Hardware Thread Scheduling Algorithms for Single-ISA Asymmetric CMPs

Nikola Markovic

Department of Computer Architecture

Universitat Politècnica de Catalunya

A dissertation submitted in fulfillment of
the requirements for the degree of

Doctor of Philosophy / Doctor per la UPC

November 2015

Advisor: Adrian Cristal
Coadvisors: Mateo Valero, Osman Unsal

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola de Doctorat

Acta de calificación de tesis doctoral	Curso académico:

Nombre y apellidos

Programa de doctorado

Unidad estructural responsable del programa

Resolución del Tribunal

Reunido el Tribunal designado a tal efecto, el doctorando / la doctoranda expone el tema de la su tesis doctoral titulada _____
_____.

Acabada la lectura y después de dar respuesta a las cuestiones formuladas por los miembros titulares del tribunal, éste otorga la calificación:

☐ NO APTO ☐ APROBADO ☐ NOTABLE ☐ SOBRESALIENTE

(Nombre, apellidos y firma) Presidente/a		(Nombre, apellidos y firma) Secretario/a
(Nombre, apellidos y firma) Vocal	(Nombre, apellidos y firma) Vocal	(Nombre, apellidos y firma) Vocal

_____, _____ de _____ de _____

El resultado del escrutinio de los votos emitidos por los miembros titulares del tribunal, efectuado por la Escuela de Doctorado, a instancia de la Comisión de Doctorado de la UPC, otorga la MENCIÓN CUM LAUDE:

☐ SÍ ☐ NO

(Nombre, apellidos y firma) Presidente de la Comisión Permanente de la Escuela de Doctorado	(Nombre, apellidos y firma) Secretaria de la Comisión Permanente de la Escuela de Doctorado

Barcelona a _____ de _____ de _____

Abstract

Through the past several decades, based on the Moore´s law, the semiconductor industry was doubling the number of transistors on the single chip roughly every eighteen months. For a long time this continuous increase in transistor budget drove the increase in performance as the processors continued to exploit the instruction level parallelism (ILP) of the sequential programs. This pattern hit the wall in the early years of the twentieth century when designing larger and more complex cores became difficult because of the power and complexity reasons. Computer architects responded by integrating many cores on the same die thereby creating Chip Multicore Processors (CMP). In the last decade, the computing technology experienced tremendous developments, Chip Multiprocessors (CMP) expanded from the symmetric and homogeneous to the asymmetric or heterogeneous Multiprocessors. Having cores of different types in a single processor enables optimizing performance, power and energy efficiency for a wider range of workloads. It enables chip designers to employ specialization (that is, we can use each type of core for the type of computation where it delivers the best performance/energy trade-off). The benefits of Asymmetric Chip Multiprocessors (ACMP) are intuitive as it is well known that different workloads have different resource requirements.

The CMPs improve the performance of applications by exploiting the Thread Level Parallelism (TLP). Parallel applications relying on multiple threads must be efficiently managed and dispatched for execution if the parallelism is to be properly exploited. 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. Thus, dynamic thread scheduling techniques are of paramount importance in ACMP designs since they can make or break performance benefits derived from the asymmetric hardware or parallel software. Several thread scheduling methods have been proposed and applied to ACMPs.

In this thesis, we first study the state of the art thread scheduling techniques and identify the main reasons limiting the thread level parallelism

in an ACMP systems. We propose three novel approaches to schedule and manage threads and exploit thread level parallelism implemented in hardware, instead of perpetuating the trend of performing more complex thread scheduling in the operating system. Our first goal is to improve the performance of an ACMP systems by improving thread scheduling at the hardware level. We also show that the hardware thread scheduling reduces the energy consumption of an ACMP systems by allowing better utilization of the underlying hardware.

Acknowledgement

This is to express my gratitude to the Universitat Politecnica de Catalunya (UPC) committee and the Barcelona Supercomputing Center (BSC) for providing me with the necessary resources that allowed me to perform my PhD studies in a highly competitive, but at the same time friendly international atmosphere. A special thank goes to Prof. Mateo Valero and Computer Science - Computer Architecture for Parallel Paradigms research group for giving me an opportunity to work in the challenging field of Hardware Thread Scheduling. I would further like to thank my advisors Adrian Cristal and Osman Unsal. I really appreciate their valuable guidance, constructive criticism and the time and concentration they spent reading all the versions of this thesis. Many thanks go to all my current and past colleagues, with whom I spent last seven years (Srdjan, Vladimir, Gokcen, Nehir, Vesna, Milovan, Adria, Oriol, Gulay, Ferad, Oscar, Tim, Xavier, Gina, Sasa, Mario, Damian, Azam, and Santosh). Maybe the best achievement in Barcelona is getting to know them. Furthermore a special thank goes to my friends (Aleksandar, Darko, Milan, Daniel, Jelena and Nikola) and my girlfriend (Marica) for encouraging me during my PhD time. I wish to sincerely thank my family (my mother Olga, my father Radovan and my brothers Nenad and Novak) for supporting me, morally above all, but also financially, and sharing all the useful life experience.

Nikola Markovic

Contents

List of Figures

List of Tables

1 Introduction

The relentless push in technology scaling driven by Moore's law has resulted in more transistors packed into a very small area. Overtime, computer architects responded by integrating many cores on the same die. Chip multiprocessors, or CMPs for short, are now the most common way to build high-performance microprocessors, for a variety of reasons. First of al, large uniprocessors are no longer scaling in performance, because it is only possible to extract a limited amount of parallelism from a typical instruction stream using conventional superscalar instruction issue techniques. In addition, one cannot simply ratchet up the clock speed on today's processors, or the power budget would become prohibitive. Compounding these problems is the simple fact that with the immense numbers of transistors available on today's microprocessor chips, it is too costly to design and debug ever-larger single-core processors every year or two.

CMPs avoid these problems by filling up a processor die with multiple, relatively simpler processor cores instead of one huge core. The complexity of a CMP's cores can vary from very simple pipelines to moderately complex superscalar processors,

but once a core has been selected the CMP's performance can easily scale across silicon process generations simply by stamping down more copies of the hard-to-design, high-speed processor core in each successive chip generation.

In addition, parallel code execution, obtained by spreading multiple threads of execution across the various cores, can achieve significantly higher performance than would be possible using only a single core with comparable amount of resources. Though, this is truth, it is not that simple to utilize the parallelism bearing in mind the interconnection networks, the cache coherency, cache/memiry bandwidth etc. of the CMP. While parallel threads are already common in many useful workloads, there are important workloads that are hard to divide into parallel threads. Nevertheless, the low inter-processor communication latency between the cores in a CMP helps make a much wider range of applications viable candidates for parallel execution than was possible with traditional, multi-chip multiprocessors.

1.1 Motivation and Scope of the Thesis

Nowadays, chip multiprocessors (CMPs) may be symmetric (SCMP), consisting of many cores of the same type, or asymmetric (ACMP), where cores may differ from one another with respect to their functionality and/or performance [2], [3]. As is shown by a number of recent studies ACMPs are likely to outperform SCMPs for a fixed budget (area or power or both) [4], [5]. Since it is well known that different workloads have different resource requirements, the benefits of ACMPs are intuitive.

The exciting rise of asymmetric multi core processors (ACMPs) has fostered a critical reevaluation of the traditional scheduling mechanisms in order to take full advantage of the new hardware resources in relation to the increasingly common thread level parallelism as well as in meeting certain system performance and power requisites. The operating system scheduler module orchestrates critical execution time junctures, selecting which jobs to be admitted next into the system and the next process to run. A technique known as fair-share scheduling is used by computer operating systems where CPU usage is equitably divided between system users or groups, in contrast to equal distribution among processes. The Linux OS scheduler, based on a fair-share scheduler strategy, is a process scheduler which was merged into the 2.6.23 release of the Linux kernel as its default scheduler [6]. It handles CPU resource allo-

cation for executing processes aimed at maximizing overall CPU utilization as well as interactive performance. Operating systems may feature up to three distinct types of schedulers, a long-term scheduler (also known as an admission scheduler or high-level scheduler), a mid-term or medium-term scheduler and a short-term or CPU scheduler. The names suggest the relative frequency with which these functions are performed.

The third type of scheduler, the short-term scheduler, commonly referred to as the CPU scheduler is the primary focus of this work. It is responsible for determining which of the ready processes (loaded into the memory by the other schedulers) should be sent for execution and on which computational core. This decision takes place periodically at interrupt points caused principally by the clock, I/O events, or OS system level calls. In relation to the long and short-term schedulers, the short-term scheduler must make scheduling decisions much more frequently. Furthermore, the short-term scheduler can be preemptive or non-preemptive based on its ability to force processes off the CPU. The preemptive method depends on a programmable interval timer that invokes a kernel level interrupt handler which implements the scheduling algorithm. A key function involved in the CPU-scheduling decision is the dispatcher which gives control of the CPU to the process selected. This function involves the context switching, changing to user mode, and jumping to the proper location of a program once it is restarted. The actual time it takes for the dispatcher to perform its job stopping one process and starting another is known as the dispatch latency typically requiring several thousands of cycles [7]. Since the dispatcher needs to analyze the program counter values, fetch instructions, and load data into the registers of the CPU every time a process switch occurs, minimizing the dispatcher latency should be a primary objective. Moreover, it is also important to avoid unnecessary context switches due to the fact that the processor remains idle for a period of time during context switches.

1.1.1 Importance of Scheduling

The need for a scheduling algorithm arises from the requirement for CMP and ACMP systems to perform multitasking (executing more than one process or thread at a time). Scheduling is the method by which threads, processes or data flows are given access per core to system resources (e.g. processor time). This is usually done to load balance and share system resources effectively or to achieve a target quality of service. Parallel

applications relying on multiple threads must be efficiently managed and dispatched for execution if the parallelism is to be properly exploited.

Multi-cores usually operate under the shared memory model, allowing parallel tasks of an application to cooperate by concurrently accessing shared resources using a common address space. Each task can be seen as a sequential thread of execution that performes useful computation. Thus, a parallel programming model has to create and manage several tasks that need to synchronize and communicate to each other. However, having concurrent parallel tasks may introduce several new classes of potential problems, of which data races (e.g., data dependencies) are the most common [8]. Today's programming models commonly target this problem via lock-based approaches.

Unfortunately, when using locks, programmers must pick between two undesirable choices. Use coarse-grain locks, where large regions of code are included as critical regions. This makes the task of adding coarse-grain locks to a program quite straightforward, but introduces unnecessary serialization that degrades system performance. On the other side, fine-grain locking aims at critical sections of minimum size. Smaller critical sections permit greater concurrency, and thus scalability. However, this scheme leads to higher complexity, and it is usually difficult to prove the correctness of the resulting algorithm.

These two choices establish a programming effort versus performance trade-off. The complexity associated with fine-grain locking can lead to incorrect synchronization, e.g., data races, which could manifest in the form of non-deterministic execution, producing incorrect results for certain executions of an application. This fact makes lock based programs difficult to debug, because bugs are hard to reproduce. Synchronization errors may also result in deadlock or livelock conditions. Using multiple locks requires strict programmer discipline to avoid cyclic dependencies where two or more threads create circular requests to acquire locks, leading to a deadlock scenario where threads are blocked and no forward progress is made. On the other hand, livelocks occur when two or more threads cease to make forward progress while performing the same piece of work repeatedly.

Nevertheless, even correctly parallelised applications may behave poorly due to coherence or unnecessary contention in critical sections caused by inappropriate thread scheduling on the CMP. Parallel applications have to modify a certain amount of

shared data. Modifying the same data in different cores causes cache-lines to move between private caches, penalizing system throughput and overall application performance. Correct mapping of parallel threads to the underlying cores is of utmost importance for further development of the chip multiprocessors as well as parallel applications.

Goal of the Thesis

Parallel applications relying on multiple threads must be efficiently managed and dispatched for execution if the parallelism is to be properly exploited. Thus, dynamic thread scheduling techniques are of paramount importance in ACMP designs since they can make or break performance benefits derived from the asymmetric hardware or parallel software. Several thread scheduling methods have been proposed and applied to ACMPs. Most 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 [9], [10], [11] among others. Though these scheduling techniques include certain performance or energy efficiency gains, their broad application remains stifled due to scalability limitations, runtime overheads, and additional hardware requirements and complexities. Our goal is to develop a scheduling policy that can be used as a foundation upon which to build practical and scalable hardware scheduling designs in order to increase the performance capabilities of ACMPs.

1.2 Key Challenges

Although parallel applications and multiapplication workloads provide opportunities for greater performance and efficiency gains, thanks in part to the amount of hardware resources or cores that can be activated to execute the different parallel portions of the workload. In order to take full advantage of these parallelism opportunities, several key concerns must be tackled, namely application bottlenecks, runtime imbalances of multithreaded programs, workload characteristics, and implementation complexity of the scheduling policy.

Firstly, to tackle the runtime imbalances, we can consider and implement different

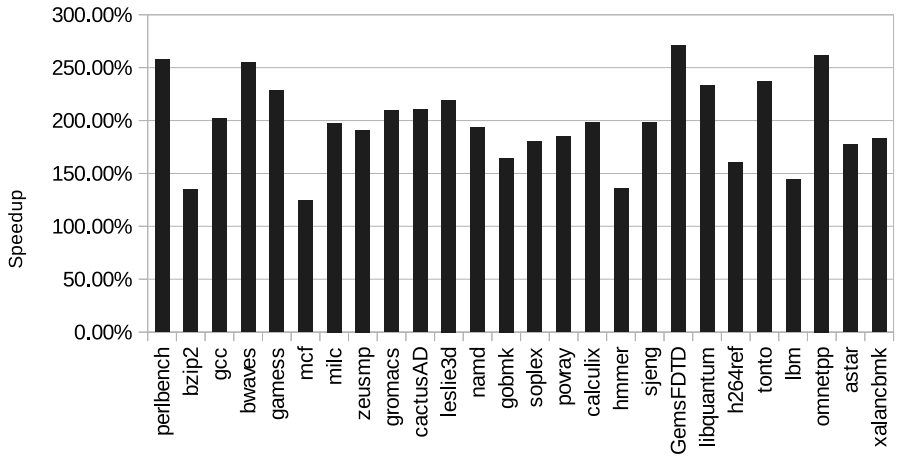

Figure 1.1: Large core speedup compared to the small cores when running the SPEC2006 benchmark.

thread scheduling techniques to achieve as much fairness as possible. For example, Van Craeynest et al. showed that in a asymmetric multicore system, a round-robin scheduler using threads pinned to cores produces no speedup compared to a lighter symmetric multicore system for most multithreaded benchmarks [12]. This behavior is caused by barrier-synchronized multithreaded workloads, because the execution progress is limited by the slowest thread. This has little meaning in a symmetric system but is significant for asymmetric systems, because the thread pinned to the simplest core will be the weakest link that all other threads will have to wait for at every barrier.

Work-stealing workloads, in contrast, allow idle large cores to steal work that normally would be run on the small cores, so that the execution time isn't as constrained. Therefore, in asymmetric multicore systems, guaranteeing fairness is fundamental for improving performance for barrier-synchronized multithreaded workloads. Fairness, defined as giving each thread equal execution time on each core or allowing each thread to make equal progress, enables all threads to reach the barriers simultaneously, and has been shown to provide average performance improvements of 14 percent (and up to 25 percent) compared with a pinned scheduler [12] for the system configuration we are using.

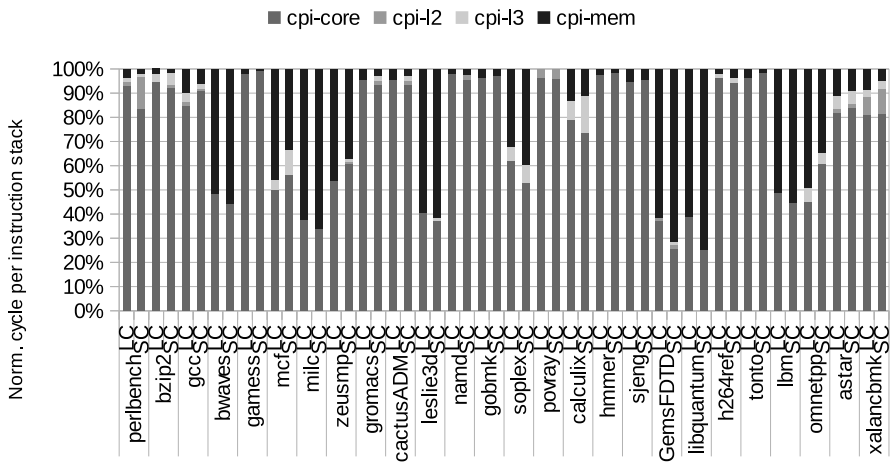

Figure 1.2: Normalized cycle per instruction stack breakdown for the large core and for the small cores when running the SPEC2006 benchmark.

Secondly, bottlenecks arising from distinct thread and memory management and sharing approaches have significant effects on performance and system usage efficiency. Moreover, code segments that produce long waits for threads tend to reduce the total amount of thread-level parallelism and can even negate the potential speedup gained by parallelization of an application. In particular, inter-thread synchronization bottlenecks (such as contended critical sections), as well as different memory structures and behaviors, can cause thread imbalances at runtime, leading to adverse performance effects. The state of the art in bottleneck acceleration on an ACMP is bottleneck identification and scheduling (BIS) [13], which outperforms the best mechanisms by 15 percent for single-application workloads. Utility-based acceleration (UBA) of multithreaded applications on ACMPs provides about 8.2 percent of additional performance improvement over BIS for an ACMP with one large core in the system [14].

Thirdly, the performance behavior of workloads on small and large cores (Fig. 1.1) can be explained by the design characteristics of each core. Fig. 1.1 compares the slowdown for SPEC CPU2006 workloads on a small core relative to a large core. A particular workload can be characterized as either compute-intensive or memory-intensive based upon the normalized CPI stack as given in Fig. 1.2. For example, milc has a memory dominant CPI stack making it memory intensive whilst the reverse is

true for tonto. These distinctions are relevant to asymmetric systems because workloads will perform differently on different cores based on their characteristics. For instance, workloads that contain large amounts of memory level parallelism (MLP) or instruction level parallelism (ILP) would be more suited to run on large cores capable of out of order execution. Conversely, workloads consisting of high amounts of thread level parallelism (TLP) and explicit ILP (i.e. parallelism that does not require dynamic extraction) are good candidates for execution on smaller cores. For example, a compute-intensive workload consisting of significant ILP may achieve adequate performance on a small core and leave the large core vacant for memory-intensive workloads which would suffer from substantial performance degradation if run on a small core. Therefore, in an asymmetric system, it may be useful to correlate a workload's execution performance with a particular core type in order to dynamically ascertain the workload's characteristics and improve the scheduler strategy.

Finally, scheduling policy applicability is directly affected by the complexity of its implementation. Most of scheduling methods proposed and applied to SCMPs as well as to ACMPs rely on the sampling or other estimation techniques, together with online or offline profiling of the performance and/or power to decide up on the most favorable thread to core mapping, when a particular event is detected or scheduling time quantum is completed, [4], [11], [15] among others. They usually require ISA extensions or changes to be made in OS or user application code along with several tables and necessary logic, implemented in hardware, to keep and manage information about threads.

1.3 Contributions of Thesis

We present the contributions of this thesis that address the previously mentioned challenges that thread scheduling faces on an ACMP in the Section 1.2.

1.3.1 Hardware Round-Robin Scheduling Algorithm

To tackle the problem of runtime imbalances during execution of multiple threads, we propose the Hardware Round-Robin Scheduling (HRRS) policy. This technique is influenced by Fairness Scheduling techniques thereby reducing thread serialization

and improving parallel thread performance.

The HRRS technique remaps hardware threads on the actual physical cores of the system at every hardware scheduling quantum, while Operating System maps and reschedules software threads on those hardware threads. The scheduler remaps hardware threads between different core types (such as large - Out-of-Order and small - In-Order) in a round-robin fashion after every time quantum expires. In this way, the HRRS approach fairly balances the workload among different cores in as ACMP system. In contrast, the Fairness Scheduler technique [12] has special hardware structure that keeps track of all software threads while mapping and rescheduling them on actual physical cores. This is explained in more detail in the section 2.2.5. Since scheduling policy applicability is directly affected by the complexity of its implementation, a possible hardware implementation of the Hardware Round-Robin Scheduling policy is also given and discussed.

We analyze the performance and energy efficiency of the HRRS policy on an ACMP and compare it to the Fairness Scheduler when running multi-threaded application workloads and when running multiple instances of the single-threaded application workloads respectively. We show that it lowers total execution time by 17.2 percent and 11.71 percent on average, while being on average 7.57 percent and 6.56 percent more energy efficient respectively.

1.3.2 Kernel to User mode Transition aware Hardware Scheduling Algorithm

The bottlenecks arising from a particular thread and memory management and sharing approaches may have significant effects on performance of the multithreaded applications. To tackle this issue, we propose a Kernel to User Mode Transition aware Hardware Scheduling Algorithm (KUTHS), which is influenced by the scheduling for fairness as well as the bottleneck identification techniques thereby reducing thread serialization and improving parallel thread performance.

The KUTHS technique builds upon the previously proposed HRRS technique. The KUTHS scheduler utilizes the kernel to user code execution transitions on cores to identify bottlenecks in multithreaded applications and make the rescheduling decision. These transitions occur when Operating System makes execution switches on

the hardware threads from executing kernel to user code and vice versa. The KUTHS remaps hardware threads after every time quantum expires. Presuming an ACMP system composed of an Out-of-Order and an In-Order core, if the "transition" does not occur on any of the cores the KUTHS remaps hardware threads between different core types in the round-robin, like the HRRS scheduler. On the other hand, if the "transition" occurs on one of the cores, the KUTHS will promote hardware thread from that core to run on an Out-of-Order core.

We analyze and evaluate the performance of the KUTHS policy on an ACMP and show that it lowers total execution time of the application by 11.5 percent (geometric mean) compared to a Fairness-aware Scheduler. It lowers total execution time of the single application and the mix of two applications by 11.1 and 8.7 percent respectively (geometric mean) compared to a Fairness-aware Scheduler on shared LLC configuration. Besides evaluating performance, we also analyze the energy efficiency of the KUTHS scheduler on an ACMP with a shared LLC and show that it is on average 9.4 percent more energy efficient than the Fairness Scheduler.

Application of KUTHS on Manycore processors

We propose the extension of the KUTHS policy to be applicable to larger many-core systems and systems where the last-level caches are private to the group of cores. It is influenced by the Fairness-aware Scheduling and bottleneck identification techniques and thereby aims at reducing thread serialization and improving parallel thread performance.An analysis and evaluation is given for the performance of the KUTHS policy on a private LLC ACMP and show that it lowers total execution time of the application by 30 percent (geometric mean) compared to Linux OS Scheduler (ACMP Scheduler).

1.3.3 Trait-aware Criticality Scheduling Algorithm

When scheduling a workload on an ACMP the performance will be affected by the fairness of scheduling, schedulers ability to identify application bottlenecks and to make the scheduling decisions based on the workload's characteristics. In order to tackle all three important issues that scheduling of different types of workloads may encounter on an AMCP, we propose a Trait-aware Criticality Scheduling (TCS) policy which is influenced by scheduling fairness, criticality, and a workload's charac-

teristics, thereby reducing thread serialization and improving performance of multi-threaded applications and multiprocess workloads on single-ISA asymmetric multi-cores.

The TCS method develops upon the earlier introduced HRRS method. The underlying technique of the TCS scheduling already tackles the problem of fairness, since the HRRS remaps hardware threads in round-robin way over different core types in an ACMP system after every time quantum expires. Besides, the TCS scheduling employs the short-term traits of workloads executing on hardware threads to improve the decision-making method during scheduling in a way that is more accustom with distinct characteristics of particular workloads. The TCS scheduler additionally uses the long-term traits of the software threads during the decision-making process to tackle the software thread criticality. A possible hardware implementation of the Trait-aware Criticality Scheduling policy is elaborated on since it is crucial for the applicability of the technique.

An analysis of the performance of the TCS policy on an ACMP is also given and is compared to the state-of-the-art schedulers in scheduling for fairness, criticality, and a workload's characteristics. It achieves an average speed up of 11 percent and 24.4 percent over the state-of-the-art mechanisms for scheduling for fairness and scheduling for criticality respectively. While being by only 2.6 percent slower compared to the state-of-the-art mechanisms on scheduling for workload characteristics.

1.4 Thesis Organization

Chapter 2 presents the scheduling in a computer system and the state-of-the art for thread scheduling schemes.
Chapter 3 describes benchmark suites and the simulator used for the experimental setup in this thesis.
Chapter 4 presents the cost that context switch of threads on the cores may have on the total execution time of the application running on the Chip Multiprocessor.
Chapter 5 introduces Hardware Round-Robin Scheduling Algorithm (HRRS) for Hardware Threads on the single-ISA asymmetric CMPs, a scheduler based on fairness scheduling techniques.
Chapter 6 presents Kernel to User mode Transition aware Hardware Scheduling Algo-

rithm (KUTHS), a scheduler based on scheduling for fairness amended by bottleneck identification for the single-ISA asymmetric CMPs.
Chapter 7 explains Trait-aware Criticality Scheduling Algorithm for Hardware Threads on the single-ISA asymmetric CMPs, a scheduler seeking the aggregation of the scheduling for fairness, scheduling for bottleneck identification and scheduling workload characteristics techniques.
Chapter 8 concludes this dissertation.

2

State-of-the-Art and Background in Scheduling Policies on an ACMP

This chapter presents the background that is relevant to the thesis. The first section explains more broadly the concept of context switch in the chip multiprocessors and the implications it has on the scheduling policies. The chapter continues with the survey of the current state-of-the-art scheduling techniques used in multiprocessor computer architecture.

2.1 Context Switching

Over the past few decades since the first CMPs emerged in 90's, both of the industry's and academia's attention have been keenly focused on conducting studies related to the practice and cost of context switching when moving thread from one core to the other. In the following sections we summarize and compare and contrast the most prominent studies which fall under three distinct categories.

2.1.1 Performance Impact of Context Switching

Several context switching studies aimed at understanding the performance impact of context switching events. For instance, Agarwal et al. [16] demonstrated that multiprogramming execution substantially degrades cache performance and increases in impact as the cache size grows. Mogul et al. [17] conducted a performance reduction estimation based on the cost of context switching and found it to be on the order of tens to hundreds of microseconds, depending on the specific cache parameters. Suh et al. [18] measured the impact of context switching on page faults and performance and endorsed a speculative prefetching scheme to mitigate the performance penalties. A novel approach is proposed by Chiou et al. [19] who suggest that contrary to conventional CPU scheduling practices, it should be the memory scheduling method, potentially at all levels of the memory hierarchy, that should drive CPU scheduling and not the other way around.

Koka et al. [20] characterized context switch misses by quantifying their effects when running transactional workloads. They evaluated the possibility of intelligent process scheduling that would minimize cache misses across context switch boundaries. In other work, Li et al. [7] concluded that overheads due to burdensome cache activity from indirect context switches is more significant than the direct overhead, where direct context switches are those that occure when rescheduling threads on the same core while indirect are those that happend when moving thread from one to the other core. Similarly, Tsafrir [21] and David et al. [22] measured the separate indirect overheads caused by context switch events for the Intel and ARM platforms. In sum, nearly all of the relevant studies found that indirect overheads caused by cache activities associated with context switch events are significant. As a result, we have kept context switch overheads as a key factor to minimize in our work.

2.1.2 Analytical Models for characterizing Context Switches

Various analytical models have been proposed which seek to justify the relationship between an application's temporal execution characteristics and its behavioral vulnerability to context switch misses. Such models require an ample scope of essential variables critical to the hardware, memory, and application workload and behavior in order to achieve an adequate conclusion detailing causality. An example of such work

is the analytical model proposed by Agarwal et al. [16] and Suh et al. [23], [24] which estimate overall cache miss rates taking into regard context switch misses. In order to obtain a continuous cache miss rate curve, Suh et al. relied on a model based upon a small fully associative cache. However, applying a fully associative cache structure to the LLC would be unrealistic and would lead to inaccurate results. The model proposed by Hwu et al. [25] focused on predicting the quantity of context switch misses for the worst case scenario. Liu et al. [26] grouped context switch misses into two classes, namely replaced misses and reordered misses. In addition, their work elaborated on an analytical model that reveals the causal correlation between cache design parameters, an application's temporal execution pattern, and the amount of context switch misses which appear during the execution of the application. This model was utilized to study the effect which prefetching and cache size can have on the amount of context switch misses.

Certain assumptions must necessarily be made when developing analytical models. For example, the model designed by Liu et al. [26] is built with the assumption of an LRU replacement policy implementation even though alternative replacement algorithms have been advanced which perform better. Another characteristic particular to analytical models to consider is that though they may be suitable for offline analysis, they often lack implementation feasibility in hardware regarding incurring a low area overhead.

2.1.3 Managing Cache Misses caused by Context Switches

Performance degradation penalties due to context switch misses may be addressed via two channels, namely either increasing the scheduling time slice or prefetching the cache state right before the new scheduling mapping is initiated. Increasing time slice is a preventive measure while cache state prefetching is more of a remedy. The general idea behind prefetching is to measure and save an application's locality at the moment when it gets swapped out due to a context switch. The locality is restored through prefetching the next time the application gets CPU time thereby mitigating the cost of additional cache misses incurred due to the context switch event. Previously advanced solutions that make use of prefetching differ in how the locality is stored and restored. An example is the work done by Cui et al. [27] who apply a Global-

history-list (GHL) in their method of prefetching. The GHL technique maintains a complete list of cache lines ordered by temporal of usage behavior. Another case is that of Daly et al. [28] who evaluated the impact of context switch miss events in highly partitioned virtualized systems. They proposed to warm the cache state by prefetching the application's working set and restoring the cache lines. The GHL and cache restoration methods differ in implementation specifics to some extent but perform relatively on par with one another although GHL performs slightly better at the cost of additional hardware overhead and complexity. In some of the most recent related work [29], the authors describe methods that mitigate the bandwidth overhead of these other prefetchers.

Brown et al. [30] proposed accelerating thread performance after a context switch event by predicting and prefetching the working set of the application. In their proposal, the access behavior of a thread is measured and saved in a compact form during pre-migration. This saved behavior is used to prefetch appropriate data to create a warm state on the core which the thread has been migrated to. Prefetching the data after a context switch event can help to solve or alleviate the problem of cold starts. However, this approach works to minimize the number of cold starts for those applications for which it matters. Zebchuk et al. [29] help to demonstrate the inability of all cache restoration prefetching techniques to dynamically adapt to the workload behavior as their main limitation.

2.2 Scheduling Policies

Due to possible performance and efficiency gains, there has been increasing interest in heterogeneous multicore architectures, and various scheduling proposals have been presented throughout the past two decades. In this section, we present the state-of-the art of the scheduling schemes on the Chip multiprocessors. We summarize, compare, and contrast the works under five different categories.

2.2.1 First Studies on Scheduling Policies on an ACMP

Since the early nineties and the introduction of the idea of the chip multiprocessor, research community have recognized the impact of scheduling. Miller [31] presented

a scheduling algorithm for an asymmetric system called Single Architecture Heterogeneous Multiprocessor or SAHM, which did not support multi-programming. Grochowski et al. [32] studied the usefulness of such cores for saving energy and improving throughput. Moncrieff et al. [33] and Menasce et al. [34] analytically studied tradeoffs of fast and slow processors in heterogeneous systems. They observe that a system with many slow and few fast processors are cost and performance effective.

An ACMP which consists of multiple cores of the same ISA but of different sizes was proposed by Kumar et al. in [35]. Their process consists of sampling for and choosing the core that will execute in the most power efficient manner each time a new phase or program is detected. This work was later expanded to include performance maximization of multithreaded applications [4].

2.2.2 Scheduling based on Profiling and Sampling

Work by Becchi [9] 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 increases. Other work in this area has been done by Saez et al. [36] 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. [10] determine optimal thread to ACMP core mapping using a biasing method estimated by the quantity of external memory stalls and internal pipeline stalls. Another approach is detailed in the work by Srinivasan et al. [15] who propose a formula based ACMP thread to core scheduling method which is used to estimate and compare thread performance on individual cores.

A main concern of sampling approaches towards thread scheduling is the runtime overhead produced when running multithreaded applications. In their work, [37] Shelepov et al. use an architectural signature based on the cache misses for different thread to core mappings collected during offline profiling in order to appropriately schedule threads and a reduce runtime overhead. However, since the signatures are

fixed and used for the duration of a program, the approach fails to take into regard the phases of the programs. On the other hand, phase classification and regression analysis is used by Khan et al. in their work [38] to optimize thread to core mapping in an ACMP.

2.2.3 Scheduling focused on Power and Energy

The first works that address the energy issue in shared-memory multiprocessors look at energy savings in cache coherence management [39], [40]. Motivated by the perception that a large part of snoops do not locate copies of data in multiple of the other caches in a snooping bus-based SMP, Moshovos, et al [39] propose Jetty to reduce the energy consumed by snoop requests. Saldanha and Lipasti [40] study the impacts of decreasing speculation in a scalable snoop-based scheme, and note significant potential of energy conservation by utilizing serial snooping for load misses.

Instead, the thrifty barrier [41] works on the CPU energy savings potential stemming from barrier imbalance in parallel applications. These techniques are not exclusive of each other and could be combined for improved overall results. With respect to microarchitectural differences, Chen et al. [42] chose to implement their ACMP with cores consisting of separate branch predictor, issue width, and L1 cache sizes that together with their scheduling method, achieve throughput and energy efficiency improvements. In a separate scheduling approach, Annavaram et al. [43] focus on staying withing a power budget by measuring the Energy per Instruction of an ACMP running multithreaded applications and using the small cores to run the parallel sections of code and then migrating to execute on the larger cores for the sequential sections. Scheduling based power management techniques have also been studied in the work by Winter et al. [44] which used several sampling based algorithms to analyze the optimal thread to core mapping.

Liu et al. [26] studied optimal scheduling of independent programs on a preemptive heterogeneous multiprocessor system. They use past non-critical thread barrier stall times to predict future thread criticality and DVFS accordingly. In contrast to this history-based approach, thread criticality predictor [45] predicts thread criticality based on current behavior, regardless of barriers. They also use the predictor to improve threading building blocks task stealing, building from the occupancy-based ap-

proach of Contreras and Martonosi [46]. But the work by Bhattacharjee and Martonosi [45] is distincts, in that they use criticality to guide task stealing for performance gains with little hardware overhead.

2.2.4 Scheduling based on Workload Characteristics

Exploiting counters for the improvement of hardware-aware scheduling policies has been elaborated on in related research work [47]. For instance, hardware performance counters are utilized to coordinate the scheduling of various independent threads which do not belong to the same application in order to reduce power consumption on CMPs. Along with similar approaches [48], [49], [50], these works demonstrate how low-level counter measurements can potentially greatly benefit the optimization of performance. The authors of [51] make use of cache-related performance measurements in order to enforce threads sharing a common data structure to also share a common last-level cache.

In a similar work, [52] the researchers demonstrate a NUMA-aware scheduler based on performance counters. Their scheduler observes memory related counters and calculates which corresponding threads are sharing data on a common Non-Uniform Memory Access (NUMA) node. The scheduler, therefore, can manage thread mapping tasks easier by placing them based on the same resource they are sharing common to the most efficient NUMA node. While approach is particular to OpenMP type of parallel applications, a more generic approach is presented in [53]where a NUMA-aware scheduler has also been introduced. The authors demonstrate that schedulers which are not necessarily aware of the hardware architecture could very well have a negative impact upon performance. Another non-NUMA approach has also been presented in [54]. The whole of these approaches best illustrate how detailed and precise scheduling policies improve performance when there is contention of hardware resources. Lastly, the works [55] and [56] provide an extra discussion of the accuracy and the benefits of different counters measurement libraries and approaches.

2.2.5 Fairness-aware Scheduling

For heterogeneous CMPs based on a single-ISA, optimizing scheduling based on a fairness criteria is of key significance. There are several fairness scheduling schemes available but an excellent example of achieving equal time and progress balancing is presented by Van Craeynest et. al. [12] detailed below.

Equal-time Scheduling

In order to properly balance the threads on different core types, the equal-time scheduling method assigns threads to cores in such a manner that all threads get equal amount of execution time on each different core type. This is achieved by observing the amount of time (in terms of scheduling quanta or time slices) each thread has been running on the distinct types of cores. If there are imbalances between threads and the amount of time each has had on a core type, then the scheduler will subsequently remap the threads at the next scheduling quantum to ameliorate the imbalance.

A straightforward implementation of a equal time scheduling scheme is a round-robin or random thread to core mapping. These methods generally ensure that all threads will spend equal time running on all core types dependent upon the duration of the scheduling quantum and the total execution time of the application. These schemes may be modified to consider preserving data locality by not migrating threads among identical cores (i.e. cores of the same type).

A drawback with equal time approaches, however, is that fairness in a broader sense is not necessarily guaranteed. For instance, while all threads may receive equal time on all core types, not all threads may be making equal progress due to slowdown from different workload characteristics and memory requisites. Therefore, unfairness may still be present in an equal time scheme due to some threads making proportionally less progress than others. Conversely, if the threads are mainly homogeneous, namely exhibiting close to identical behavior, this imbalance ceases to exist and equal time scheduling will have the same effect as equal progress scheduling. Heterogeneous workloads, on the other hand, do not experience an identical one to one relationship of time running and progress made between different threads and hence must be managed more carefully than an equal time scheme to ensure fairness. Some homogeneous workloads may actually also end up exhibiting these same characteristics as

heterogeneous workloads especially if the different threads end up processing separate sections of an application's input data.

Van Craeynest et. al. take this possible imbalance into regard in their solution by proposing a scheduling scheme that uses equal progress in order to more closely achieve fairness.

Equal-progress Scheduling

To achieve fairness for all threads, equal progress scheduling's main focus is on enabling all threads to make equal progress or at least experience equal slowdown. This scheme hence involves observing the slowdown of each thread and producing a thread to core mapping such that the slowest threads are accelerated by scheduling them to execute on the fastest cores. As the thread experiencing the most slowdown (compared to the other threads) is accelerated, the other threads will end up running on the smaller and slower cores. Fairness is consequentially achieved as the slowdown of all threads converge.

Correctly calculating a thread's slowdown is a critical feature in the equal progress approach and is computed by comparing the total execution time of the thread on the heterogeneous CMP with the total execution time when running the thread only on a large core. The execution time of the thread on the heterogeneous CMP can be observed by accumulating the number of scheduling quanta or time slices TS_i the thread has been executing thus far on all core types. The total execution time of the thread on the single large core, however, is non-trivial to know in real time and must be estimated dynamically. For instance, Van Craeynest et. al. estimate the execution time of a thread running solely on the large core by firstly observing the number of quanta the thread was run on the large core compared with the smaller cores. Then, they rescale the time a thread has spent running on the small cores using an approximate scaling factor R of the large versus small cores, see Equation (2.1).

$$S_i = \frac{T_{het,i}}{T_{big,i}} = \frac{TS_{big,i} + TS_{small,i}}{TS_{big,i} + TS_{small,i}/R_i} \tag{2.1}$$

This scaling factor is estimated using a model-based scheduling technique which constantly monitors the CPI on both core types. This information is fed into an ana-

lytical model which produces the corresponding large versus small core scaling factor. An advantage of using this technique is that the scaling factor is continuously updated as the monitored CPI values change on the different core types. In contrast to other static methods of gathering CPI values such as history-based or sampling scheduling techniques, Van Craeynest et. al. use what they term the PIE model [11] to estimate the large versus small core scaling factor.

It is important to note that model-based approaches, such as PIE, require substantial hardware support for constant CPI monitoring and analysis. The effectiveness of PIE relies on the accuracy in monitoring and the analytical model.

2.2.6 Scheduling targeting Bottlenecks in Parallel Applications

Different mechanisms have been recently proposed which seek to overlap the execution of bottlenecks as long as they do not modify shared or synchronized data. One of these such approaches which targets critical sections is known as Transactional Memory (TM) [57]. The TM approach is dependent upon the lack of data conflicts between threads and bottlenecks. That is to say, that TM can only execute two transaction instances in parallel if they do not share data conflicts. One of the disadvantages of TM is the need for programmers to write code using transactions, but other proposals such as Speculative Lock Elision [58], Transactional Lock Removal [59], and Speculative Synchronization (SS) [60] can overlap the execution of non-conflicting instances of the same critical section without this implicit programming requirement. In fact, as long as no data conflicts exist between instances, SS can also reduce the overhead of barriers by speculatively allowing threads to execute past them.

Other research proposals focus on accelerating Amdahl's serial bottleneck. For instance, while Annavaram et al. [61] use frequency throttling to achieve benefits, Morad et al. utilize the large core of an ACMP for a single application [62] and for multiple applications [63]. Conversely, Suleman et al. [64] seek to accelerate accelerate critical sections on a large core of an ACMP. In their other work, Suleman et al. [65] allocate cores to stages in order to reduce stage imbalance on pipeline-parallel programs. Joao et al. [13] speedup multiple types of bottlenecks on large cores of an ACMP, using a criticality metric which depends upon the amount of thread waiting each bottleneck causes. These proposals accelerate only specific bottlenecks and

though they improve the performance of applications that are limited by those bottlenecks they do not take advantage of the large cores of an ACMP for applications which they do not identify a particular bottleneck. In addition, neither of these proposals are designed to accelerate bottlenecks from multiple applications, except for [63], which is also limited to just serial bottlenecks. In their work, Joao et al. [14] seek to solve this specificity concern by suggesting that it is more optimal focus on accelerating code segments which may be either bottlenecks or lagging threads. They do so by fully utilizing any number of large cores in order to improve performance of both single and multiple applications.

Alternative approaches including Meeting Points [66], Thread Criticality Predictors (TCP) [45] and Age-Based Scheduling [67] intend to reduce overheads caused by barriers via detecting and accelerating threads which last end up at the barrier. Age-Based Scheduling, on the other hand, makes use of the history from the previous instance of the loop in order to determine the optimal candidates for acceleration. Their scheme, however, is limited to iterative kernels showing similar behavioral characteristics across multiple executions. TCP uses a prediction approach by counting a combination of L1 and L2 cache misses in order to guess which threads are the most probable to arrive early at each barrier and then slows their execution in order to save power, though it may also be used to manage the acceleration of the lagging thread. Lastly, Meeting Points uses software hints to measure and identify lagging threads which are most likely to be bottlenecks.

3 Methodology

This chapter gives an overview of the benchmarks and the simulator we have used for our experiments. First, we describe SPEC CPU2006 and SPLASH2 benchmark suites and the instruction stream characteristics particular to them. For conducting our experiments we have made use of the Sniper simulator.

3.1 Benchmark Suites

In this thesis we use two comprehensive benchmark suites in order to evaluate our proposed simulation techniques: SPEC2006INT, SPEC2006FP and SPLASH2. These suites allow a broader scope of experiments since SPEC2006 is more intensive both in terms of computation and memory footprint, or workload size, and SPLASH2 allows for multi-threaded application simulations. SPEC benchmarks utilized in this thesis are single-threaded applications and therefore to run on a CMP system we run several application instances concurrently. Conversely, the SPLASH2 benchmarks utilized in this thesis are generally run as one instance of an application with various

threads executing concurrently on the cores of the system. This allows us to make use of parallelism both from the application level and the individual thread level in order to maximize system resource utilization on many core architectures. Both of the benchmarks are compiled with gcc 4.8 using -O3 optimization level and executed on top of an unmodified Ubuntu 14.04.3 LTS (GNU/Linux 3.13.0-61-generic x86_64) operating system.

3.1.1 SPEC Benchmark Suites

The SPEC [68] benchmark suite is popularly used within the research community in order to evaluate the performance and energy traits of various computer systems and techniques. The SPEC CPU2006 suite, in a manner similar to its former predecessors is divided into two parts, the integer component (CINT2006 benchmarks) and the floating point component (CFP2006 benchmarks). This classification divide in essence considers the number of floating points instructions that appear in a particular application's instruction stream. For instance, if a higher quantity than thirty percent of all dynamically executed instructions are classified as floating point operations, then the application is defined as floating point program and placed within the floating point benchmark (FP), otherwise it is classified under the integer benchmark (INT) suite. In terms of the SPEC2006INT integer group, the suite consists of twelve separate programs written in C as well as C++ while the floating point group, on the other hand, includes seventeen individual programs written in C, C++, and FORTRAN languages. For the experiments we have conducted and present in this thesis, we simulated twenty six applications from the SPEC benchmark suite. The individual applications and their specific input parameters are summarized in Table 3.1.

In this section we present an overview of the particular runtime characteristics of SPEC CPU2006 integer and floating point benchmarks in terms of their machine level instruction mix. We have obtained the runtime characteristics by measuring their performance on an Intel(R) Core(TM) i7-4600U CMP processor [1] system whilst running Ubuntu 14.04.3 LTS (GNU/Linux 3.13.0-61-generic x86_64). It is important to recollect that the benchmarks were compiled using the gcc/g++ and FORTRAN compiler V9.1. To account for the performance counter measurements, we have made habitual use of the dynamic instrumentation tool Pin[69].

Table 3.1: A list of the SPEC Benchmarks used in simulations, along with the most relevant execution command-line arguments.

Benchmarks	**Input parameters**
400.perlbench	-I. -I./lib checkspam.pl 2500 5 25 11 150 1 1 1 1
401.bzip2	chicken.jpg 30
403.gcc	166.i -o 166.s
410.bwaves	
416.gamess	< cytosine.2.config
429.mcf	inp.in
433.milc	< su3imp.in
434.zeusmp	
435.gromacs	-silent -deffnm gromacs -nice 0
436.cactusADM	benchADM.par
437.leslie3d	< leslie3d.in
444.namd	–input namd.input –iterations 38 –output namd.out
445.gobmk	–quiet –mode gtp 13x13.tst
450.soplex	-m3500 ref.mps
453.povray	SPEC-benchmark-ref.ini
454.calculix	-i hyperviscoplastic
456.hmmer	–fixed 0 –mean 500 –num 500000 –sd 350 –seed 0 retro.hmm
458.sjeng	ref.txt
459.GemsFDTD	
462.libquantum	
464.h264ref	-d foreman ref encoder main.cfg
465.tonto	
470.lbm	3000 reference.dat 0 0 100 100 130 ldc.of
471.omnetpp	omnetpp.ini
473.astar	rivers.cfg
483.xalancbmk	-v t5.xml xalanc.xsl

Table 3.2: Dynamic Instruction Count and Instruction Mix of SPEC CPU2006 Integer and Floating-Point Benchmarks.

Benchmarks	**Inst. Count (Billion)**	**Branches (Percent)**	**Loads (Percent)**	**Stores (Percent)**
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 illustrates in detail the dynamic instruction count and instruction mix for the programs of the benchmarks. 24 out of the 29 benchmarks have a dynamic instruction count in the order of a few trillion instructions. Comparing this total count with the CPU2000 benchmark suite which had a maximum of few hundred billion instructions per program, we can clearly understand the exacerbating effect which more contemporary applications have on execution time and speedup in general. On the other hand, a greater total amount of instructions can provide extra opportunities for extracting instruction level parallelism, offloading program kernels using multi-threading, and making use of other parallelization techniques. In addition, there are several interesting observations from the instruction mix of the programs. For example, the percentage of branches in the dynamic instruction stream of the programs classified as falling within the integer suite is close to the typical 20 percent. However, it is crucial to note that the percentage of branches within dynamic instruction streams of two programs, 456.hmmer and 464.h264ref consist of only 7 percent. Conversely, the 483.xalancbmk program which is one of the three applications written in C++ within the integer suite contains 25 percent branches within the instruction stream. In comparison, the two other C++ programs, 471.omnetapp and 473.astar, follow more typical patterns by having around 20 percent and 15 percent branch instructions per instruction stream respectively. Among the instruction streams of the floating-point programs, only 450.soplex and 453.povray show a high branch discrepancy by having approximately 15 percent branches whereas most of the other floating-point programs have less than 5 percent branch instructions. On the other side, the floating-point programs, 410.bwaves, 470.lbm, 436.cactusADM, each show less than 1 percent of branches in their corresponding instruction stream. These program show a large average dynamic basic block size suggesting a considerable amount of parallelism may be exploited by utilizing out-of-order and multi core architectures.

3.1.2 SPLASH2 Benchmark Suites

Compared to the SPEC2006 benchmark suite, the SPLASH-2 suite is relatively older being from 1995 and consists of a mixture of full applications as well as computational kernels. It features eight complete applications and four kernels, which showcase a variety of computational tasks based on the fields of scientific, engineering, and graphics

Table 3.3: SPLASH2 Benchmarks used in simulations, along with the problem sizes.

Benchmarks	**Input problem size**
barnes	65,536 particles
choleskey	tk29.O
fft	4,194,304 data points
fmm	65,536 particles
lu.count	1024 x 1024 matrix, 64 x 64 blocks
lu.ncout	1024 x 1024 matrix, 64 x 64 blocks
ocean.count	514 x 514 grid
ocean.ncount	514 x 514 grid
radiosity	large room
radix	8,388,608 integers
raytrace	car
water.nsq	4096 molecules
water.sp	4096 molecules

computing. Some of the original programs of the SPLASH suite have been improved and others have been removed due to either poor composition for evaluating medium-to-large scale parallel machines, or because they are not practically maintainable. The SPLASH benchmark applications and their input parameters are summarized in Table 3.3. Below we briefly describe the applications and kernels.

The Barnes application reproduces the fundamental interaction of a system of bodies akin to galaxies or particles in a three dimensional space over a number of timesteps, using the Barnes-Hut hierarchical N-body method. Cholesky is a blocked sparse factorization kernel which factors a sparse matrix to become the product of a lower triangular matrix and its transpose. The complex FFT program is a one-dimensional variant of the typical "Six-Step" FFT which is described in [70]. The FMM application also simulates a system of bodies for a given number of timesteps in a similar manner as Barnes. However, it simulates interactions in two dimensions using a different hierarchical N-body method called the adaptive Fast Multipole Method [71]. The

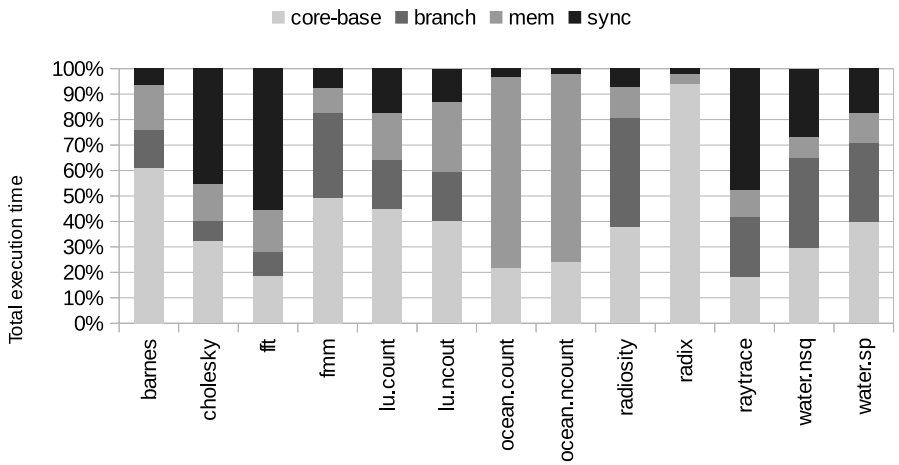

Figure 3.1: Breakdown of the execution time for the problem sizes in Table 3.3 on a four processor machine. The execution time is broken down into total time application spends in computation (core-base), in branches (branch), in memory accesses (mem) and in synchronization (sync). The synchronization represents the time spent in the barriers, locks and pauses (flag-based synchronizations) encountered across all processors.

LU kernel factors a dense matrix into the product of a lower triangular and an upper triangular matrix. The Ocean application studies large-scale ocean movements based on eddy and boundary currents. The Radiosity application computes the equilibrium distribution of light in a scene using the iterative hierarchical diffuse radiosity method [72]. The integer radix sort kernel is based on the method described in [73]. The Raytrace application renders a three-dimensional scene using ray tracing [74]. The Water-Nsquared application evaluates forces and potentials that occur over time in a system of water molecules. The Water-Spatial application solves the same problem as Water-Nsquared, but uses a more efficient algorithm.

Fig. 3.1 shows the execution time of the SPLASH2 benchmarks broken down into four parts, on computation, branch, memory access and synchronization, marked as “core-base”, “branch”, “memory” and “sync” respectively. The synchronization bottleneck is a byproduct of imbalances between the thread lock sections or barriers that incur sizable waiting latencies before being able to acquire a desired lock. The large average portions of execution time spent on synchronization as well as memory ac-

cesses in these programs. Also, the low percentage of branches in applications like cholesky, fft and ocean suggests relatively large average dynamic basic block sizes and high ILP. This high degree of the parallelism (ILP) may be much better exploited by out-of-order microarchitectures. These two factors suggest a high degree of parallelism that can be exploited by better scheduling on an asymmetric single-ISA Chip Multicore Processors consisting of out-of-order and in-order cores.

3.2 Sniper Infrastructure

Sniper is a high-speed, parallel and hardware validated x86 architectural simulator [75]. It is a versatile multi-core simulator based on the interval core model and also includes the Graphite simulation infrastructure, providing quick and accurate simulation. The simulator also allows for tradeoffs of simulation speed for accuracy in order to provide a greater width of simulation options for evaluating different homogeneous and heterogeneous many core systems.

Sniper is capable of performing timing simulations for both multi-programmed workloads as well as multi-threaded, shared-memory applications executing on many cores. The simulation itself occurs at relatively high speeds as compared with other existing simulators. One of the key features distinguishing the Sniper simulator is its use of interval simulation and mechanistic core models. By 'jumping' between miss events, called intervals, interval simulation raises the architectural simulation's level of abstraction which in turn provides for faster simulator development and evaluation times. Sniper provides performance prediction errors within 25 percent on average at a simulation speed of several MIPS and has furthermore been validated against multi-socket Intel Core2 and Nehalem systems.

The interval simulation [76] technique is a recent tool for conducting simulations including multi-core and multiprocessor systems. As previously mentioned, this approach enables a higher level of abstraction compared to current practices of detailed cycle-accurate simulations. Utilizing a mechanistic analytical model to abstract core performance, interval simulation drives the timing of each particular core without needing to keep detailed track of all individual instructions passing through a core's pipeline stages. The functional basis of the technique is keeping track of miss events such as branch mispredicts, cache and TLB misses, and serialization instruc-

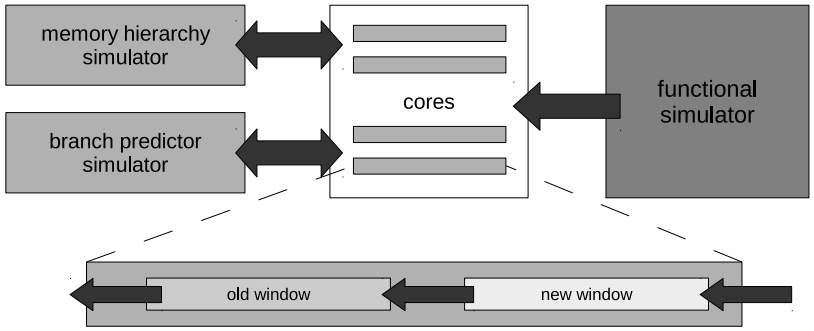

Figure 3.2: Sniper interval simulation model

tions among others to transpose the streaming of instructions through the pipeline as intervals. Separate simulators for the branch predictor, memory hierarchy, cache coherence and interconnection network help to determine the miss events and latency which feeds back into the analytical model which uses them to calculate the timing for each interval. Collaboration between the mechanistic analytical model and the miss event simulators enables modeling the often complex and dependent performance of threads concurrently executing on many core processors.

Fig. 3.2 shows how the interval simulator models the timing for the individual cores in a multi core system. For each simulated core, the simulator maintains a window of instructions . This instruction window corresponds to the size, or number of entries, of the reorder buffer of a superscalar out-of-order processor, used to find miss events that are overlapped by long-latency load misses. The functional simulator supplies instructions into this window through the tail of the window. The timing simulation, or in other words, progress at the core-level is determined based upon the instruction at the head of the window. For instance, in case of an I-cache miss, the simulated core time is augmented by the miss latency penalty. In case of a branch mispredict, the resolution time of the branch is added to the front-end pipeline depth (e.g., pipeline flush cost) with is summed to the core simulated time. In other words, a the simulator must model the penalty for executing the chain of dependent instructions which has led to the mispredicted branch plus the number of cycles needed to flush and refill the pipeline front-end . In the case of a last-level cache miss, cache coherence miss, or other long-latency load operation, the corresponding miss latency penalty is added to the simulated core time. Additionally, the simulator searches the

instruction window for independent miss events (e.g., cache and branch misses) that are overlapped by the long-latency load and other second-order effects. For serializing instructions, the window drain time is added to the simulated core time to ensure proper accounting for the latency penalty. If none of the above cases appears, instructions are dispatched at the effective dispatch rate, that takes into consideration inter-instruction dependencies and individual instruction execution latencies.

The Sniper simulator, and the interval core model as a whole is useful for system-level studies requiring extra detail compared to typical one-IPC simulation models, but where cycle-accurate simulators remain too slow to enable workloads of significant sizes to be simulated. An additional benefit peculiar to the interval core model is that it allows for the generation of CPI stacks showing the number of cycles lost due to different characteristics of the system. This is helpful for understanding how the cache hierarchy or branch predictor is affecting the total system performance and what system components should be improved to maximize system efficiency. CPI stacks enables researchers to use Sniper for application characterization and hardware/software co-design. The Sniper simulator is free to use for academic research and maybe be found online [77].

3.2.1 Sniper Extensions

In a similar manner to other cycle-accurate or interval simulators, Sniper pins hardware threads to the simulated cores. We have chosen to use Sniper simulator because of the two principle reasons. First, since all scheduling algorithms we propose in this thesis perform scheduling on hardware threads, we had to introduce extensions to the Sniper simulator in order to allow the movement of the hardware threads from one core to the other. Due to the nature of the interval simulation [76] technique and simulator organization [75] this was a relatively straightforward task. The second main reason for choosing Sniper was that it can be relatively easy to use for hardware/-software co-design extensions, where some of the scheduling algorithms we propose in this thesis require hardware-software communication. A detailed explanation of the hardware extensions for each of the proposed scheduling algorithms is provided throughout the following chapters.

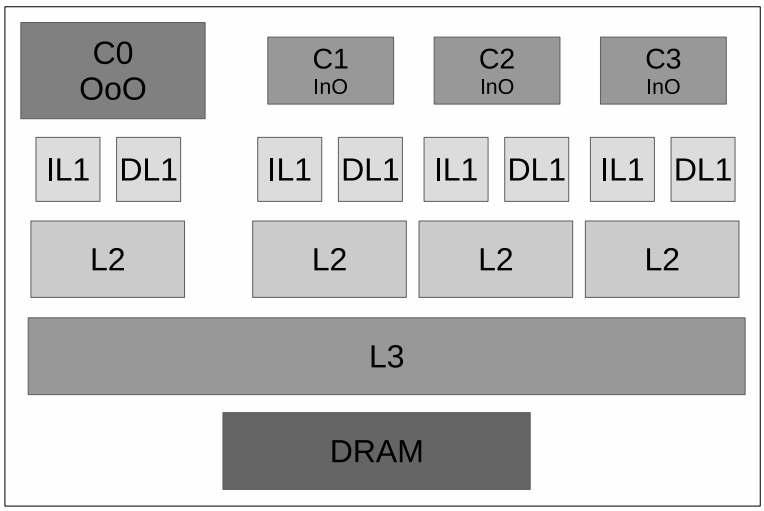

Figure 3.3: Simulated Architecture

3.2.2 Simulated Architecture - Sniper configuration

We use Sniper simulator [75] since it allows us to perform timing simulations for both multi-program workloads and multi-threaded, shared-memory applications with multiple cores having different capabilities, at a high speed when compared to existing simulators. This simulator is useful for system-level studies that require more detail than the typical one-IPC models, but for which cycle-accurate simulators are too slow to allow workloads of meaningful sizes to be simulated.

We configured the simulator to model an ACMP made up of one large core and three small cores as shown on the Fig. 3.3. The differences between the core types lie in the pipeline complexity (out-of-order for large, in-order for small). In order to isolate the causes of potential performance differences, the clock frequency (2.6GHz), issue width (4-wide), number of available thread contexts (one hardware context per core), and cache sizes are the same for both core types. We assume a cache hierarchy with separate and private 32 KB L1 instruction and data caches, private 256 KB L2 caches, and a shared 4 MB L3 last-level cache (LLC). All the caches employ a LRU replacement policy and we assume the memory controllers are on-chip.

Table 3.4 shows detailed system configurations. Similar to the work in [12], we utilize a conservative hardware-quantum of 1ms and a software-quantum of 4ms even though it is typically upwards of this range [6]. All the differences in the configuration

Table 3.4: Asymmetric Chip Mmultprocessor System configurations for Sniper simulator used in experiments

Small core	4-wide, 5-stage in-order, 2.66GHz
Large core	4-wide, 12-stage out-of-order, 128-entry ROB, 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 4MB, write-back, 30 cycle, 16-way
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	1048576 entries, 16-way DRAM, modeling all queues and delays, latency 45 ns, controller bandwidth 7.6 GB/s

of the simulator for conducting a particular experiments are further described in the relevant sections of the thesis.

4

Context Switch on the CMP

In this chapter we present the way we manage the context switch as well as the cost that context switch of threads on the cores may have on the total execution time of the application running on the shared last level cache memory of a Chip Multiprocessor.

4.1 Managing the Context Switches

The proposed scheduling techniques, described in the following chapters 5, 6 and 7, does not facilitate hardware overheads to be able to store and restore the architecture state in the cores. They utilize the x86 hardware context switching mechanism, called Hardware Task Switching in the CPU manuals [78].

The Hardware Task Switching, like Operating System, uses a special data structure called a Task State Segment (TSS) to store and restore the architectural state of the CPU in and out the memory during a context switch. Before invoking it, our hardware schedulers need first to pass two addresses to the CPU. The first one is the address where to save the existing CPU state while the second is the address to load from the

new CPU state. The Hardware Task Switching uses the long version of CALL and JMP instructions to invoke a context switch after passing to the CPU addresses where to store and load its old new architectural state respectively. The segments of the addresses indicate the position of the a TSS Descriptor in the Global Descriptor Table (GDT) while their offsets are ignored. The TSS descriptor specifies the base address as well as the limit of the TSS, which are used to store and load the old and new CPU state respectively, during the context switch. The CPU has a register called the Task Register (TR). This register tells which TSS will receive the old CPU state. The TR is consisting if two parts, one is visible and another invisible to the user. The visible part can be read and changed by instructions whereas the invisible portion can not be read by any instruction. The core maintains the invisible part of the TR. The core uses the invisible portion to cache the base and limit values from the TSS descriptor. When the Hardware Task Switching invokes an "LDTR" instruction on the core, which loads the TR register, the CPU checks the GDT entry and performs two actions in parallel. First, it loads the visible part of TR with the GDT entry. Second, it loads the private part of the base and limit of the GDT entry. The core uses the private part of TR when it saves the CPU state.

The Hardware Task Switching mechanism can be used to change all of the CPU's state except for the FPU/MMX and SSE state. There are a few options to store and restore the FPU/MMX as well as the SSE state during the context switch. One option is to save the data explicitly. The other option is that the CPU generates an exception the first time it uses an FPU/MMX or SSE instruction. With the second choice, 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.

When our scheduling techniques perform rescheduling after a given time quantum expired, explained in detail in following chapters, they need to take special care in two cases. To be precise, let as take an example of an ACMP system as described in section 3.2.2. An ACMP system consists of one large (Out-of-Order) core and three small identical (In-Order) cores. In the case that large core is in the kernel mode (such as handling an interrupt) our scheduling techniques will wait until it returns to user mode to mark rescheduling. While if the small core that is to have its thread swapped with the large core is in the kernel mode, the scheduler will choose the next small

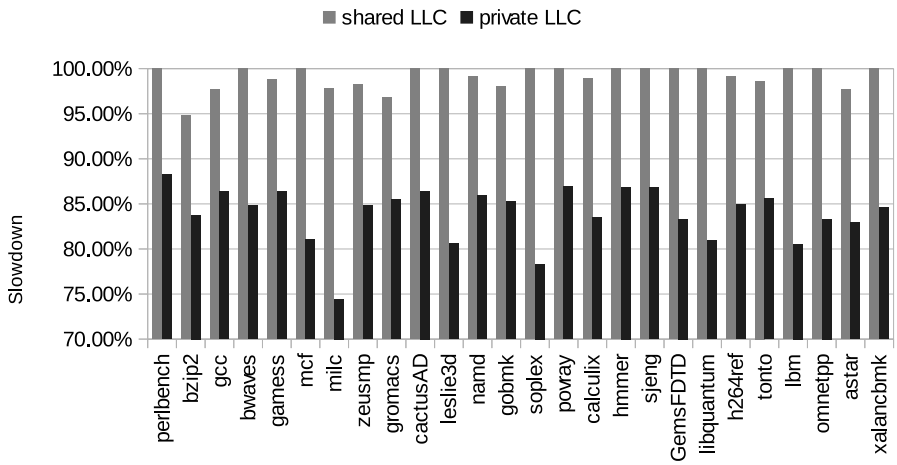

Figure 4.1: Context switch cost including flushing cores pipeline and register file as well as working set transfer (warming of the cache hierarchy) between two out-of-order cores on the CMP with the shared and private last level cache for the SPEC2006 benchmark

core to switch threads with large core in an adequate way, depending on the particular scheduling technique.

4.2 Context Switch Cost

Implementing dynamic scheduling requires the migration of workloads between different cores. This leads to overheads incurred by context switches. A context switch overhead consists of three major components. First, a context switch incurs a cost for storing and restoring the architecture state - register file (at most a few kilobytes). Second, processor's pipeline flushing, which takes the least time compared to all other costs incurred by the context switch. Third is the migration of the working sets that is the loading of the working-sets into the private caches of the destination cores. The last one is the most important one since it is orders of magnitude larger than the other two.

A previous study [11] has shown the total migration execution time overhead to be less than 1.5% across different types of workloads, ranging form memory-intensive

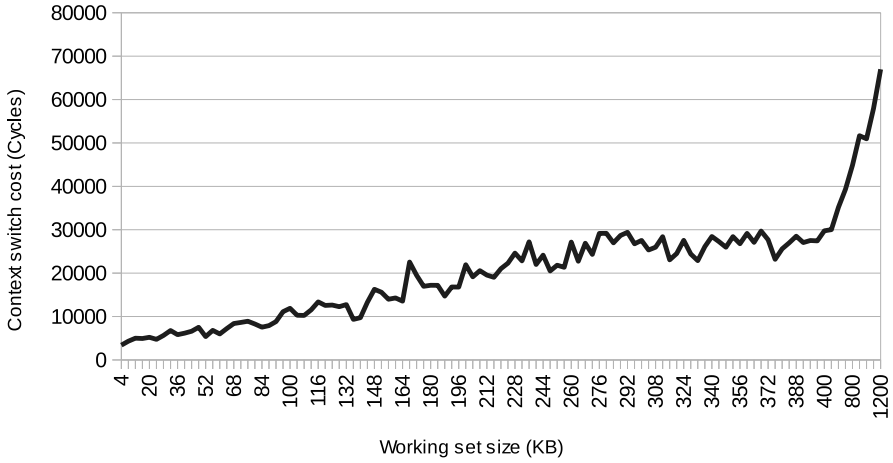

Figure 4.2: Context switch cost including flushing cores pipeline and register file as well as working set transfer (warming of the cache hierarchy) between cores on Intel(R) Core(TM) i7-4600U CMP[1]

to compute-intensive, for a 4 MB shared LLC using a 1ms hardware-quantum. We have estimated the overhead caused by context switches occurring at every 1ms for the SPEC2006 benchmark. To do so, we have configured the Sniper simulator to simulate a system similar to Intel i7-4600U processor [1], simulationg two four issue wide out-of-order cores. We configured a cache hierarchy with separate and private 32 KB L1 instruction and data caches, private 256 KB L2 caches, and with a shared or a private 4 MB L3 last-level cache (LLC). The results on the Fig. 4.1 represent the slowdown caused by switching contexts from one to the other core at every 1ms. Fig. 4.2 represents context switch costs with different working set sizes ranging from four up to one thousand two hundred KB in size on Intel i7-4600U core[1].

This relatively low overall overhead of the context switch comes as the effect of the so called Smart Cache architecture that allows all cores to dynamically share access to the last level cache. Smart Cache is a level 2 or level 3 caching method for multiple-execution cores, introduced by Intel [79]. Smart Cache shares the actual cache memory among all cores in the system. In comparison to a dedicated per-core cache, the overall cache miss rate decreases in times where not all cores need equally much of the cache space. Consequently, a single core can use the full level 2 cache

or level 3 cache, if the other cores are inactive [80]. Furthermore, the shared cache makes it faster to share memory among different execution cores [81].

In our simulations, we presume a fixed 1,000 cycle penalty [7],[12] for storing and restoring architectural state of the core i.e. registrar file. We account for the warming up of the cache hierarchy needed after a context switch as well as for processors pipeline flushing in our simulations.

5

HRRS: Hardware Round-Robin Scheduler for Hardware Threads

In section 1.2 we have noted the importance and the impact scheduling for fairness may have on the potential speedup that can be achieved from the parallelization of multi-threaded applications and multiprocess workloads on single-ISA asymmetric multi-cores.

In an asymmetric multi-core system, a scheduler using threads pinned to cores produces no speedup compared to a lighter symmetric multi-core system for most multithreaded benchmarks [12]. This behavior may be caused by barrier-synchronized multithreaded workloads since the execution progress is limited by the slowest thread which has little meaning in a symmetric system, but is significant for asymmetric systems since the thread pinned to the simplest core will be the weakest link that all other threads will have to wait for at every barrier. Work-stealing workloads, in contrast, allows for idle large cores to steal work that would normally be run on the small cores so that the execution time is not as constrained. Also, when running workload con-

sisted of multiple applications on an asymmetric system the overall performance may suffer if scheduler does not guarantee fairness [11]

Therefore, in asymmetric multi-core systems, guaranteeing fairness is fundamental for improving performance of multithreaded and multiapplication workloads. Fairness, as defined by giving each software thread equal execution time on each core or allowing each thread to make equal progress, enables all threads to reach the barriers simultaneously, and has been sown to provide average performance improvements of 14% (and up to 25%) compared with a pinned scheduler [12] for the system configuration we are using.

We propose a simple and efficient scheduling mechanism that remaps hardware threads on physical cores at every hardware scheduling quantum, while offering an easy hardware implementation in contrast to the previously proposed technique [12]. In the next two subsections we describe the proposed Hardware Round-Robin Scheduling (HRRS) policy and discuss its hardware implementation.

5.1 HRRS Algorithm

Fig. 5.1 is used to illustrate the inner workings of the HRRS approach, we will assume a system composed of 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, which correlate to four identical hardware threads. 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 HRRS in turn maps the threads running on the logical cores to the physical cores as shown in Fig. 5.2 at every hardware-quantum. In essence, the HRRS 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 HRRS algorithm must produce a new scheduling scheme after every hardware-quantum of time passes (as opposed to the software-quantum which invokes the OS scheduler). In order to minimize the amount of overhead in implementing the scheduling policy, the HRRS algorithm determines the next scheduling scheme to apply before the beginning of the

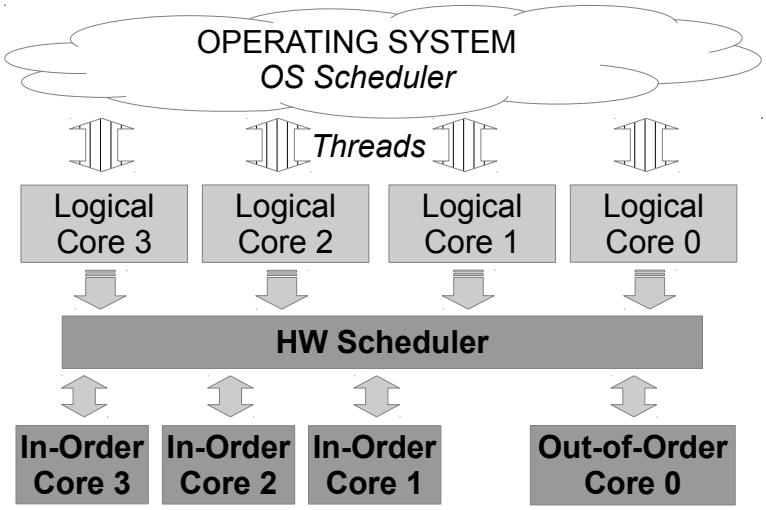

Figure 5.1: HRRS scheduling - All logical cores (which correlate to hardware threads) are the same while the large physical core is represented by Core 0 and the small physical cores are shown as Cores 1,2 and 3

ALGORITHM 5.1: HRRS algoritham pseudo code

buffer: holds entries small core IDs;
ID_{large}: holds large core ID;
while *at every scheduling quantum or when large core is in the idle state* **do**
    tmpID_{small} = buffer.first;
    switch contexts of tmpID_{small} and ID_{large};
    buffer.last(ID_{large});
    ID_{large} = tmpID_{small};
end

next hardware-quantum.

The defining characteristic of the HRRS algorithm is that it evenly rotates threads (scheduled onto the logical cores by the OS scheduler) running on the physical cores after every hardware-quantum. The OS scheduler, on the other hand, is triggered at every software-quantum which happens much less frequently than that of the hardware-quantum. Additionally, the HRRS algorithm does not need to take into account whether the OS scheduler has activated and swapped one of the currently executing threads on a logical core for another thread from its ready queue. In such cases, the thread 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 all of which is performed by the triggered OS scheduler routine. Consequentially, the HRRS scheduling policy guarantees that 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. Algorithm 5.1. presents the pseudo code of the HRRS.

The Difference compared to the Fairness-aware Scheduler

There are two fundamental differences between the HRRS and Fairness-aware scheduler [12] algorithms. First, the difference between the HRRS and Fairness-aware Scheduler algorithms is that HRRS maps logical cores (e.g. hardware threads) onto the physical cores while Fairness-aware Scheduler schedules software threads onto the physical cores on every hardware scheduling quantum. Second, in both approaches, a thread running on one of the smaller physical cores is swapped with the thread running on the large physical core after a given time quantum. However, while Fairness-aware scheduling strives at achieving fairness by guaranteeing even progress using specific heuristic for each software thread, it does not necessarily enforce swaps of threads between large and small core every scheduling quantum but prefers to leave threads to run on the same physical core. In contrast, the HRRS policy runs each logical core, hardware thread, on each physical core type for a specified amount of time. After every quantum, the HRRS triggers a swap between the thread running on the large core with one executing on a small core that is chosen using a round-robin selection algorithm.

5.1.1 Hardware Implementation

Hardware Round-Robin Scheduling leaves the operating system level scheduling 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 [82]. Both of these approaches (HRRS and Fairness approaches [12]) may also be implemented at the OS level by extending the OS scheduler but the advantage of a hardware approach, in addition to minimizing scheduling overheads, is that it provides a finer level of granularity for the scheduling quanta and requires no changes to the OS code

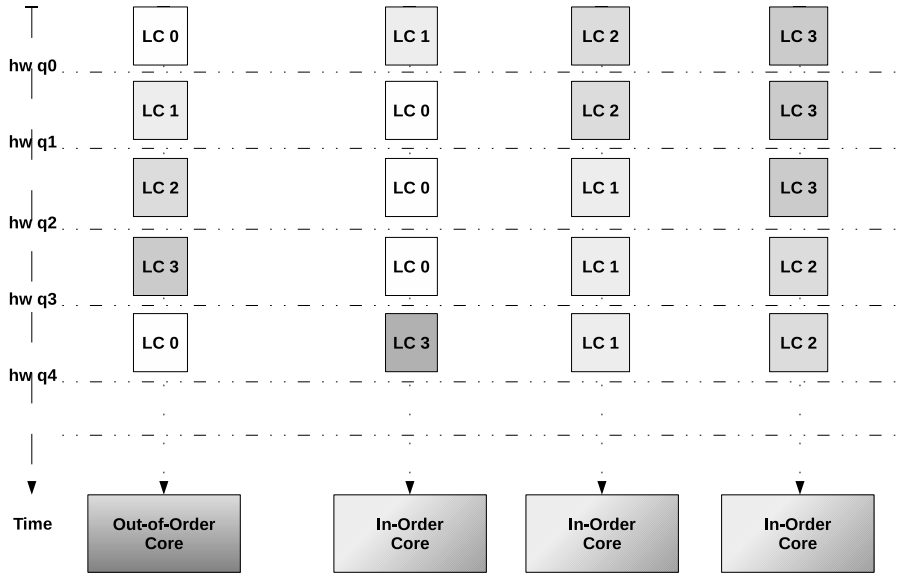

Figure 5.2: An example of the HRRS scheduling logical cores on actual physical cores at every hardware-scheduling quantum. At the beginning logical core 0 is running on the large physical core while logical cores 2, 3 and 4 are running on small physical cores. After a first hardware scheduling quantum, Logical core 1 will be moved to large physical core and logical core 0 will be moved to a small physical core.

[11]. While OS scheduler typically performs rescheduling of the software threads after every 4 ms on average [6], hardware implementation allows HRRS to perform rescheduling of the hardware threads after every 1 ms.

HRRS has hardware additions which include two counters, one register and two decoders to facilitate round-robin mechanism, as shown on the Fig. 5.3. An “active” bit per hardware thread indicates weather the thread is running user code or not (i.e. running kernel code or being idle). Register holds the ID of the logical core that is currently executing on the large physical core. One counter is incremented after every hardware scheduling quantum to point to the next logical core running on small physical core to be swapped with logical core running on the large physical core in the next scheduling quantum. The two decoders help determine weather logical cores that are supposed to be swapped are currently executing user code or not. The other counter keeps track of the hardware scheduling quantum. The size of these depends on

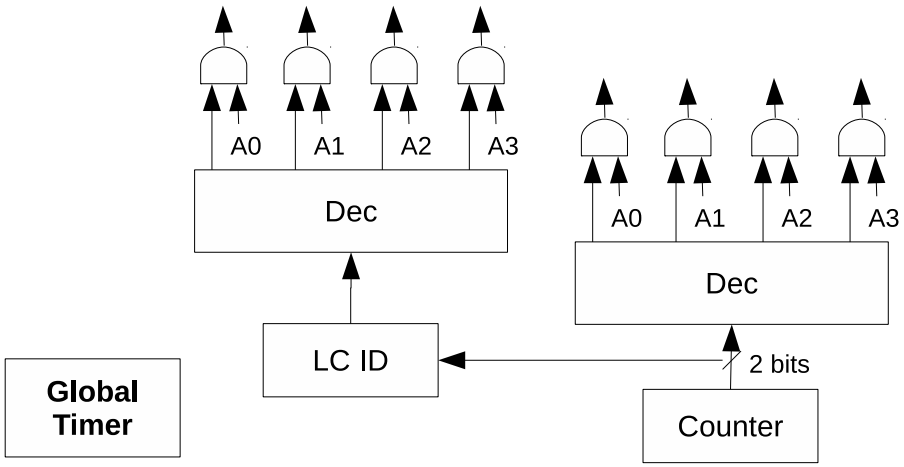

Figure 5.3: Hardware implementation layout example for HRRS scheduling policy on the four core ACMP consisting of one large and three small cores (1 OoO + 3 InO), where 'A' is the active bit per hardware thread.

the number of the cores in the system e.g. for four core system we need 2-bit counter.

The complexity of the hardware implementation of the Fairness-aware scheduler [12] compared to the HRRS is significant. The Fairness-aware scheduler relies on the PIE [11] model-based scheduling technique that constantly monitors the CPI on cores. Therefore, it requires additional counters, registers as well as logic circuits to calculate features like last level cache misses of each core.

As oppose to other dynamic schedules [13], [14], the HRRS scheduling technique does not need special hardware overheads to store and restore the architecture state in the cores. It uses he x86 hardware context switching mechanism, called Hardware Task Switching in the CPU manuals [78]. In order to use it, the HRRS needs to instruct the CPU where to save the existing CPU state, and where to load the new one. This is fully explained in section 4.1.

5.2 Evaluation

Here we evaluate the Hardware Round-Robin Scheduling (HRRS) approach and compare it to the Linux OS scheduler and Fairness approach [12]. First, we describe the

bechmarks and the simulator configuration we have used for this study. Second, we present potential ideal speedup gains of the HRRS scheduling policy. Finally, we evaluate the performance and energy efficiency of the HRRS scheduler over the Fairness and Linux OS scheduler on an ACMP when running multi-threaded applications (running dependent threads) and multiple instances of the single-threaded applications (running independent threads) on an ACMP system.

5.2.1 Simulated Architecture and Workloads

For conducting the simulation experiments we have used Sniper [75], a parallel, hardware - validated, x86-64 multi-core simulator capable of running both multi-program and multi-threaded applications, described in Section 3.2. Section 3.2.2 shows detailed system configurations used for the experiments. Similar to the work in [12], we utilize a conservative hardware-quantum of 1ms and a software-quantum of 4ms even though it is typically upwards of this range.

We use the SPLASH-2 [83] and SPEC2006 [68] benchmarks in our experiments. The SPLASH-2 benchmarks are designed to represent multi-threaded applications to evaluate hardware architectures when running several thread contexts. The SPEC2006 benchmark is an industry-standardized, CPU-intensive benchmark suite, stressing a system's processor, memory subsystem, and compiler. We have utilized the SPEC2006 benchmarks to run multiple instances of traces from different phases of the single threaded applications concurrently on the system. We run each benchmark on the four simulated cores with each core capable of executing one hardware thread context at a time. All simulated workloads are run from start to finish. Each thread or process of the workload has a fixed amount of work to perform. We declare that the evaluation finished once all threads/processes finished their preassigned work. In the case of having several threads running at the same time, the threads that finish first are restarted until the last one of initially started threads finishes its preassigned work, such that the number of threads running at any given time through evaluation on the system remains constant. Once the longest one of original threads/processes has been completed, the simulation is ended.

We evaluate single multithreaded application workloads running an equal or greater number of threads per application as the number of available hardware contexts, i.e.

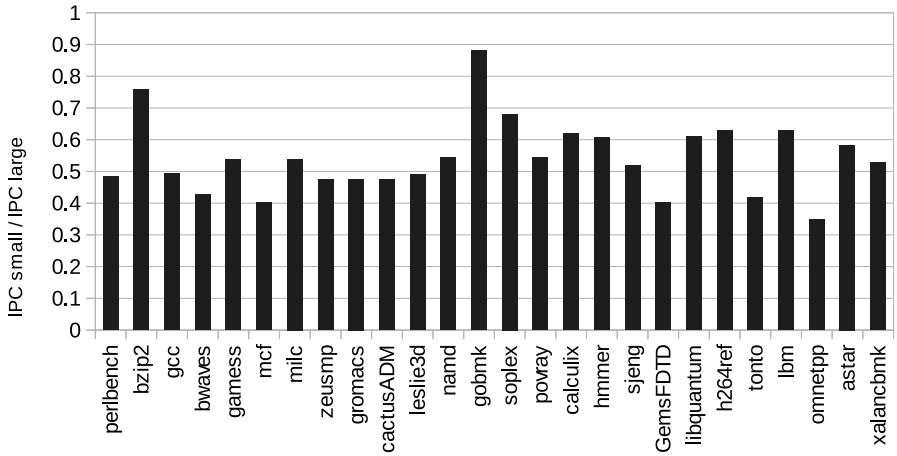

Figure 5.4: Small core slowdown (expressed as IPC small / IPC large) compared to the large cores when running SPEC2006 benchmark

maximum number of cores. When we run multiple instances of the single-threaded application workloads, we run at least as many different trace instances of the same application, as there are available hardware contexts, i.e. the maximum number of cores. The indicated method is common when executing non-I/O-intensive applications [14]. For example, when we run one multi-threaded application on a system simulated with four cores, we use the minimum of four threads for that application. The results shown in the following sections are averaged for running a different number of threads, e.g. 4, 8 and 16 threads, per workload on the system.

5.2.2 Ideal Performance gains and Scalability

In an ACMP system cores do not provide a uniform computing capability for each thread/process/application. While one application might exibit a very good performance gains when executing in a more powerful core, the other might not because it can not take advantage of the additional hardware resources available to it.

To illustrate this point, Fig. 5.4 shows the performance of the SPEC2006 [6] applications on a small core normalized to their performance on a large core. The large and small cores are representative of state of the art out-of-order and in-order

cores, respectively. While the absolute values are not relevant, their distribution shows a significant variation among applications. Most of these applications can harness around a 2x speedup on the large core, but there are those too, that are only able to gain a modest 10 percent.

In order to calculate the ideal performance gains we have introduced two major presumptions. First, the ideal context switch. That is, the flushing of the cores pipeline and transfer of the register file as well as the working set migration (the major component) are ideal, i.e. with zero overhead. Second, that there are four identical independent threads running in the four core ACMP system (1 OoO + 3 InO).

Considering that the current practice in the Linux OS scheduler is that software threads are pinned to the cores, e.g. when thread is assigned to a core it is rarely reassigned to continue its execution on any other core in the system. It rather shares resources of that core with other software threads assigned to the core. This is a simple and reasonable way to manage load balancing since nowadays CMPs consist of the cores of equal capabilities. However, this kind of scheduling would cause the performance to be bound by the capabilities of the smallest core in an ACMP system.

On the other hand, Fair scheduler strives to provide fairness by guaranteeing equal progress of all software thread, either in terms of time or executed instructions. Therefore, it does not necessarily promote a thread from the small core to the large core if that thread makes sufficient progress, compared to the others threads. The detail overview and the model of the Fair scheduler is presented in Section 2.2.5.

Equation (5.1) represent the ideal analytical model of the execution time T of all workloads on all cores for the HRRS scheduler. N represents the total number of cores and hardware threads, Q is the number of necessary scheduling quanta, $Workload(time)$ represents the workload per thread (equal per thread), $L = IPC_{large}$ and $S = IPC_{small}$.

$$T = \frac{Workload(time)}{L + (N-1) * S} * N * Q \tag{5.1}$$

Fig. 5.5 represents the ideal analytical comparison of the HRSS and Fairness schedulers on an ACMP consisted of one large and three small cores for different performance ratios between large and small cores. The maximum ideal speedup of

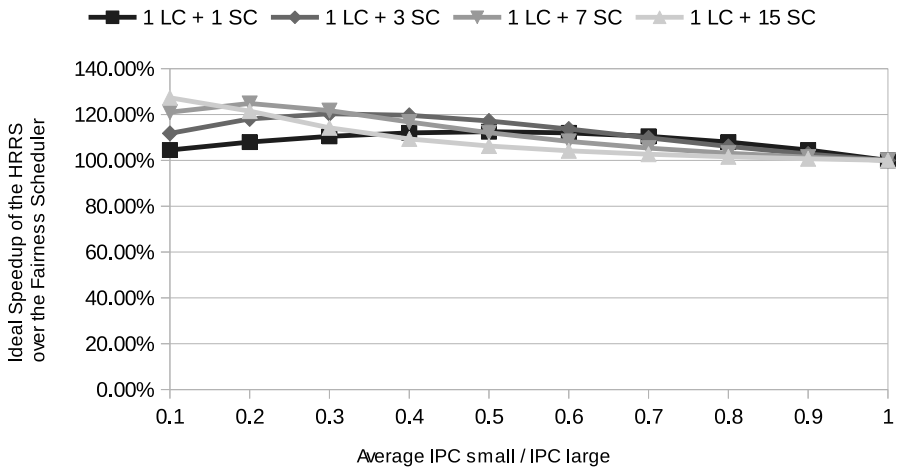

Figure 5.5: Ideal Speedup comparison of the HRRS over Fairness scheduler on an ACMP systems consisted of 2 (1 OoO + 1 InO), 4 (1 OoO + 3 InO), 8 (1 OoO + 7 InO) and 16 (1 OoO + 15 InO) cores respectively.

the HRRS over Fairness scheduler is around 20 percent in case that thread exhibits around 2x speedup when executing on the large compared to small core. When compared with an ACMP system with the regular software scheduler, in the case that thread has over 3x performance benefit when running on the large core, the HRRS has significant performance gains, over 1.5x and up to 3x. It is important to note that neither one of the SPEC2006 applications have not exhibited 3x performance gains when running on large core compared to a small core, presented on the Fig. 5.4.

Although we have used an ACMP composed of one large and three small cores to conduct our test, our scheduling technique can be easily applied on an ACMP with the different ration of large and small cores, see Fig. 5.5. The following chapter 6 offers more insight on scalability of our schedulers in many-core systems.

5.2.3 Performance Evaluation

We discuss the performance of the HRRS scheduling policy by examining it two scenarios and comparing it to the Fairness scheduler. First, when running independent threads in an ACMP system. Second, when running dependent threads in an ACMP system. In both cases the results are scaled to a Linux OS scheduler where the operat-

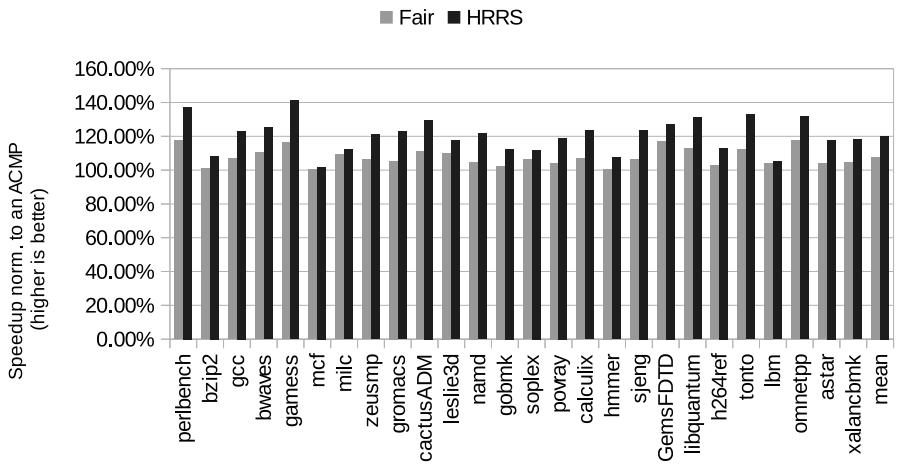

Figure 5.6: Speedup comparison of the HRRS and Fairness scheduler normalized to Linux OS scheduler for the SPEC2006 benchmark suite running multiple instances of traces collected from different phases of an application on four cores ACMP (1 OoO + 3 InO)

ing system has a notion of the underlying hardware. Under the Linux OS scheme, in the case of a symmetric CMP, the operating system pins individual threads to each of the cores in a round-robin fashion until 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 (when there are more than one thread assigned per core). When using Linux OS scheduler threads on an asymmetric CMP, the operating system does not necessarily pin the threads to the cores, nor does it tend to swap the threads running on the large core with those on the small core at every scheduling quantum until threads pinned to the large core are all stalled or finished their execution. Rather, the OS scheduler tries to ensure quality-of-service for the threads by ensuring that all threads pinned to the core share cores resources equally. This reflects the current practice in contemporary operating system schedulers, as exemplified in the Linux 2.6 kernel [6].

Fig. 5.6 shows the results when running independent threads on an ACMP system. The HRRS policy has an average speedup of the 11.71 percent and 20.25 percent over Fairness and the Linux OS scheduler respectively. When running independent threads in the system we run multiple instances of the single-threaded applications

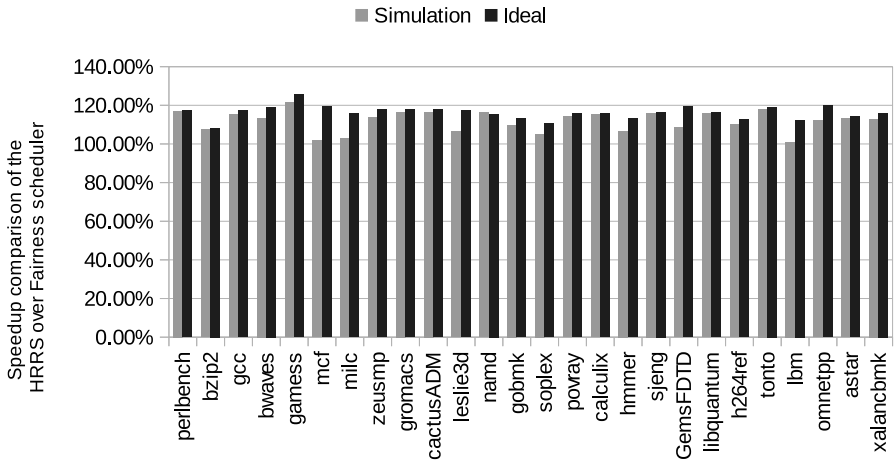

Figure 5.7: Speedup comparison of the ideal and simulation results of the HRRS over Fairness scheduler for the SPEC2006 benchmark suite running four instances of traces collected from different phases of an application on four cores ACMP (1 OoO + 3 InO)

(SPEC2006) on a four core ACMP system. This allows us to compare our simulation results with the ideal performance gains. By looking at the results presented on Figures 5.4, 5.5 and 5.6 we can the compare the ideal and simulation achieved speedup of the HRRS over Fairness scheduler on a four core ACMP consisted of the one large and three small cores. Fig. 5.6 shows the comparison of the ideal and simulation achieved speedup. The major difference between ideal and simulation results comes as result of the context switch cost, since ideal model presumes ideal context switch cost for each application while simulation accounts for it. This is especially evident with the *mfc*, *milc* and *lbm* application, which are memory rather than execution bounded.

Fig. 5.8 shows the speedup of the HRRS scheme over the hardware implementation of the Fairness scheduler on four core system running multi-threaded application workloads. The results are scaled to a Linux OS scheduler where the operating system has a notion of the underlying hardware. The average speedups of HRRS over the Fairnes and Linux OS scheduler when running the Splash2 workloads are 16.5 percent and 37.7 percent respectively on a four core system.

A key element driving these performance benefits comes from the redistribution of the workloads amongst the cores. Fig. 5.9 shows per benchmark LLC access

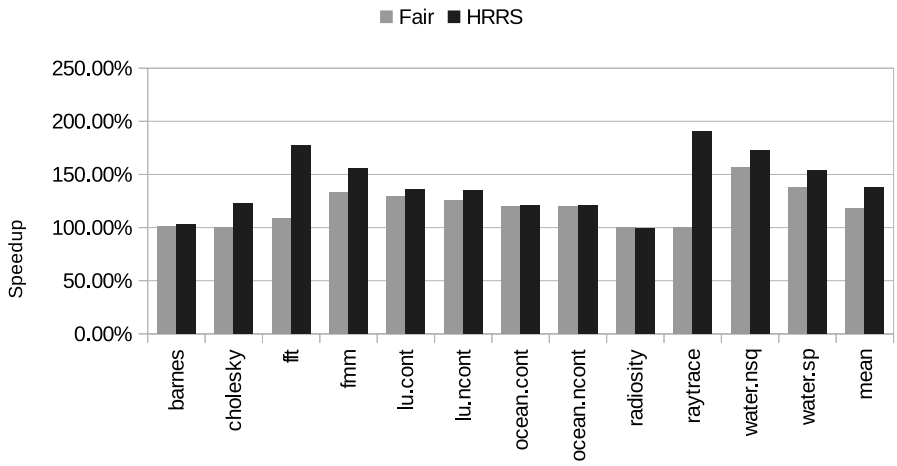

Figure 5.8: Speedup comparison of the HRRS and Fairness scheduler normalized to Linux OS scheduler for the SPLASH-2 benchmark suite running on four cores (1 OoO + 3 InO)

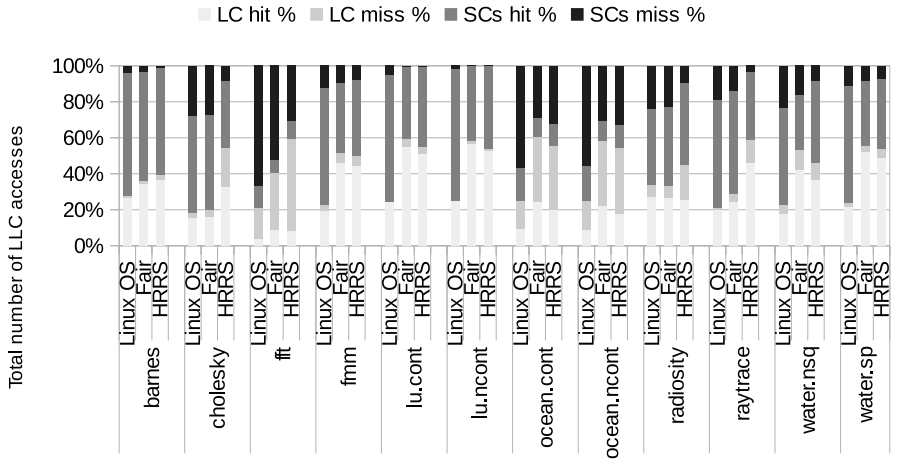

Figure 5.9: The LLC cache accesses breakdown for the large and small cores of the Linux OS, Fairness and HRRS scheduler for the SPLASH-2 benchmark running on four cores ACMP (1 OoO + 3 InO)

distribution between large core and small cores, while the total number of the LLC accesses grows up by only up to 1.5 percent for Fair and HRRS schedulers compared to Linux OS scheduler. The HRRS scheme produces a higher proportion of LLC accesses originating from the large core. Fundamentally, the large core can better support the extra burden of LLC cache accesses since the large out-of-order instruction window allows for a greater quantity of instructions to be processed concurrently, which enables it to hide the additional latency caused by the extra cache accesses and still, even after including overheads from the context swap, outperform the small cores in thread execution time. This hit/miss ratio and considerable change in the total number of LLC accesses between large and small cores are clearly noticeable with *fft*, *cholesky* and *raytrace* benchmarks. Therefore these benchmarks have the highest performance gains. This can be noticed on the Fig. 5.8.

Hardware vs. Software Implementation

Hardware Round-Robin Scheduling provides a consistent view of the underlying hardware to the operating system, therefore introducing no changes to an existing operating system. The default Linux scheduling quantum is defined in the Linux kernel as RR_TIMESLICE (include/linux/sched/rt.h), and its default value is set to 100 ms. And a clock tick is typically 1 ms to 6 ms for the Linux kernel 2.6.8 and onwards (defined in kernel/sched/fair.c by sysctl_sched_min_granularity and sysctl_sched_latency). It can have a minimum quantum of 4 ms (a case rarely seen and still 4x the latency of the hardware quantum) [12]. The hardware scheduling quantum we have used in our experiments is 1 ms as is mentioned in the first paragraph of Sec. 5.2.1. Fig. 5.10 represents the speedup that a hardware implementation, with a scheduling quantum of 1ms, has over a software implementation (baseline), with a scheduling quantum of the 4ms, for the Fairness-aware and the HRRS scheduler. We can see that a hardware implementation results in an average speedup of 6.98 percent over a software implementation for the HRRS scheduler.

5.2.4 Energy Efficiency Analysis

We have used the commonly applied power-delay product (PDP) in order to evaluate energy efficiency of the Fair and HRRS scheduling policies applied on an ACMP

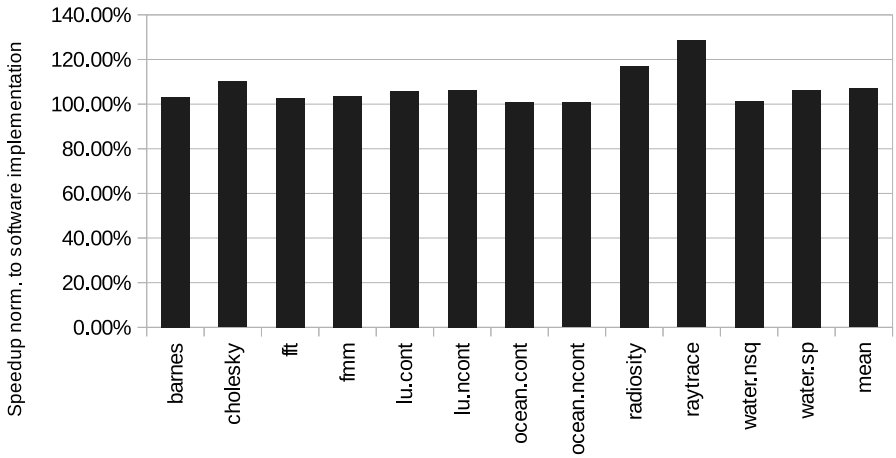

Figure 5.10: Speedup of the hardware over the software implementation (baseline) for the HRRS scheduler, where scheduling quanta are 1ms and 4ms for hardware and software implementations respectively

system. Since average energy consumption of the Fair and HRRS scheduling units are negligible compared to the whole CMP system, several million times less, we did not consider them separately in this study.

Fig. 5.11 and Fig. 5.12 represent normalized energy efficiency (less is better) of the Fairness scheduler and HRRS scheduler over Linux OS scheduler on an ACMP when running dependent and independent threads in the system respectively. When running independent threads in an ACMP system HRRS is on average 10.73 percent and 6.56 percent more energy efficient than the Linux OS and Fairness schedulers respectively. On the other hand, when running dependent threads in an ACMP system, HRRS policy is 7.57 percent more energy efficient than the Fairness scheduling policy, while consuming 5.71 percent more energy on average than the Linux OS scheduler.

5.3 Summary

We have presented the Hardware Round-Robin Scheduler (HRRS). This work is influenced by the rise of many core processors, particularly the asymmetric core multiprocessors (ACMPs) and their dependence on dynamic schedulers such as the com-

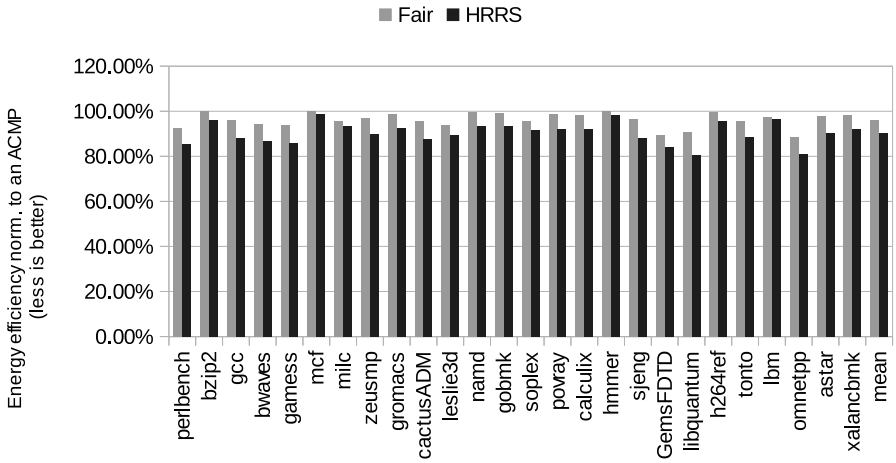

Figure 5.11: Energy efficiency comparison of the HRRS and Fairness scheduler normalized to Linux OS scheduler for the SPEC2006 benchmark suite running four instances of an application on four cores ACMP (1 OoO + 3 InO)

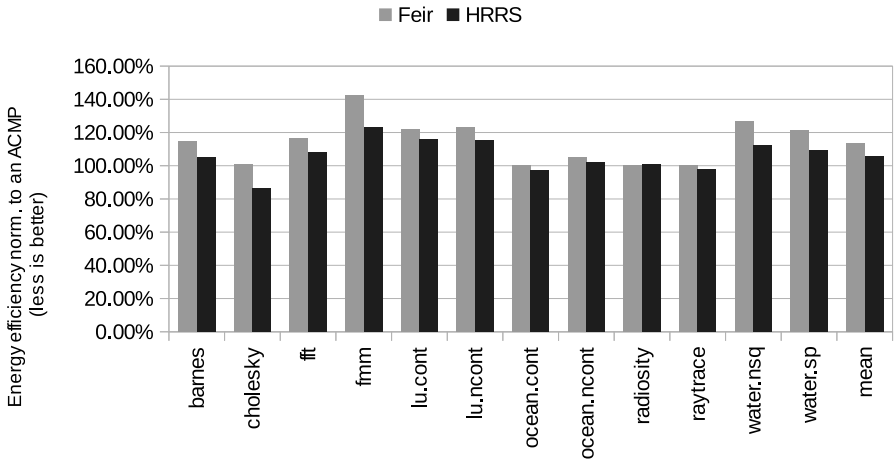

Figure 5.12: Energy efficiency comparison of the HRRS and Fairness scheduler normalized to Linux OS scheduler for the SPLASH-2 benchmark suite running on four cores ACMP (1 OoO + 3 InO)

modity Linux OS CPU scheduler in order to achieve fair and balanced performance between active threads. Our initial objective was to achieve these performance benefits from running parallel workloads on ACMPs without the need for substantial hardware extensions, sampling, or runtime overheads. Incorporating minimal hardware additions, our HRRS policy promotes a balanced distribution of execution time for threads per core type. We have shown that HRRS provides greater opportunity for all threads to share time running on the more efficient large core, selected via a round-robin algorithm, which produces generous performance benefits even after including scheduler and context swap overheads as well as latencies arising from the additional cache accesses needed for loading the working data sets. By using the HRRS policy on an ACMP, we got a total execution time speedup of 37.7 percent and 16.5 percent respectively compared to the state-of-the-art Linux OS scheduler and Fairness scheduler when running a multi-threaded application workloads (Splash2).

6

KUTHS: Kernel to User mode Transition aware Scheduler for Hardware Threads

In section 1.2 we have commented the effects, and the consequences that bottlenecks can have on the potential speedup reached through the parallelization of multi-threaded applications on single-ISA asymmetric multi-cores.

The performance gain of parallel execution is in direct relation to the amount of time spent in synchronization of the workload. The higher the concurrency throughout the execution of the parallel workload the higher the performance. If the threads need to wait for each other, the less work gets done concurrently, which reduces the possible speedup gain. Programmers are trying hard to reduce bottlenecks that lead to thread waiting, to achieve higher concurrency. Much effort is put into attempting to reduce issues such as false sharing, thread synchronization, thread load imbalance. Some of the earlier research on computer architecture focuses on this problem as well, e.g., transactional memory [57], primitives like test-and-test-and-set [84], lock elision [58], speculative synchronization [60], etc. The main idea is to speed up the bottlenecks that

cause thread waiting.

Our proposal, Kernel to User mode Transition aware Scheduler for Hardware Threads (KUTHS), builds upon the previously proposed HRRS technique, described in section 5. KUTHS extension consists of two parts. First, KUTHS utilizes kernel to user code execution transitions to identify possible critical sections and make a rescheduling decision. Second, KUTHS accelerates these bottlenecks by localizing and executing them on the large core. In the next two sections, we describe the proposed KUTHS policy and discuss its hardware versus software implementation.

6.1 KUTHS Algorithm

The KUTHS approach builds upon the HRRS approach, described in section 5.1. Therefore, we use the same ACMP system to present KUTHS technique. An ACMP is composed of one large out-of-order core and three smaller and identical in-order cores. The KUTHS approach presents an abstracted homogeneous hardware view comprised of four identical logical cores to the operating system. The OS scheduler maps software threads to the logical cores which enables the OS scheduling policies and implementation to be left unmodified. In turn, the KUTHS remaps the software threads running on logical cores to physical cores after each hardware-quantum of time passes (as opposed to the software-quantum which invokes the OS scheduler). In essence, KUTHS can be viewed as mapping the logical cores that the OS sees and schedules software threads onto, to the physical cores of the underlying hardware which actually execute the threads.

The defining characteristic of the KUTHS 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. It is important to note that these transitions (the ones which activate the OS scheduler) are due to a thread reaching a synchronization point in its code and having to wait on a lock (operating system futex) or thread waiting on the Operating system routine (e.g. I/O) to finish. 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 or complex profiling.

In order to determine these execution level transitions, the KUTHS 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 KUTHS checks to see which of the "transition-flags" belonging to the four cores are set. If none are set, the KUTHS proceeds to schedule based on a round-robin selection scheme akin to the Hardware Round-Robin Scheduler (i.e. using round-robin selection, it chooses one of the three logic cores running on the small cores to swap with the logical core running on the large core). If only the large core has its "transition-flag" set, then no swap is made. If only one of the small cores 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 via round-robin selection 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 can not be selected to be swapped.

A thread running on one of the smaller physical cores is always swapped with the thread running on the large physical core after each hardware-quantum for all three approaches, the KUTHS, the HRRS and the Fairness-aware scheduling. The fundamental difference between KUTHS and HRRS or Fairness-aware Scheduler algorithms is the way in which the threads are selected to be mapped onto the physical cores. The KUTHS scheduling tries to enhance scheduling benefits by discovering and running the critical sections of code on the larger cores, n contrast to the HRRS and the Fairness-aware approaches. The other difference between the KUTHS and Fairness-aware Scheduler algorithms is that KUTHS maps logical cores (e.g. hardware threads) onto the physical cores while Fairness-aware Scheduler schedules software threads onto the physical cores on every hardware scheduling quantum.

The difference between KUTHS to state-of-the-art in bottleneck acceleration found in BIS [13] and UBA[14] is that the KUTHS does not require any ISA extensions that have impacts on the re-usability of code. Conversely, the ease of the KUTHS implementation comes at the cost of two things. Firstly, the ease of the KUTHS implementation comes at the cost of not being able to catch all of the critical sections compared to the BIS[13] and UBA[14] 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. Also, KUTHS may potentially trigger some thread swaps that are not caused by the critical sections but by OS scheduling policy, thus substantially entailing additional overheads compared with BIS and UBA approaches. Secondly, when applied on single-threaded applications, KUTHS follows a typical HRRS policy described in the previous chapter.

6.1.1 Hardware Implementation

From the perspective of the physical hardware, the KUTHS 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 or it is executing kernel code 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 KUTHS policy attempts to promote the thread, upon acquiring the lock, to be scheduled on the large physical core to continue its execution. However, KUTHS does not necessarily execute all the critical sections of a thread on a large core.

Unlike to BIS[13] and UBA[14], the KUTHS scheduling technique does not facilitate hardware overheads to be able to store and restore the architecture state in the cores. It utilizes the x86 hardware context switching mechanism, called Hardware Task Switching in the CPU manuals [78]. To use it KUTHS needs to tell the core where to save the existing state, and where to load the new state. This is explained in detail in section 4.1.

Like the HRRS scheduler hardware implementation, presented in section 5.1.1, KUTHS also needs two counters, one register and two decoders to facilitate round-robin mechanism. The size of these components depends on the number of the cores in the system e.g. for a four core system we need a 2-bit counter. Besides those, KUTHS

has hardware additions that include a “transition-bit” on every core and a separate unit with a vector that holds all of the “transition-bits”. In contrast to the low additional overhead needed by KUTHS, UBA requires the Lagging Thread Identification (LTI), the Bottleneck Identification (BI) and the Acceleration Coordination (AC) [14] 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 [13].

6.2 KUTHS Algorithm extension for Many-Core Systems

Having explained the essence of the KUTHS algorithm, it is important to note the results in the Table 6.1 before applying it on a many core processor. Every context switch introduces overhead for a workload migration from one core to another. The main elements are: 1) a few thousand cycle penalty for storing and restoring the architecture state (register file), at most a few kilobytes 2) the overhead due to the time it takes to drain a core’s pipeline prior to migration, which is modeled in our simulations 3) the migration overhead due to cache effects, which is accounted for in our simulations. The last point is clearly the largest and most significant element. Results in Table 6.1 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 KUTHS mechanism is suffering slowdowns for certain types of workloads when applied on a system with private LLC. If each core in the system has a private LLC, the KUTHS 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 KUTHS 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.

Table 6.1: Cost of workload migration (in cycles) during context switch for workloads ranging from a few kilobytes to a few thousand kilobytes

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

6.3 Evaluation

We now evaluate the KUTHS approach and compare it to Fairness-aware Scheduling [12], Hardware Round-Robin Scheduler presented in the previous section, and UBA scheduling [14].

6.3.1 Simulated Architecture and Workloads

To analyse the proposed scheduling technique, we use the SPLASH-2 benchmarks [83]. The SPLASH-2 benchmarks are designed to represent multi-threaded applications to evaluate hardware architectures when running several thread contexts. In our experiments, we run each benchmark on the four simulated cores with each core capable of executing one hardware thread context at a time. All applications are run from start to finish. Each thread or process has a fixed amount of work to perform, and we declare that the evaluation finished once all threads/processes finished their preassigned work. For example, several threads start at the same time. The threads that finish first, of initially started ones, are restarted as they finish its preassigned work. Once the longest thread, among all, initially started threads, has been completed its preassigned work for the first time, the simulation ends. Such that the number of threads running at any one time, until the end of the simulation, in the system remains constant. We evaluate single multi-threaded application workloads running an equal or greater number of threads per application as the number of available hardware contexts, i.e., the maximum number of threads. This is common practice for running non-I/O-intensive applications [14]. For example, when we run one multi-threaded application on a system simulated with four cores, we use four or more threads for that application.

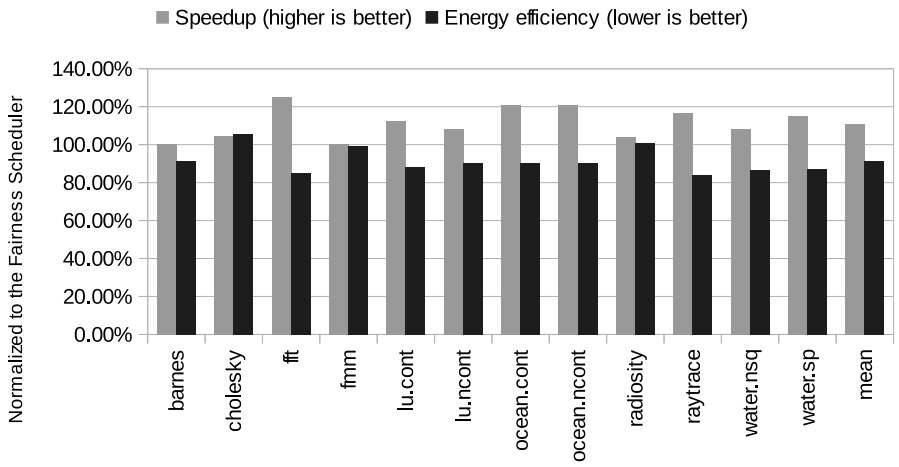

Figure 6.1: Speedup and Energy efficiency comparison of the KUTHS and the Fair scheduler for the SPLASH-2 benchmark suite in a shared LLC four core system (1 large + 3 small)

We use Sniper [75] for conducting the simulation experiments in this thesis chapter. Sniper is a parallel, hardware-validated, x86-64 multi-core simulator capable of running both multi-program and multi-threaded applications. Section 3.2.2 shows detailed system configurations of four core system consisted of one large and three small cores, used for the experiments.

Here we also test the scalability of proposed scheduling technique on an ACMP system configurations of 8, 16 or 32 cores. The 8/16/32 core configurations composed of 2/4/8 groups of four cores (1 large + 3 small) configurations, where each group of four cores share a 4MB region of the LLC. LLC regions are connected with the ring network. We call these configurations the private LLC systems, By private we mean that each group of four cores has a portion of the LLC assigned only for itself to use hence there is no sharing between the groups (even though the four cores within a group may share their portion of the LLC). Similar to the work in [12], we utilize a conservative hardware-quantum of 1 ms whereas the software-quantum is typically upwards of around the 4 ms range.

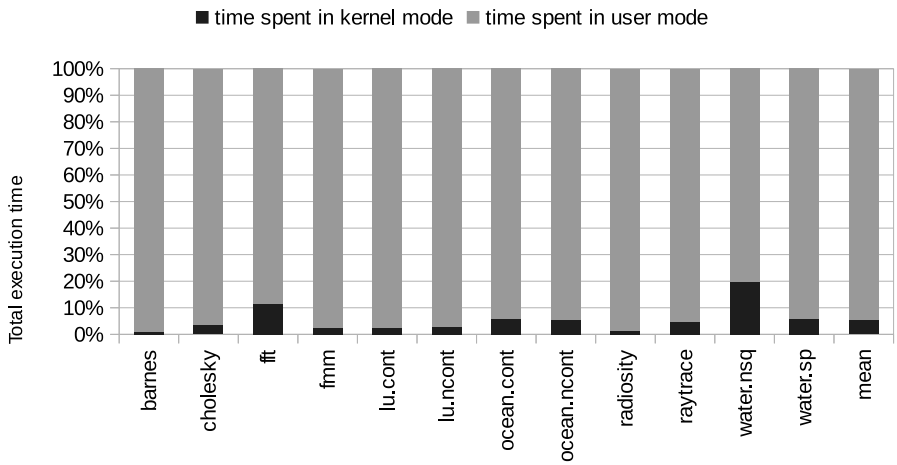

Figure 6.2: Distribution of the total execution time of the SPLASH-2 benchmark applications

6.3.2 Performance and Energy Efficiency evaluation on a Shared LLC System

Fig. 6.1 shows the speedup of the KUTHS over the Fairness-aware Scheduler (higher is better). Depending on the application, the speedup is due to the acceleration of not only the critical sections but also of the threads that were sleeping for too long while waiting for critical sections and need to catch up. The benefits arise from several facts. First, as Fig. 6.2 shows the time spent executing user and kernel code averages to 94.45 percent and 5.55 percent respectively. On average, 90.48 percent 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 synchronization and the percentage of threads continuing execution on the large core after exiting synchronization based system calls is 32.18 percent and 64.22 percent for the Fairness-aware and the KUTHS scheduler respectively as shown in Fig. 6.3. The KUTHS policy localizes more of the synchronization sections onto the large core therefore optimizing a program's execution by minimizing the time spent on synchronization sections in the slower small cores. Second, crucial communication and data sharing behavior between threads during the entire parallel phase of the applications *barnes, cholesky, fmm and radiosity*

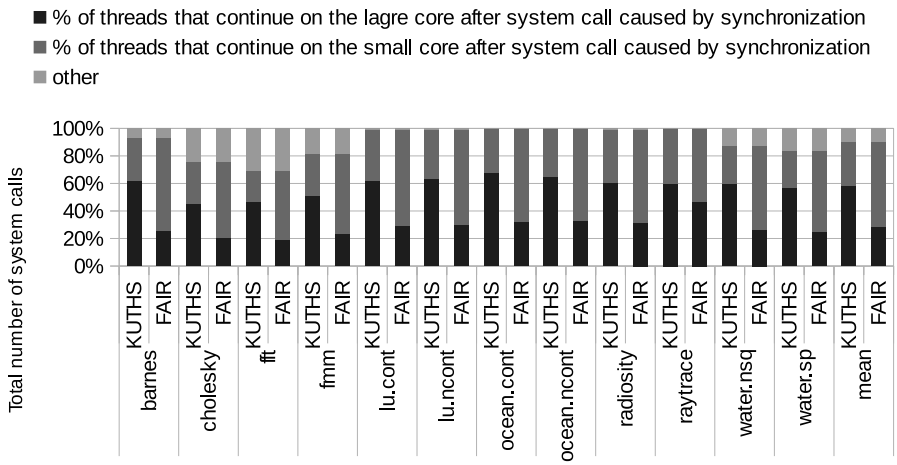

Figure 6.3: Percentage of threads that continue execution on the large or small core after synchronization based system calls in the parallel section of the application for the Fairness-aware and the KUTHS scheduler respectively

is irregular [85]. Therefore the performance of these applications does not benefit as much from the KUTHS scheduler compared to the Fairness scheduler.

Fig. 6.4 represents the speedup of the KUTHS and HRRS over the Linux OS scheduler on the Splash2 benchmark suite. The average speedup of the KUTHS over HRRS scheduler is 10.95 percent. Similar to the comparison to the Fairness-aware scheduler the speedup is consequence of the acceleration of not only the critical sections but also of the threads that were sleeping for too long while waiting for critical sections and need to catch up.

The dynamic energy efficiency (commonly used power delay product PDP) comparison of the KUTHS scheduler over Fairness scheduler on an ACMP with shared LLC (less is better) is also shown on the Fig. 6.1. Note that we do not consider energy spent in the hardware scheduler separately due to the fact that it is orders of magnitude less compared to the energy spent in the cores or caches. Our results show that KUTS is on average 9.4 percent more energy efficient than Fairness scheduler on an ACMP with the shared LLC. Fig. 6.5 represents speedup and energy efficiency comparison of the KUTHS and the Fairness scheduler while running two randomly selected applications from the SPLASH-2 benchmark suite. Each application runs as

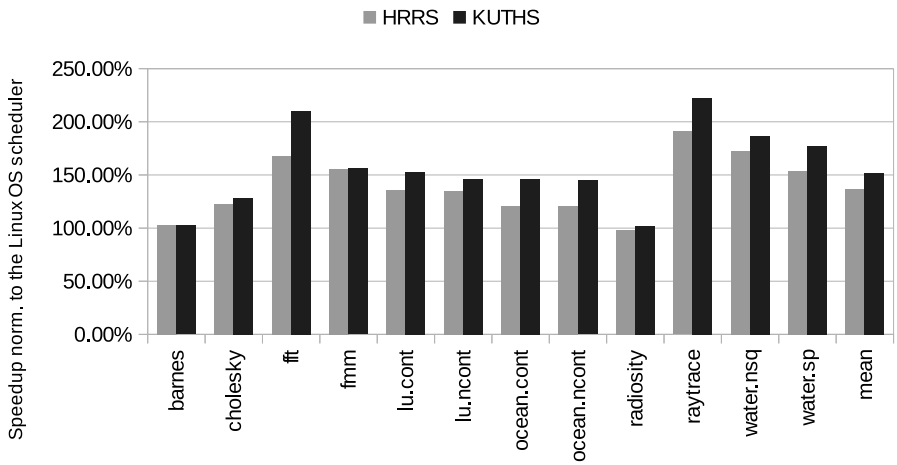

Figure 6.4: Speedup comparison of the KUTHS and the HRRS scheduler for the SPLASH-2 benchmark suite in a shared LLC four core system (1 large + 3 small)

many threads as there are physical cores in the system, therefore stressing the system with the larger number of dependent and independent software threads. Our results show that KUTHS still outperforms the Fairness scheduler by 8.7 percent while being 8.4 percent more energy efficient.

6.3.3 Hardware vs Software Implementation

KUTHS 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 [82]. Both of these approaches (KUTHS 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 [11].

The default Linux scheduling quantum is defined in the Linux kernel as RR_TIMESLICE (include/linux/sched/rt.h), and its default value is set to 100 ms. A clock tick

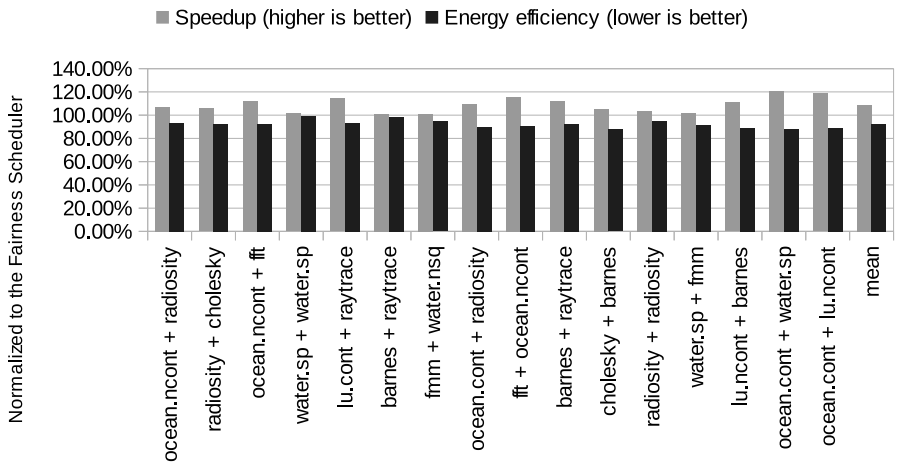

Figure 6.5: Speedup and Energy efficiency comparison of the KUTHS and the Fair scheduler running two applications from the SPLASH-2 benchmark suite in a shared LLC four core system (1 large + 3 small)

is typically 1 ms to 6 ms for the Linux kernel 2.6.8 and onwards (defined in kernel/sched/fair.c by sysctl_sched_min_granularity and sysctl_sched_latency). It can have a minimum quantum of 4 ms (a case rarely seen and still 4x the latency of the hardware quantum) [12]. The hardware scheduling quantum we have used in our experiments is 1 ms which is mentioned in the first paragraph of the Sec. 6.3.1. Fig. 6.6 represents the speedup that a hardware implementation, with a scheduling quantum of 1ms, has over a software implementation (baseline), with a scheduling quantum of the 4ms, for the Fairness-aware and the KUTHS schedulers. We can see that a hardware implementation results in an average speedup of 7 percent and 12.1 percent over a software implementation for the Fairness-aware and the KUTHS schedulers respectively.

Though a software implemented scheduler can directly provide information as to whether each thread is coming back from a synchronization-related interrupt, we have decided not to go in the direction of trying to capture all of the critical section of threads execution. When schedulers were able to identify all or most of the critical section, using online or offline profiling or ISA extensions, they experienced the problem of threads piling up and waiting in the queue on the large core and increased

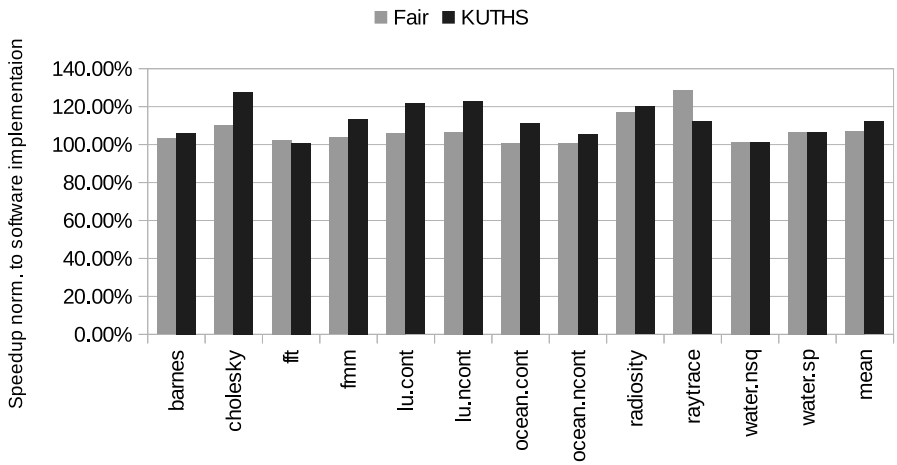

Figure 6.6: Speedup of the hardware over the software implementation (baseline) for the Fairness-aware and the KUTHS scheduler, where scheduling quanta are 1ms and 4ms for hardware and software implementations respectively

number of unused small cores in the system [13], [14], [15]. This was especially noticeable in many-core systems when running a large number of threads. Therefore they had to incorporate additional software or hardware mechanisms in order to balance the workload.

6.3.4 Performance Evaluation on Private LLC System

Fig. 6.7 shows the speedups of our proposal for 8/16/32 simulated cores with as many simulated threads per application, relative to the ACMP configuration. 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 percent respectively. On the other hand, BIS [13] proposal with 52 small cores having 3 large cores gives average performance benefits of 42 percent, while UBA [14] outperforms it by 8 percent. If we compare it to KUTHS 32 cores (8 groups) configuration, BIS [13] outperforms it by 8 percent and UBA [14] by 16 percent on average, with less large cores in the system. This comes as consequence of KUTHS lightweight yet coarser grained approach which makes it unable to identify all bottlenecks in the multithreaded applications and send them only to be

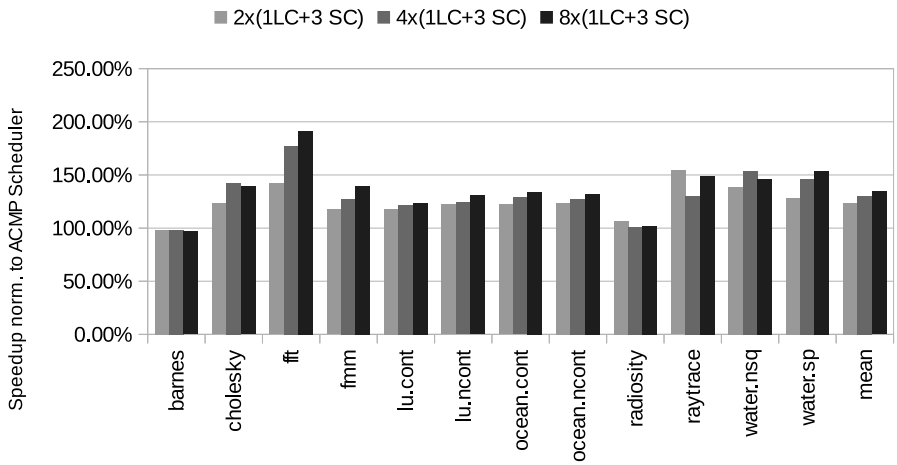

Figure 6.7: Speedup comparison of the KUTHS and the Linux OS (ACMP) Scheduler for the SPLASH-2 benchmark suite in the private last-level L3 cache 8/16/32 cores system configurations where each group of one large and three small share a 4MB L3 cache

executed on the large cores.

6.4 Summary

We have presented the Kernel to User mode code Transition aware Hardware Scheduling (KUTHS) method. Our work is heavily influenced by Fairness-aware Scheduling as well as bottleneck identification techniques. We seek to provide performance benefits from running parallel workloads on ACMPs without the need for substantial hardware extensions, sampling, or runtime overheads. Incorporating minimal hardware additions, our KUTHS policy promotes the execution of the critical sections of code on the larger rather than smaller cores within an ACMP resulting in performance gains of 11.1 percent and 30 percent (geometric mean) compared to the state-of-the-art Fairness-aware Scheduler and Linux OS Scheduler respectively, while being slower by 8 percent compared to one of the most complex and sophisticated bottleneck identification techniques running SPLASH-2 benchmarks.

Although the hardware implementation of the ACMP scheduler achieves signifi-

cant performance benefits over other conventional approaches, there are still several challenges that can enhance the previous proposal. Improvements in both the scheduling design and implementation layout of the KUTHS mechanism could lead to efficiency gains and power reduction. Therefore, we seek to incorporate a smarter criticality predictor based on monitoring cores' instructions per cycle, cache hit/miss ratios, and thread activity. The following section presents the scheduler with some of these improvements applied. These enhancements require additional hardware counters but lets us to better identify and manage a significant amount of the critical sections of the user code while maintaining hardware implementation feasibility. They also allow us to make better scheduling decisions based on the characteristics of the user code currently running.

7

TCS: Trait-aware Criticality Scheduler for Hardware-Threads

The parallel execution of multi-threaded applications, as well as multiprocess workloads on single-ISA asymmetric multi-cores, offers a potential speedup gain. In section 1.2 we have noted the importance and the impact that scheduling based on fairness, criticality and a workload's characteristics may have on the potential speedup. These features, especially a workload's characteristics, are relevant to asymmetric systems on account of workloads will perform differently on different core types based on their characteristics. Consequently, in an asymmetric system, it may be beneficial to correlate a workload's execution behaviour with a particular core type to ascertain dynamically the workload's characteristics and enhance the scheduler strategy.

We propose the Trait-aware Criticality Scheduling (TCS) policy, which is an improvement upon the HRRS scheduling policy. While the HRRS scheduling already tackles the problem of fairness, the TCS scheduling uses the short-term traits of the hardware threads to enhance the decision making process during scheduling in a man-

ner that is more in tune with a workload's specific characteristics. Additionally, in order to adapt the scheduling policy to tackle the software thread criticality, the TCS scheduler uses the long-term traits of the software threads during the scheduling decision making process. In the next sections we describe the proposed Trait-aware Criticality Scheduling (TCS) policy and discuss its hardware implementation.

7.1 TCS Algorithm Basis

Since the basis of the TCS scheduler is the HRRS scheduling policy, that we presented in chapter 5, we briefly describe it here again to help composting the two approaches. To expalin the mechanics of the HRRS approach, Fig. 7.1 showes an x86 ACMP system containing one large out-of-order (OoO) core and three smaller and identical in-order cores. Four identical logical cores form the figure correlate to four identical hardware threads. The HRRS maps the threads running on the logical cores to the physical cores after every hardware-quantum. This provides an abstracted homogeneous hardware view to the operating system. The OS scheduler maps software threads to the logical cores, e.g. hardware threads. This allows the OS scheduling policies and implementation to be left unchanged. The OS scheduler, being triggered after every software-quantum, executes much less frequently than that the HRRS, beeing triggered after every hardware-quantum. In essence, the HRRS 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.

It is important to note the defining characteristic of the HRRS algorithm. The HRRS algorithm evenly rotates threads (scheduled onto the logical cores by the OS scheduler) running on the physical cores after every hardware-quantum.

7.1.1 Differences to the HRRS Algorithm

While both approaches, TCS and HRRS, swap a thread running on one of the smaller physical cores with the thread running on the large physical core after a given time quantum, here are two fundamental differences between the two algorithms. The first significant difference is in the way in which the threads are selected to be mapped onto the physical cores. The HRRS scheduling runs each logical core and hardware

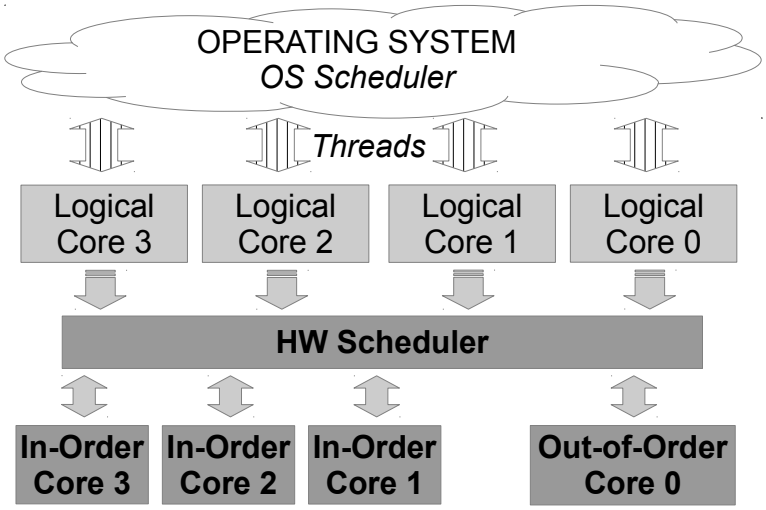

Figure 7.1: TCS scheduling - All logical cores (which correlate to hardware threads) are the same while the large physical core is represented by Core 0 and the small physical cores are shown as Cores 1,2 and 3

thread, on each physical core type for a specified amount of time, namely one hardware quantum, in the round-robin order. In contrast, the TCS policy uses a short-term progress counter to make its decision regarding which of the logical cores, hardware threads, to map on which physical core type in the following hardware quantum. The short-term progress characteristic of a thread is obtained by dividing the hardware quantum interval by the number of committed instructions of a thread. After every quantum, the TCS triggers a swap between the thread running on the large core with one executing on a small core if the short-term progress counter of one of the small cores has a higher value than the large core. In the case that the short-term counter of more than one small core has a smaller value than the large core's short-term counter, the thread running on the small core with the highest value of the short-term counter will be chosen to swap with the thread running on the large core.

The second difference between the schedulers is that the HRRS algorithm does not need to take into account whether the OS scheduler has activated and swapped one of the currently executing threads on a logical core for another thread from its ready queue. In other words, if there are several threads pinned to an individual logical core, the OS scheduler is responsible for swapping the executing thread with a ready thread on a logical core, and this does not affect the HRRS method. In such cases, the thread

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 which is all performed by the triggered OS scheduler routine. The TCS scheduling policy, however, is notified every time a software thread is not performing useful work (e.g. spinning or scheduled out by the operating system). The following subsection explains our approach of identifying these threads.

7.2 TCS approach of determining Critical Threads

To determine a thread's criticality, we use a similar method used in work by Du Bois et. al. [86]. The metric is based upon a thread's idleness and inter-thread dependency. For example, if a thread is busy waiting or spinning it is assumed that it is idle. Similarly, if other threads are waiting for it, for instance for a synchronization event, then it is denoted as having a more critical level of inter-thread dependency. A detailed definition of how we identify running threads is found in Section 7.3.1.

Since it is challenging to recognize which are the most critical threads in parallel workloads, we identify a thread as critical if the progress of the entire program is susceptible to the execution time of that individual thread. This is exemplified best in the case where several threads reach a barrier much before another thread and must remain idle until the last thread advances to the same point. In this case, the thread whose criticality level is highest can be either the longest running or longest waiting thread.

In tackling this problem, the TCS scheduling policy is able to make use of the criticality metric which takes into consideration each thread's long-term characteristics before performing a scheduling mapping at every hardware quantum. This criticality metric depends both on the running time and the number of committed instructions of running threads. Each running thread's long-term characteristic is calculated by dividing the time of an interval by the number of committed instructions of the thread doing useful work during that period which is then added to each thread's long-term characteristic sum. In other words, this metric essentially weights time and work done, increasing the level of importance for active threads which other threads may need to wait upon.

Fig. 7.2 shows an example program with 5 threads. Thread t0 starts off on the large

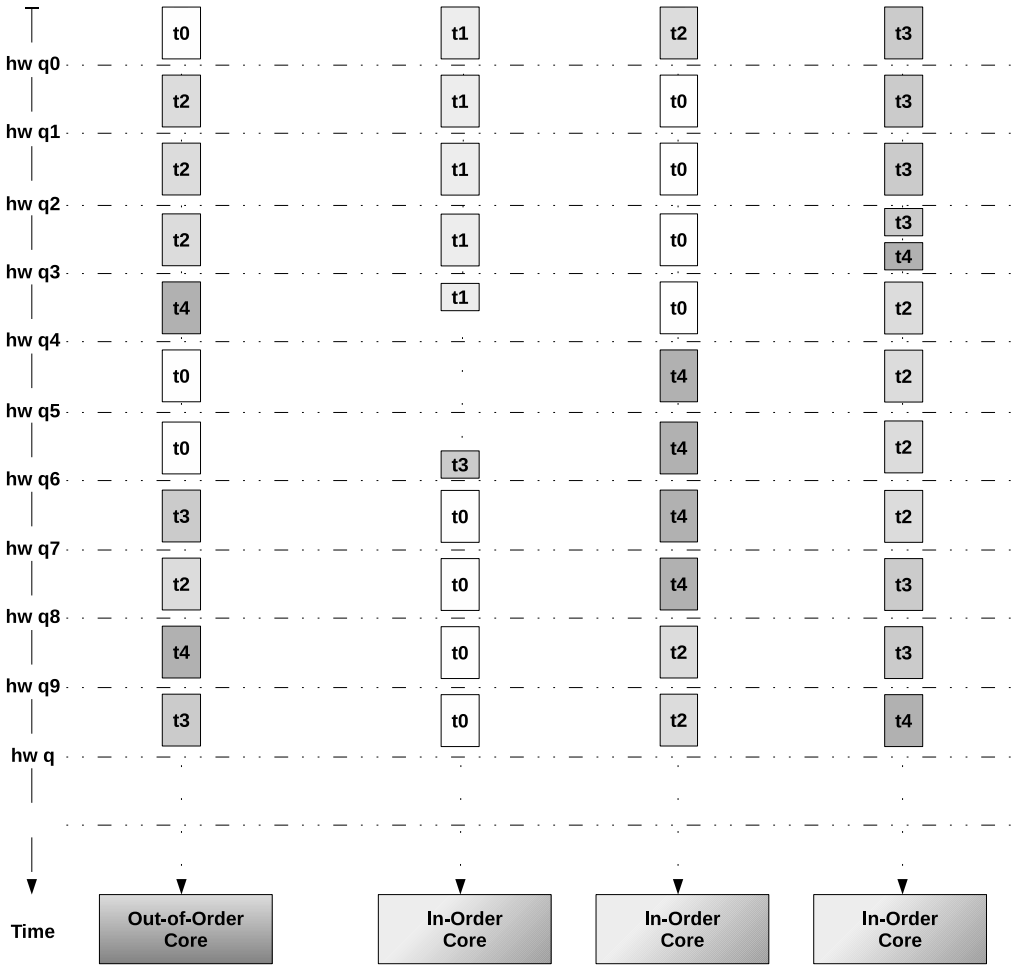

Figure 7.2: An example of the TCS scheduling threads on physical cores during every hardware-scheduling quantum. At the beginning, thread 0 is running on the large physical core while threads 2, 3 and 4 are running on the small physical cores. After every hardware scheduling quantum, threads will be rescheduled based on their short-term characteristics. Whenever a thread stalls or continues, rescheduling is performed based on the long-term thread characteristics.

core while threads t1, t2, and t3 start execution on the smaller cores. After the first hardware quantum (hw q0), the TCS uses the short-term characteristics of the threads

and determines that thread t2 has executed the least amount of instructions during the last hardware quantum and hence swaps it with thread t0 so that it can commence executing on the larger core so that it may hopefully catch up. Thread t2 is furthermore considered as the most critical short term thread during the next two hardware quanta as well so the TCS does not swap it out of the large core. After hardware quantum q3, a new thread t4 begins execution due to an OS software scheduling swap which triggers the TCS long-term scheduling policy. Since the total criticality sum of the new thread t4 is less that either t0, t1, or t2, all of which have been previously executing, the TCS schedules t4 to begin executing on the large core at the start of hardware quantum 4. The rest of the figure illustrates a similar process where at the end of each hardware quantum, the TCS invokes either the short-term (i.e. at the end of hw q7, q8, and q9) or long-term scheduling approaches (i.e. at the end of hw q5 and q6).

7.2.1 Differences to the other Criticality Schedulers

The fundamental differences between the TCS and the criticality stacks study [86] are based on the manner in which the thread criticality metric is computed as well as how rescheduling is triggered. In contrast to the TCS, the criticality stacks technique takes into account the running time and the number of running threads. In other words, the time of an interval is divided by the number of threads doing useful work during that period, and this is added to each thread's criticality sum. This metric essentially weights time, opposed to the TCS metric which weights time and the work done. While both techniques divide execution time into a number of intervals, the criticality stacks approach begins a new interval whenever any thread changes state as a result of synchronization behavior from active to inactive or vice versa, while TCS is based on a fixed hardware quantum.

Unlike KUTSH approach, TCS does need OS library extensions to be made (pthread). Because of it, the TCS policy is able to identify all critical sections marked by the user. Bottleneck Identification and Scheduling (BIS) [13] tries to identify parallel bottlenecks, and migrates threads executing these bottlenecks to a large core in a heterogeneous multicore. While they speed up bottlenecks that restrict parallel execution, we rather discover the thread(s) essential to overall execution. A bottleneck could be on the critical for the performance of one thread, but not for the overall performance

of the application. Hence, accelerating bottlenecks does not inevitably enhance performance, and could still needlessly speed up threads, reducing energy efficiency [86].

7.3 Hardware Implementation

Compared to other state-of-the-art criticality based scheduling methods, the amount of overhead with our implementation is relatively similar and lightweight. In this section, we first describe the manner of identifying running threads and then present an example case of a possible hardware implementation of the TCS policy. At the end of this section we elaborate on the manner in which context switches are managed.

7.3.1 Identification of the Running Threads

A thread identified as idle will typically be either spinning, that is to say running in a waiting look for some reason such as checking the status of a synchronization variable, or scheduled out of physical execution by the operating system. While the latter is easily detected since the OS can communicate when it has scheduled threads in or out of physical execution, to detect when a thread is spinning is non-trivial due to the fact that the spinning thread is still executing instructions.

This may be solved both at the hardware and software level. Hardware approaches typically use tables within processors to keep track of backward branches, the frequency of which may help to identify a spinning loop, or repetitive operation such as loading a condition variable. In order to improve accuracy of spinning identification, additional conditions are checked alongside of the tables including architectural state changes and load address modifications. Conversely, software based solutions involve the insertion of extra instructions into the executable code which serve to denote when a thread is spinning. To avoid the need for developers to manually insert such instructions, they are conventionally added directly into threading libraries such as Pthreads.

Each approach has advantages as well as drawbacks. The software approach, by using semantic information from the program, will never falsely identify spinning loops and is able to determine the exact start and end of the loop. Relying on semantic information, however, stipulates that non-instrumented user-level spinning will not

be detectable. On the other hand, hardware solutions typically detect spinning loops once a certain threshold and condition set has been reached which can have adverse impacts on its efficiency. As a result, hardware approaches may also produce false negatives (e.g., when the number of loop iterations is lower than the threshold) and false positives (e.g., non-monitored architectural state modifications).

In our work we utilize a software based approach to identifying spinning due to its efficacy and ease of implementation. We evaluate benchmarks that perform synchronization using threading libraries (Pthreads and OpenMP) which we instrument to be able to capture all events involving spinning including barriers, locks, and condition variables. Our instrumentation allows us to notify the hardware that a thread has become either active or inactive via a call-down when the application enters or exits a spinning loop. Next we describe how based on these call-downs, the hardware is able to dynamically calculate the criticality of a thread.

7.3.2 Hardware Component Description

We propose a small hardware component that keeps track of the number of committed instructions and time elapsed per hardware thread and calculates the criticality of each thread. The hardware component, shown in the Fig. 7.3, consists of: two 64-bit criticality counters per hardware thread (short and long term counters) marked as STC and LTC on the figure, an "active" bit that indicates whether the thread is running or not, and a number of committed instructions counter (64-bits) per hardware thread. In order to calculate the criticality as defined in the previous section, we need to know which threads performe useful work during the hardware quanta. The active flag of each thread is set or reset via the software calls either activating or deactivating a thread. Besides this, the hardware component also has one global timer and four per hardware thread timers. These timers are independent of a core's frequency and monitor absolute time. Global counter keeps track of the hardware quantum. Once the timer reaches the quantity of time of a hardware quantum, it resets the committed instruction counters and per-hardware thread timers and signals the update of criticality counters. Hardware thread timers are used to calculate the criticality metric.

Short and long term counters are updated every hardware quantum with the criticality result of each thread (i.e., the absolute time of the hardware quantum divided by

the number of committed instructions) being overwritten in its short term criticality counter and added to its total long term criticality counter sum. This is followed by resetting the quantum timer and committed instructions counter. When a software call is received, it triggers a reset of the active bit of a particular thread, we add its criticality result to a threads long term criticality counter before updating the hardware state.

Although each software thread needs to be associated with a long term criticality counter, we need to implement only one physical counter per hardware context even though there may be more software threads and hardware threads. Since only running software threads need to update their criticality counters, all that is needed in hardware is one criticality counter, a quantum timer, and an active bit per hardware thread. When a context switch occurs, the operating system will store the long term criticality state associated with the swapped out thread and initialize the core's long term criticality state to be associated with the newly active thread. Therefore, our implementation may work with many more software threads than physical cores or hardware threads.

7.3.3 Managing the Context Switches

The TCS scheduling method does not provide the hardware overheads with the ability to store and restore the architecture state in the cores, as opposed to some other dynamic schedulers [13], [14]. It utilizes the x86 hardware context switching mechanism, called Hardware Task Switching in the CPU manuals [78]. The HRRS must instruct the core where to store the existing CPU state, and where to load the new CPU state. There is a detailed explanation of this in section 4.1.

7.4 Evaluation

In this section we evalute the TCS schedulin policy. We compare it to the state-of-the-art policies in scheduling for fairness, workload characteristics and thread criticality.

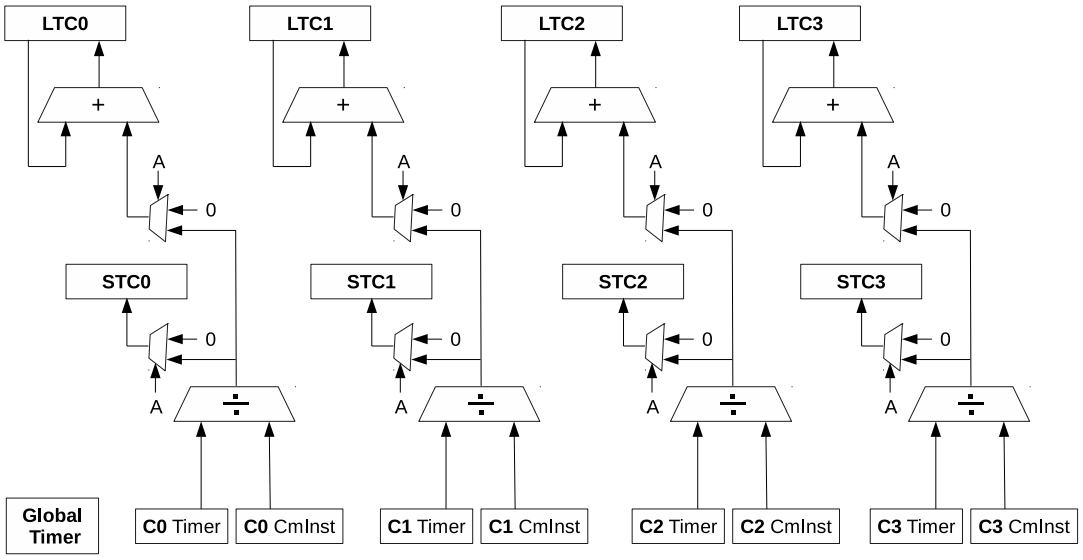

Figure 7.3: Hardware implementation layout example for TCS scheduling policy on the four core ACMP consisting of one large and three small cores (1 OoO + 3 InO), where "A" is the active bit per hardware thread and "CmInst" is the number of committed instructions in given time quantum per hardware thread.

7.4.1 Simulated Architecture and Workloads

We utilize the SPLASH-2 [83] and SPEC2006 [68] benchmarks in our tests. The SPLASH-2 benchmarks are intended to embody multi-threaded applications to assess hardware designs when running several thread contexts. The SPEC2006 benchmark is an industry-standardized, CPU-intensive benchmark suite, stressing a system's processor, memory subsystem, and compiler. We have employed the SPEC2006 benchmarks to run multiple instances of the various single threaded applications concurrently on the system. We run each workload on the four simulated cores with each core able of executing one hardware thread context at a time.

The individual workload may consist of one or several processes or threads originating from one or more applications. Each thread or process has a fixed amount of work to perform, and we declare that the evaluation finished once all threads/processes finished their preassigned work. In the event of possessing several applications or threads at the same time, the threads that finish earliest are restarted so that the

number of threads running at any one time on the system remains constant. Once the longest thread/application has been finished, the simulation ends. We evaluate single multi-threaded application workloads running an equal number of threads per application as the number of available hardware contexts, i.e., the maximum number of threads. When we run workloads consisted of the single-threaded applications, we run at least as many different applications as there are available hardware contexts, i. e. max number of cores. This is common practice for running non-I/O-intensive applications [14]. For example, when we run one multi-threaded application on a system simulated with four cores, we use four or more threads for that application.

For performing the simulation experiments we have used Sniper [75], a parallel, hardware - validated, x86-64 multi-core simulator capable of running both multi-program and multi-threaded applications, described in Section 3.2. We configured the simulator to model an ACMP made up of one large core and three small cores respectively. Section 3.2.2 shows detailed system configurations used for the experiments. Similar to the work in [12], we utilize a conservative hardware-quantum of 1ms and a software-quantum of 4ms even though it is typically upwards of this range.

7.4.2 Comparison to the Scheduling for Fairness

Fig. 7.4 illustrates performance (system throughput or weighted speedup) comparison of the TCS and HRRS scheduler relative to the pinned scheduling method for random workload mixes on a heterogeneous multi-core with one large and three small cores. The workload mixes consist of four randomly selected applications from the SPEC2006 benchmark suite. From the figure we can see that for some mixes of applications TCS underperforms compared to the pinned scheduler. Small (e.g., in-order) cores provide adequate performance for compute-intensive workloads whose following instructions in the dynamic instruction stream are for the most part independent (i.e., high levels of inherent instruction level parallelism or ILP). On the other hand, large (e.g., out-of-order) cores provide decent performance for workloads where the ILP must be obtained dynamically, or the workload presents a substantial quantity of memory level parallelism (MLP). Consequently, scheduling decisions on heterogeneous multi-cores can be significantly enhanced by taking into account how well a small or large core can utilize the ILP and MLP properties of a workload. Since the

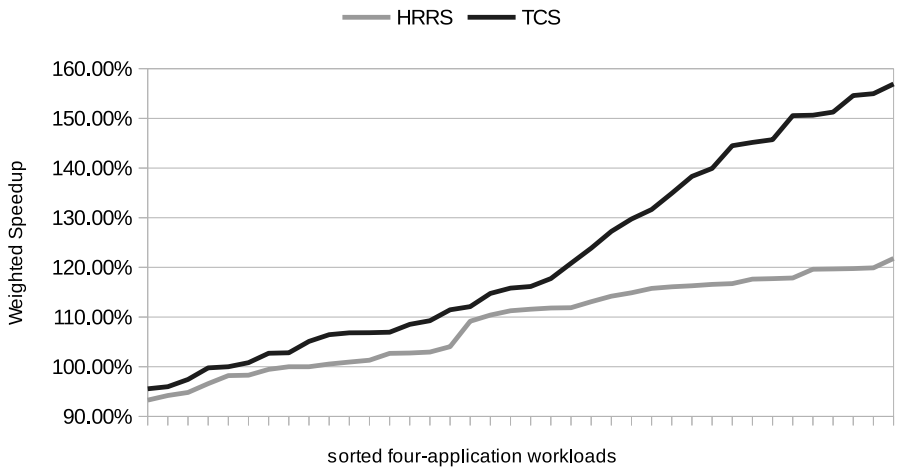

Figure 7.4: Weighed speedup (system throughput) comparison of the TCS and HRRS scheduler normalized to the pinned scheduler on an ACMP consisting of four cores (1 OoO + 3 InO) for the SPEC2006 benchmark.

TCS does not take into account the threads' MLP, it will keep rescheduling threads on every hardware quantum although threads might not have high enough MLP or ILP to overcome the cost of the introduced context switches.

7.4.3 Comparison to the Scheduling for Workload Characteristics

Fig. 7.5 presents performance improvement (system throughput or weighted speedup) over the pinned scheduler for random workload mixes on a heterogeneous multi-core with one large and one small core. The workload mixes consist of two randomly selected applications from the SPEC2006 benchmark suite. We have been running hundred different randomly selected application mixes. For the most workloads the TCS is outperforming the pinned scheduler, but in a few cases it may suffer up to 5 percent slowdown. This happens in cases when we have workload consisted of two heavily memory bounded applications. Since TCS is not able to grasp weather an application is memory bounded or not it will continue rescheduling applications among large and small cores though in such a cases it may be better not to.

In Fig. 7.6 we compare TCS scheduling against random, memory-dominance (mem-dom) scheduling, MLP ratio based scheduling, and the state-of-the-art PIE scheduling

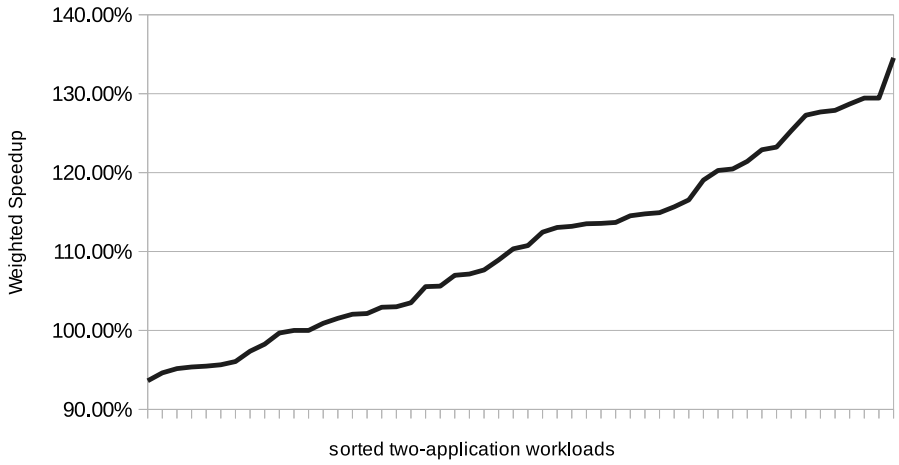

Figure 7.5: Weighed Speedup (system throughput) comparison of the TCS over the pinned scheduler on an ACMP consisting of two cores (1 OoO + 1 InO) for the SPEC2006 benchmark.

[11]. Memory-dominance scheduling refers to the conventional practice of always scheduling memory-intensive workloads on the small cores. Our results shows that TCS underperforms compared to the state-of-the-art scheduler for workload characteristic (i.e. PIE) by only as much as 2.6 percent on average even for heterogeneous multi-core with the one large and one small core. This comes as the consequence of the fact that TCS metric is based on ILP and does not consider the thread's MLP separately. These results highlight the significance that MLP detection plays in optimizing scheduling. Though, the PIE outperforms the TCS by 2.6 percent, the complexity of the hardware implementation of the PIE scheduler over the TCS is significant since PIE requires additional counters, registers as well as logic circuits to calculate features like last level cache misses of each core.

7.4.4 Comparison to the Scheduling for Criticality

Fig. 7.7 shows performance (system throughput or weighted speedup) relative to pinned scheduling for multi-thread workloads from the SPLASH2 benchmark suite on a heterogeneous multi-core with one large and one small core. We compare our TCS approach to the KUTHS and criticality stacks [86] (criticality) approaches. While the

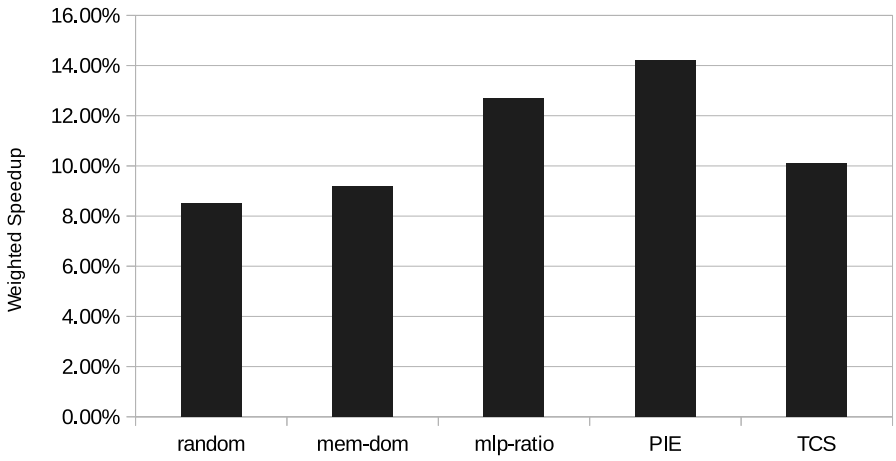

Figure 7.6: Comparing absolute average weighed speedup (system throughput) of the different scheduling policies over the pinned scheduler on an ACMP consisting of two cores (1 OoO + 1 InO)

KUTHS and the criticality stack techniques exhibit similar performance benefits, TCS outperforms both aproaches by about 24 percent on average. While KUTHS accelerates bottlenecks that limit parallel performance, the criticality stacks technique and the TCS approach instead find the thread(s) which are critical to overall performance. For example, a bottleneck could be on the critical path for one thread, but not for others, consequently, speeding up bottlenecks do not significantly increase overall performance. Though, the KUTHS technique is not able to identify every bottleneck throughout the application execution, the ones it does identify it accelerates by executing it on the large core. Thus, its underlying approach to scheduling for fairness compensates for not being able to catch all bottlenecks and leads to similar performance benefits as gained via the criticality stack technique. In contrast to the criticality stacks and the KUTHS approachs, the TCS reschedules running threads applying all three methods: scheduling for fairness, scheduling for criticality and scheduling for workload characteristics. This results in significant performance benefits because the threads are checked for rescheduling during every hardware quantum but are not necessarily moved from a small to the large core therefore it does not introduce as many unnecessary context switches as KUTHS which frequently schedules the slow-

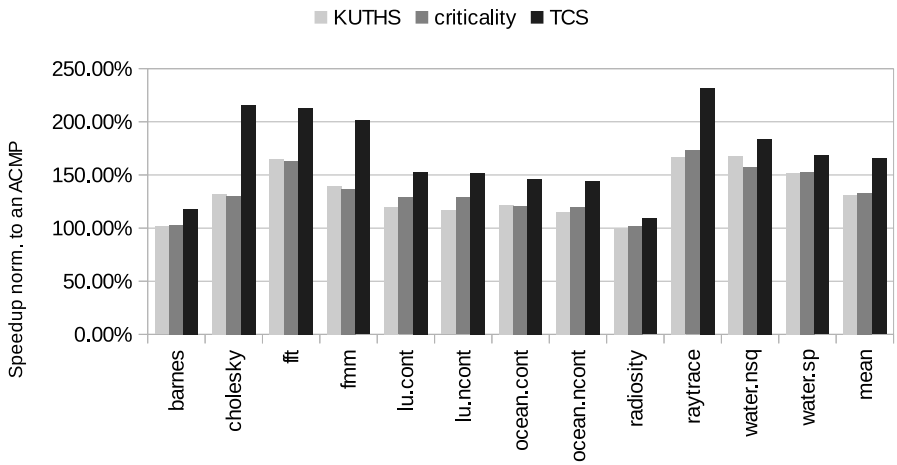

Figure 7.7: Weighed Speedup (system throughput) comparison of the KUTHS, criticality and TCS schedulers normalized to the pinned scheduler on an ACMP consisting of four cores (1 OoO + 3 InO) for the SPLASH2 benchmark.

est thread out from the large core after only one hardware quantum. The criticality stack approach reschedules threads only when it receives a notification from software that the thread has been scheduled in or out from the core. This leads to much less frequent and unfair scheduling since a slower thread might end up on the small core for a large period of time therefore reducing overall application performance.

7.5 Summary

Single-ISA asymmetric multi-cores are typically composed of small (e.g., in-order) cores and large (e.g., out-of-order) cores. Employing various core types on an individual die has the potential to increase performance. However, the success of asymmetric multi-cores is directly dependent on how well a scheduling policy maps workloads to the best core type (large or small). Incorrect scheduling decisions can unnecessarily degrade performance.

We have presented the Trait-aware Criticality Scheduling method of the hardware threads on an ACMP. Our work is heavily influenced by scheduling for fairness, scheduling for workload characteristics as well as bottleneck identification techniques.

We have proposed a simple but effective scheduling mechanism using a thread criticality metric. We have applied this metric, which uses the amount of instructions completed per hardware quantum, to gain insight into short-term and long-term thread criticality. Our Trait-aware Criticality Scheduler (TCS) uses both of these criticality measures to produce a mapping of threads to cores which takes into consideration the runtime performance characteristics of individual workloads. We seek to provide performance benefits from running parallel and multi-application workloads on an ACMP without the need for substantial hardware extensions, sampling, or runtime overheads. Incorporating minimal hardware additions, our TCS policy promotes the execution of the critical threads on the larger rather than smaller cores within an ACMP resulting in performance gains of 11 percent and 24.4 percent (geometric mean) compared to the state-of-the-art techniques in scheduling for fairness and scheduling for criticality respectively, while being slower by only 2.6 percent compared to one of the state-of-the-art techniques in scheduling for workload characteristics.

8

Conclusion and Future Work

Single-ISA asymmetric multi-cores are typically composed of small (e.g., in-order) cores and large (e.g., out-of-order) cores. Using different core types on a single die has the potential to improve performance. However, the success of asymmetric multi-cores is directly dependent on how well a scheduling policy maps workloads to the best core type (e.g. in-order or out-of-order). Incorrect scheduling decisions can unnecessarily degrade performance.

In Chapter 5, we have presented the Hardware Round-Robin Scheduler (HRRS). Our work is influenced by the rise of many core processors, particularly the asymmetric core multi-processors (ACMPs) and their dependence on dynamic schedulers such as the commodity Linux OS CPU scheduler in order to achieve fair and balanced performance between active threads. Our initial objective was to achieve these performance benefits from running mitiple single-threaded applications and parallel multi-threaded workloads on ACMPs without the need for substantial hardware extensions, sampling, or runtime overheads. We discussed the possible hardware implementation of the HRRS scheduler where by incorporating minimal hardware additions

our HRRS policy promotes a balanced distribution of execution time for threads per core type. We have shown that HRRS provides greater opportunity for all hardware threads to share time running on the more efficient large core, selected via a round-robin algorithm, which produces generous performance benefits even after including scheduler and context swap overheads as well as latencies arising from the additional cache accesses needed for loading the working data sets. By using the HRRS policy on an ACMP, we got an average speed up of 17.2 percent and 11.71 percent, while being on average 7.57 percent and 6.56 percent more energy efficient compared to the Fairness scheduler when running a multi-threaded application workloads (Splash2) and when running multiple instances of the single-threaded applications (SPEC2006) respectively. This work was a foundation upon which we have built and tested our proposed hardware scheduling designs in order to further improve the performance and energy efficiency capabilities of ACMPs.

In Chapter 6, we presented the Kernel to User mode code Transition aware Hardware Scheduling (KUTHS) method. Our work is heavily influenced by Fairness-aware Scheduling as well as bottleneck identification techniques. We seek to provide performance benefits from running parallel multi-threaded workloads on ACMPs without the need for substantial hardware extensions, sampling, or runtime overheads. Incorporating minimal hardware additions, our KUTHS policy promotes the execution of the critical sections of code on the larger rather than smaller cores within an ACMP resulting in performance gains of 11.1 percent and 30 percent (geometric mean) compared to the state-of-the-art Fairness-aware Scheduler and Linux OS Scheduler respectively, while being slower by 8 percent compared to on of the most complex and sophisticated bottleneck identification techniques running SPLASH-2 benchmarks.

In Chapter 7, we presented the Trait-aware Criticality Scheduling method of the hardware threads on an ACMP. Our work is heavily influenced by scheduling for fairness, scheduling for workload characteristics as well as bottleneck identification techniques. We have proposed a simple but effective scheduling mechanism using a thread criticality metric. We have applied this metric, which uses the amount of instructions completed per hardware quantum, to gain insight into short-term and long-term thread criticality. Our Trait-aware Criticality Scheduler (TCS) uses both of these criticality measures to produce a mapping of threads to cores which takes into consideration the runtime performance characteristics of individual workloads. We seek to provide per-

formance benefits from running parallel and multiapplication workloads on an ACMP without the need for substantial hardware extensions, sampling, or runtime overheads. Incorporating minimal hardware additions, our TCS policy promotes the execution of the critical threadss on the larger rather than smaller cores within an ACMP resulting in performance gains of 11 percent and 24.4 percent (geometric mean) compared to the state-of-the-art techniques in scheculing for fairness and scheculing for criticality respectively, while being slower by only 2.6 percent compared to on of the state-of-the-art technique in scheduling for workload characteristics.

8.1 Future Work

When considering an LLC cache for many-core processors, an popular option gaining traction in the industry is to distribute the cache into separate blocks, therefore appearing as unified rather than being physically unified. This is a similar approach to that taken by the IBM Power8 architecture [87] where each core has an 8MB low-latency LLC cache and a high-speed cache-coherent ring is used to connect all of the cores. Thus joined, the LLC cache blocks can be viewed as a shared 96MB cache with a nonuniform latency. Access to the local 8MB LLC is speedy, but access to remote LLC cache blocks will require additional cycles to traverse the ring. By comparison, a large unified LLC cache would have a constant access time slower than the local cache but faster than remote blocks. The IBM design keeps hot data in the CPU's local LLC, reducing the average LLC latency.

In a many-core processor with this kind of distributed LLC cache configuration, a latency problem may arise due to frequent context switches among cores being connected to a different LLC cache segments. Therefore, the TCS scheduling heuristic many need to be adjusted to account for the added latencies of the LLC and moreover can be tuned to allow for the scheduling of threads to be such as to take advantage of the distribution of cache blocks. Perhaps it may become viable to not only swap threads from large to small cores but also from small to small depending on which LLC segments their data sets are located. Furthermore, it would be beneficial to implement the TCS scheme on an FPGA in order to gauge the feasibility of the design as well as raise the level of accuracy concerning the latency overheads.

9

Publication List

The content of this thesis led to following publications:

Journals

- N. Markovic, D. Nemirovsky, O. Unsal, M. Valero, and A. Cristal, Thread Lock Section-aware Scheduling on Asymmetric Single-ISA Multi-Core, IEEE Computer Architecture Letters, DOI:10.1109/LCA.2014.2357805, Volume pp., Issue 99., pages 1-1, October 2014.

- N. Markovic, D. Nemirovsky, O. Unsal, M. Valero, and A. Cristal, Kernel-to-User-Mode Transition-Aware Hardware Scheduling, IEEE Micro, Volume 35., Issue 4., pages 37-47, August 2015.

Conferences

- N. Markovic, D. Nemirovsky, V. Milutinovic, O. Unsal, M. Valero, and A. Cristal, Hardware Round-Robin Scheduler for Single-ISA Asymmetric Multi-core, Euro-Par 2015: Parallel Processing, Lecture Notes in Computer Science, Volume 9233, pages 122-134, August 2015

- N. Markovic, D. Nemirovsky, O. Unsal, M. Valero, and A. Cristal, TCS: Trait-aware Criticality Scheduling for Hardware-Threads on Single-ISA Asymmetric Chip Multiprocessor, The 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Under review, March 2016.

Publications related but not included into this thesis:
Conferences

- N. Markovic, D. Nemirovsky, O. Unsal, M. Valero, and A. Cristal, Performance and energy efficient hardware-based scheduler for Symmetric/Asymmetric CMPs, The 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), October 2015.

Publications not related to this thesis:
Journals

- D. Nemirovsky, N. Markovic, O. Unsal, M. Valero, and A. Cristal, Reimagining Heterogeneous Computing: a Functional Instruction Set Architecture (F-ISA) Computing Model, IEEE Micro, Accepted for publication, 2015.

Workshops

- N. Markovic, D. Nemirovsky, O. Unsal, M. Valero, and A. Cristal. Object Oriented Execution Model (OOM). The 2nd Workshop on New Directions in Computer Architecture (NDCA-2), held in Conjunction with the 38th International Symposium on Computer Architecture (ISCA-38), June 2011.

Bibliography

[1] Intel Corporation. Intel® core™ i7-4600u processor, 2013. [Online] Available: ark.intel.com/products/76616/Intel-Core-i7-4600U-Processor-4M-Cache-up-to-3_30-GHz. xiii, 26, 40

[2] P. Greenhalgh. big.little processing with arm cortex-a15 & cortex-a7, 2011. [Online] Available: http://www.arm.com/files/downloads/bigLITTLE_Final_Final.pdf. 2

[3] NVIDIA. Variable smp: A multi core cpu architecture for low power and high performance, 2011. [Online] Available: http://www.nvidia.com. 2

[4] R. Kumar, and D. M. Tullsen, and P. Ranganathan, and N. P. Jouppi, and K. I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *Proc. 31st Annu. Int. Symp. Comput. Archit.*, page 64, 2004. 2, 8, 17

[5] R. Rodrigues, and A. Annamalai, and I. Koren, and S. Kundu, and O. Khan. Performance per watt benefits of dynamic core morphing in asymmetric multicores. In *Proc. Int. Conf. Parallel Architectures Compilation Tech.*, pages 121–130, 2011. 2

[6] M. Jones. Inside the linux 2.6 completely fair scheduler, 2009. [Online] Available: http://www.ibm.com/developerworks/library/l-completely-fair-scheduler/l-completely-fair-scheduler-pdf.pdf. 2, 35, 47, 50, 53

[7] S. Li, and J. Ho Ahn, and R. D. Strong, and J. B. Brockman, and D. M. Tullsen, and N. P. Jouppi. Quantifying the cost of context switch. In *Proc. Workshop Exp.Comput. Sci.*, pages 2–es, 2007. 3, 14, 41

[8] R. Netzer, and B. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1:74–88, 1992. 4

[9] M. Becchi, and Patrick Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. *J. Instruction-Level Parallelism*, 10:1–26, 2008. 5, 17

[10] D. Koufaty, and D. Reddy, and S. Hahn. Bias scheduling in heterogeneous multi-core architectures. In *Proc. 5th Eur. Conf. Comput. Syst.*, pages 125–138, 2010. 5, 17

[11] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer. Scheduling heterogeneous multi-cores through performance impact estimation (pie). In *Proc. 39th Annu. Int. Symp. Comput. Archit.*, pages 213–224, 2012. 5, 8, 22, 39, 44, 47, 48, 70, 87

[12] K. Van Craeynest, S. Akram, W. Heirman, A. Jaleel, and L. Eeckhout. Fairness-aware scheduling on single-isa heterogeneous multi-cores. In *Proc. 22nd Int. Conf. Parallel Archit. Compilation Tech.*, pages 177–187, 2013. 6, 9, 20, 35, 41, 43, 44, 46, 48, 49, 56, 66, 67, 71, 85

[13] J. Joao, and M. A. Suleman, and O. Mutlu, and Y. N. Patt. Bottleneck identification and scheduling in multithreaded applications. In *Proc. 17th Int. Conf. Architectural Support Program Languages Operating Syst.*, pages 223–234, 2012. 7, 22, 48, 64, 65, 72, 80, 83

[14] J. Joao, and M. A. Suleman, and O. Mutlu, and Y. N. Patt. Utility-based acceleration of multithreaded applications on asymmetric cmps. In *Proc. 40th Annu. Int. Symp. Comput. Archit.*, pages 154–165, 2013. 7, 23, 48, 50, 64, 65, 66, 72, 83, 85

[15] S. Srinivasan, and L. Zhao, and R. Illikkal, and R. Iyer. Efficient interaction between os and architecture in heterogeneous platforms. *Operating Syst. Rev.*, 45:62–72, 2011. 8, 17, 72

[16] A. Agarwal, J. Hennessy, and M. Horowitz. Cache performance of operating system and multiprogramming workloads. *ACM Trans. Comput. Syst.*, 6-4:393–431, 1988. 14, 15

[17] J. C. Mogul and A. Borg. The effect of context switches on cache performance. In *Proc. Int. Conf. Architectural Support Program Languages Operating Syst.*, pages 75–84, 1991. 14

[18] G. E. Suh, E. Peserico, S. Devadas, and L. Rudolph. Job-speculative prefetching: Eliminating page faults from context switches in time-sharing systems, 2001. [Online] Available: `http://csg.csail.mit.edu/pubs/memos/Memo-442/memo-442.pdf`. 14

[19] D. Chiou, S. Devadas, J. Jacobs, P. Jain, V. Lee, E. Peserico, P. Portante, L. Rudolph, G. E. Suh, and D. Willenson. Scheduler-based prefetching for multilevel memories, 2001. [Online] Available: `http://csg.csail.mit.edu/pubs/memos/Memo-444/memo-444.pdf`. 14

[20] P. Koka and M. H. Lipasti. Opportunities for cache friendly process scheduling. In *Proc. Workshop on Interaction between Operating System and Computer Architecture*, page 1, 2005. 14

[21] D. Tsafrir. The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops). In *Proc. Workshop on Experimental Computer Science*, page 1, 2007. 14

[22] F. M. David, J. C. Carlyle, and R. H. Campbell. Context switch overheads for linux on arm platforms. In *Proc. Workshop on Experimental Computer Science*, page 1, 2007. 14

[23] G. E. Suh, S. Devadas, and L. Rudolph. Analytical cache models with applications to cache partitioning. In *Proc. Int. Conf. on Supercomputing (ICS)*, pages 1–12, 2001. 15

[24] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Proc. Int. Conf. on High Perf. Comp. Arch. (HPCA)*, pages 117–128, 2002. 15

[25] W. Hwu and T. M. Conte. The susceptibility of programs to context switching. *IEEE Transactions on Computers*, 43-9:994–1003, 1994. 15

[26] F. Liu and Y. Solihin. Understanding the behavior and implications of context switch misses. *ACM Trans. Archit. Code Optim.*, 7-4:1–28, 2010. 15, 18

[27] H. Cui, and S. Sair. Extending data prefetching to cope with context switch misses. In *Proc. Int. Conf. on Comp. Design (ICCD)*, pages 260–267, 2009. 15

[28] D. Daly, and H. W. Cain. Cache restoration for highly partitioned virtualized systems. In *Proc. Int. Conf. on High Perf. Comp. Arch. (HPCA)*, pages 1–10, 2012. 16

[29] J. Zebchuk, H. W. Cain, V. Srinivasan, and A. Moshovos. Recap: a region-based cure for the common cold cache. In *Proc. Int. Conf. on High Perf. Comp. Arch. (HPCA)*, pages 83–94, 2013. 16

[30] J. A. Brown, L. Porter, and D. M. Tullsen. Fast thread migration via cache working set prediction. In *Proc. Int. Conf. on High Perf. Comp. Arch. (HPCA)*, pages 193–204, 2011. 16

[31] L. Miller. A heterogeneous multiprocessor design and the distributed scheduling of its task group workload. In *Proc. of the 9th Ann. Symp. on Comp. Arch.*, pages 283–290, 1982. 16

[32] E. Grochowski, and R. Ronen, and J. Shen, and H. Wang. Best of both latency and throughput. In *Proc. of the Int. Conf. on Computer Design (ICCD)*, pages 236–243, 2004. 17

[33] D. Moncrieff, and R. E. Overill, and S. Wilson. Heterogeneous computing machines and amdahl's law. *Parallel Computing*, 22:407 – 413, 1996. 17

[34] D. Menasce and V. Almeida. Cost-performance analysis of heterogeneity in supercomputer architectures. In *Proc. of the 4th Int. Conf. on Supercomputing*, pages 169–177, 1990. 17

[35] R. Kumar, and K. I. Farkas, and N. P. Jouppi, and P. Ranganathan, and D. M. Tullsen . Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Proc. 36nd Annu. IEEE/ACM Int. Symp. Microarchit.*, page 81, 2003. 17

[36] J. C. Saez, and M. Prieto, and A. Fedorova, and S. Blagodurov. A comprehensive scheduler for asymmetric multicore systems. In *Proc. 5th Eur. Conf. Comput. Syst.*, pages 139–152, 2010. 17

[37] D. Shelepov, and J. C. Saez Alcaide, and S. Jeffery, and A. Fedorova, and N. Perez, and Z. F. Huang, and S. Blagodurov, and V. Kumar. Hass: a scheduler for heterogeneous multicore systems. *ACM SIGOPS Operating Systems Review*, 43:66–75, 2009. 17

[38] O. Khan, and S. Kundu. A self-adaptive scheduler for asymmetric multi-cores. In *Proc. of the 20th Symp. on Great lakes symposium on VLSI*, pages 397–400, 2010. 18

[39] A. Moshovos, G. Memik, B. Falsafi, and A. Choudhary. Jetty: Filtering snoops for reduced energy consumption in smp servers. In *Proc. Int. Conf. on High Perf. Comp. Arch. (HPCA)*, pages 85–96, 2001. 18

[40] C. Saldanha, and M. Lipasti. Power efficient cache coherence. In *Proc. Workshop on Memory Performance Issues*, page 1, 2001. 18

[41] J. Li, J. Martinez, and M. Huang. The thrifty barrier: Energy-aware synchronization in shared-memory multiprocessors. In *Proc. Int. Conf. on High Perf. Comp. Arch. (HPCA)*, page 14, 2005. 18

[42] J. Chen, and L. K. John. Efficient program scheduling for heterogeneous multi-core processors. In *Proc. Annu. Design Automation Conf. (DAC)*, pages 927–930, 2009. 18

[43] M. Annavaram, E. Grochowski, and J. Shen. Mitigating amdahl's law through epi throttling. In *Proc. of the 32nd Ann. Symp. on Comp. Arch.*, pages 298–309, 2005. 18

[44] J. A. Winter, and D. H. Albonesi, and C. A. Shoemaker. Scalable thread scheduling and global power management for heterogeneous many-core architectures. In *Proc. Int. Conf. Parallel Archit. Compilation Tech. (PACT)*, pages 29–40, 2010. 18

[45] A. Bhattacharjee, and M. Martonosi. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. In *Proc. 36th Ann. Intl. Symp. on Comp. Arch. (ISCA)*, pages 290–301, 2009. 18, 19, 23

[46] G. Contreras, and M. Martonosi. Characterizing and improving the performance of intel threading building blocks. In *Proc. IEEE Intl. Symp. on Workload Characterization*, pages 57–66, 2008. 19

[47] K. Singh, M. Bhadauria, and S. A. McKee. Cache performance of operating system and multiprogramming workloads. *ACM SIGARCH Comput. Archit. News*, 37:46–55, 2009. 19

[48] S. Hsin-Ching, S. Bor-Yeh, Y. Wuu, and L. Jenq-Kuen. Migrating java threads with fuzzy control on asymmetric multicore systems for better energy delay product. In *Proc. International Conference on Computing and Security (ICCD)*, pages 1–12, 2011. 19

[49] R. Bertran, M. Gonzalez, X. Martorell, N. Navarro, and E. Ayguade. Decomposable and responsive power models for multicore processors using performance counters. In *Proc. 24th ACM International Conference on Supercomputing (ICS)*, pages 147–158, 2010. 19

[50] M. Y. Lim, A. Porterfield, and R. Fowler. "softpower: fine-grain power estimations using performance counters. In *Proc. 19th ACM International Symposium on High Performance Distributed Computing (HPDC)*, pages 308–311, 2010. 19

[51] R. West, P. Zaroo, C. A. Waldspurger, and X. Zhang. Online cache modeling for commodity multicore processors. *ACM SIGOPS Oper. Syst. Rev.*, 44:19–29, 2010. 19

[52] C. Su, D. Li, D. Nikolopoulos, M. Grove, K. W. Cameron, and B. R. de Supinski. 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)*, pages 19–20, 2011. 19

[53] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova. A case for numa-aware contention management on multicore systems. In *Proc. USENIX Annual Technical Conference (USENIXATC)*, pages 1–15, 2011. 19

[54] S. Blagodurov, S. Zhuravlev, and A. Fedorova. Contention-aware scheduling on multicore systems. *ACM Trans. Comput. Syst.*, 28:1–45, 2010. 19

[55] D. Zaparanuks, M. Jovic, and M. Hauswirth. Accuracy of performance counter measurements. In *Proc. International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 23–32, 2009. 19

[56] S. Eranian. What can performance counters do for memory subsystem analysis? In *Proc. ACM SIGPLAN workshop on Memory Systems Performance and Correctness (MSPC)*, pages 26–30, 200. 19

[57] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proc. 20th Ann. Intl. Symp. on Comp. Arch. (ISCA)*, pages 289–300, 1993. 22, 61

[58] R. Rajwar, and J. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proc. 34nd Annu. IEEE/ACM Int. Symp. Microarchit.*, pages 294–305, 2001. 22, 61

[59] R. Rajwar, and J. Goodman. Transactional lock-free execution of lock-based programs. In *Proc. 10th Int. Conf. Architectural Support Program Languages Operating Syst.*, pages 5–17, 2002. 22

[60] J. F. Martinez, and J. Torrellas. Speculative synchronization: applying thread-level speculation to explicitly parallel applications. In *Proc. 10th Int. Conf. Architectural Support Program Languages Operating Syst.*, pages 18–29, 2002. 22, 61

[61] M. Annavaram, and E. Grochowski, and J. Shen. Mitigating amdahl's law through epi throttling. In *Proc. 32st Annu. Int. Symp. Comput. Archit.*, pages 298–309, 2005. 22

[62] T. Morad, U. C. Weiser, A. Kolodny, M. Valero, and E. Ayguade. Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors. *IEEE Comp. Arch. Letters*, 5:14–17, 2006. 22

[63] T. Morad, A. Kolodny, and U. Weiser. Scheduling multiple multithreaded applications on asymmetric and symmetric chip multiprocessors. In *Proc. 3rd Intl. Symp. on Parallel Arch., Alg. and Prog. (PAAP)*, pages 65–72, 2010. 22, 23

[64] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *Proc. 14th Int. Conf. Architectural Support Program Languages Operating Syst.*, pages 253–264, 2009. 22

[65] M. A. Suleman, M. K. Qureshi, Khubaib, and Y. N. Patt. Feedback-directed pipeline parallelism. In *Proc. 19th Int. Conf. Parallel Archit. Compilation Tech. (PACT)*, pages 147–156, 2010. 22

[66] Q. Cai, J. Gonzalez, R. Rakvic, G. Magklis, P. Chaparro, and A. Gonzalez. Meeting points: using thread criticality to adapt multicore hardware to parallel regions. In *Proc. Int. Conf. Parallel Archit. Compilation Tech. (PACT)*, pages 240–249, 2008. 23

[67] N. B. Lakshminarayana, J. Lee, and H. Kim. Age based scheduling for asymmetric multiprocessors. In *Proc. IInt. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, page 25, 2009. 23

[68] J. Henning, Sun Microsystem. Spec cpu2006 benchmark descriptions. In *Proc. of of the ACM SIGARCH Computer Arch. News*, pages 1–17, 2006. 26, 49, 84

[69] C.K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin:building customized program analysis tools with dynamic instrumentation. In *Proc. ACM SIPLAN Conf. on Prog. Lang. Design and Impl.*, pages 190–200, 2005. 26

[70] D. Bailey. Ffts in external or hierarchical memory. *Journal of Supercomputing*, 4:23–35, 1990. 30

[71] J. Singh, C. Holt, J. Hennessy, and A. Gupta. A parallel adaptive fast multipole method. In *Proc. Conf. on Supercomputing*, pages 54–65, 1993. 30

[72] P. Hanmhan, D. Satzmarr, and L. Aupperle. A rapid hierarchical radiosity algorithm. In *Proc. 18th Ann. Conf. on Comp. Graph. and Iter. Tech. (SIGGRAPH)*, pages 197–206, 1991. 31

[73] G. Blelloch, C. Leiserson, B. Maggs, C. Greg Plaxton, S. Smith, and M. Zagha . A comparison of sorting algorithms for the connection machine cm-2. In *Proc. Symp. on Parallel Alg. and Arch.*, pages 3–16, 1991. 31

[74] J. Pal Singh, A. Gupta, and M. Levoy. Parallel visualization algorithms: Performance and architectural implications. *IEEE Computer*, 27:45–55, 1994. 31

[75] T. E. Carlson, and W. Heirman, and L. Eeckhout. 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)*, pages 1–12, 2011. 32, 34, 35, 49, 67, 85

[76] D. Genbrugge, S. Eyerman and L. Eeckhout . Interval simulation: Raising the level of abstraction in architectural simulation. In *Proc. 16th Int. Symp. High Perform. Comput. Arch.*, pages 1–12, 2010. 32, 34

[77] Sniper. The sniper multi-core simulator, 2015. [Online] Available: `http://snipersim.org`. 34

[78] Intel. Intel® 64 and ia-32 architectures developer's manual, 2015. [Online] Available: `http://www.intel.com/content/www/us/en/architecture-and-technology/`

64-ia-32-architectures-software-developer-manual-325462.html. 37, 48, 64, 83

[79] Intel Corporation. http://www.intel.com. 40

[80] Intel Corporation. Intel smart cache, 2012. [Online] Available: https://software.intel.com/en-us/articles/software-techniques-for-shared-cache-multi-core-systems/?wapkw=smart+cache. 41

[81] Intel Corporation J. Doweck. Inside intel® core tm microarchitecture and smart memory access, 2011. [Online] Available: web.archive.org/web/20111229193036/http://software.intel.com/file/18374/. 41

[82] NVIDIA. Tegra 3 (kal-el) quad-core mobile processor, 2011. [Online] Available: http://www.nvidia.com/object/tegra-3-processor.html. 46, 70

[83] S. Woo, and M. Ohara, and E. Torrie, and J. P. Singh, and A. Gupt. The splash-2 programs: Characterization and methodological considerations. In *Proc. 22nd Annu. Symp. Comput. Archit.*, pages 24–36, 1995. 49, 66, 84

[84] L. Rudolph and Z. Segall. Dynamic decentralized cache schemes for mimd parallel processors. In *Proc. 20th Ann. Intl. Symp. on Comp. Arch. (ISCA)*, pages 340–347, 1984. 61

[85] N. Barrow-Williams, C. Fensch, and S.Moore. A communication characterisation of splash-2 and parsec. In *Proc. IEEE Int. Symp. on WL. Char.*, pages 86–97, 2009. 69

[86] K. Du Bois, S. Eyerman, J. Sartor, and L. Eeckhout. Criticality stacks: Identifying critical threads in parallel programs using synchronization behavior. In *Proc. 40th Annu. Int. Symp. Comput. Archit.*, pages 511–522, 2013. 78, 80, 81, 87

[87] E. J. Fluhr, and et. al. Power8tm: A 12-core server-class processor in 22nm soi with 7.6tbs off-chip bandwidth. In *Proc. IEEE Int. Solid-State Circuits Conf. Digest of Technical Papers (ISSCC)*, pages 96–97, 2014. 93