheduling problem. In *Cybernetics (CYBCON), 2017 3rd IEEE International Conference on*, 1–8, IEEE. 134

[28] DOUGHERTY, M. (1995). A review of neural networks applied to transport. *Transportation Research Part C: Emerging Technologies*, **3**, 247–260. 92

[29] DUESTERWALD, E., CASCAVAL, C. & DWARKADAS, S. (2003). Characterizing and predicting program behavior and its variability. In *Parallel Architectures and Compilation Techniques, 2003. Proceedings. 12th International Conference on*, 220–231, EEE. 21, 99, 135

[30] DURAN, A., AYGUADÉ, E., BADIA, R.M., LABARTA, J., MARTINELL, L., MARTORELL, X. & PLANAS, J. (2011). Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, **21**, 173–193. 137

[31] E. GROCHOWSKI, AND R. RONEN, AND J. SHEN, AND H. WANG (2004). Best of both latency and throughput. In *Proc. of the Int. Conf. on Computer Design (ICCD)*, 236–243. 132

[32] EECKHOUT, L. & DE BOSSCHERE, K. (2001). Hybrid analytical-statistical modeling for efficiently exploring architecture and workload design spaces. In *Parallel Architectures and Compilation Techniques, 2001. Proceedings. 2001 International Conference on*, 25–34, IEEE. 135

[33] ESMAEILZADEH, H., BLEM, E., ST AMANT, R., SANKARALINGAM, K. & BURGER, D. (2011). Dark silicon and the end of multicore scaling. In *ACM SIGARCH Computer Architecture News*, vol. 39, 365–376, ACM. 14

[34] EYERMAN, S., EECKHOUT, L. & DE BOSSCHERE, K. (2006). Efficient design space exploration of high performance embedded out-of-order processors. In *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, 351–356, European Design and Automation Association. 135

[35] FATAHALIAN, K., HORN, D.R., KNIGHT, T.J., LEEM, L., HOUSTON, M., PARK, J.Y., EREZ, M., REN, M., AIKEN, A., DALLY, W.J. *et al.* (2006). Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 83, ACM. 137

[36] FEDOROVA, A., VENGEROV, D. & DOUCETTE, D. (2007). Operating system scheduling on heterogeneous core systems. In *TProceedings of the Workshop on Operating System Support for Heterogeneous Multicore Architectures*. 135

[37] G. CONTRERAS, AND M. MARTONOSI (2008). Characterizing and improving the performance of intel threading building blocks. In *Proc. IEEE Intl. Symp. on Workload Characterization*, 57–66. 132

[38] GREENHALGH, P. (2011). big.little processing with arm cortex-a15 & cortex-a7. [Online] Available: http://www.arm.com/files/downloads/bigLITTLE_Final_Final.pdf. 1, 14, 131

[39] GURESEN, E., KAYAKUTLU, G. & DAIM, T.U. (2011). Using artificial neural network models in stock market index prediction. *Expert Systems with Applications*, **38**, 10389–10397. 3

[40] HARLAND, D.M. (1988). *Rekursive: Object-oriented Computer Architecture*. Halsted Press, New York, NY, USA. 137

[41] HELMY, T., AL-AZANI, S. & BIN-OBAIDELLAH, O. (2015). A machine learning-based approach to estimate the cpu-burst time for processes in the computational grids. 3–8. 134

[42] HORNIK, K., STINCHCOMBE, M. & WHITE, H. (1989). Multilayer feedforward networks are universal approximators. *Neural networks*, **2**, 359–366. 3

[43] Intel (2015). Intel 64 and ia-32 architectures developer's manual. [Online] Available: http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-manual-325462.html. 77

[44] J. A. Winter, and D. H. Albonesi, and C. A. Shoemaker (2010). Scalable thread scheduling and global power management for heterogeneous many-core architectures. In *Proc. Int. Conf. Parallel Archit. Compilation Tech. (PACT)*, 29–40. 132

[45] J. C. Saez, and M. Prieto, and A. Fedorova, and S. Blagodurov (2010). A comprehensive scheduler for asymmetric multicore systems. In *Proc. 5th Eur. Conf. Comput. Syst.*, 139–152. 133

[46] J. Chen, and L. K. John (2009). Efficient program scheduling for heterogeneous multi-core processors. In *Proc. Annu. Design Automation Conf. (DAC)*, 927–930. 131

[47] J. Henning, Sun Microsystem (2006). Spec cpu2006 benchmark descriptions. In *Proc. of of the ACM SIGARCH Computer Arch. News*, 1–17. 46, 127

[48] J. Joao, and M. A. Suleman, and O. Mutlu, and Y. N. Patt (2012). Bottleneck identification and scheduling in multithreaded applications. In *Proc. 17th Int. Conf. Architectural Support Program Languages Operating Syst.*, 223–234. 21, 77, 78, 83

[49] J. Joao, and M. A. Suleman, and O. Mutlu, and Y. N. Patt (2013). Utility-based acceleration of multithreaded applications on asymmetric cmps. In *Proc. 40th Annu. Int. Symp. Comput. Archit.*, 154–165. 77, 78, 83

[50] J. Liu and A. Yang (1974). Optimal scheduling of independent tasks on heterogeneous computing systems. In *Proc. of the 1974 annual conference*, 38–45. 132

[51] Jeff, B. (2013). big.little technology moves towards fully heterogeneous global task scheduling. Tech. rep., ARM. 26

[52] Jeffers, J., Reinders, J. & Sodani, A. (2016). *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Morgan Kaufmann. 14

[53] Jiménez, D.A. & Lin, C. (2001). Dynamic branch prediction with perceptrons. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, 197–206, IEEE. 135

[54] Jones, M. (2009). Inside the linux 2.6 completely fair scheduler. [Online] Available: `http://www.ibm.com/developerworks/library/l-completely-fair-scheduler/l-completely-fair-scheduler-pdf.pdf`. 2, 25, 50

[55] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer (2012). Scheduling heterogeneous multi-cores through performance impact estimation (pie). In *Proc. 39th Annu. Int. Symp. Comput. Archit.*, 213–224. 26, 27, 78, 79

[56] K. Van Craeynest, S. Akram, W. Heirman, A. Jaleel, and L. Eeckhout (2013). Fairness-aware scheduling on single-isa heterogeneous multi-cores. In *Proc. 22nd Int. Conf. Parallel Archit. Compilation Tech.*, 177–187. 26, 50, 78, 109, 111, 127, 133

[57] Kim, C., Burger, D. & Keckler, S. (2002). An adaptive, non-uniform cache structure for wire-dominated on-chip caches. In *Proc. of Int. Conf. Architectural Support Program Languages Operating Syst.*, 211–222. 132

[58] Kim, C., Burger, D. & Keckler, S. (2003). Nuca: a non-uniform cache access architecture for wire-delay dominated on-chip caches. 64

[59] Kongetira, P., Aingaran, K. & Olukotun, K. (2005). Niagara: A 32-way multithreaded sparc processor. *IEEE micro*, **25**, 21–29. 14

[60] Krizhevsky, A., Sutskever, I. & Hinton, G.E. (2012). magenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, 1097–1105. 2, 27

[61] Lattner, C. & Adve, V. (2004). Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, 75, IEEE Computer Society. 35

[62] LeCun, Y., Kavukcuoglu, K. & Farabet, C. (2010). Convolutional networks and applications in vision. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, 253–256, IEEE. 2, 27

[63] Li, C.V., Petrucci, V. & Mossé, D. (2016). Predicting thread profiles across core types via machine learning on heterogeneous multiprocessors. In *Computing Systems Engineering (SBESC), 2016 VI Brazilian Symposium on*, 56–62, IEEE. 134

[64] M. Annavaram, and E. Grochowski, and J. Shen (2005). Mitigating amdahl's law through epi throttling. In *Proc. 32st Annu. Int. Symp. Comput. Archit.*, 298–309. 14

[65] M. Annavaram, E. Grochowski, and J. Shen (2005). Mitigating amdahl's law through epi throttling. In *Proc. of the 32nd Ann. Symp. on Comp. Arch.*, 298–309. 132

[66] M. Becchi, and Patrick Crowley (2008). Dynamic thread assignment on heterogeneous multiprocessor architectures. *J. Instruction-Level Parallelism*, **10**, 1–26. 26, 133

[67] MATLAB (2017). *version R2016b*. The MathWorks Inc., Natick, Massachusetts. 53, 103

[68] Matsunaga, A. & Fortes, J.A. (2010). On the use of machine learning to predict the time and resources consumed by applications. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, 495–504, IEEE Computer Society. 135

[69] Miller, L. (1982). A heterogeneous multiprocessor design and the distributed scheduling of its task group workload. In *Proc. of the 9th Ann. Symp. on Comp. Arch.*, 283–290. 132

[70] Mishra, A.K., Vijaykrishnan, N. & Das, C.R. (2011). A case for heterogeneous on-chip interconnects for cmps. *ACM SIGARCH Computer Architecture News*, **39**, 389–400. 133

[71] N. Markovic, D. Nemirovsky, O. Unsal, M. Valero, and A. Cristal (2014). Thread lock section-aware scheduling on asymmetric single-isa multi-core. *IEEE Computer Architecture Letters*, **DOI:10.1109/LCA.2014.2357805**, 1. 9, 26, 50, 51, 57, 73, 81, 87

[72] N. Markovic, D. Nemirovsky, V. Milutinovic, O. Unsal, M. Valero, and A. Cristal (2015). Hardware round-robin scheduler for single-isa asymmetric multi-core. *Euro-Par 2015: Parallel Processing, Lecture Notes in Computer Science*, **9233**, 122–134. 2, 26, 50, 51, 92, 109, 111, 127

[73] Negi, A. & Kumar, P.K. (2005). Applying machine learning techniques to improve linux process scheduling. In *TENCON 2005 2005 IEEE Region 10*, 1–6, IEEE. 134

[74] NVIDIA (2011). Tegra 3 (kal-el) quad-core mobile processor. [Online] Available: `http://www.nvidia.com/object/tegra-3-processor.html`. 1, 14, 78

[75] O. Khan, and S. Kundu (2010). A self-adaptive scheduler for asymmetric multi-cores. In *Proc. of the 20th Symp. on Great lakes symposium on VLSI*, 397–400. 133

[76] O'Connor, J.M. & Tremblay, M. (1997). picojava-i: The java virtual machine in hardware. *IEEE Micro*, **17**, 45–53. 137

[77] Pabla, C.S. (2009). Completely fair scheduler. *Linux Journal*, **2009**, 4. 111, 127

[78] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M. & Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, **12**, 2825–2830. 53, 123

[79] Pimentel, A.D., Polstra, S., Terpstra, F., Van Halderen, A., Coffland, J.E. & Hertzberger, L.O. (2002). Towards efficient design space exploration of heterogeneous embedded media systems. In *Embedded Processor Design Challenges*, 57–73, Springer. 135

[80] Pinel, F. & Dorronsoro, B. (2014). Savant: Automatic generation of a parallel scheduling heuristic for map-reduce. *International Journal of Hybrid Intelligent Systems*, **11**, 287–302. 134

[81] R. Kumar, and D. M. Tullsen, and P. Ranganathan, and N. P. Jouppi, and K. I. Farkas (2004). Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *Proc. 31st Annu. Int. Symp. Comput. Archit.*, 64. 4, 14, 133

[82] R. Kumar, and K. I. Farkas, and N. P. Jouppi, and P. Ranganathan, and D. M. Tullsen (2003). Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Proc. 36nd Annu. IEEE/ACM Int. Symp. Microarchit.*, 81. 131

[83] R. Rodrigues, and A. Annamalai, and I. Koren, and S. Kundu, and O. Khan (2011). Performance per watt benefits of dynamic core morphing in asymmetric multicores. In *Proc. Int. Conf. Parallel Architectures Compilation Tech.*, 121–130. 4, 14

[84] Radojković, P., Čakarević, V., Moretó, M., Verdú, J., Pajuelo, A., Cazorla, F.J., Nemirovsky, M. & Valero, M. (2012). Optimal task assignment in multithreaded processors: a statistical approach. *ACM SIGARCH Computer Architecture News*, **40**, 235–248. 134

[85] Rai, J.K., Negi, A., Wankar, R. & Nayak, K. (2012). A machine learning based meta-scheduler for multi-core processors. In *Technological Innovations in Adaptive and Dependable Systems: Advancing Models and Concepts*, 226–238, IGI Global. 135

[86] Robison, A.D. (2013). Composable parallel patterns with intel cilk plus. *Computing in Science & Engineering*, **15**, 66–71. 31

[87] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova (2011). A case for numa-aware contention management on multicore systems. In *Proc. USENIX Annual Technical Conference (USENIXATC)*, 1–15. 134

[88] S. Li, and J. Ho Ahn, and R. D. Strong, and J. B. Brockman, and D. M. Tullsen, and N. P. Jouppi (2007). Quantifying the cost of context switch. In *Proc. Workshop Exp.Comput. Sci.*, 2–es. 27, 79

[89] S. Li, and J. Ho Ahn, and R. D. Strong, and J. B. Brockman, and D. M. Tullsen, and N. P. Jouppi (2009). Mcpat: An integrated power, area, and timing modeling framework for multicore architectures. In *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchit.*, 469–480. 16, 43, 61

[90] S. Srinivasan, and L. Zhao, and R. Illikkal, and R. Iyer (2011). Efficient interaction between os and architecture in heterogeneous platforms. *Operating Syst. Rev.*, **45**, 62–72. 133

[91] S. Woo, and M. Ohara, and E. Torrie, and J. P. Singh, and A. Gupt (1995). The splash-2 programs: Characterization and methodological considerations. In *Proc. 22nd Annu. Symp. Comput. Archit.*, 24–36. 46

[92] Saez, J.C., Pousa, A., Castro, F., Chaver, D. & Prieto-Matias, M. (2017). Towards completely fair scheduling on asymmetric single-isa multicore processors. *Journal of Parallel and Distributed Computing*, **102**, 115–131. 133

[93] Schoeberl, M. (2008). A java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, **54**, 265–286. 137

[94] Sherwood, T., Perelman, E., Hamerly, G., Sair, S. & Calder, B. (2003). Discovering and exploiting program phases. *IEEE micro*, **23**, 84–93. 21, 135

[95] Shulga, D., Kapustin, A., Kozlov, A., Kozyrev, A. & Rovnyagin, M. (2016). The scheduling based on machine learning for heterogeneous cpu/gpu systems. In *NW Russia Young Researchers in Electrical and Electronic Engineering Conference (EIConRusNW), 2016 IEEE*, 345–348, IEEE. 135

[96] Smith, J.E. (1998). Decoupled access/execute computer architectures. In *25 Years of the International Symposia on Computer Architecture (Selected Papers)*, ISCA '98, 231–238, ACM, New York, NY, USA. 137

[97] Smith, J.E., Dermer, G.E., Vanderwarn, B.D., Klinger, S.D. & Rozewski, C.M. (1987). The zs-1 central processor. *SIGPLAN Not.*, **22**, 199–204. 137

[98] Smith, J.E., Sastry, S., Heil, T. & Bezenek, T.M. (1998). Achieving high performance via co-designed virtual machines. In *Innovative Architecture for Future Generation High-Performance Processors and Systems, 1998*, 77–84, IEEE. 33

[99] Smith, J.E., Sastry, S., Heil, T. & Bezenek, T. (1999). Achieving high performance via co-designed virtual machines. In *In International Workshop on Innovative Architecture*, 77–84. 136

[100] Snavely, A. & Tullsen, D.M. (2000). Symbiotic jobscheduling for a simultaneous mutlithreading processor. *ACM SIGPLAN Notices*, **35**, 234–244. 133

[101] Sniper (2015). The sniper multi-core simulator. [Online] Available: `http://snipersim.org`. 43

[102] Stone, J.E., Gohara, D. & Shi, G. (2010). Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, **12**, 66–73. 31

[103] Strigl, D., Kofler, K. & Podlipnig, S. (2010). Performance and scalability of gpu-based convolutional neural networks. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, 317–324, IEEE. 3

[104] Suleman, M.A., Hashemi, M., Wilkerson, C., Patt, Y.N. *et al.* (2012). Morphcore: An energy-efficient microarchitecture for high performance ilp and high throughput tlp. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 305–316, IEEE Computer Society. 131

[105] T. E. Carlson, and W. Heirman, and L. Eeckhout (2011). Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, 1–12. 41

[106] T. Li, and D. Baumberger, and D. A. Koufaty, and S. Hahn (2007). Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Proc. of the ACM/IEEE conference on Supercomputing*, 53. 14

[107] Treleaven, P. (1991). Neural computing and the galatea project. In *Proceedings on Parallel Architectures and Languages Europe*, 25–33, Springer-Verlag New York, Inc., New York, NY, USA. 136

[108] Ungar, D., Blau, R., Foley, P., Samples, D. & Patterson, D. (1984). Architecture of soar: Smalltalk on a risc. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, ISCA '84, 188–197, ACM, New York, NY, USA. 137

[109] Unsal, O.S., Koren, I., Khrishna, C. & Moritz, C.A. (2004). Cool-fetch: A compiler-enabled ipc estimation based framework for energy reduction. In *Interaction between Compilers and Computer Architectures, 2004. INTERACT-8 2004. Eighth Workshop on*, 43–52, IEEE. 99, 135

[110] Van Haastregt, S. & Knijnenburg, P. (2007). Feasibility of combined area and performance optimization for superscalar processors using random search. In *Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE'07*, 1–6, IEEE. 132

[111] Witten, I. & Cleary, J. (1983). An introduction to the architecture of the intel iapx 432. *Software and Microsystems*, **2**, 29–34. 137