UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

EXPLOITING HETEROGENEOUS MULTICORE PROCESSORS THROUGH FINE-GRAINED SCHEDULING AND LOW-OVERHEAD THREAD MIGRATION

A DISSERTATION

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

DOCTOR OF PHILOSOPHY

By

LINA HAKAM SAWALHA
Norman, Oklahoma
2012

EXPLOITING HETEROGENEOUS MULTICORE PROCESSORS THROUGH FINE-GRAINED SCHEDULING AND LOW-OVERHEAD THREAD MIGRATION

A DISSERTATION APPROVED FOR THE SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

BY

Dr. Ronald D. Barnes, Chair

Dr. Monte P. Tull, Co-Chair

Dr. Joseph P. Havlicek

Dr. James J. Sluss

Dr. Jon G. Bredeson

Dr. John K. Antonio

To My Parents

For their support, care and love

Acknowledgements

I wish to thank my advisor, Dr. Ronald D. Barnes, for his help, advice, encouragement and support he provided during my PhD studies. His insight and direction enabled me to produce quality research. I would also like to thank my co-advisor, Dr. Monte P. Tull, for his help, support, encouragement and valuable discussions. Moreover, I would like to thank my PhD committee members Dr. Joseph Havlicek, Dr. James Sluss, Dr. Jon Bredeson and Dr. John Antonio for their support and advice.

I would like to thank Xiaolei Yan and Dr. Amy McGovern for their work on the reinforcement-learning-based scheduler. I also wish to thank in particular the Soonergy Lab team, Sonya Wolff, Nick Freeman, Frederic Bard and Dustin Northup for their work on the Soonergy Framework used throughout this work.

I also would like to thank my parents for their ultimate support, endless love, valuable advice and encouragement. Without their support and encouragement I would not be able to finish my PhD studies. Furthermore, I thank my siblings, family and friends for being there for me and for their support not only for my graduate studies but in all other aspects of my life as well.

Finally, I would like to thank the school of electrical and computer engineering for the teaching assistantship and the support of the National Science Foundation (Grant No. 1018771).

Table of Contents

List of Tables

List of Figures

Abstract

EXPLOITING HETEROGENEOUS MULTICORE PROCESSORS THROUGH FINE-GRAINED SCHEDULING AND LOW-OVERHEAD THREAD MIGRATION

Lina Hakam Sawalha, PhD
The University of Oklahoma, 2012

Supervisors: Ronald D. Barnes
Monte P. Tull

Heterogeneous (also known as asymmetric) multicore processors (HMPs) offer significant advantages over homogeneous multicores in terms of both power and performance. Power-efficient cores can be paired with higher-performance cores to achieve advantageous power/performance tradeoffs. Particular cores could also be tailored to efficiently meet the demands of particular application domains. Unfortunately, HMPs also create unique challenges in effective mapping of running processes to cores. The greater the diversity of cores, the more complex this problem becomes. Existing dynamic scheduling approaches for HMPs fall into two categories: sampling and prediction. Sampling approaches permute running applications to across core types to find the best-performing assignment. This sampling step hurts performance and power due to time spent migrating threads through non-optimal assignments. Alternatively, prediction-based approaches estimate the performance for each application on different types of cores and choose the schedule with the best estimated performance. Prediction eliminates the cost of sampling but may result in sub-optimal scheduling decisions. This dissertation introduces new and novel phase-identification-based online schedulers for HMPs that combine aspects of both sampling and prediction approaches by *identifying* phases of execution (instruction sequences with similar behavior), sampling new phases, *recognizing* repeating phases and *reusing* recorded phase information to *predict* the best performing schedule and optimize the schedule for either performance or power consumption. While previous approaches utilized only phase-change *detection* to begin evaluating new schedules, the proposed approaches recognize the current phase of each executing thread and reuse phase information recorded in a Signature History Table when the same or similar behavior of programs reoccurs. This dissertation further proposes machine-learning based schedulers that learns effective scheduling policies using the same characteristics of these program phases.

Exploiting differences between relatively short duration phases using the presented scheduling techniques results in frequent thread migrations that can harm performance. Operating system (OS) context switching can be time consuming. To reduce this context switching overhead, a context switching circuit that both accelerates thread switches among cores in HMPs and reduces switching cost within each core (multitasking) is further introduced in this work. This novel context switch circuit enables low-overhead hardware-level thread migration between cores on a chip and results in up to1380X speedup as compared to an OS context switch. Together with the presented scheduling approaches, this mechanism enables efficient and fine-grained scheduling for HMPs.

Chapter 1

Introduction

As microelectronic technology has advanced, transistor sizes have become smaller, and the number of transistors on a chip have increased. This allows computer architects to place more hardware logic on the same chip area. Architects improved processor performance by building deeper pipelines to increase processor frequency, building larger components such as caches and branch predictors to increase the number of instructions per cycle, and exploiting instruction level parallelism (ILP) to increase the number of executed instructions per cycle. Recently, power constraints have limited these increases in frequency to improve performance. Increasing performance by only increasing frequency is infeasible due to the amount of power consumed, heat generated and cooling required to prevent the processor from damage. Achieving higher performance through exploiting ILP almost reached the limit because it depends on the number of independent instructions that can execute simultaneously for each thread or application. With the contemporary limits on higher frequency processors and the small room for improvement that can be achieved by exploiting ILP, computer architects have moved to implementing increasing numbers of processing cores on the same chip to achieve improved system performance. A multicore processor implemented on a single silicon die is also known as chip multiprocessor (CMP). CMPs allow users to run multiple applications in parallel. They also allow multithreaded applications, however, they do nothing to accelerate the sequential programs or sequential regions of programs.

Most of the currently available CMPs are homogeneous; each processor core is identical to other cores on a chip. Homogeneous multicore processors can be targeted for either single-thread efficiency, or thread-level parallelism (TLP), or specific domain applications. However, future high-performance processors are expected to be heterogeneous with larger number of cores than those currently available [25]. Heterogeneous multicore processors (HMPs) are CMPs that combine different types of processing cores on the same die area. While homogeneous CMPs are easier to design and verify, heterogeneous architectures can be exploited for power efficiency and targeted for performance or domain specific applications. However, HMPs create new challenges for designers and/or programmers in mapping applications to the different types of cores. Because user applications are heterogeneous in natures, they differ in the amount of resources they require, power they consume and performance they result in, heterogeneous multicore processor serves those applications' needs more efficient than homogeneous processors. Moreover, those applications may change their behavior over time. A static assignment that maps applications to core before hand does not adapt to the dynamic workload changes and can result in a decreased performance and increased power consumption. Thus, a dynamic scheduling mechanism is required to adapt to the dynamically changing workloads, and improve system performance and/or reduce power consumption.

This dissertation demonstrates that fine-grained online scheduling techniques with fast thread migration mechanisms can maximize performance and minimize energy consumption for HMPs. The proposed scheduling techniques identify changes in applications behavior to re-evaluate the current schedule. The techniques combine both sampling and prediction approaches to adapt to dynamic changes in workload behaviors and find the best map of threads to the dissimilar cores in HMPs. While frequent sampling (permuting applications on the different types of cores to find the best thread-to-core map) hurts performance, prediction (anticipating the performance for different thread-to-core assignments) may not produce accurate results and may hurt performance. Combining both sampling and prediction approaches can result in better throughput through performance sampling when a new behavior of a thread is encountered and predicting system's throughput of different possible assignments for new and repeating phases. However, fine-granularity scheduling results in frequent thread migrations, which causes performance degradation when performed in software. This dissertation further proposes hardware/software cooperative techniques that reduces the cost of thread migrations.

Processing cores in an HMP contain a mix of two or more general purpose processors, special purpose processors, accelerators and/or programmable logic such as field programmable gate arrays (FPGAs). All cores can either share the same instruction set architecture (ISA), execute subsets of an ISA, or each core (or a set of cores) can execute a different ISA. This work utilizes single-ISA HMPs where all cores execute the same ISA, but can be applied for other types of HMPs as well. Single-ISA heterogeneous systems lack the specialization of the instruction set that could be found in a general heterogeneous system, but maximize the flexibility in scheduling or mapping computation to processors. In such systems, any processing core may run any application thread. However, not all cores provide equal levels of performance or efficiency.

1.1 Motivation and Challenges for Heterogeneous Multicore Processors

According to Moore's Law [51, 68], the number of transistors on an integrated circuit doubles every two years as the size of transistors becomes smaller. Exploiting this, computer architects make use of the extra transistors on a chip to improve systems' throughput by: increasing the frequency(by exploiting deeper pipelines), improving the number of instructions executed per cycle (through enhanced branch predictors, caches, instruction dispatch and the number of execution units) and building more processing cores on a chip. With multiple cores, users can run more applications or threads in parallel, which leads to an increased system's performance. HMPs can further increase the performance of a system by serving a wider range of applications more efficiently than homogeneous CMPs. Heterogeneous multicore processors can efficiently execute both single-threaded applications and parallel (TLP) applications. HMPs can be exploited for both performance and power. Some processing cores on an HMP can be built to target some domain specific applications.

The number of applications that support multiple threads is increasing. Thus, the ability to handle multiple parallel threads is crucial for high performance processors. Designers aim for a large number of cores to support the execution of a large number of parallel threads. Numerous simple, power-efficient cores are more desirable than a few high performance cores in this case. While executing multiple parallel threads is crucial for parallelizable program pieces, handling the serial part of program is vital for the overall system performance [32]. Amdal's Law [3, 32] states that the speedup of a program using multiple processors in parallel is limited by the time needed for the execution of sequential fractions of the program. Hence, a high-performance core is desired for the serial part of a program. For these reasons, a heterogeneous mix of cores are being considered.

With heterogeneous cores, a processor can have more processing cores compared to a homogeneous processor, which contains high single-thread performance cores, occupying the same area [42]. For instance, a small in-order core occupies less area than a big powerful superscalar out-of-order core. Having more processing cores allow the system to execute more threads in parallel and service interrupts simultaneously. In HMPs, dissimilar types of cores result in different performance for any given application. In addition, a typical application's behavior changes over time, thus transitioning through distinct program phases [44]. Each phase of execution has different characteristics that may lead to a different behavior of the application or thread. These characteristics include the type and number of executed instructions, the degree of instruction-level parallelism and the utilization of available resources. Thus, each phase of a thread may result in different throughput. For example, a program phase that contains many memory accesses with last-level-cache misses may perform much better running on an out-of-order processor core than on an in-order core, due to the out-of-order core's ability to better tolerate long, variable latencies.

1.1.1 Challenges of Designing Heterogeneous Multicore Processors

In addition to the increased core design and verification complexity, there are several challenges in designing HMPs–based system. Some of the challenges include: efficiently scheduling threads to the different processing cores, switching threads among cores, memory design and managing core interconnection overheads.

Scheduling Tasks on Heterogeneous Multicore Processors

Effective scheduling of threads on HMPs can be a more difficult problem than scheduling for homogeneous multicores, because of the difference between the processing cores. These differences can include the microarchitecture, numbers and types of execution units, sizes of execution units, sizes of local caches, types and sizes of branch predictors, ISA, etc. An efficient scheduling algorithm maps threads to cores by taking into consideration the characteristics, behavior and relative performance of applications while executing on cores of the various types, and matching them to the relevant characteristics of the cores.

Moreover, applications change their behavior over time, such that some execution phases may have different behavior characteristics and performs better on a different core type. A thread might achieve the best performance when running on one core for some phases, and running on another core for some other phases. While static scheduling approaches (offline thread-to-core assignments) provide a fixed assignment of applications to cores before execution and avoid thread migrations, dynamic schedulers are capable of reassigning threads online to adapt to the dynamic changes in workload behavior. A dynamic scheduler can benefit from the heterogeneity in the system to increase applications performance and/or reduce their energy consumption.

Thread Migration

The operating system (OS) typically is responsible for switching the contexts of running threads. With time-multiplexing between threads, the CPU saves the context of the old thread and launches another thread. Similarly, when a thread requests an IO (e.g. data from disk), the CPU does not wait for the request to be served (reading to finish). Instead, it switches to another thread, and when the first thread finishes reading, the CPU is interrupted with the result of the read. In HMPs, the same need for operating system context switches obviously exists. However unless a single, fixed core assignment is used, the CPU needs to be able to switch a thread among its cores depending on the thread's relative characteristics and behavior in the current phase of execution. The overhead and extensive computations associated with context switching in software limits the number of switches per second for a CPU. Thus, enabling more frequent or fine-grained thread reassignments in HMPs requires a faster thread migration mechanism among cores.

1.2 Dissertation Motivation

As an example of the varying behavior of a program, figure 1.1 shows the number of executed instructions per cycle (IPC) over a short interval of 10,000 instructions of an application, *bzip2*, run on both in-order and out-of-order cores. Figure 1.1 demonstrates that the behavior of *bzip2* varies over relatively short intervals of execution. For most intervals, the performance of the application running on the out-of-order core is better than the performance of the same instruction interval running on the in-order core. This is not surprising as the out-of-order processor is able to dynamically exploit available instruction-level and memory-level parallelism by selecting instruction execution in an order other than program order. However, the opposite is true for other intervals. As can be more clearly seen in Figure 1.2, the in-order core actually outperforms the out-of-order core for some intervals, due to its shorter pipeline, larger cache and increased instruction issue width.

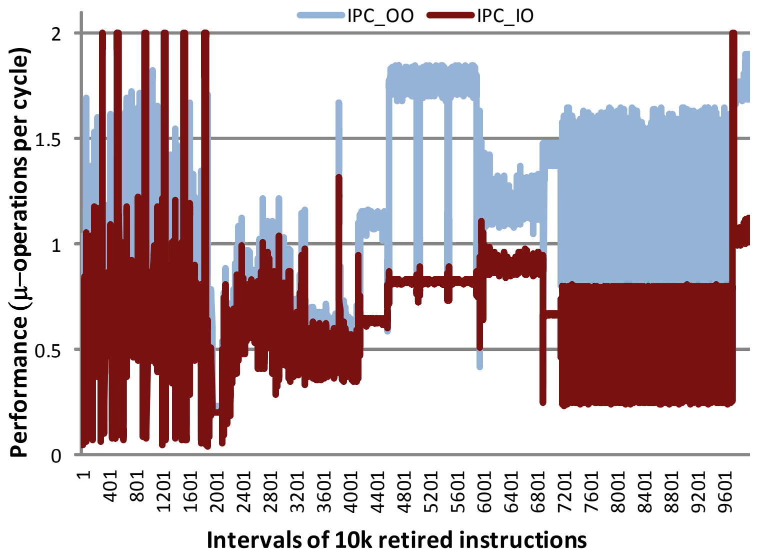


Figure 1.1: IPC of an in-order core and an out-of-order core, for *bzip2* application.

In general, to maximize throughput by benefiting from the heterogeneity in a system, an effective on-line scheduling technique that reassigns threads-to-cores dynamically is required to adapt to the dynamic changes in workload behavior [66, 67]. The dynamic scheduler is responsible for evaluating thread-to-core assignments over time and changing the schedule to improve performance and/or reduce energy consumption. One way to adapt to the dynamic changes in workload is to detect relatively short changes in program behavior and re-assign jobs to cores on the fly. This work represents an adaptive online scheduler for HMPs, which detects fine-grained changes in programs' behavior

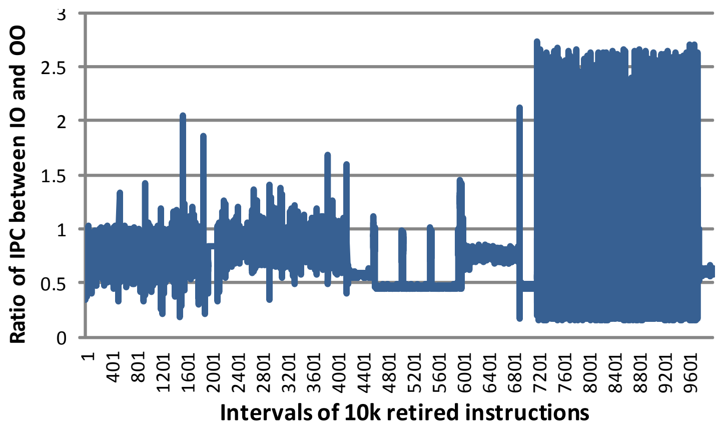


Figure 1.2: The ratio of IPC of an in-order core over an out-of-order core, for *bzip2* application.

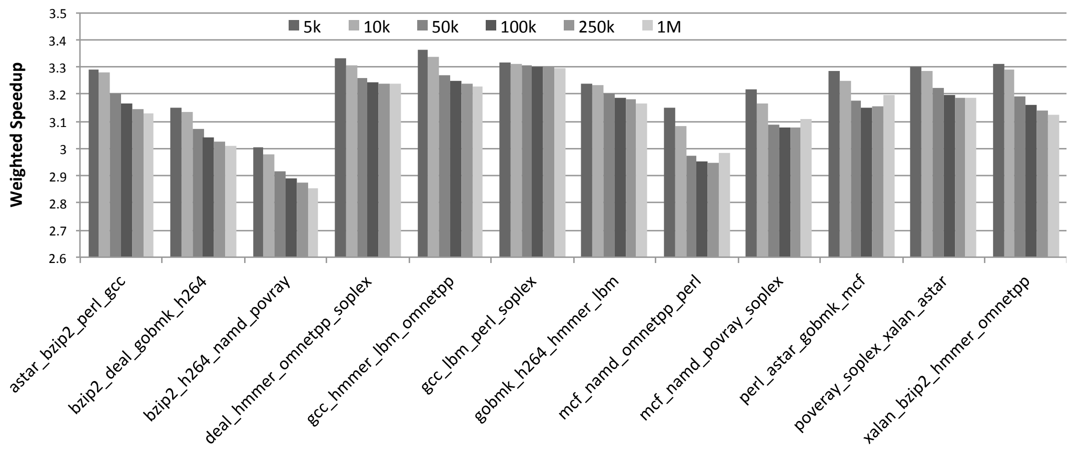


Figure 1.3: The performance of applications on a quad core processor using different scheduling granularity.

and efficiently reassigns threads to the different cores on a chip whenever a change in a program's behavior or a new phase of execution is detected. The scheduler keeps track of active threads behavior (or phases of execution) and their performance or best assignment, recognizes recurring phases and reuses the recorded information to predict the best map of threads to the cores.

Figure 1.3 shows the results of executing four-tuple applications on a quad-core processor with different sizes for windows of executed instructions, over which execution phases are detected. The figure demonstrates that, with ignoring context switching overhead, exploiting shorter changes in program behavior can result in an improved performance. However, fine-grained scheduling yields a large number of thread switches among the heterogeneous cores. Frequent thread migrations hurt performance. Thus, to support fine-grained scheduling and to overcome the overhead of thread migration, a faster context switching is required.

1.3 Dissertation Contributions

This dissertation makes the following contributions:

- It analyzes the different design options of HMPs as well as industry trends for building HMPs, and it proposes different designs for single-ISA HMPs.
- It proposes online scheduling algorithms that adapt to fine-grained changes in programs' behavior to benefit from the asymmetry in the design to maximize performance and minimize energy consumption. The scheduling algorithms combine both sampling and prediction approaches to produce more accurate decisions for both performance and energy consumption.
- It introduces a machine learning technique to make the schedule intelligently learn when to switch to a different assignment to maximize performance.
- It demonstrates that the finer the granularity of scheduling, the higher the performance of HMPs can be–ignoring the cost of context switching.

It also analyzes both the direct cost and indirect cost of thread migrations between the cores on a chip.

- It proposes a hardware switching circuit that drastically reduces the direct cost of context switching, the cost of copying the processor state.

1.4 Dissertation Organization

This dissertation is organized as follows: Chapter 2 provides background information and related work. Chapter 3 shows the design, simulation and evaluation parameters used through all the dissertation. Chapter 4 introduces different scheduling approaches for HMPs. Those approaches are applied to scheduling for both performance and energy consumption. The chapter also provides comparisons between my scheduling approaches and other methods. Chapter 5 analyzes the overhead of context switching and provides solutions for the direct cost of thread migration. Chapter 6 proposes a reinforcement learning-based scheduling for many-core processors. Finally, Chapter 7 concludes the dissertation and provides future work.

Chapter 2

Background

In the history of computing, there has been three main computing domains: high-performance computing domain, personal computing domain and embedded computing domain. The high performance computing domain concerns with multi-threaded and multiprogram applications throughput. It serve multiprogram and multithreaded applications through server systems. The personal computing domain concerns with single-application performance. The demand of personal applications performance was at the beginning through personal desktop computers. The embedded computing domain concerns mainly with power-consumption and serving real-time applications. There are several embedded systems that serve the demand of reduced power consumption, such as cell-phones, tablets, aircraft controllers, etc.

Furthermore, there has been many applications and devices that lie in the intersection of some of these domains such as personal laptops. When laptops first came to the market, they were designed to serve both demands of single-application performance and low-power consumption. Recently, there is a convergence in computer architecture such that many applications and devices requires a sufficient service level for the three demands. For example, most of the mobile devices today such as laptops, tablets, notebooks and cell phones contain multi-core processors to serve more than one application and at the same time single-application performance is important. In addition, battery life in mobile devices is a vital issue, thus these processors are designed to consume less power than processor designed for other applications.

2.1 Multicore Processors

The first microprocessor, Intel 4004 [2], was introduced in 1971. It consisted of only 2300 transistors and operated at 784KHz. Since then, the number of transistors that microprocessors' manufacturers are capable of fabricating on a chip has doubled roughly every 18 months, closely following Gordor Moore's famous observation [51]. Computer architects has used the increased number of transistors to improve performance by increasing parallelism through additional components on chip and increasing frequency through ever-deeper pipelines. The frequency of current processors has reached the GHz scale. However, significantly improving the performance of monolithic processors by increasing frequency is no longer an option for a cost-effective design because of power and thermal limitations. The dynamic increasing the frequency of a processor increases the dynamic power dissipation linearly as shown from Equation 2.1 below:

$$P = ACV^2F \tag{2.1}$$

where P is the dynamic dissipated power, V is the voltage, F is the operating frequency and A is the activity factor. However, frequency is also closely related to operating voltage making the relationship super linear. More dissipated power yields more generated heat; this often means that more expensive heat dissipation methods are required. Moreover, improving performance through extracting instruction-level parallelism (ILP) has reached diminishing returns.

Dynamically finding this parallelism requires power-hungry support and requires larger numbers of independent instructions that are often not available in typical programs. In addition, exploiting increased ILP increases the verification complexity and cost. For all these reasons, designers have moved to more power-aware and complexity-aware approaches to computing.

To benefit from the increased number of transistors on a chip to improve performance, architects have added more cores on a chip, which is known as chip multiprocessors (CMPs). CMPs improve the system performance by allowing multiple applications or threads to run on parallel. Instead of looking for concurrency at the instruction-level, multicore processors looks for concurrency at a coarser granularity–thread-level parallelism (TLP). Multi-threaded applications execute threads on different cores and communicate with each other through message passing or shared memory. In addition, parallelizing applications is often a complicated work for some programmers. In a more straightforward fashion, multicore processors can exploit a coarser granularity than TLP at the application level parallelism. Multicore processors can execute more than one application simultaneously, known as multi-programmed workload. Multicore processors can be either homogeneous processors where all cores on a chip are the same, or heterogeneous where some cores on a chip are designed differently than others. Most of the available multicore processors are homogeneous containing multiple symmetric processing cores.

2.1.1 Homogeneous Multicore Processors

Multicore processors can be classified depending on their applications domain, power/performance, memory model and architecture design [10]. With contemporary power limitations, power/performance tradeoffs are becoming the main processor design concern. Although there has been convergence in computer architecture, most multicore processors lie on two points in the power/performance range: energy-efficient processors and high-performance processors.

Energy-Efficient Homogeneous Processors

Typical power-efficient processors are composed of small identical cores, usually with in-order execution model, or small, power-efficient out-of-order execution model. In in-order processors, the pipeline executes instructions in program order. Out-of-order execution model reorder instructions and execute them out of their program order to increase the number of instructions executed per cycle, while preserving the dependency order or between instructions. Power-efficient homogeneous processors can be further classified to either low-latency or high-throughput processors.

For applications where processor's power-consumption is very important such as mobile devices, few power-efficient low latency cores are used. However, in mobile devices, along with power consumption, applications performance is also important. Combining both demands, power-efficient with relatively low-latency processors are desired for real-time and mobile applications. For instance, ARM's cortex A9 MPCore processor can contain up to four symmetric cores on a chip. Each processing core contains an out-of-order eight-stage pipeline [4]. Cortex A9 provides relatively low-latency (or hight-performance) in low-power constraint devices such as smart phones, digital TVs, etc.

The second type of power-efficient homogeneous processors is the high-throughput processors where the system throughput is the main concern. In parallel computing, many processing cores are desired for multithreaded applications to execute many threads simultaneously. Small, power-efficient processing cores occupy less area than larger aggressive cores, hence more cores can be fabricated on a single chip than larger cores. In addition, small power efficient cores consume less power and require less sophisticated cooling than larger aggressive cores. Because of their small area and reduced energy consumption, those power-efficient cores are desired for parallel applications where single-thread throughput is not as important as the whole system throughput. In this case, more executing threads results in an increased throughput even if the single-thread performance is lower than that of a higher performance processor. This type of processor usually contain several small in-order cores to support a large number of threads with low power consumption.

An example of the high-throughput power-efficient processors is Intel's single-chip cloud computer (SCC) [1]. The SCC is a research chip that Intel built to study many-core CPUs. The SCC consists of 48 cores, where each couple of cores form a tile. Each core is based on a simple, in-order processor. The SCC tiles are connected using a 6x4 synchronous mesh fabric [33]. The chip has multiple voltage and frequency domains and can be dynamically targeted for fine-grain power and performance management.

A group of researchers in MIT developed the Raw processor [83]. It is made up of a set of programmable tiles that are connected through a tightly integrated programmable interconnects. Each tile contains an in-order pipeline, and private data and instruction memories. The Raw processor is mainly targeted for parallel and multimedia applications, and it allows for custom operations. This domain-specific application processor supports several multimedia applications or threads at the simultaneously while the over-all system throughput is the main concern.

One of Sun's multicore processors is the UltraSPARC T-1 [39]. This processor consists of eight simple in-order four-way simultaneous multithreaded cores. The T-1 is targeted at multithreaded applications. Thus, up to 32 threads can execute simultaneously on the processor. UltraSPARC T-1 may result in a lower single-thread performance than high-performance processors, but it increases system throughput by supporting the execution of many threads simultaneously. This processor is suitable for servers that does not require huge amount of computations such as web servers.

High-Performance Homogeneous Processors

Unlike the power-efficient processors described above, high-performance multicore processors are typically composed of a few larger aggressive superscaler cores with out-of-order execution model than the power-efficient cores. This type of processors aim for the highest performance of single-thread applications. This type of processor can be targeted for a very high-performance applications.

Intel has recently produced the latest in a series of multicore processors. Core i7 is an Intel's quad core processor with hyper threading (HT) technology, Intel's version of simultaneous multithreading. With HT, two threads can execute simultaneously on each core. The highest end version of the i7 has six cores and is capable of handling 12 threads simultaneously. Because Core i7 is designed for servers and desktops, it results in high performance for individual threads. However, it contains fewer cores and less multithreading support, and is much less power efficient than small power-efficient cores such as the UltraSpace T-1 and T-2.

2.1.2 Heterogeneous Multicore Processors

A heterogeneous multicore architecture is one system design approach to meeting user's expectations of increased software capabilities and performance even in the presence of tight power constraints. By balancing specialization, single-thread performance and efficient parallel multiprocessing, such systems have the potential to outperform homogeneous architectures while at the same time providing a more power-efficient solution.

When designing a heterogeneous multiprocessor system, the choice of the different sets of processor cores is crucial for power, performance and programmability. Some processor cores may produce higher single-thread performance but require significantly more power or greater chip area. Cores may have different number of execution units, support varying degrees of out-of-order execution, have a different type/size of branch predictors, and feature a different size of private caches. The power consumption and die area requirements of each core play important roles in choosing the cores in a heterogeneous multicore processor.

Heterogeneous multicore systems can either be composed of processing cores that execute specialized instruction sets for a particular domain, or they can execute the same instruction set but feature heterogeneity in the types of characteristics mentioned above. Today, most heterogeneous systems follow the former approach. For example, IBM's Cell Broadband Engine [31] is a heterogeneous multicore processor targeted for specific applications related to streaming media and similar scientific applications. The Cell processor is composed of one Power Processing Element (PPE), and eight Synergetic Processing Elements (SPEs). The PPE is a two-way multithreaded general purpose core, which handles most of the control and coordination and acts as the controller for the eight SPEs. The PPE is similar to a 64-bit PowerPC (PPC) RISC processor. The SPEs are single-instruction multiple-data (SIMD) vector processors with instruction sets focused on SIMD vector instructions, similar to the SIMD vector instructions on the PowerPC. By combining one general purpose core with 8 small but computationally powerful cores, the Cell is an efficient and high performance processor for the multimedia application domain. Similarly, multicore architectures [13] that combine general purpose processing units with specialized graphics processing units (GPUs) offer the exciting promise of exploiting GPU SIMD hardware for accelerating computation.

However, partitioning computation between disparate cores with different ISAs is a challenging task, and one that must be done at software development level. Performing this partitioning automatically can be extremely difficult. In one approach to partitioning Cell applications, Blagojevic et al. [9] introduced a model of multigrain parallelism (MMGD) for parallelizing tasks on heterogeneous parallel architectures. They utilized a phased hierarchical task graph (HTG) to partition applications into multiple phases of execution and split these phases into nested sub-phases and evaluated their technique on a Cell Broadband Engine consisting of two PPEs (host processor unit) and 16 SPEs (accelerator processor unit). While approaches like this one for mapping application code to heterogeneous architectures are promising, multi-ISA heterogeneous architectures make it difficult, if not impossible, to dynamically partition application threads using run-time information.

Recently, ARM introduced a heterogeneous multicore processor named big.LITTLE [30]. This processor combines both a large superscalar processor (ARM Cortex-A15) for high performance, and a small in-order processor (Cortex-A7) for energy efficiency that both execute the same instruction set architecture. Two different implementations for this big.LITTLE processor exists. In one implementation, only one core is powered on at a time and the other is turned off. For this implementation, applications switch transparently between the two cores; they switch to Coretex-A7 to reduce power consumption, or to Cortex-A15 for increased performance. In the second implementation, both cores are switched on if there are more than one application running. Applications are statically mapped to their best fit core beforehand. While static mapping can prevent thread switching between the two cores, my dynamic scheduling techniques with the low-switching overhead described in this dissertation would work efficiently for this big.LITTLE processor and adapt to the dynamic changes of applications' behaviors.

Pericas et al. [57] proposed a flexible heterogeneous multicore processor (FMC). The FMC allows changing instruction window size at runtime based on an execution locality concept. Rather than allocating a single core for each thread, FMC allows threads to use as many resources as they need from a pool of available cores.

2.2 Scheduling Applications to Cores

Heterogeneity significantly increases the complexity of scheduling the different types of applications on the dissimilar cores. Additionally, applications changes their behavior over time and adapting to these dynamic changes in applications' behaviors makes scheduling even more complicated. There has been several work on scheduling for heterogeneous systems, such as heterogeneous multiprocessor systems [29, 55] and distributed systems [5, 6]. However, scheduling for a single-chip heterogeneous multicore processor is different than these larger systems because multiprocessors and distributed systems consist of physically separated node locations. The time that is taken to migrate a task from one processor node to another differs widely depending on the distance between the two nodes. Distributed scheduling algorithms need also to account for recovery from node failures and relocation.

To adapt to the dynamic workload and the different execution phases of applications within a heterogeneous multicore processor requires dynamic reassignment of threads or applications to cores. There are many previous works that present scheduling algorithms for heterogeneous multicore processors. Each of these scheduling techniques can be classified as: static (off-line) scheduling, static/dynamic scheduling and dynamic (on-line) scheduling techniques.

2.2.1 Static (Off-line) Scheduling Techniques

In static scheduling approaches the processing core to which each application will be mapped is decided prior to execution based on certain characteristics of applications. Most HMP off-line schedulers rely on previous analysis of benchmarks characteristics. In one such approach, after off-line profiling of microarchitectural independent characteristics, a signature is composed for each application representing the resources required by that application [69]. These architectural signatures are inserted into the executable binaries of applications' headers. The heterogeneity-aware signature-supported (HASS) scheduler uses those signatures to estimate the performance of entire applications on the different core types within an HMP, and maps applications to their highest-performing core. This approach was evaluated on symmetric cores operating under different frequencies using dynamic voltage frequency scaling (DVFS), and did not consider different types of architectures.

Chen and John [16] proposed another static scheduling algorithm that matches cores to programs. Similar to HASS, the scheduling software analyzes program characteristics and the hardware configurations of each core. The scheduler then matches runtime programs to cores depending on the characteristics of the program, resource demands, and the physical characteristics of cores. This algorithm looks at the inter-program diversity and does not adapt to dynamic changes in workloads.

2.2.2 Static/Dynamic Scheduling Techniques

Static/dynamic schedulers combine both off-line analysis of applications and on-line rescheduling of jobs to the processing cores. In one static/dynamic approach, an offline program phase detection and marking were employed in HMP scheduling [74]. Phase marker code fragments were instrumented at statically detected phase transition points in application executable binaries. These markers perform dynamic performance analysis of these phases and handle threads' reassignment. Similarly, Cong and Yuan [20] uses static analysis to determine loops and function boundaries to determine program phases, and optimizes the scheduler assignments for reduced energy-delay product (EDP).

Instrumentation functions are inserted in the executable binaries of programs at the boundaries of loops and functions. The instrumentation functions are responsible for measuring the call time and number of instructions for each loop and function call. A call graph is constructed from the detected loops and functions, and then major program phases are identified. To overcome the loss of performance due to thread migrations from one core to another, thresholds are used for the number of instructions and invocations. At runtime, when an instrumented code is reached, the EDP is predicted on both core types using regression model, and the scheduler predicts the lowest EDP schedule.

The aforementioned approaches require offline analysis of program behavior. While the scheduling approaches presented in this dissertation exploit the phase behavior of programs, they utilize a dynamic detection and identification of program phases that does not require special modification of code to be executed. Rather than relying on application code to make scheduling decisions as in [74], these techniques perform scheduling on a system-wide basis in a way that is transparent to the running applications.

Chen and John [15] proposed an energy-efficient scheduling mechanism for heterogeneous multicore processors. Instruction-level parallelism, branch-transition rate and data dependency distance characteristics are chosen to measure the suitability of cores issue width, branch predictor size and L1 cache size respectively. Programs are analyzed off-line for these three characteristic. Fuzzy logic was used to combine the individual suitabilities, determine an overall suitability that indicates a degree to find the best program-to-core map. This approach requires previous analysis of programs and is not adaptable to new applications.

2.2.3 Dynamic (On-line) Scheduling Techniques

Dynamic schedulers are capable of changing job-to-core assignments on the fly without the need for off-line analysis or profiling. In this way, they can adapt to dynamic changes in program behaviors without prior profiling or analysis of applications. In a heterogeneous multicore processors, dynamic schedulers have increased potential to improve the system throughput through on-line rescheduling by adapting to both changes in programs behavior and need of resources. Dynamic scheduling techniques can be classified as: sampling, prediction and sampling/prediction techniques.

Sampling-Based Scheduling Techniques

Kumar et al. [42] designed a sampling-based heuristic scheduling approach for assigning jobs to cores dynamically. Like this work, they focused on single-ISA heterogeneous multicore architectures in which cores can vary in performance and power consumption but not in ISA. Their scheduling algorithm is divided into two different phases, a sampling step and a steady step. In the sampling step, all of the possible assignments of jobs to cores are examined and a weighted speedup is recorded for each assignment. In their evaluation, each assignment is run for two million cycles. Then the assignment with the best weighted speedup was chosen as the schedule for the steady step. Calculating the weighted speedup for this method requires before-hand knowledge of the performance of each application executed along on the system. Becchi and Crowley [7] proposed a similar approach called IPC-driven, in which an IPC ratio between a fast core and a slow core is presented as a more practical evaluation criteria. In either approach, whenever an execution phase change is detected or a certain number of cycles elapsed, the dynamic scheduler begins a new sampling phase. Note, that while both studies utilize a simple mechanism to detect when a phase change occurs, they do not seek to identify the particular phases. This distinction with the study in this dissertation is further clarified in Chapter 4.

A limitation of Kumar's scheduling algorithm [42] is its inefficiency in estimating the performance of particular mappings. The scheduler must examine all the possible assignments before it chooses the best assignment and enters the steady step . For a small number of cores and core types, this may be feasible but the number of possible mappings grows factorially with the number of unique core types. It also grows linearly with the number of total cores for fixed number of asymmetric core types. Thus, with a larger number of permutations only some of the possible assignments can be attempted. Furthermore, a significant amount of time is spent in the sampling phase, most of which is spent with a sub-optimal thread-to-core assignment. This can significantly hurt performance. The sampling approaches use coarse grain changes in program behavior, however, performance sampling can happen during fine grain changes in program behavior or during different sampling intervals, thereby misleading the scheduler decision. For instance, if a thread transits through a short phase that contains several last-level cache misses during one sampling interval and then for the next sampling interval the application transits through a phase with high instruction-level parallelism (ILP) the performance is different for both phases even on the same core types. This may cause the scheduler to choose a suboptimal assignment.

A scheduling algorithm called hierarchical hungarian was proposed for many-core HMPs is proposed [85]. The cores in a processor are divided into clusters and the hungarian algorithm is applied for each cluster. Sampling is used to get performance information of each application on the different core types.

Prediction-Based Scheduling Techniques

The prediction-based scheduling approach relies on predicting the performance of threads on different core types, and/or predicting the assignment of threads on the different core types. In a related approach, Jooya et al. [37] introduced the history-aware, resource dynamic (HARD) scheduler for heterogeneous chip multiprocessors. The HARD scheduler performs reassignment of jobs to cores when an application phase change occurs by “upgrading” or “downgrading” job assignment to a higher-performance or more power-efficient core, respectively. Like Kumar’s technique, the HARD scheduling approach relies on a change in an application’s performance (in this case throughput and core utilization) to detect a change in a program phase. While HARD avoids permuting applications amongst different types of cores, it relies upon a strict performance ordering for processor core types. This may not always be the case, as some cores may have better performance for certain application domains.

Other scheduling algorithms exploit off-chip performance, such as memory accesses, to detect changes in programs behaviors [40, 60]. Such algorithms do not account for the differences in the architecture such as the execution units, pipeline, etc. [60]. In one approach, Koufaty et al. [40] correlates an application’s behavior with internal (on-chip) and external (off-chip) stalls. They estimate a bias for each application using performance counters that keep track of the external stalls. The algorithm uses overlapping windows of instructions to calculate a running average of each of the metrics. The bias of an application changes if the amount of stalls goes over or below a certain threshold. This algorithm can be implemented with any type of operating system scheduler without changing the properties of the scheduler such as responsiveness and fairness. However, it periodically checks the load balance of the system and migrates the thread with the highest bias to a "big" core. Saez et al modified the HASS scheduler (discussed in Section 2.2.1) to dynamically assign threads to cores by detecting program phases [61]. Similarly, the last-level cache miss rate is used to estimate a speedup factor. Initially, the speedup factor is assigned to a default value and after that it is calculated using the profiled information of last-level cache misses. The scheduler detects coarse-grain program phases and updates the speedup factor before making a decision.

Some scheduling algorithms for HMPs are concerned with multithreaded applications performance [12, 43]. In the age-based scheduling technique [43], the length of threads are predicted, using history of previous instances of code for predicting the next barrier or end of thread. The threads with the longest estimated run time are scheduled to run on the fastest cores. This scheduling technique improves the throughput of parallel applications by overcoming the barrier bottleneck through accelerating the longest thread. However, for single-threaded applications workloads, this approach acts the same as static approaches. Instead of measuring thread length, Poovey et al [12] measured thread complexity through analyzing dependance chain. The scheduler evaluates the current assignment every 100 ms and resets the dependence chain.

A dynamic scheduling approach for assigning threads in an HMP that includes a central processing units (CPUs) and a graphical processing unit (GPU) was performed [46]. In [46], a dynamic compilation of programs to native machine codes is performed at run time to adapt to changes in the environment. The scheduler, Qilin, uses an empirical method to map computations to cores. The first time a program runs on Qilin, it is considered a training run. The input of the program is divided in two parts: one part runs on the GPU and the other part run on the CPUs. The execution-time projections are kept in a database, such that when the same program run again with different input size, the execution-time projection stored in the database are used to determine the mapping of computations to the processing cores. The algorithm attempts to find the fraction of work to run on CPU to minimize execution time. This method focuses on multimedia and parallel applications and assumes that programs are repeated before their information are forced out of the database.

The above prediction-based scheduling techniques rely on performance estimation. Another way to predict threads-to-cores map is to learn over time the best assignment that results in the best throughput when encountering certain features of programs [26, 87].

Prediction/Sampling Based Scheduling Techniques

Prediction/sampling scheduling approach combines both prediction and sampling techniques to improve the performance of the system. While sampling alone hurts performance when performed frequently, prediction may result in sub-optimal assignments. Thus, combining both prediction and sampling improves system throughput, by sampling when new behaviors of programs are detected and predicting when similar behavior are encountered again. This type of scheduling techniques is used in this dissertation for scheduling on heterogeneous multicore processors because of its accuracy and reduced sampling cost. Wu et al. [86] proposes a hardware/software co-designed heterogeneous multicore processor that contains narrow out–of-order (OoO) with x86 ISA and wide in-order (IO) very long instruction window (VLIW) virtual machine. The code is dynamically translated from x86 to the VLIW machine. When the code is running on OoO core, there is no need for dynamic translation and it executes native machine code. Dynamic profiling is performed, and when one or more hot spots are detected the code is dynamically translated and optimized for the IO core. Two predictors are used to predict continuing on the same core and to predict switching to the other core in case of sampling. The first predictor compares data collected on both core types to decide whether to stay on the same core or not. Continuation is allowed for only K continuous times and then sampling is forced. If the scheduler predicts not to continue on the same core, the other core is activated and sampling is performed for a short interval to collect information. Based on the collected information the scheduler decides whether to stay or switch again to the first core.

2.3 Context Switching

Scheduling jobs on heterogeneous multicore processors requires switching among the different cores on chip frequently to benefit from the heterogeneity. Context switching is the process of saving the central processing unit (CPU) state (context) and restoring it when switching to another process. The conventional mechanism for this migration is through software context switching. The context of a CPU consists of a process control block (PCB), which includes the state of the process, CPU registers and information about memory management. Context switching is time consuming and depends on the objects to be switched–register, thread or process switching. It also depends on the size of data to be copied, which in turn depends on the behavior of the running program.

The OS is typically responsible for switching tasks to allow several threads to share the CPU (time-multiplexing) and to be able to switch the CPU to another thread when there is an interrupt or the current thread requests to use an IO device. On the other hand, some processors (including modern Intel x86 processors) have hardware support for context switching, by saving the processor state in a system data structure called the task state segment (TSS). The TSS consists of two types of fields: dynamic fields and static fields. The dynamic fields include the general purpose registers and the segment selector, while the static fields contain local descriptor table (LDT), CR3 control register, stack pointers and I/O map address [35]. The TSS is intended to automate switching between programs. However, it restricts how OS programmers can configure context switching. Instead of taking advantage of the TSS, many x86 OSs use their own context switching mechanism. For instance, Linux does not use the TSS feature. Instead, it simply creates only one TSS for each processor and modifies it for each process. However, Linux does use the static fields of Intel's TSS such as the control register CR3 while switching tasks.

There are two types of context switching overhead: direct cost and indirect cost [45]. The direct cost includes the time copying the context of the

CPU, such as CPU registers and TLB, and flushing the processors pipeline in which the switched thread is running. The direct cost can be measured using Ousterhout's method by forking a child process and sending a message forth and back between the parent and the child processes, using two pipes, periodically [56]. McVoy and Staelin improved Ousterhout's technique by eliminating system call overhead ([22, 48]). The indirect cost comprises the performance degradation of such a system caused by resource sharing. For example, switching a thread between two cores might result in more L1 cache misses and branch miss prediction since the thread was using the previous core's resources, which in turn affects the performance of the system. The indirect cost of context switch resulting from cache performance loss, ranges between several microseconds to few thousands microseconds [45], [22]. Other researchers have also examined the indirect cost of context switching for caches [50, 75, 76], and the cost of context switching for branch predictors [18]. Some hardware efforts to speed up context switch for ARM architecture are: fast address space switching (FASS) [84], and fast context switch extension (FCSE) [14]. Cho et al. [17] proposed a protocol that is ensures a deadlock-free thread switches for fine-grained migration architectures. Chapter 5 will discuss both types of switching overhead and possible solutions in more details.

2.4 Program Phase Detection

Many applications exhibit behavior in which program execution occurs in distinct phases, where each phase consists of a set of code blocks that are executed with a high degree of temporal locality. Many common types of programs exhibit this execution phase behavior [44]. There are several different mechanisms which have been proposed to detect application phase changes. Some of these techniques can be used to identify unique application phases.

In one of the single-ISA heterogeneous scheduling approaches described in Sectiion 2.2, Kumar used a simple mechanism to determine when an application's phase changes by monitoring the instructions executed per cycle (IPC) over some windows. When this value changes by more than 50% for one application or by a total of 100% changes for all applications [42], a phase change is said to have occurred. This approach makes no attempt to identify the individual phases, but instead only detect coarse changes in program behavior. It further lacks the ability to detect more subtle changes in application behavior because it relies on a significant change in the IPC.

More precise phase detection methods rely on statistical sampling of executing instructions. Such samples can be subsequently analyzed in software to determine phase composition. Hardware performance monitoring counters or the program counter can be sampled to gather low-overhead profiles, but such profiles do not support the differentiation of one phase from another. Basic block distribution analysis [70] combines intense, periodic sample-based profiling to determine the composition of repetitive phases.

Special purpose hardware can be used to reduce the overhead of accurate phase detection and identification. The hot spot detector [49] is a hardware mechanism for detecting and identifying program phases based on the address and taken/not-taken direction of retired branch instructions. Working set signatures [23] can be used to provide an efficient, compressed representation of an application's phases, which are composed of windows of retired instructions. Exploiting the correlation between a working set phase and program behavior, this approach was used to control reconfigurable hardware resources, and provides a relatively general mechanism for finding relatively short phases. Sherwood et al. [72] used a similar technique but weighted the profiled code by a phase's frequency of execution. Their architecture provides an accurate way of not only identifying program phases, but also predicting when a phase change will occur and which phase will execute next.

Many performance metrics correlate strongly to application phases, including cache behavior, branch predictor behavior, utilization of core resources and IPC [72]. Any of the mechanisms for identifying application phases could be used with our approach, but for the purposes of this work, I utilize the relatively straightforward working set signatures mechanism [23], which is described in more detail in Chapter 3. Only improved results from using more accurate phase identification techniques are expected.

Chapter 3

Methodology

This chapter describes the experimental setup and evaluation metrics used in this work. It also includes a description and initial evaluation of some parameters of the phase detection method used in this dissertation.

3.1 Simulation Environment

The simulation environment used to evaluate the proposed techniques includes both a cycle-accurate simulator and a simulation model described below.

3.1.1 Cycle-Accurate Simulation Framework

Detailed evaluations of the techniques proposed in this dissertation was performed using a cycle-accurate simulation framework, Soonergy, a microarchitectural simulator developed by the Soonergy Architecture Research Lab at the University of Oklahoma [27]. This simulator provides timing simulations comparable with a physical hardware design because it provides a cycle-by-cycle timing and performance simulation. The simulator is based on the x86 instruction set architecture (ISA), and is developed to simulate different architectural designs (including multicore architectures) for performance and power. The simulator runs on Windows platform systems and allows users to simulate different processor configurations.

Two types of experiments were performed using Soonergy: single-core simulations and multicore simulations. Single-core simulations are used to evaluate the different characteristics of the executed programs and as inputs for the performance estimation model model described in Section 3.1.2. Each single-core/single-benchmark ran for 300 million instructions. The Soonergy simulator profiled performance and microarchitectural behavior. Multicore simulations include heterogeneous mixes of different x86 cores. Different types of cores are distinct in micro-architectural design including pipeline depth and architecture, number and sizes of execution units, different cache sizes, and branch predictors. Dual-core and quad-core simulations were performed. In the presented experiment, the simulator runs all benchmarks on the multicore system simultaneously until the slowest benchmark executes 300 million. Faster benchmarks execute more than 300 million by looping through benchmark's trace files.

3.1.2 Performance Estimation Model

A model was developed to estimate multicore processor performance using single core runs as inputs to the emulator. It specifically emulates different application scheduling techniques for heterogeneous multicore processors (HMPs). Because Soonergy is a detailed cycle accurate simulator, multicore runs are relatively slow. Thus, it is time consuming to run many processor configurations and all sets of benchmarks on Soonergy simulator. Instead a performance estimation model was used to estimate the performance of HMPs using different scheduling algorithms and different phase-detection parameters with all combinations of benchmarks. The model gives a reasonable performance estimate compared with the cycle-accurate simulator, such that initial evaluations for core and parameter choices can be done using the model. Moreover, many more results can be generated for different the configurations and combinations of benchmark inputs.

3.1.3 Processor Configurations

Three different types of x86 cores are used for this study which I refer to as: *Corei7_like*, *Core2duo_like* and *Atom_like*. The cores differ in the issue width, cache size, branch predictors and other characteristics as shown in Table 3.1. All cores share a last-level cache of 6MB for both dual-core and quad-core processors. Single core runs included a last-level cache of 2MB, 1MB and 512KB for the *Corei7_like*, *Core2duo_like* and *Atom_like* cores respectively.

Table 3.1: Processor configurations.

Parameter	Corei7_like	Core2duo_like	Atom_like
Pipeline	Out-of-order	Out-of-order	In-order
Fetch width	6	4	2
Issue width	5	3	2
Number of stages	16	16	18
L1 cache	32KB	32KB	24KB
L2 cache	256KB	-	–
Branch predictor	GShare (32KB) and Bimodal (4KB)	GShare (32KB) and Bimodal (4KB)	GShare (4KB)
Reservation station	36	32	–
Reorder buffer	128	96	–
Integer latency	1	1	1
Floating point	5	4	3
Packed latency	5	4	2
Multiplication/division latency	10	9	7(int), 8(fp)
L1 cache latency	4	2	2

3.1.4 Experimental Workload

A subset of SPEC CPU 2006 benchmark suite were simulated using Soonergy simulator. The chosen benchmarks are the C and C++ written applications that can be compiled with Visual Studio 10. Tables 3.2 shows a description of these benchmarks. Many of the benchmarks have more than reference input such as (*astar* (2), *bzip2* (6), *gcc* (9), *gobmk* (5), *h264* (3), *hmmer* (2), *perl* (3), *soplex* (2)). Each of the reference inputs were considered individually. Each application was executed for 300 million x86 instructions chosen from a statistically relevant portion of the program. The SimPoint tool [71] was used to determine the number of instructions skipped for each benchmark to reach the 300 million instruction point.

Table 3.2: SPEC2006 benchmarks simulated.

Program	Type	Description
astar	Integer	Computer game, artificial intelligence, path finding
bzip2	Integer	Compression program
dealII	Floating Point	Partial differential equations solver
gcc	Integer	C language optimizing compiler
gobmk	Integer	Artificial intelligence, game playing
h264ref	Integer	Video compression
hmmer	Integer	Search engine for a genetic database
lbm	Floating point	Computational Fluid Dynamics, Lattice Boltzmann
mcf	Integer	Combinatorial optimization
namd	Floating point	Scientific, structural biology, classical molecular dynamics simulation
omnetpp	Integer	Discrete event simulation
perlbench	Integer	Programming language
povray	Floating point	Computer visualization
soplex	Floating point	Simplex linear program solver
xalancbmk	Integer	XSLT processor transforming XML documents to html, text or other

Each application was executed on all of the core types described in Table 3.1. The following statistics were collected: the number of instructions executed per cycle (IPC), private caches miss and hit rates, shared-level cache miss and hit rates, branch predictor accuracy and instruction-type mix. Figures 3.1 shows the percentage of all types of instructions for each program.

Figures 3.2 shows the performance of the benchmarks executed on the three types of cores described earlier. In general, the largest superscalar core (*Corei7_like*) outperforms both the other superscalar (*Core2duo_like*) and the in-order (*Atom_like*) cores. However, over short windows of intervals, the smaller cores can result in better performance as illustrated earlier in Section 1.2.

3.1.5 Evaluation Metrics

Choosing an evaluation metric is a critical issue for online schedulers. While instructions executed per cycle (IPC) represents system throughput, using this metric to evaluate different thread-to-core assignments favors high throughput threads/applications over low-throughput ones. A fair metric that does not favor threads over others is desired for improved system performance. Snavely and Tullsen [73] proposed a weighted speedup as an evaluation metric that provides an equal contribution of each thread to the total work. This evaluation metric represents the summation of IPC ratios between each running thread on its current core and the IPC of that thread while running alone on the system (Equation 3.1)).

$$Weightedspeedup = \sum_{i=1}^{n} \frac{IPC_{thread_i-current-core}}{IPC_{thread_i-alone-on-system}}. \tag{3.1}$$

This evaluation metric is used by Kumar et al. [42] in their HMP scheduler to evaluate all possible assignments. However, this evaluation metric requires a previous knowledge of the IPC of each thread after running alone on the system. Such metric is only feasible for online dynamic schedulers when the final results of different scheduling approaches are evaluated and compared with each other. In this study, a similar weighted speedup is used by dividing the IPC of a thread for the current sampling interval over the best IPC across all sampling interval on the different core types as shown in Equation 3.2. This provides an intuition on how threads perform during different phases of

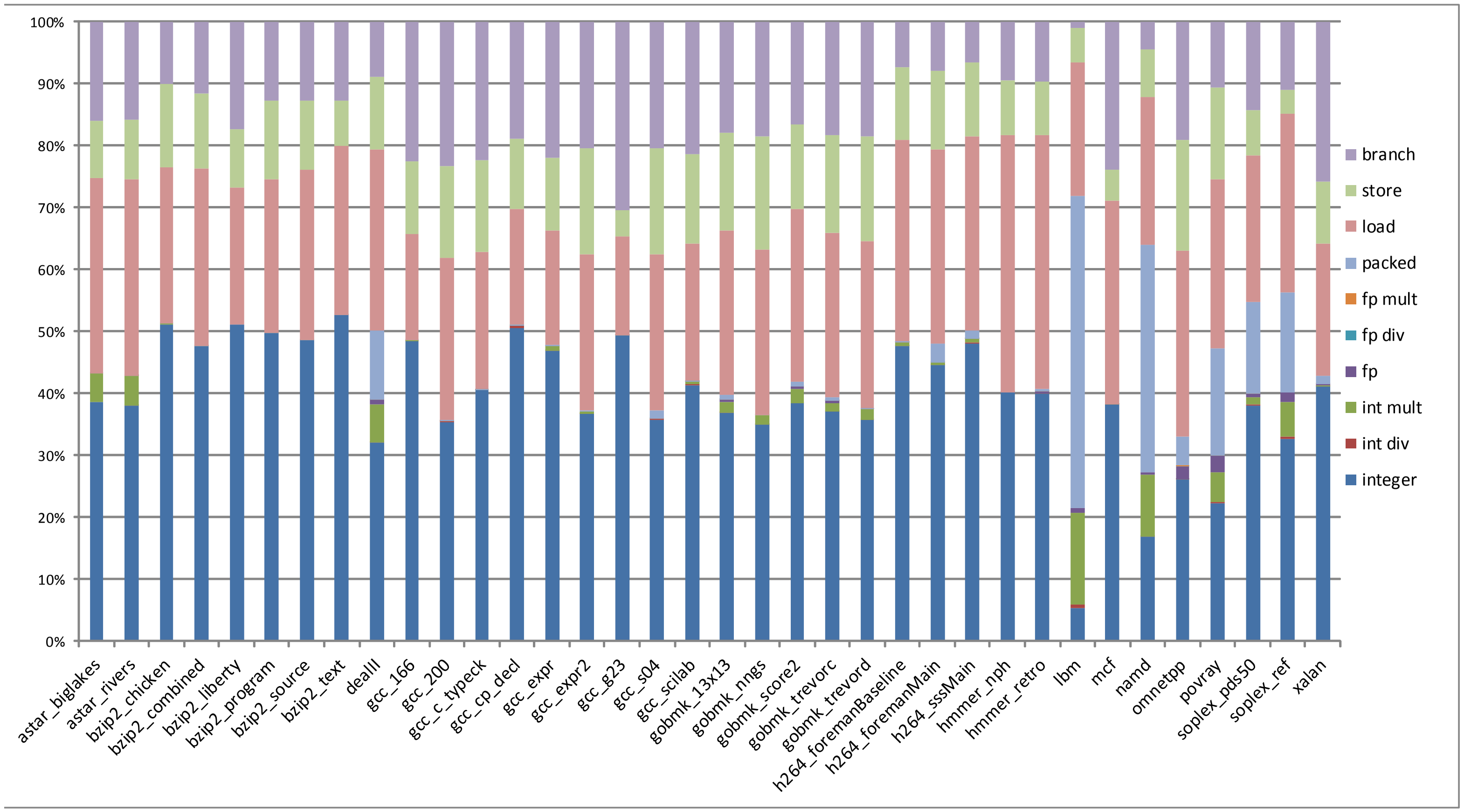


Figure 3.1: Instruction Mix of SPECCPU2006.

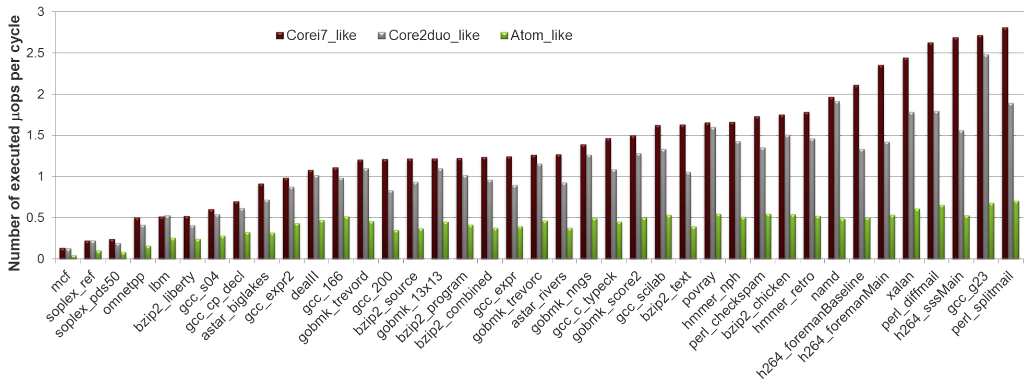


Figure 3.2: A Comparison of the performance on three cores (Corei7_like, Core2duo_like and Atom_like) described in Table 3.1 for SPECCPU2006.

execution. The IPC is calculated online for each thread over program sampling intervals for each sampling occurrence.

$$Weightedspeedup = \sum_{i=1}^{n} \frac{IPC_{thread_i - currentcore}}{IPC_{thread_i - fastestcore}}. \tag{3.2}$$

3.1.6 Program Phase Detection Method

The phase-identification based scheduling mechanisms presented in this work utilize and reuse performance and assignment information for previously encountered phases. A working set signature (WSS) approach was used to detect and identify program phases [23]. A working set signature is a highly compressed representation of a program's working set of retired instructions. The WSS approach uses non-overlapping windows of retired instructions to generate programs signature for each window. In [23], a window size of 100k instructions is used and a signature size of 256 to 1024 bits is proposed to capture reasonably sized phases within feasible hardware requirements. A signature identifies a working set of instructions. During temporally local program segments, these signatures will be similar for several consecutive windows, thus composing a phase. When the same program phase repeats at a later point in time, the same (or very similar) signature indicates a repeated phase. This dissertation proposes methods to exploit fine-grained phase detection and attempt different relatively small window sizes and signatures. The next chapter includes an evaluation of different window sizes and signature sizes for this phase detection method.

The WSS is calculated using a portion of the bits of the program counter hashed into a working set signature vector as shown in Figure 3.3. In the presented evaluations, the least-significant 6-bits of the program counter are excluded, corresponding to the instruction cache block size of 64 bytes. Thus, each instruction in the same cache block will hash to the same bit in the WSS vector. When an instruction that hashes into a particular bit in the current signature vector is executed, this bit is set. That means, the process of computing a WSS is that of setting specific bits in the current signature. The signature is cleared at the beginning of subsequent windows and the next window signature is computed.

A new phase is detected if the signature of the current working set is significantly different from the previous one. The relative distance between the two signatures is computed as the number of bit positions with different values between the two signatures divided by the population count of ones in the union of the two signatures. Because the boundaries of working set signature windows and a program phase do not necessarily match, noise is often observed when comparing two signatures. To account for this noise, a threshold of 50% is used by [23] as well as my work. If the difference between the two signatures

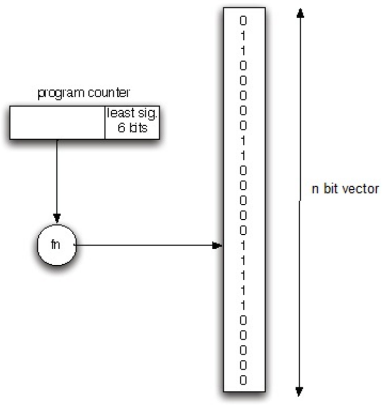


Figure 3.3: Computation of current working set signature.

is less than 0.5, they are identified as the same phase.

To demonstrate that the phases are in fact highly correlated to the performance of applications, several SPEC2006 benchmarks were evaluated on different types of processor cores as described in Section 3.1.3. For each phase in the application, the standard deviation of the IPC was computed across each occurrence of the phase. IPC varies little across different occurrences of the same phase. In particular, for applications that have strong phase behavior for the segment of execution analyzed, IPC varies significantly less during the same phase (including repeated occurrences of that phase) compared to the variation in IPC over the application segment of execution. These results are described in greater detail in Section 4. Since the IPC is relatively stable over the execution of a phase, the IPC seen during the first instance of a phase is a good predictor of the IPC of future occurrences of the same phase.

Figure 3.4 shows an example of applications phases over execution time. The performance of *xalan* changes over time depending on its execution phase. Each phase can consist of different numbers and types of instructions, different

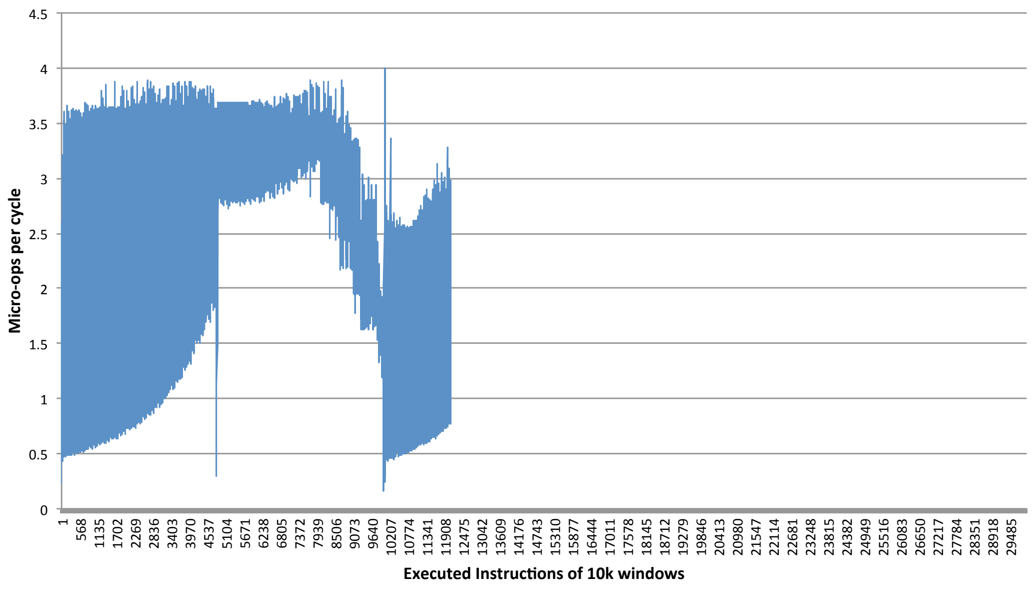


Figure 3.4: Executed micro-operations per cycle over 10 000 instruction windows for a CPU 2006 benchmark *xalan*.

caches miss and branch prediction behavior. The relationship between performance characteristics such as L2 cache Misses is highly correlated to the execution phases. Figure 3.5 shows how the number of L2 cache misses corresponds to the execution phases in Figure 3.4.

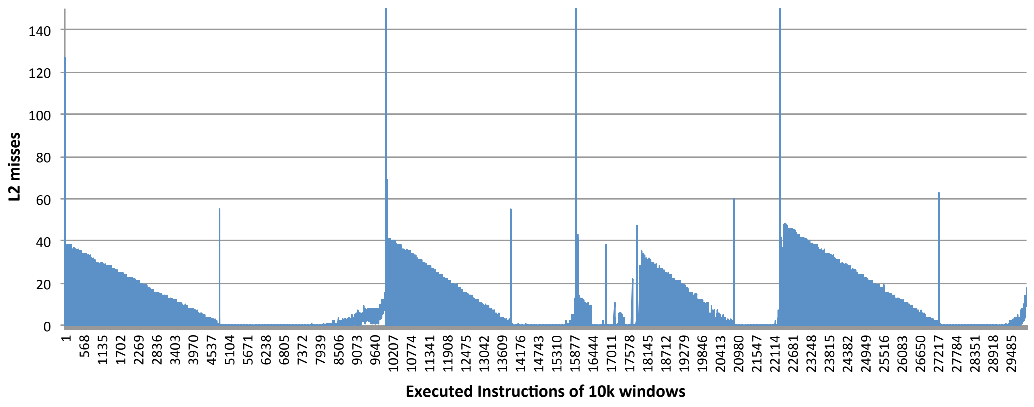


Figure 3.5: The number of L2 cache misses for *xalan*.

Chapter 4

Scheduling for Heterogeneous Multicore Processors

Heterogeneous systems could take advantages of ISA diversity by matching applications to cores whose ISA can be best exploited by particular applications. Some processing cores can support applications targeted accelerated instructions, such as with SSE4.2 in Intel's Corei7. However, single-ISA HMPs allow a single piece of software to be scheduled and rescheduled among distinct processing nodes without the need of compiling programs for different ISAs or using an intermediate byte-code or a virtual instruction set architecture. This dissertation focuses on single-ISA HMPs, also known as asymmetric chip multiprocessors (ACMPs), however the same scheduling techniques can be applied to multiple-ISA HMPs.

As described in Chapter 4, Kumar et al. [42] designed a heuristic scheduling approach for dynamically assigning jobs to cores for single-ISA HMPs. In this approach, a scheduler evaluates all possible assignments before choosing the best assignment. A reassignment is evaluated whenever there is a drastic change in the performance of one of the executing threads. Trying all possible assignments of threads on unique core types can cause significant execution time to be spent in suboptimal arrangements. Permuting threads between core types also quickly becomes impractical as the number of different types of cores increases.

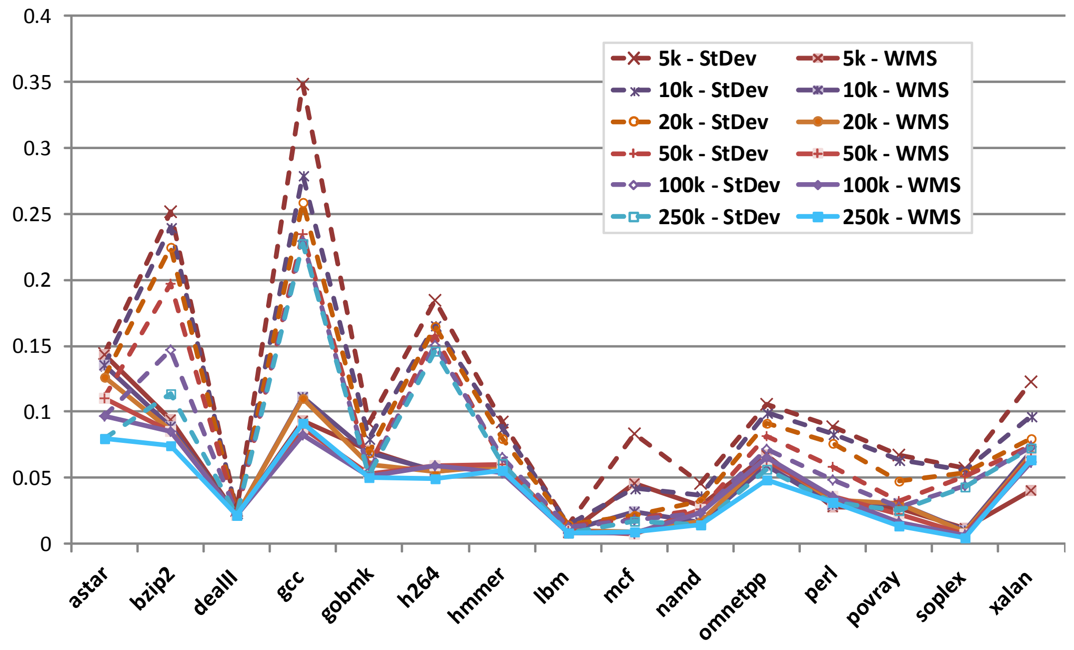
This work proposes novel phase-identification-based scheduling approaches for single-ISA HMP systems. Each phase represents a sequence of program execution with similar behavior. Whereas previous approaches used phase-change *detection* to initiate the evaluation of new schedules, my approach seeks to actually *identify* the current phase of each executing thread and reuse performance evaluations whenever previously recognized phases reoccur. Specifically, the working set signature (WSS) [23] approach, described earlier in Chapter 3, is used to identify program phases. A signature, identifying the currently executing application working set, is computed over some window of execution. A phase change is encountered when the signature of the current working set is significantly different from that of the previous one.

4.1 Phase-Based Scheduling

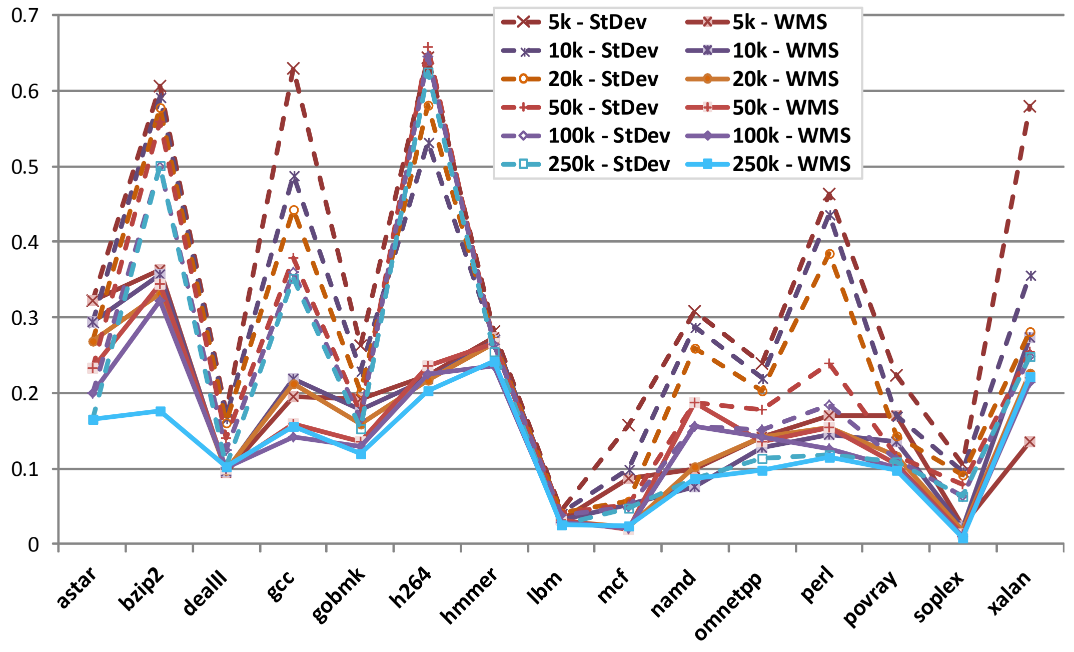
A working set signature (WSS) is a highly compressed representation of a program's working set of retired instructions. The WSS approach uses non-overlapping windows of retired instructions to generate programs signature for each window. A signature identifies an application's code over a working set or windows of instructions. Dhodapkar et al. [23] used a window size of 100 000 instructions to detect medium length program phases. They also suggested a signature size from 256 bits to 1024 bits to detect program phases. During temporally local program segments, these signatures will be similar for several consecutive windows, composing the duration of a phase. When the same program phase recurs at a later point in time, the same (or very similar) signature indicates a repeated phase.

The WSS is calculated using a portion of the bits of the program counter hashed into a working set signature vector as shown previously in Figure 3.3. The least-significant 6-bits of the program counter are excluded, corresponding to the instruction cache block size of 64 bytes. Thus, each instruction in the same cache block will hash to the same bit in the WSS vector. When an instruction that hashes into a particular bit in the current signature vector is executed, this bit is set. That means, the process of computing a WSS is performed simply by setting those specific bits in the current signature.

To assess the suitability of working set phases for predicting an effective thread-to-core mapping, the IPC of each application was measured over windows of different number of executed instructions. The standard deviation across all windows of an application was computed to reflect the variation in IPC over the run of each application. The weighted mean was also computed for the standard deviations of retired IPC within phases detected using the same windows. The standard deviation for all windows are weighted depending on the number of windows occurring in each phase. These are shown for fifteen SPEC CPU2006 benchmarks in Figure 4.1. Single-window phases are excluded in this comparison because they result in a zero standard deviation. The weighted mean of the standard deviation (WMS) of instruction throughput of windows within the same program phase is always less than the standard deviation over all windows (StDev). Thus, the throughput of each application (IPC) during each phase is relatively more consistent, typically varying much less during occurrences of the same phase than it does throughout the entire execution of the thread. Note that for few applications, such as *astar, lbm* and *dealII*, this difference appears quite small because only a very small number of long-running phases are detected.



(a) Core 0 (In-order).



(b) Core 1 (Out-of-order).

Figure 4.1: Standard deviation of IPC over the entire application (StDev) and weighted mean standard deviation across windows of the same phase for each benchmark.

Because program behavior often repeats over time, the phase-aware scheduler can learn from the past history of applications to predict the performance/power on the different types of core for each application. When a phase repeats, its performance/power is expected to be similar to the previous recurrence of that phase.

4.1.1 Scenario

Consider a scenario where there are two programs, represented by threads *A* and *B*, and two cores *C1* and *C2*. Suppose that each program has two different phases, and both program threads alternate between their two phases. The phase lengths are different, and each thread exhibits different IPC values for each phase, as shown in Table 4.1. From the table, we can see that *C1* is always slower than *C2*. Combining phases of both threads into one execution phase (or set of phases), the scheduler may detect four different execution phases (Table 4.2). The scheduler first samples the performance of threads' phases on the different core types. Initially, the scheduler assigns thread *A* to *C2* and thread *B* to *C1* randomly, entering a sampling interval. During sampling intervals, the scheduler permutes the assignment of threads *A* and *B* among cores *C1* and *C2*. The scheduler then switches the threads to perform the second sampling interval. While sampling, the scheduler profiles the execution of threads and then uses the generated profiles to choose the new assignment for the current phase of execution. It calculates the weighted speedup for each thread using Equation 3.1. Table 4.2 shows the assignments of threads *A* and *B* to the cores *C1* and *C2* for the four possible phases.

Threads *A* and *B*, each executes a phase that performs differently ac-

Table 4.1: IPC and number of instructions of each phase of threads A and B.

Threads phases	C1	C2	No. Instrcutions
Thread A Phase AI	0.5	2.7	3000
Thread A Phase AII	1.0	2.0	7000
Thread B Phase BI	1.3	2.0	4000
Thread B Phase BII	0.4	1.5	6000

Table 4.2: Possible sets of phases, combining both threads A and B.

Execution phase	A	B	C1	C2
Phase_set_1	Phase AI	Phase BI	B	A
Phase_set_2	Phase AI	Phase BII	A	B
Phase_set_3	Phase AII	Phase BI	B	A
Phase_set_4	Phase AII	Phase BII	A	B

cording to the type of the core it is running on. Thus, each thread requires a different number of cycles to finish the execution of any particular phase. Assume threads A and B run for 10000 instructions each and the scheduler detects three distinct phases (Table 4.3). When sampling is completed, the scheduler chooses *A* to run on *C2* and *B* to run on *C1*. Now, we can calculate the number of cycles it takes for each execution phase, using the number of instructions for each phase and the IPC, see table 4.3 from the possible set of phases in Table 4.2. *Phase AI* of thread *A* runs on the faster core and has fewer instructions than *Phase BI* in thread *B*, thus, it finishes execution faster. Next, thread *A* changes its execution phase to *Phase AII* while thread *B* is still in *Phase BI*, but the scheduler detects a new execution phase (*Phase_set_3*) and samples again. The same thing happens when thread B changes its execution phase. As shown in Table 4.3, *Phase BII* of thread *B* finishes execution faster than *Phase AII* of thread *A*, so *Phase AII* continues to run while thread B enters a new phase (*Phase BI*). This results in a different execution phase

Table 4.3: Execution phases–identified when running threads for 10 000 *A* and *B*.

Execution phase	A	B	No. cycles	C1-C2
Phase_set_1	Phase AI	Phase BI	1429	B-A
Phase_set_3	Phase AII	Phase BI	1648	B-A
Phase_set_4	Phase AII	Phase BII	4000	A-B
Phase_set_3	Phase AII	Phase BI	2390	B-A

(*Phase_set_3*), which is a repeated phase. Therefore, the scheduler does not need to sample again, instead it retrieves the assignment from its history table.

The result of the scenario for hardware scheduling using working set signatures is shown in Figure 4.2. On the left of the figure, there are two data sets that represent static assignments. The phase-aware scheduler performs better and spends less time sampling than the two possible static assignments. Furthermore, for this particular scenario the phase-based scheduler performs better than the previous work in [42] since it detects short phases. The heuristic method in [42] does not detect fine-grained phases and in this case performs exactly the same as the static scheduler, because the sampling period used in [42] is two million cycles. This is longer than several, short changes in program behaviors.

4.1.2 Phase_Sampling Approach

In the first proposed phase-based scheduling approach, *Phase_Sampling*, a sampled performance evaluation (IPC) of each thread on each core type is initiated when a previously unencountered *combination* of executing phases is identified. The weighted speedup is calculated using the sampled IPC information on each core by dividing that IPC on the highest IPC of all the samples. This perfor-

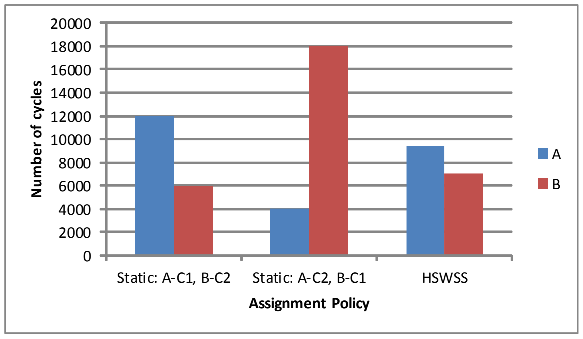


Figure 4.2: Number of executed cycles for different assignment polices for two threads running on a dual-core HMP.

mance evaluation is akin to the sampling done in Kumar's work [42]. However, Kumar et al. [42] divided the IPC of each sample on the total IPC of each application as if it is running alone in the system [42]. This required a priori knowledge of the isolated performance alone on the system. Such a requirement likely makes the sampled evaluation methodology of Kumar [42] impractical for real systems. In *Phase_Sampling*, the highest performance schedule for the given set of program phases is selected and that schedule is recorded in a Signature History Table (SHT) for that *set of phases*. An example SHT for the Phase_Sampling approach is shown in Figure 4.3. When a phase change is detected, the scheduler searches the SHT for the combination of currently executing threads. The signatures of executing threads are compared with the ones previously recorded in the SHT. Recognizing the same set of phases again enables for the reuse of a previously determined schedule without any additional evaluation.

The main limitation of *Phase_Sampling* is that to reuse a schedule for a recorded set of phases, each running thread must be in the same phase as

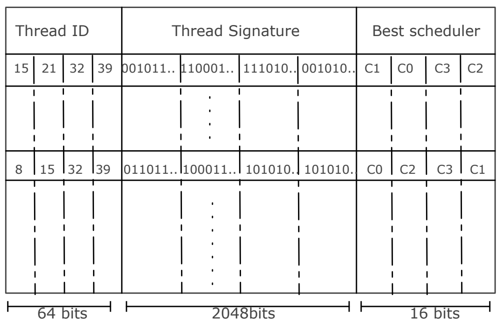


Figure 4.3: Signature History Table for *Phase_Sampling* Approach.

before. The second proposed phase-identification based approach, *Phase_IPC* described in Section 4.1.3, relaxes this requirement.

4.1.3 Phase_IPC Approach

The second scheduling approach, *Phase_IPC*, attempts to more fully exploit the correlation between repeating program phases and the performance of that phase on a processor core. In this approach, when a phase change occurs, the scheduler is invoked to determine a potentially new mapping between executing program threads and the processor cores. The signature of the new individual phase is compared with those in a hardware Signature History Table (SHT), and if a match is found a previously identified phase has been detected. The observed performance of that phase (measured in instructions per cycle), on each type of processor core, is used as the prediction of the performance for this detected phase. Figure 4.4 shows a SHT for *Phase_IPC* on a heterogeneous system with four different types of cores. For the phase signature shown in the table, the performance prediction for the thread's current phase if it was to execute on *Core 0* is 1.0, while the prediction of performance on *Core 3* is 1.9. Only when a previously unencountered phase is detected, is it necessary to evaluate the performance of that phase on each core. Note that unlike the sampling approaches, this does not require permuting all of the mappings of threads to cores, but rather only rotating a particular thread to each core for an evaluation period. If four different types of cores are executing four threads, there are 24 possible mappings of the four threads to the different core types. However, *Phase_IPC* approach only requires evaluating the performance of a previously unencountered phase of execution of each thread on each of the four cores. When, individually, each application is executing a previously seen phase (for that application), the performance prediction from the SHT is used to predict which mapping of application threads to cores will yield the highest throughput. Thus, in this case, even if the exact set of phases currently executing has not been encountered together, evaluating the performance of the threads on different cores can be avoided as long as the phases have previously been evaluated separately.

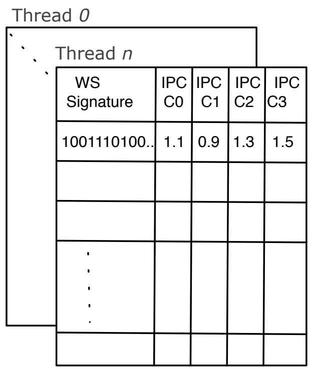


Figure 4.4: Signature History Table for *Phase_IPC* Approach.

Table 4.4: processor configurations

parameter	core 0	core1	core 2	core 3
execution	io	ooo	ooo	ooo
issue width	4	2	3	4
l1 cache	32kb	16kb	16kb	32kb
l2 cache	256kb	256kb	512kb	512kb
rob	–	64	96	128
rs	–	16	24	32

4.1.4 Evaluation and Results

To further evaluate the phase-based scheduling techniques using WSS for fine-grained scheduling, an appropriate signature size and window size over which phases are detected are evaluated.

Evaluation of Window Size for Computing Working Set Signature

To determine an appropriate working set's window size over which program phases are calculated, each application was run and analyzed using several different window sizes: 2k, 5k, 10k, 20k, 50k, 100k and 500k executed instructions for total of 250 million instructions. Each program was evaluated on the cores shown in Table 3.1 and Table 4.4 using the multicore performance estimation model described earlier in Chapter 3.

Figure 4.5 shows a characterization of each window for a 250 million instruction segment run using 256-bit signature size. In this figure, these windows are divided into three categories: single-window phases, repeated-window single phases and repeated-window repeated phases. As shown in Figure 4.5, on average, the percentage of single-window phases increases as the window size decreases. These phases are transitory phases that can be considered noise associated with computing phase signatures over small windows. In general, the number of windows that compose repeated phases is higher than those that compose transitory or single-stable phases. Note, that while a relatively large portion of some application's execution is spent in repeated-window single phases (phases that do not reoccur during this 250 million instruction segment), these phases in all likelihood may recur over much longer periods of instructions. This has been observed anecdotally for several benchmarks but multi-billion instruction runs of each application have not been done due to the significant simulation time required.

In choosing an appropriate window size, parameters such as the number of program phases identified and the number of repeated phases encountered should be considered. In general, as the number of instructions in a window decreases, the number of detected phases increases. This trend is seen in Figure 4.6 and this is because smaller window sizes results in finer detected phases. Window sizes smaller than 5 000 instructions are not considered because they contain a significant amount of noise. Moreover, because a large number of phases are detected in these cases, there are also a large number of possible thread switches. Thus, such small sizes can be excluded for practical reasons. Rescheduling threads at the granularity of even moderate window sizes may have significant overhead. On the other hand, the use of large window sizes does not allow for detecting fine-grained changes in program behaviors. Because of the overhead of thread migration, scheduling may need to be performed with a period equal to several of the windows used for detecting program phase behavior. In this study, windows are considered to detect execution phases, and scheduling is performed on phases-granuralities. Phases vary in length during program execution and each phase may contain one to several windows

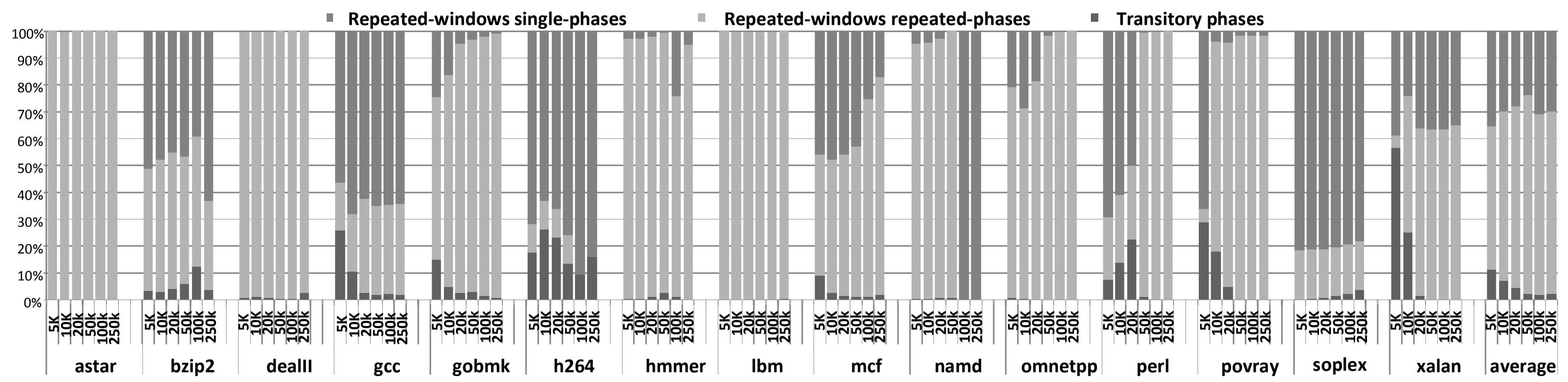


Figure 4.5: Classification of windows in each application for varying window sizes.

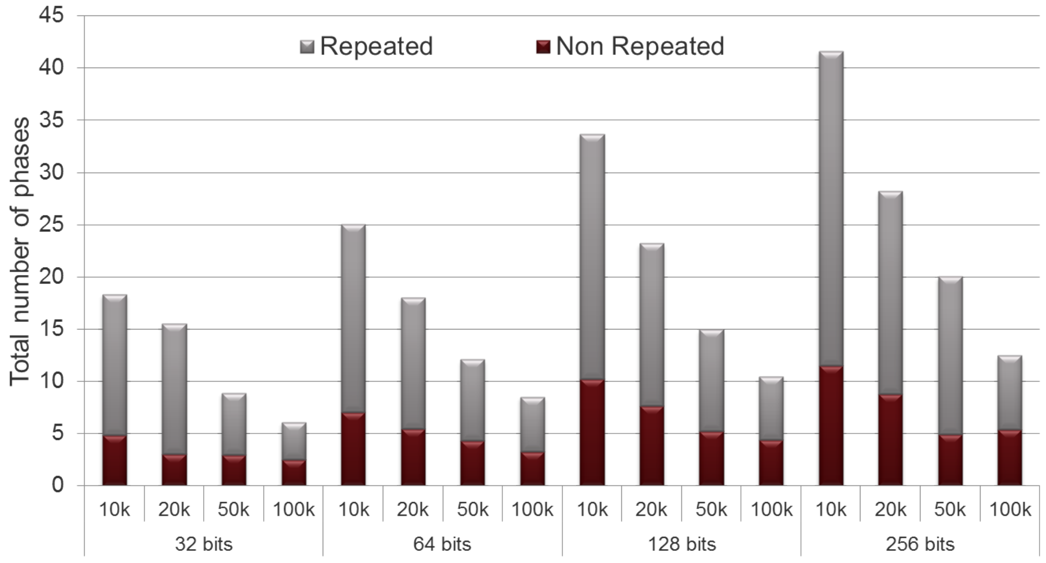


Figure 4.6: Average number of phases and repeated phases detected using different signature sizes.

of instructions.

Evaluation of Signature Size for Computing Working Set Signatures

To evaluate the impact of signature size on detecting fine-grained application phases, applications ran using 32-bit, 64-bit, 128-bit, 256-bit and 512-bit signatures sizes. Signatures larger than 256-bits are ignored because of the large space required to store them in the SHT table. Figure 4.6 shows the average number of detected phases and repeated phases for SPEC2006 benchmark applications using different signature sizes smaller than 512-bits. The average number of detected phases increases with increasing the signature size because larger signatures are detecting ever finer differences between detected phases. The difference in the number of detected phases among the different signature sizes is bigger for smaller window sizes.

In addition to the effect of window size and signature size on the number of detected and repeated phases, the performance estimation model is used to find the highest possible performance using the *Phase_IPC* scheduling mechanism for the different window sizes. In general, the smaller the window size, the higher the performance is. Conversely, the smaller the signature size, the lower the performance. The lowest signature size is neglected because it produces the lowest weighted speedup, and the highest signature size is also neglected because of the large number of bits required to store the signature.

Multicore Results Using the Performance Estimation Model

To evaluate the proposed phase-guided approaches, a quad-core system composed of one of each of the two types of cores (2 Corei7_like, 2 Atom_like) was simulated. Five hundred sets of four benchmarks were chosen randomly from the set SPEC CPU2006 benchmarks evaluated as workloads for this heterogeneous multicore processor system. The two different types of scheduling algorithms, *Phase_Sampling* and *Phase_IPC*, described earlier, were evaluated and compared to a fine-tuned heuristic scheduling method. The *Phase_Sampling* method attempts to match the four-tuple of application phases (running on the four cores) and reuse any previously determined schedule for that set of phases. When a new set of phases is encountered, this method requires 24 different sampling intervals, evaluating each thread-to-core-type mapping. The second method, the *Phase_IPC* method, records the IPC for a new phase as it is sampled on each of the core types. Because it records the performance for each application thread in isolation (without regard to the phase in which other running applications are in), it only needs to perform four sampling intervals for each phase–one sampling interval for each core type. Sampling is performed *only* the first time a particular program phase is encountered. Each time a previously encountered phase occurs, the scheduler simply reuses the results of the previous sampling intervals of those recurring phases to predict the assignment that will produce the highest summation of weighted speedups and utilizes that assignment as the current schedule.

The final results of these scheduling approaches are evaluated using the weighted speedup across all four applications. The weighted speedup for each thread is computed using IPC achieved for each application thread over a specified interval on the core onto which it was scheduled divided by its performance on the best performing core during the entire run(Equation 3.1). Note that this weighted speedup is different from the one used for dynamic scheduling to evaluate the different possible assignment in Equation 3.2. The goal of each scheduling approach is to achieve the highest possible weighted speedup. The weighted speedups were summed across the four applications and averaged across each of the intervals over the entire one billion instruction execution (250 million instructions per application thread). Note, that a weighted speedup of four would mean that the multicore processor achieved the performance of a four-core, homogeneous system whose processors are all implemented as the best performing core type.

Figure 4.7 shows the weighted speedup of the two proposed scheduling mechanisms compared with a fine-tuned heuristic, the ideal assignment for each interval, the worst case assignment and the average of the 24 static possible assignments. The ideal assignment is an approximation of the best schedule for each working set of instructions computed as the weighted speedup of the

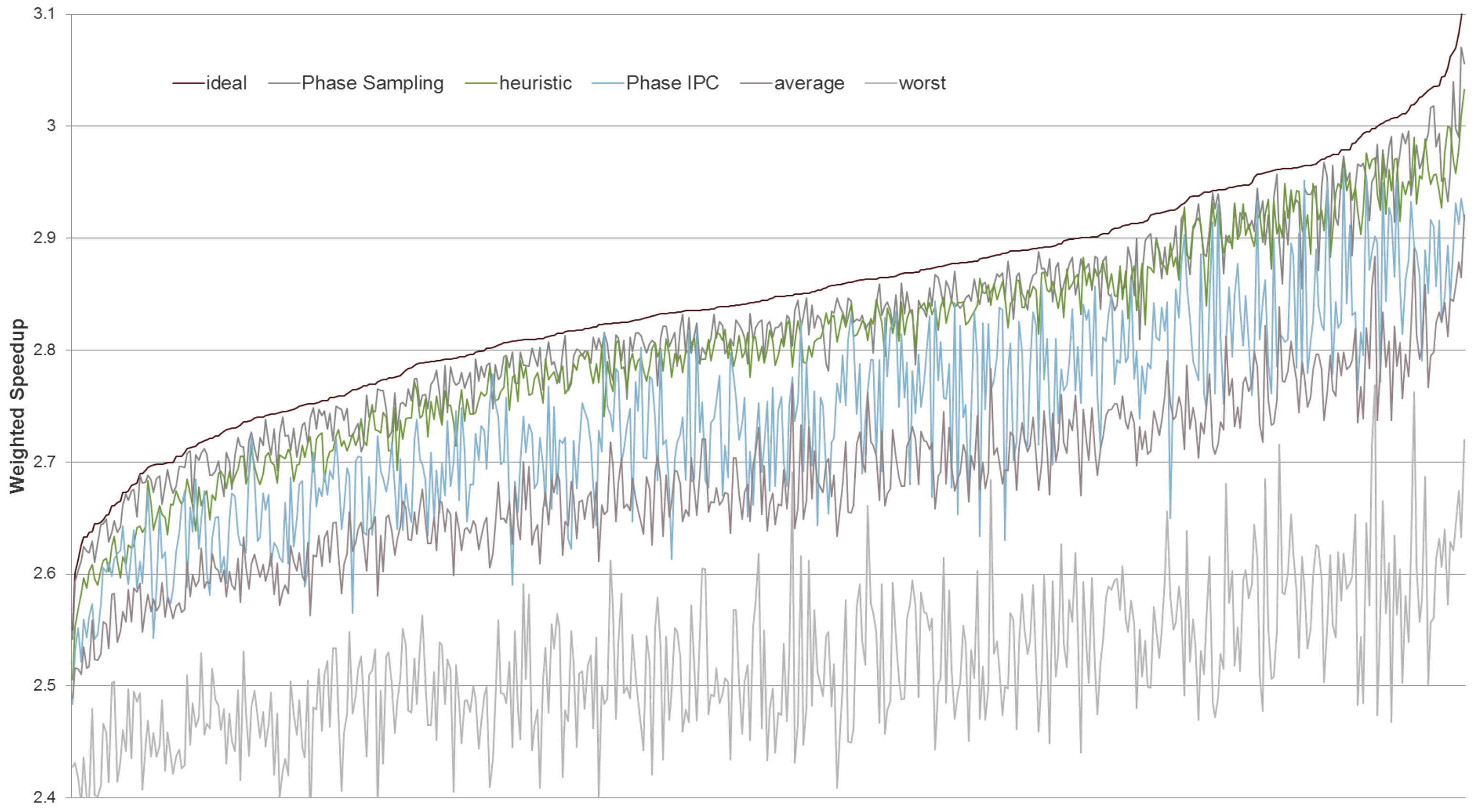


Figure 4.7: Weighted speedup for different scheduling algorithms on 500 sets of 4-tuple benchmarks.

highest-performance mapping for each window. The worst case assignment similarly represents the assignment that results in the worst weighted speedup for the 24 different static scheduling possibilities. The average of the 24 static assignments is also provided for comparison. This would be the expected value of the performance of a randomly picked assignment. Figure 4.7 shows that in terms of overall weighted speedup, the *Phase_Sampling* method performs almost as good as the ideal assignment and outperforms *Phase_IPC* and the fine-tuned heuristic approaches. *Phase_Sampling* achieves a higher weighted speedup than the fine-tuned heuristic approach because *Phase_Sampling* initiates scheduling decisions based on an accurate signature-based phase identification compared to the coarse-grained phase change detection used by Kumar [42]. However, *Phase_Sampling* requires more sampling (often significantly more). This is due to the fact that relatively fine-grained phases are being detected and that a match is required of all four executing phases across the four applications in order to reuse a previously determined schedule. The number of sampling intervals for this approach can be reduced by requiring a phase change in more than one application to initiate a search for a phase set match in the SHT or a sequence of sampling. However, this could also negatively impact the weighted speedup achieved. The fine-tuned heuristic approach yields a lower weighted speedup than *Phase_Sampling*. *Phase_IPC* approach typically yields a weighted speedup slightly below that of the . However, the *Phase_IPC* requires only roughly half the sampling intervals and application thread switches as that of the fine-tuned heuristic method. *Phase_IPC* bases its scheduling on the IPC that was measured during the first window of the first time that a phase is identified. By reusing this measurement, the amount of sampling required is greatly reduced. The performance of *Phase_IPC* could likely be improved by detecting when this estimation does not accurately represent the overall behavior of the phase and reinitiating sampling.

Multicore Results Using the Cycle-Accurate Simulator

Two types of multicore experiments are performed on the cycle accurate simulator: dual-core simulations and quad-core run. The dual-core processor contains an Atom_like core and a Corei7_like core. The quad-core processor contains two Atom_like and two Corei7_like cores. These type of cores where chosen to represent a realistic asymmetric system that combines both a superscalar high performance processor such as the Corei7_like and an energy-efficient in-order processor such as the Atom core. *Phase_IPC* scheduling is evaluated using the cycle-accurate simulator, Soonergy, on both processor types. Results in this chapter ignore the direct cost of thread migrations but do take the indirect cost of thread migration into consideration.

Dual-Core Simulations

Phase_IPC is evaluated on the two core processor neglecting the direct cost of thread migration. The indirect cost represents warming up of microarchitectural structures such as caches and branch predictors after switching, and is considered in this work. Branch predictors are not invalidated after switching the running thread to a different core; the new thread continues from the same state where the other thread was and warms up the branch predictor while executing. Two approaches for dealing with caches are evaluated. The first approach is the invalidation of the data caches. Cache invalidation can cause more overhead after switching a thread from one core to the other. Figures 4.8

and 4.9 show the results of *Phase_IPC* including the overhead from invalidating caches. *Phase_IPC* results are compared to the heuristic approach in Kumar et al. [42]. The second way of handling caches is by using an update coherence protocol, which allows some cache information for the next time the thread migrates back to the same core. Only cache blocks that are evicted by data accesses from the new thread allocated to the current core will be missing. Although applications that are considered in this study are single-threaded, when an application switch from one core to another, data from writes to locations in both caches are kept coherent. Constantinou et al. [21] showed that if a thread migrates to a previously visited core for a system with Private L1 caches and a shared L2 cache, the performance loss can be minimized. Figure 4.8 shows the weighted speedup of *Phase_IPC* compared to Kumar's heuristic method [42]. Note again that Kumar's method [42] requires a *priori* knowledge about each application running separately on the system. For some combination of applications that benefit the most from exploiting finer granularity of phase changes. For instance, *astar_namd* input combination benefit significantly from *Phase_IPC* through exploiting fine-grained changes in their behaviors. Specifically *namd* application consists of very fine-grained phases that result in many L2 cache misses; by adapting to these fine changes in *namd*, *Phase_IPC* performs better than the heuristic sampling method [42]. For some other input combinations, the IPC of some benchmarks such as *gobmk* does vary much through the entire simulation run, thus, exploiting fine-grained scheduling to adapt to the change of the other inputs may reduce the weighted speedup of applications such as *gobmk*. In general, *Phase_IPC* yields a 1.8% improvement in the weighted speedup over Kumar's sampling heuristic approach [42].

Phase_IPC also results in 1701 switches on average which is significantly more than Kumar's approach (37.2 average switches). Thus, *Phase_IPC* has a better potential to provide even better performance with reducing the indirect cost of thread migrations such as warming up caches than the work in [42]. This overhead can be greatly reduced by using cache prefetching mechanisms such as Suleman's et al. [77]. The weighted speedup of the *Phase_IPC* algorithm is similar to the heuristic algorithm on average. When using an update cache coherency protocol the weighted speedup showed only a slight improvement, however, larger improvement is observed on the execution time.

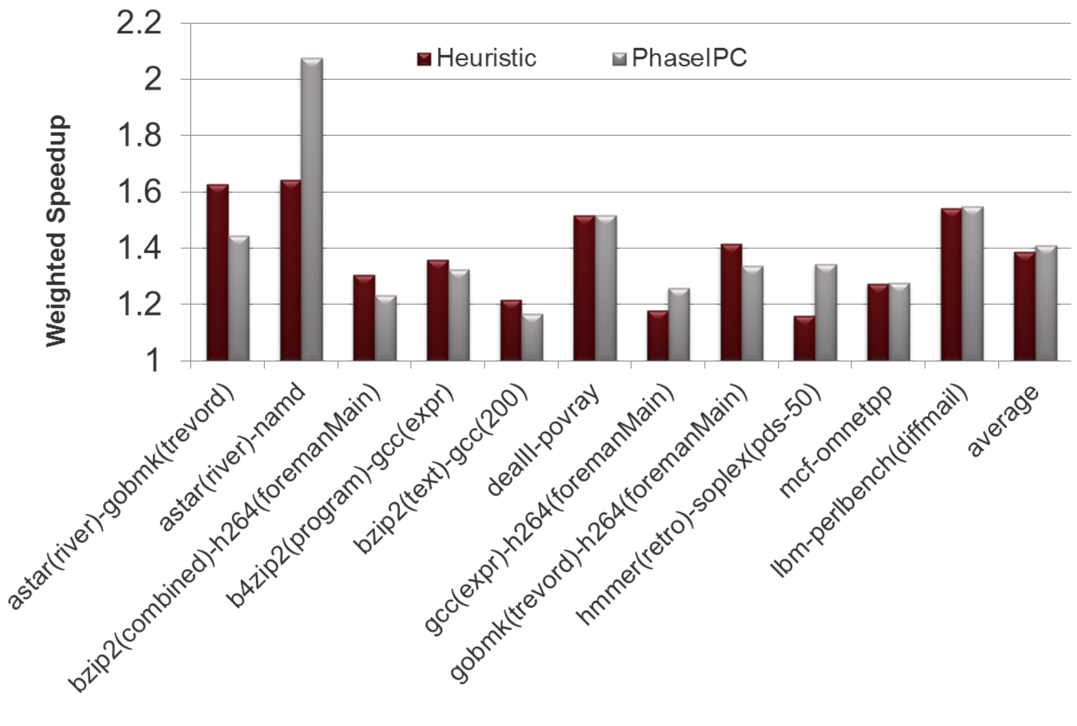


Figure 4.8: Weighted speedup from *Phase_IPC* compared to Kumar's sampling method [42] on a dual-core system.

Although the improvement over a previous sampling approach is only 1.8%, *Phase_IPC* results in a shorter run time of both benchmarks for most of the input combinations as shown in Figure 4.9. Note again, that the heuristic sampling approach by Kumar et al. [42] uses before-hand knowledge about the IPC of each application as if it is running alone in the system. In general, *Phase_IPC* yields 12.5% reduction in execution time for both applications. The heuristic sampling approach by [42] may result in a *cognitive bias* while evaluating the different assignments. Because the sampling duration used by Kumar et al. [42] is long (two million cycles), this can cover more than one phase change and affects the IPC of that sample. A similar effect on the IPC for other different assignments can be observed if they are sampled on the same set of instructions, however, other assignments may observe different application behaviors. Thus, the evaluation method might be biased towards one assignment because different application's behaviors occur during sampling itself. This also can be true for small sampling intervals when sampling is not triggered at the boundaries of phases.

Quad-Core System

A quad-core processor was also simulated using the *Phase_IPC* scheduler and a random static assignment for eight different four-tuple combintations of benchmarks (randomly chosen). *Phase_IPC* shows a great improvement over a random static assignment. Figure 4.10 shows the weighted speedup of both *Phase_IPC* and a random assignment; *Phase_IPC* improves the weighted speedup of the system by 9.5% on average. Similar to the dual-core system, *Phase_IPC* observes a slight improvement on system performance when using a coherency update protocol. Figure 4.11 shows the execution time of the quad-core processor for *Phase_IPC* compared with a random static; *Phase_IPC* results in 37.2% reduction in execution time on average over the random static

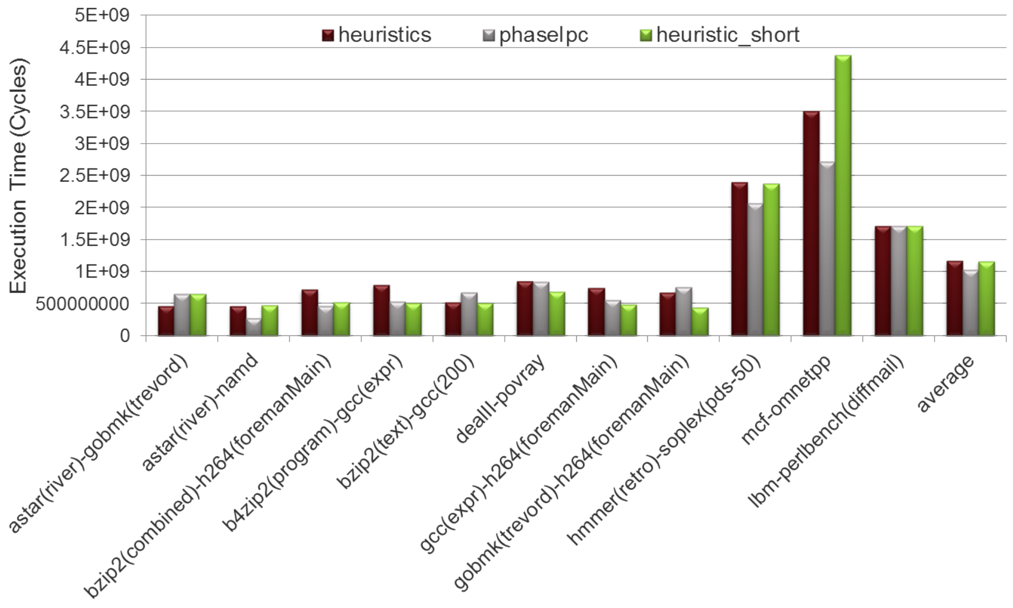


Figure 4.9: Weighted speedup of Phase_IPC compared to heuristic method on a dual-core system.

assignment. In addition, it results in an average of 27.15% increase in system throughput over a random static assignment as shown in Figure 4.12.

4.2 Energy-efficient Scheduling Algorithm

HMPs can be exploited for either increased performance or reduced power consumption. If the goal is improved (reduced) power consumption, then an approach similar to *Phase_IPC* can be used. In such an approach, program phase information is used to dynamically map applications to cores of various types on a single-ISA HMP processor targeting reduced energy consumption [64]. Program execution phase "signatures" are calculated on the fly, and when a phase change occurs the scheduler re-evaluates the assignment to optimize for energy-delay product. Phase-aware scheduling is applied for energy consump-

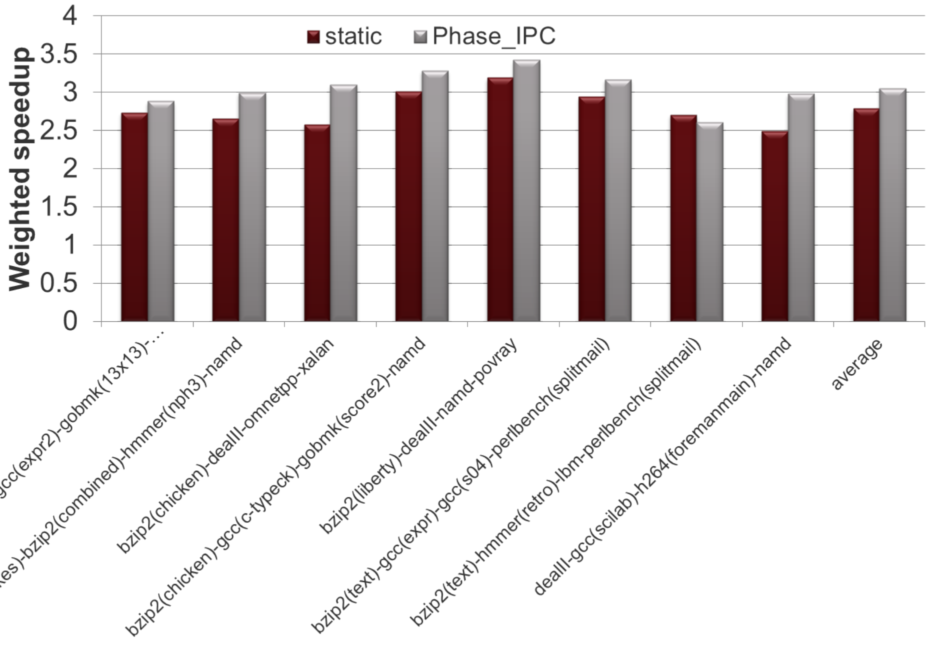


Figure 4.10: Weighted speedup of Phase_IPC compared to random static assignment on a quad-core system.

tion reduction using energy-delay product as an evaluation metric rather than performance. The energy-delay product is a common metric for evaluating energy-efficiency; it balances between energy consumption and performance.

4.2.1 Phase_EDP Scheduling Algorithm

The proposed energy-efficient phase-identification based scheduling approach, *Phase_EDP*, exploits the correlation between repeating program phases and the EDP of each phase on a processor core. When a phase repeats, the EDP of the repeated phase is expected to be similar to the EDP of the first occurrence of that phase. Figure 4.14 shows that the EDP behavior of 100 000 instruction intervals repeats over time for the *bzip2* application. Similar behavior was observed for the other simulated programs. Thus, the EDP information calcu-

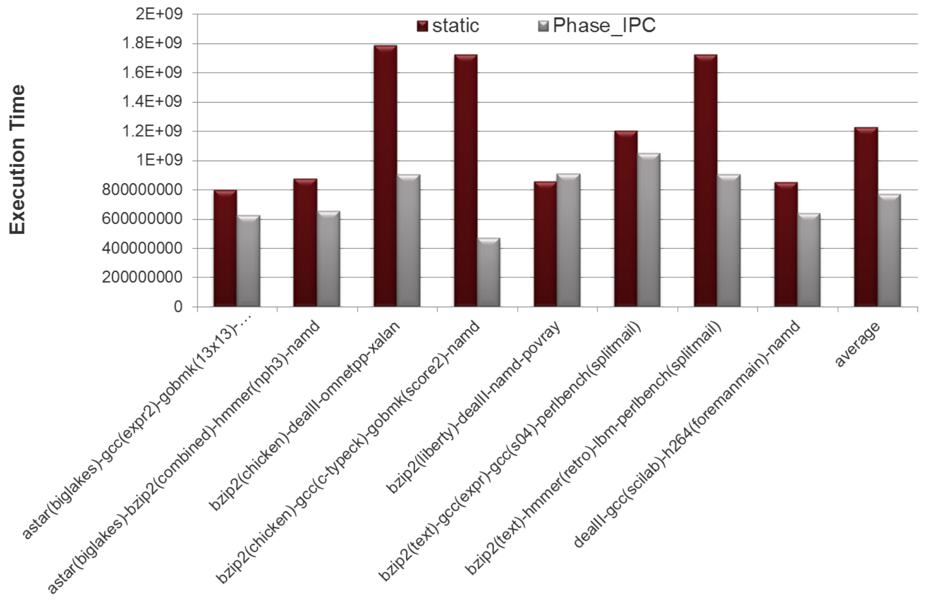


Figure 4.11: Execution time of Phase_IPC compared to heuristic method on a quad-core system.

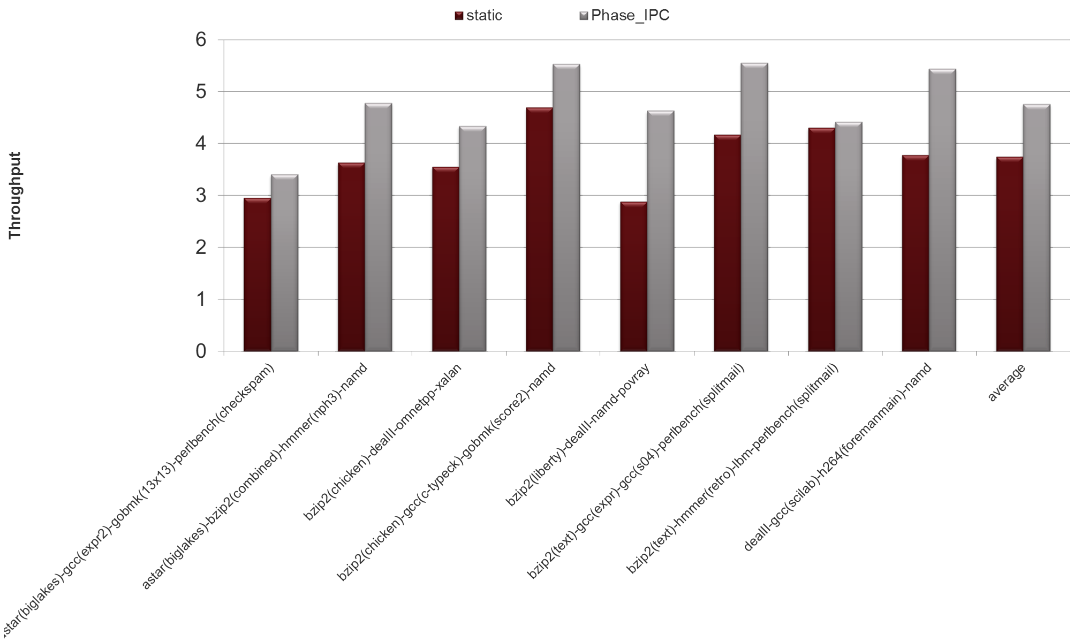


Figure 4.12: Throughput of Phase_IPC compared to heuristic method on a quad-core system.

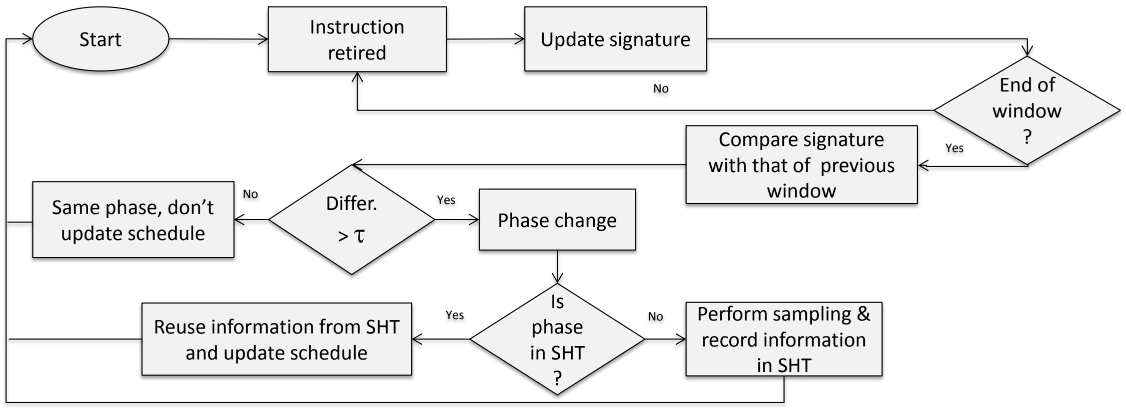

Start
Instruction retired
Update signature
End of window ?
No
Yes
Compare signature with that of previous window
Same phase, don't update schedule
No
Differ. > τ
Yes
Phase change
Is phase in SHT ?
Yes
No
Reuse information from SHT and update schedule
Perform sampling & record information in SHT

Figure 4.13: Flow chart showing *Phase_EDP* scheduling technique.

lated over the first window of the first occurrence of a phase is used to predict the schedule that leads to the lowest total EDP, for all set of phases, when a phase change occurs in one or more of the executing programs with repeated phases.

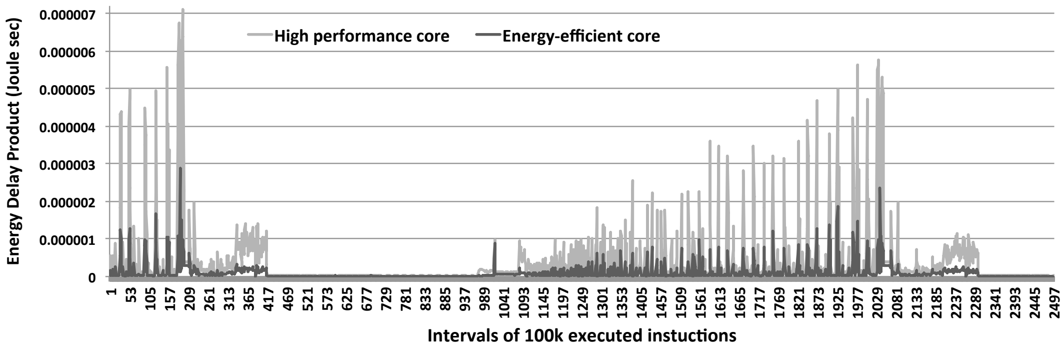


Figure 4.14: Energy-delay product for *bzip* application on an energy efficient in-order core and a less energy efficient, high-performance out-of-order core.

When a previously unidentified phase is encountered, the scheduler evaluates the EDP for the thread in that phase on each core type by sampling the execution of that thread on each core. The EDP for each currently executing application thread is recorded separately in a Signature History Table (SHT) modified to store EDP rather than IPC. When a detected phase signature matches one in the SHT, the previously observed EDP of that phase on the respective core types is used as a prediction of the EDP for the current detected phase. This is then used to predict the minimum EDP arrangement of threads for the available core types. Through the use of these signatures to identify repeated occurrences of previous phases, actual EDP evaluations are necessary only when a previously unencountered phase is detected.

Figure 4.13 shows a flow chart diagram of the *Phase_EDP* scheduling process. First, after each instruction in the pipeline finishes execution, a bit in the signature vector is set. The signature is calculated over a window of instructions. At the end of each window, the signature of the window is compared with that of the previous window. If the difference between the two signatures is below a certain threshold (τ), the two windows are assumed to be in the same execution phase. Otherwise, a phase change is detected, in which case the signature is compared to the ones recorded earlier in the SHT to find if the current phase is a repeated phase. If the difference between signatures is above τ, a new phase is detected and recorded in the SHT. The new phase is sampled on the different core types for EDP and the EDP value is also recorded in the SHT. Next, the scheduler chooses the assignment that leads to the lowest EDP. On the other hand, if the signature was similar to one in the SHT, the phase is considered a repeated phase, and the EDP value from the SHT is used to predict the schedule with the lowest EDP.

Phase_EDP is evaluated using a multicore processor consisting of a mix of one simple energy-efficient IO core and larger, higher performance out-of-order (OoO) cores with different configurations as shown in Table 4.4. All

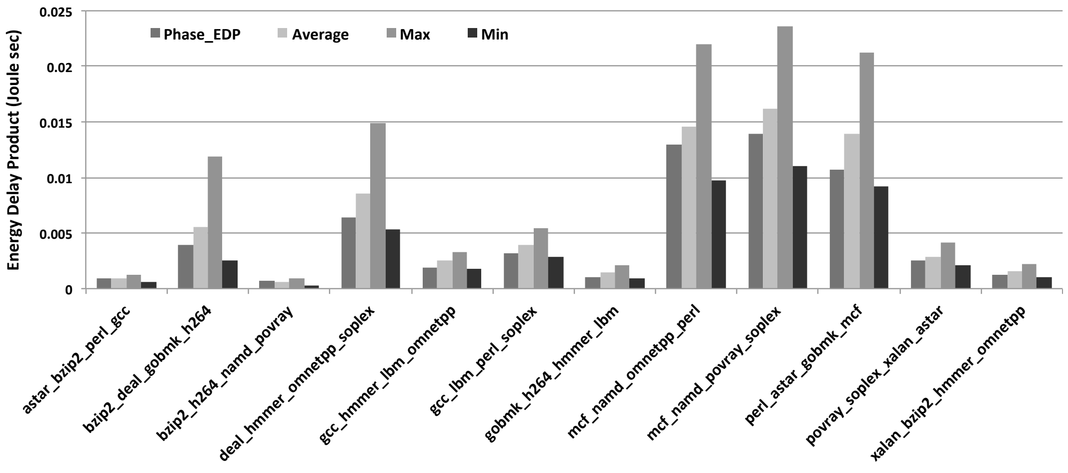


Figure 4.15: Energy-delay product for four-tuple of programs (250 million instructions) using different scheduling methods.

cores possess private level one (L1) and level two (L2) caches. The OoO cores vary in instruction issue width, L1 cache size, reorder buffer (ROB) size and number of reservation stations (RS) listed for each type of core. In general, for OoO cores, the higher the issue width the better the performance. However, but the more energy consumption is associated with extra overhead needed to find additional instructions to execute in parallel. Similarly for L1 cache, ROB and RS, the larger the size, the better the performance. However, this comes along with greater energy consumption. Different integer and floating point applications from the SPEC CPU2006 suite were used in our evaluations.

Initial evaluation of *Phase_EDP*, was performed for different benchmarks on a quad-core processor consisting of one core of each type. Energy-delay product information was computed over 300 million instruction runs of all programs using an architectural power modeling approach based on [11] with scaling for the different core types. Sets of four different applications were then chosen to create several multiprogrammed workloads. Figure 4.15 shows the results

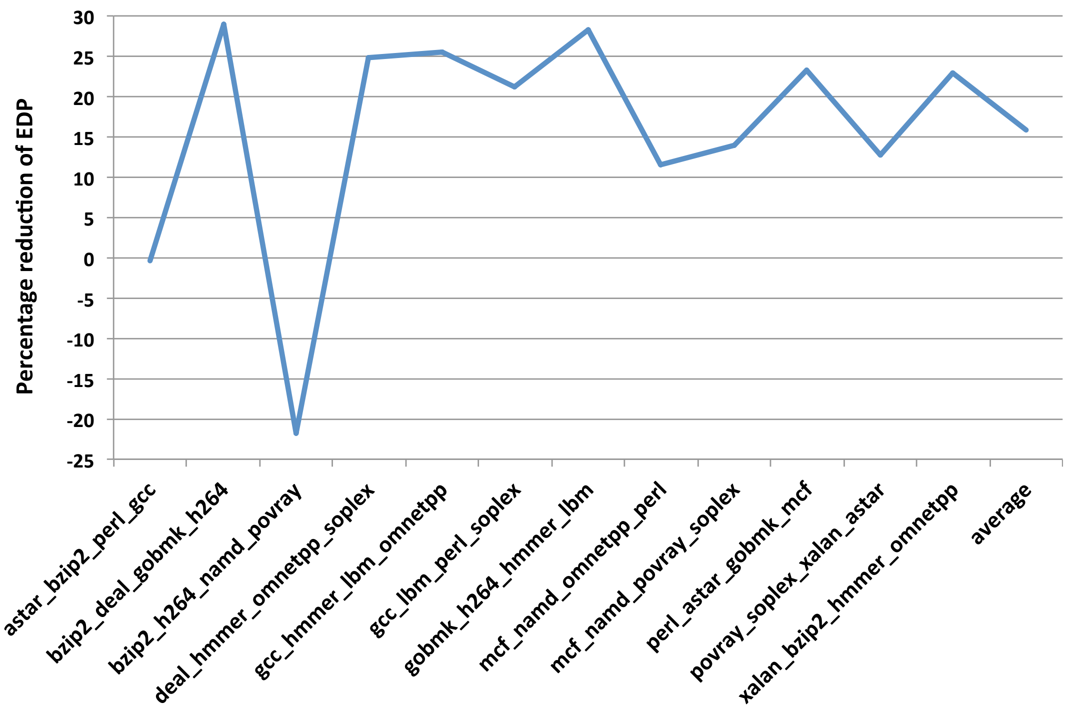


Figure 4.16: Percentage reduction in EDP from *Phase_EDP* over average of all static assignments.

of the energy-delay product for our scheduling algorithm compared with those of the average of the 24 static different possible assignments of applications on each of the core types. The assignment with the maximum EDP and the assignment with the minimum EDP are also shown. For most workloads, the EDP of the phase-aware scheduling approach results in a reduced energy-delay product compared to the average assignment. Note that this average represents the EDP of a randomly chosen static assignment. Figure 4.16 shows the percentage reduction in EDP through the *Phase_EDP* scheduling method over the average static assignment. The mean percentage of EDP reduction is 15.9% over the average static assignment. Although some workloads show only a slight reduction in EDP from *Phase_EDP* over the average static as-

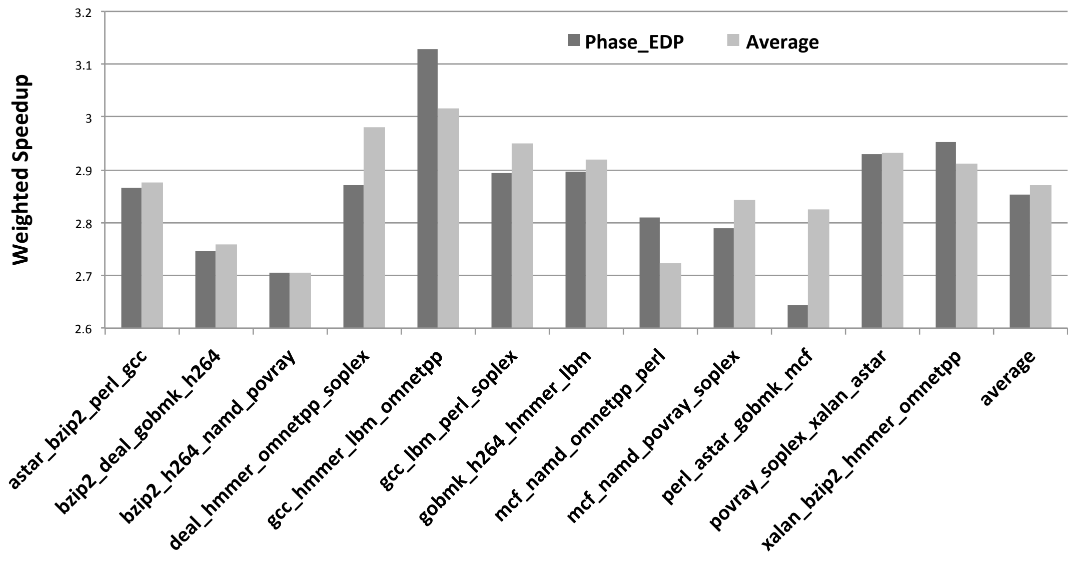


Figure 4.17: Weighted speedup of *Phase_EDP* compared with the average of all static assignments.

signment in Figure 4.15, the percentage decrement of EDP is significant as shown in Figure 4.16. Although *Phase_EDP* results in a significant reduction in power consumption, it is important that this reduction does not come at a corresponding significant reduction in performance.

Figure 4.17 shows a comparison of the weighted speedup for *Phase_EDP* and the average of the 24 different possible static assignments. The weighted speedup represents the sum of the ratios of the performance of each application over the performance of that application on the best performing core. While the *Phase_EDP* method provides 16% reduction in EDP over the average, it typically does not result in worse performance. The weighted speedup of the *Phase_EDP* is very similar to that of the average of the 24 assignments and even better in some cases. Thus, *Phase_EDP* results in a reduced EDP without harming the performance.

Chapter 5

Hardware Support for Fast Context Switching

In computer systems, context switching, the task of saving the state of one process or thread in memory and loading the state of another, is often time consuming. This overhead increases the granularity of the time-slices each process must be appropriated to avoid excessive time during which the system does no useful work. The penalty of context switching includes both a direct cost of saving and loading process state and an indirect cost, which includes the time that is needed to repopulate microarchitectural structures such as caches and predictors. This chapter evaluates the direct cost of migrating a thread from one core to another and presents a novel hardware context switching circuit that drastically reduces the direct cost of context switching.

Heterogeneous (or asymmetric) multicore processors (HMPs) can particularly benefit from faster process migration [65]. Differences between cores enable the exploitation of fine-granularity changes in program behavior but this requires frequent process migration. In addition to fine-grained scheduling for HMPs [66], faster thread migration benefits many other migration-based techniques including load balancing [63], thermal and power management [19], coherence protocols [38] and manufacturing-fault tolerance [58].

In multicore and multi-processor systems, the operating system (OS) is responsible for switching executing thread contexts. With time-multiplexing between threads, the central proceeding unit (CPU) saves the context of the old thread and launches another thread. Similarly, when a thread requests an IO (such as reading data from disk), the CPU does not stall waiting for the I/O read to finish. Rather, it switches to another process, and when the first thread finishes reading, the CPU is interrupted with the result of the read. In dynamically scheduled HMPs, the same need for operating system context switches obviously exists. The CPU needs to be able to switch a thread among its cores depending on the thread's relative characteristics and behavior in the current phase of execution. For example, a thread might achieve the best performance when running on one core for some phases, and running on another core for some other phases. The overhead and extensive data transfer associated with software context switching limits the number of switches per second for a CPU.

5.1 Methods to Support Frequent Thread Migrations

Accelerating thread migrations between the different cores on a chip can be done in software, hardware or software/hardware. Software acceleration of context switching can be achieved by modifying the operating system. Strong et al. [76] modified the Linux operating system to reduce the amount of time spent on both the direct overhead and indirect overhead of migrating threads from one core to another. Nellans et al. [53] continued on Strong's work to reduce the indirect cost of switching by adding a L2 OS-cache separate from User cache. Software methods have been shown to reduce the cost of context switching by a sizable percentage but are not able to eliminate the majority of context switching overhead. While software solutions are inexpensive, they are still relatively slow especially when considering frequent context switching. An alternative to software solutions is to perform context switching entirely in hardware. Hardware solutions should be faster than software but can not be dynamically configured or set threads' priorities. One of ARM's big.LITTLE processor designs utilize a such hardware context switching mechanism [30]. However, this particular design assumes that one core is powered on at a time while the other is turned off. A running application transfers from one core to the other dynamically depending on its performance needs. This dissertation proposes an efficient fast switching mechanism that is a cooperative hardware/software technique. By benefiting from fast switching in hardware while at the same time allowing the operating system to reconfigure the hardware switching mechanism, the context switch process can handle multitasking, serve interrupts, and reset applications' priorities.

5.2 Measuring the Overhead of Context Switching

The indirect cost of context switching consists of the performance degradation of such a system caused by resource sharing and warming up microarchitectural structures. For example, switching a thread between two cores might result in more L1 cache misses and branch miss predictions, which in turn affects the performance of the system. The direct cost of context switching includes time that is spent copying the context of the CPU, such as CPU registers, and flushing the processors pipeline in which the switched thread is running. Tsafrir [79] measured the direct cost of context switching on Pentium IV (2.2 GHz) processor to be 16.4% to 59.4% for LMbench benchmark suite depending on the way the operating system handles the wall-clock time. Although the indirect overhead of context switching is greater than the direct cost, the indirect cost consist of more than one type of overhead. Thus, decreasing the indirect cost of switching requires addressing each cost component separately (such as warming up instruction caches, data caches, branch predictors, etc). The direct cost of context switching including the execution of kernel code and saving the state of one process and loading the state of another is a major single component of the total overhead of context switching. Thus, I measure the direct overhead of thread migrations and provide a new hardware/software solution to this major component of the total overhead.

The direct cost can be measured using Ousterhout's method by forking a child process and periodically sending a message forth and back between the parent and the child processes using two pipes [56]. McVoy and Staelin improved Ousterhout's technique by eliminating system call overhead ([48]). In this work, the direct cost of context switching for a multicore processor was measured by changing the processor affinity for a process that does not require any memory accesses over 10 000 000 processor core switches. The system contains a quad-Core i5 x86 (2.67 GHz) processor running a Linux operating system. The direct cost calculation is measured as the real time of the system reduced by the user time and then divided by the number of switches (equation 5.1).

$$time_(switch) = \frac{time_{real} - time_{user}}{No.Switches} \tag{5.1}$$

To reduce the noise in calculations, this process was repeated for 500 times. The direct cost of context switching is measured to be $9.306 \pm 0.15\mu s$ with a marginal error of $0.0115\mu s$. This means that for the 2.67GHz processor, the direct cost of a context switch consumes on average 24 848 clock cycles with a marginal error of 30 cycles.

The total switching cost was also measured for one program, *dealII*, on the same system in which the direct cost was measured, by forking two children processes: one responsible for running *dealII* and the other responsible for switching the processor affinity for which *dealII* is running. The total cost of a context switch in this experiment was measured as 21.8μ seconds. Thus the indirect cost represents 42.6% of the total context switch overhead for *dealII* in this experiment.

5.3 Hardware Thread Migration and Context Switching

This dissertation proposes a novel hardware context switching circuit that enables low-overhead hardware thread migration between cores in a single-chip multiprocessor and cooperate with software, such that the OS is responsible for setting threads' priorities and choosing the threads to run on the next time interval. This switching circuit supports multiple simultaneous thread switches and can store the context of both currently running and time-multiplexed threads. This circuit both accelerates migration of threads between cores in a multicore processor and reduces the direct cost of context switching within each processing core. The thread switching circuit responsible for switching threads among different cores consists of control logic, a crossbar switch, additional registers and a *Shared Context Unit* (SCU). Figure 5.1 shows a block diagram of the hardware context switch.

The SCU contains multiple context sets (CSs). Each context set consist of all the registers that are required to save the state of a processing core. To accelerate switches between time-multiplexed threads, the number of CSs in the

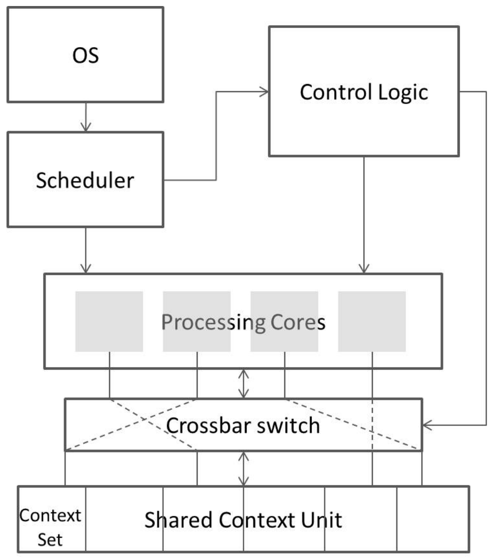


Figure 5.1: Block diagram for the context switch circuit.

SCU can exceed the number of cores. When the CPU runs multiple processes, it records information about these processes in different context sets and does not require the operating system to copy the CPU state and registers from memory when switching between processes. Having multiple context sets for each CPU (or processing core) on a chip supports multitasking for each core. Other architectural approaches similarly utilize multiple register files including simultaneous multithreading (SMT) [24] and checkpointing [34]. In SMT [24], multiple register files support multiple threads executing simultaneously within the same core. In out-of-order execution with checkpointing, additional copies of register values are used to save the processor state at appropriate points of execution [34]. These copies are used to repair register contents to a previous state when exceptions or branch mispredictions occur. By necessity, the SCU should be physically laid out near each of the cores. Because the architectural CPU registers must be closely integrated with the retirement stage of the CPU's pipeline, long wire delay would be intolerable. For a processor with large numbers of cores, it may be almost impossible to share the same context unit amongst all the cores. Some of the cores will be located farther away than others, and sharing the context unit among all of them may cause a considerable amount of latency in writing the architectural registers. Because of these layout constraints, a system with many cores on a chip can consist of clusters of cores, such that few cores form a cluster and share one context unit. If a process/thread migrates from one cluster to another the scheduler must copy the entire context of the migrating process/thread from one cluster's context unit to another, possibly requiring OS coordination.

Figure 5.2 shows a detailed circuit the hardware context switch. This circuit is responsible for switching threads among different cores and consists of control logic, a crossbar switch, additional registers and the SCU. The control logic receives information about threads and their scheduling assignment to different cores from the OS and the scheduler. The scheduler could be part in software as part of the OS or a hypervisor, or could be implemented in hardware. Additional registers are used to record scheduling information and the indices to the CSs in the SCU. Figure 5.3 shows a flow chart of the hardware context switch technique. The circuit is composed of three main partially overlapping sets of components utilized for context switching: multitasking support, thread migration and CS pointer switch shown in Figures 5.4, 5.6 and 5.7 respectively. The circuits in each figure show four cores (n=4) and eight CSs (m=8) but these can be extended for reasonable m and n, where $n \leq m$.

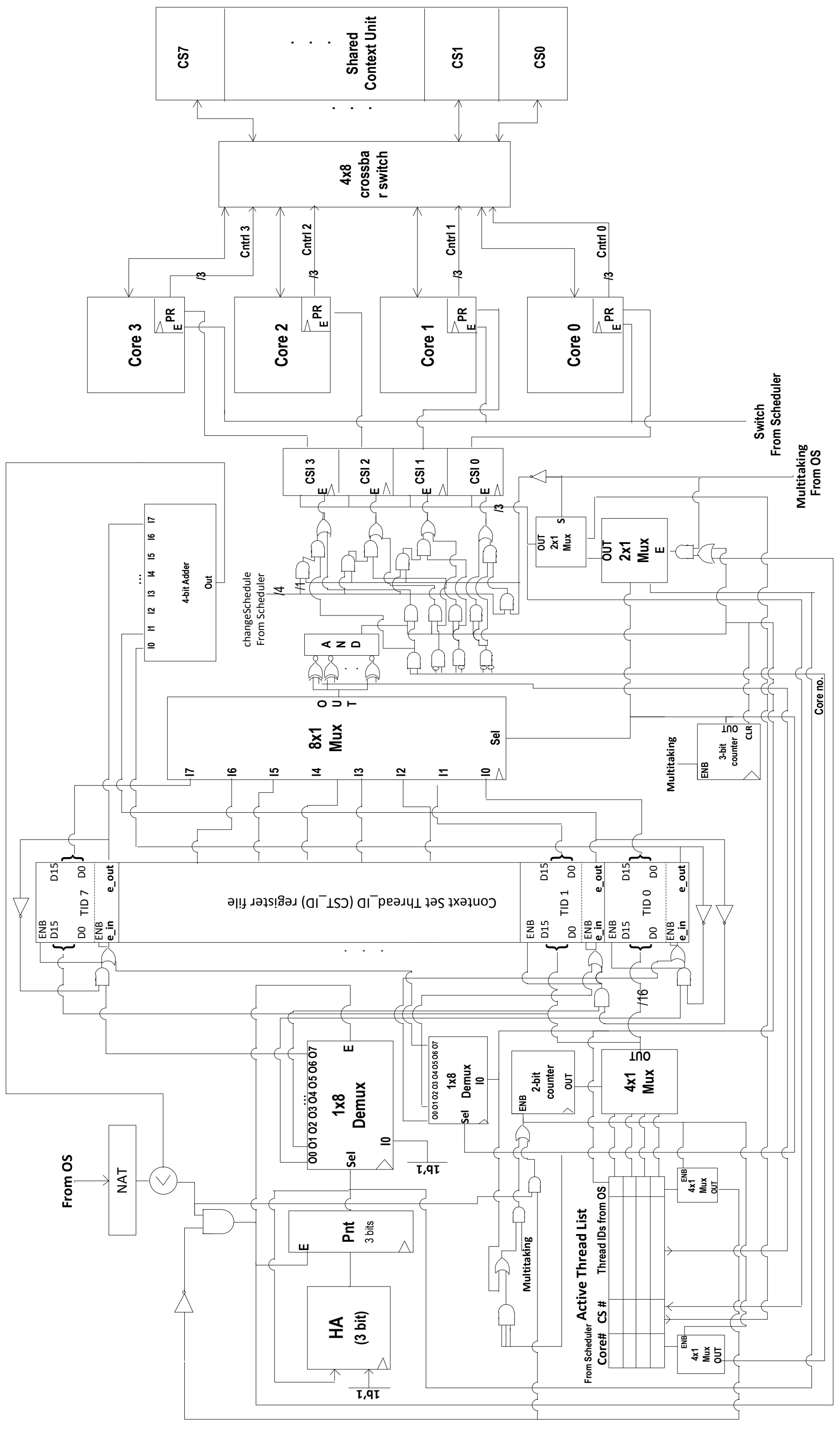


Figure 5.2: Thread context switching circuit.

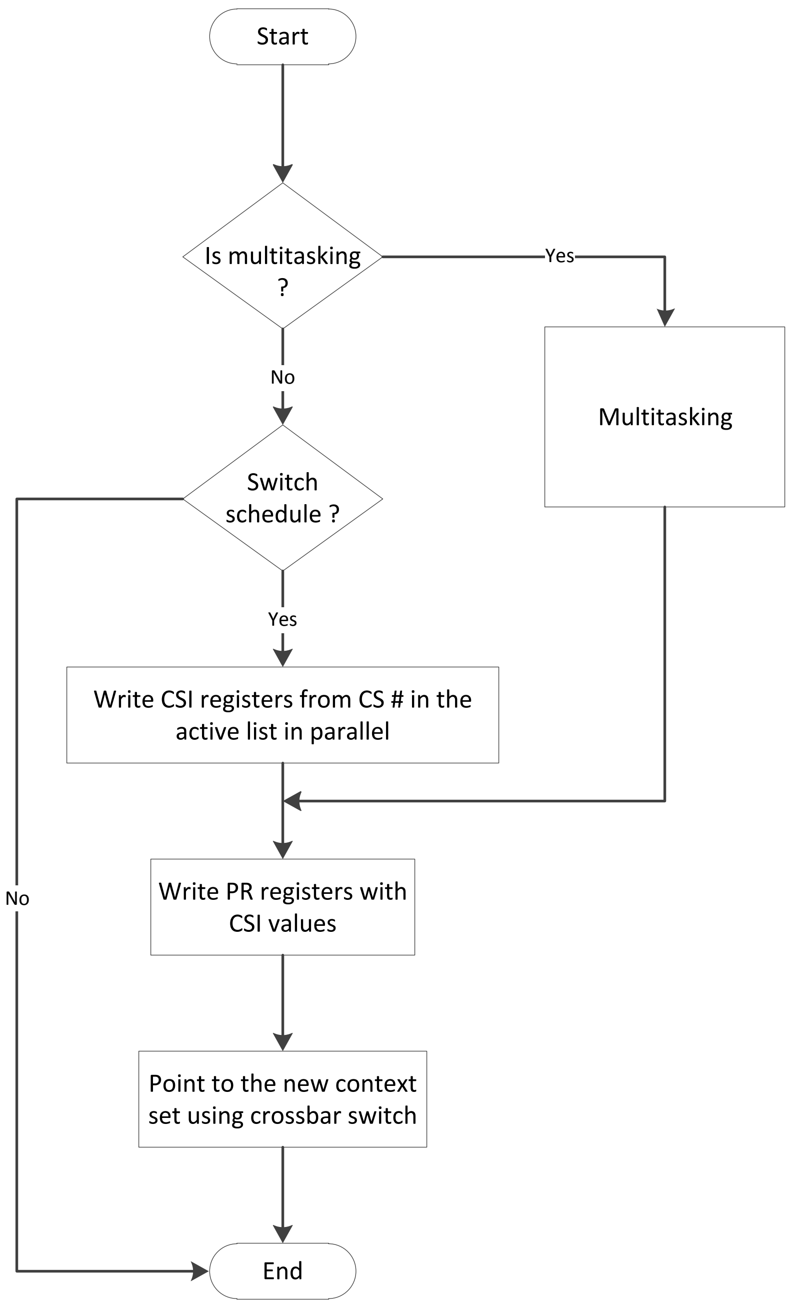

Start
Is multitasking ?
Yes
Multitasking
No
Switch schedule ?
Yes
Write CSI registers from CS # in the active list in parallel
No
Write PR registers with CSI values
Point to the new context set using crossbar switch
End

Figure 5.3: Hardware context switching flow chart.

Multitasking Support

In the first set of components, multitasking support, shown in Figure 5.4, the OS assigns IDs of threads that are scheduled to run during the next time interval to an *active-thread list* (ATL). The number of entries in the ATL is the same as the number of processing cores (n). Along with the thread ID for each entry, a $\lceil \log_2 m \rceil$-bit context set index, a $\lceil \log_2 n \rceil$-bit core index and an *exist* bit are present for each active thread. The context set index is written after each thread is assigned a CS. The scheduler inserts the core number that each active thread is assigned. The *exist* bit is reset before multitasking processing is started and set whenever the thread, associated with the same entry in the ATL is found in one of the CSs. Otherwise that thread is assigned a new CS in the SCU. A *context_set_thread_ID* (CST_ID) register file contains a number of registers equal to the number of CSs (m) in the SCU. Each register is composed of a 16-bit thread ID and a 1-bit *exist* flag (e). Each register corresponds to one CS in the SCU containing the context (state) of that thread. The CST_ID register file is sequentially searched for a thread ID match from the thread IDs in the ATL. A $\lceil \log_2 n \rceil$-bit counter (within the multitasking support control in Figure 5.4) and an nx1 multiplexer is used to select one thread ID in the active list every 1 to m cycles (depending on the time required to search for that ID as subsequently explained). Another $\lceil \log_2 m \rceil$-bit counter is used to point to the current CST_ID register as the CST_ID register file is searched for a match with the thread_ID from the ATL. This counter is reset once IDs match. Each thread ID in the ATL is compared to each thread ID in the register file (one per cycle) until the thread ID is found or all registers are searched. If the thread ID is found, the *exist* bit is set to one in both the CST_ID register and the corresponding entry in the ATL, and the $\lceil \log_2 n \rceil$-bit counter is enabled. The next thread_ID is then searched. Figure 5.5 shows a flow chart for the multitasking step.

The process of searching all thread IDs takes between (1+2+...+l) and mxl cycles, where l is the number of active threads ($l \leq n$). After all thread IDs are searched, the number of the *exist* bits in the register file that are set is calculated using an $\lceil m/2 \rceil$-bit full adder. The $\lceil \log_2 n \rceil$ least significant bits of the summation are compared to the value of the *number of active threads* register (NAT). If the result is less than the NAT value, this means that some threads do not exist in the SCU. Each thread that does not exist in the SCU is then assigned an available CS in the SCU and a corresponding spot in the CST_ID register file. The CSs in the SCU are assigned round-robin to insure fair utilization of all sets.

Thread Migration

Hardware support for thread migration, shown in Figure 5.6, is divided into two scenarios, migration due to multitasking and migration due to dynamic thread reassignment to adapt to the dynamic changes in workload behavior (e.g. to increase performance or reduce power consumption in an HMP [67]). In the case of multitasking, after all threads are assigned a CS from the SCU pool, the scheduler stores the new assignment of active threads to the processing cores in the ATL. Thread IDs in the CST_ID register file are compared to thread IDs in the ATL. The indices to the CSs of active threads are then recorded in n *context set index* (CSI) registers, shown in Figure 5.6, each of ($\log_2 m$) bits size. Each CSI register corresponds to one of the n cores. In the second scenario,

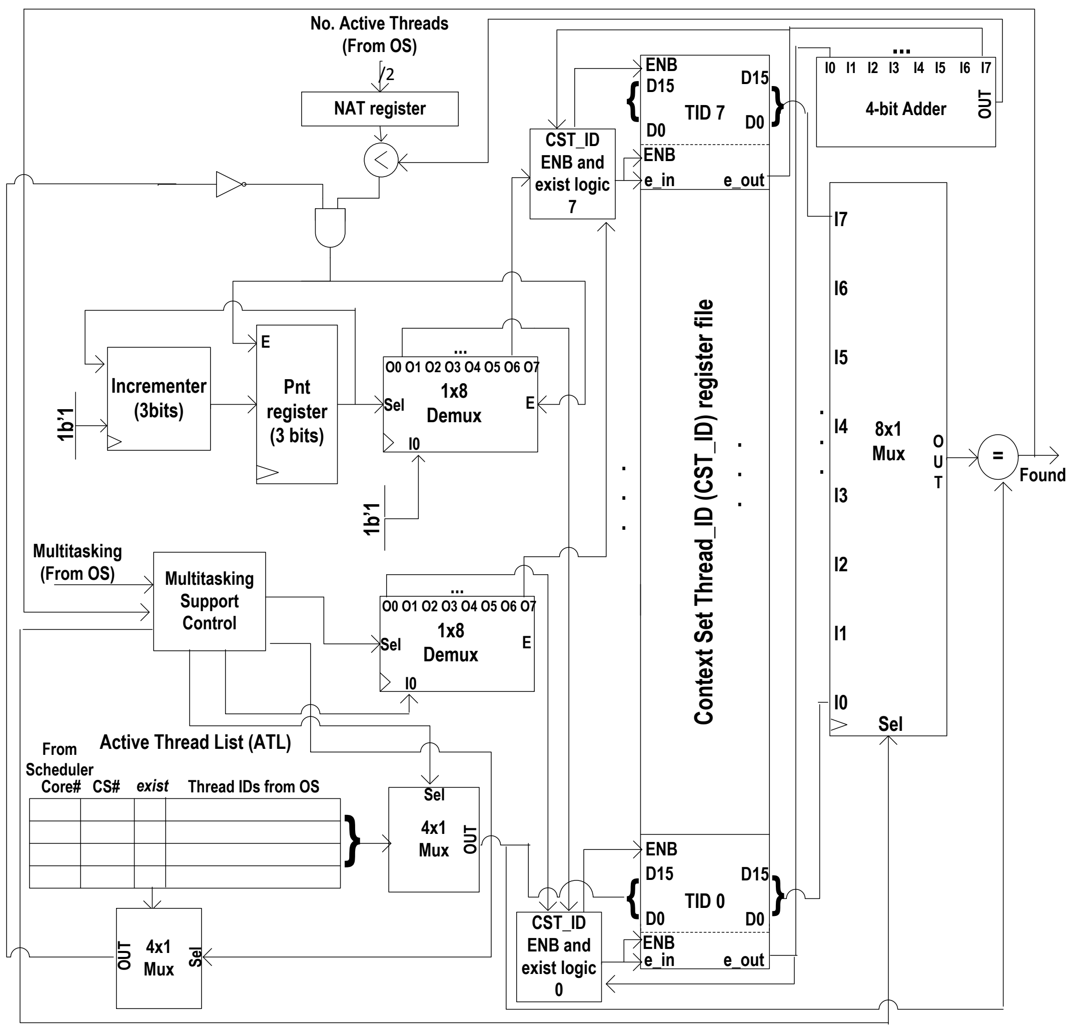


Figure 5.4: Components for multitasking support (n=4, m=8).

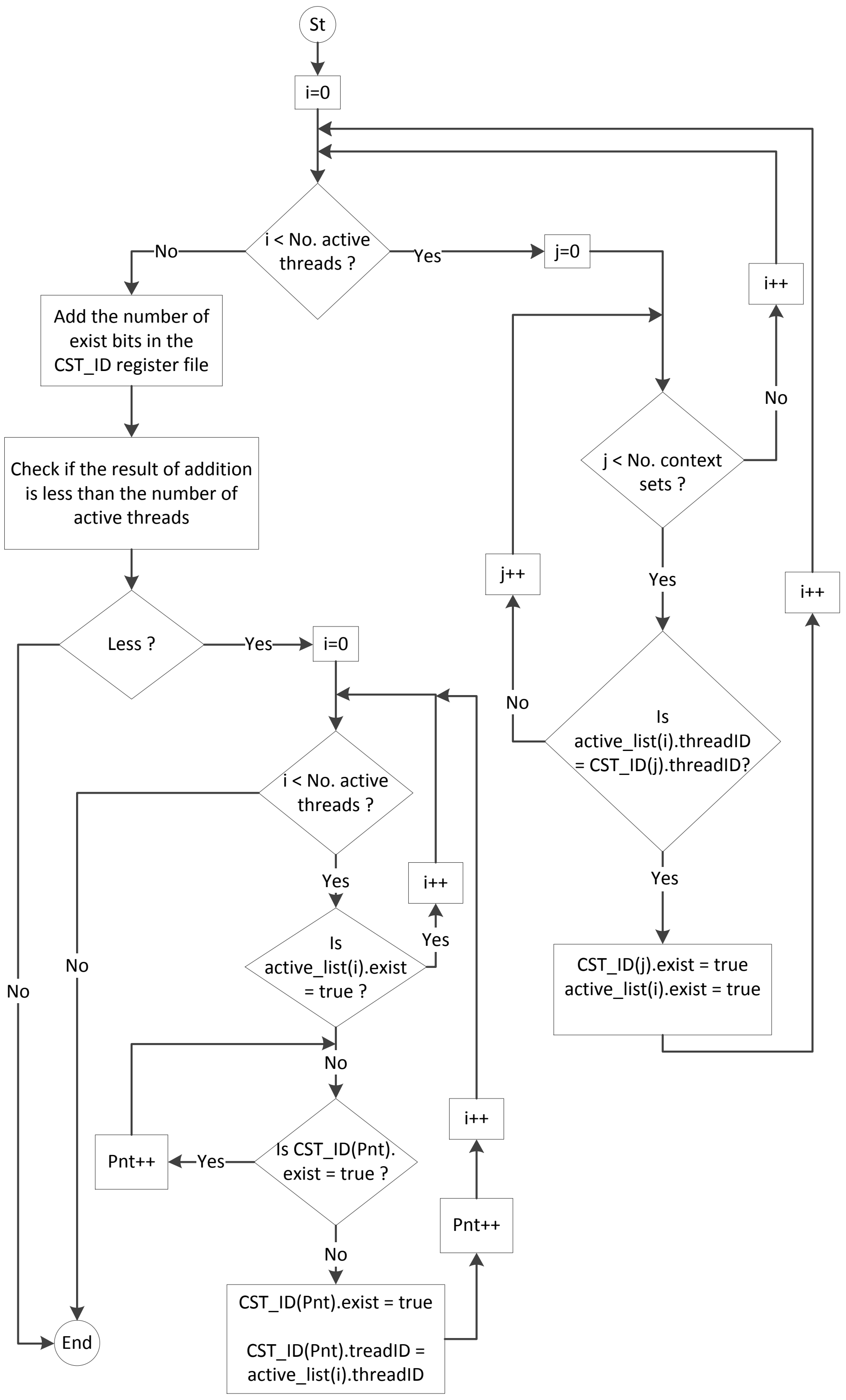

St
i=0
i < No. active threads ?
No
Yes
j=0
Add the number of exist bits in the CST_ID register file
i++
No
j < No. context sets ?
Check if the result of addition is less than the number of active threads
Yes
j++
i++
Less ?
Yes
i=0
No
Is active_list(i).threadID = CST_ID(j).threadID?
i < No. active threads ?
Yes
i++
Yes
Yes
Is active_list(i).exist = true ?
CST_ID(j).exist = true
active_list(i).exist = true
No
No
No
i++
Pnt++
Yes
Is CST_ID(Pnt). exist = true ?
Pnt++
No
End
CST_ID(Pnt).exist = true
CST_ID(Pnt).treadID = active_list(i).threadID

Figure 5.5: Hardware context switching Multitasking support step flow chart.

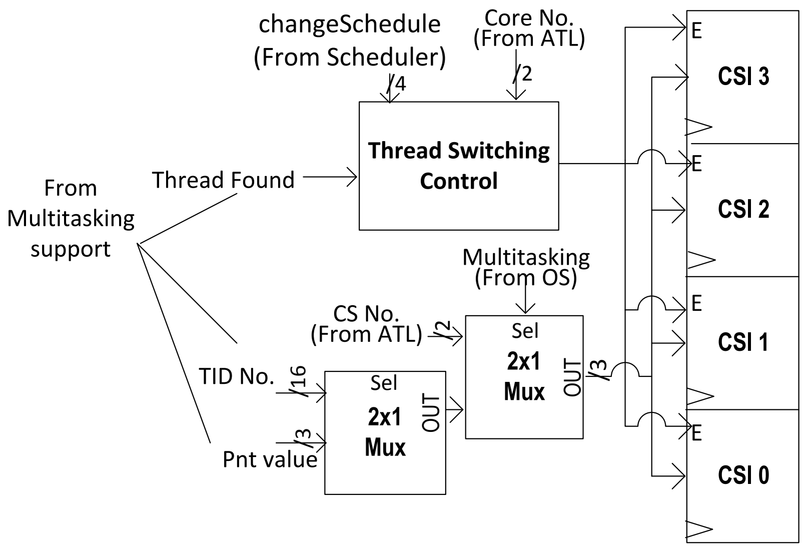


Figure 5.6: Components for thread switching (n=4, m=8).

if the scheduler changes the assignment of the same threads that are currently executing, it writes the new core numbers for each thread ID in the ATL and asserts *changeSchedule* signals for each core that is switching to another thread. Next, the CSs' indices in the CSI registers are updated using the information recorded in the ATL in parallel.

Context Set Pointer Switch

After all pipelines that are switching to different threads are flushed (or drained), a *switch* signal is set. Each core receives a new index to the proper CS for a new thread to execute. The index to the CS is stored in a $\lceil \log_2 m \rceil$ bit *pointer register* (PR) as shown in Figure 5.7. Each PR register is updated with a new value from its corresponding register in the CSI register file after the *switch* signal is set. Next, all cores are assigned to point to their proper CSs for the applications/threads they are executing through an $n \times m$ crossbar switch. The crossbar switch is capable of connecting all n cores to any n CSs out of the m

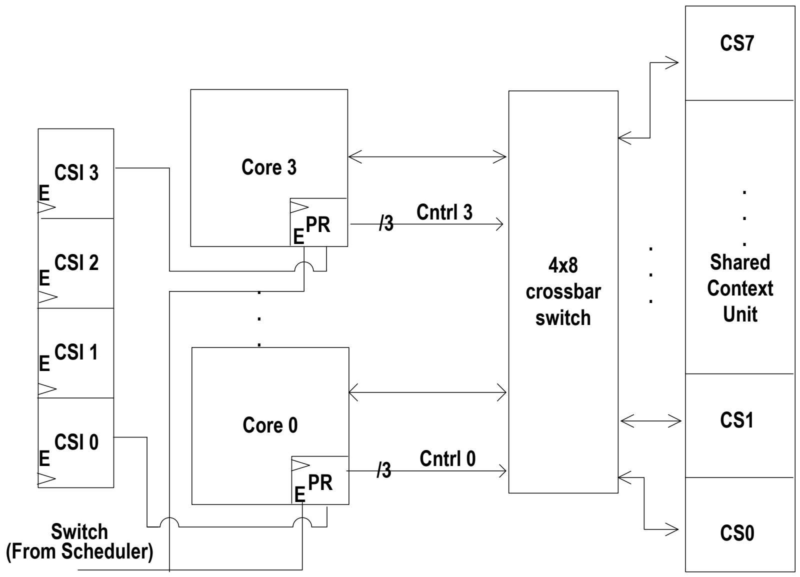


Figure 5.7: Context set pointer switching components (n=4, m=8).

sets in the context unit simultaneously. Because a cluster of cores is assumed to contain only a few cores sharing a single SCU, the cost of the crossbar switch is limited. Each core points to only one CS at a time but is capable of accessing any of the CSs through the crossbar switch.

5.4 Evaluation

Each CS in the SCU contains a copy of the processor context including the basic execution environment (584 bytes for a x86 64-bit processor), debug, control and design-specific registers, etc. ($\sim$1000 bytes). Using CACTI [52], a CS's area was estimated as 1971 nm^2 for a 32 nm process with each 8-byte access consuming 1.4 pJ. This fast switching mechanism results in approximately 1 380X improvement for migrating the same executing threads between cores and 407X improvement for migrating threads with multitasking support (time-multiplexing) for a quad-core system with eight context sets over the traditional

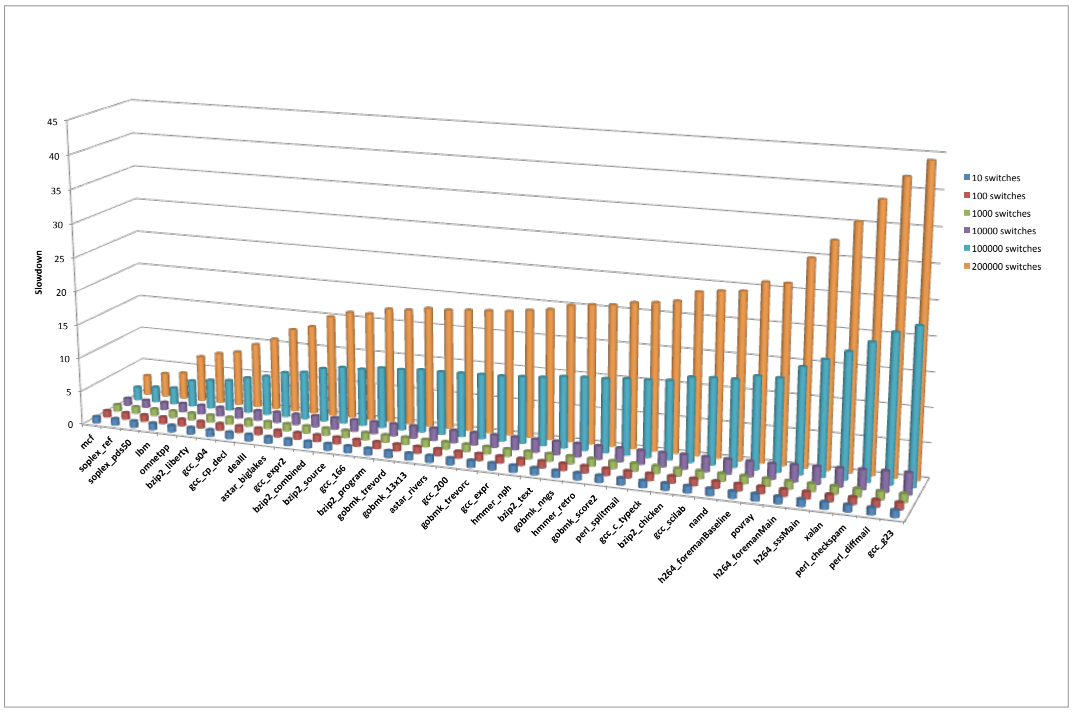


Figure 5.8: Slowdown of executed applications due to the direct cost of switching only.

operating system switch mechanism.

For frequent thread migration, the direct cost of switching results in a significant slowdown of applications. Figure 5.8 shows the slowdown of 300 Million instruction for various applications using different number of context switches. With 1000 switches, the average percentage slowdown is 9%, however with more switches the slowdown in performance grows from 2X for 10 000 switches to 19.5X for 200 000 switches on average. This means, that the direct cost alone makes it almost impossible to exploit frequent switching and thus fine-grained scheduling.

Chapter 6

Application Scheduling for Many-Core Processors

Previous chapters focused on heterogeneous multicore processors (HMPs), in which each processor contains a small number of cores. The described sampling/predictions approaches can exploit such processors to improve performance (*Phase_IPC* and *Phase_Sampling*) or reduce energy delay product (*Phase _EDP*). Through fine-grained phase-based approaches, the aforementioned scheduling techniques works efficiently for all lengths of benchmarks (small, medium, or long). However, when the number of cores grows, the requirements to support these algorithms grows linearly. For instance, the scheduler requires a larger signature history table (SHT) table, which is used to record all signature of all executing applications, to achieve the same performance of multicore processors. Unfortunately, the SHT is not free, a larger sized SHT is costly: area and power. This chapter proposes a new approach to exploit fine-grained scheduling for heterogeneous many-core processors (HMCPs) containing ten's, hundreds, or even thousands of processing cores on the same chip.

6.1 Scheduling for HMCPs using Machine Learning

Previous methods described in this study used a signature vector to represent the history of program behaviors. Because using an SHT table would be infeasible for many-core processors with hundreds and thousands of cores on a chip, more systematic ways to detect and represent program behaviors is desired such as machine learning techniques. Machine learning is an artificial intelligence method that provides computers the ability to learn without being programmed and make predictions based on new data [62]. Machine learning has been used for different computer problems, including scheduling. Berral et al. [8] proposed a machine learning approach for an scheduling in data centers targeting energy-efficiency. There are several machine learning algorithms, specifically reinforcement learning (RL) algorithm is used in this dissertation. Reinforcement learning has been used for resource allocation problems [28, 54, 78, 80–82]. McGovern et al. [47] used RL to build a basic block instruction scheduler, which produces higher performance than already available commercial schedulers. Ipek et al. [36] proposed an RL-memory scheduler for memory controllers. Their results showed that the controller significantly improves the performance of parallel applications on chip multiprocessors through optimized DRAM bandwidth utilization. It also showed that RL-based memory scheduling is feasible for hardware implementation. Federova et al. [26] proposed an RL solution to heterogeneous multicore scheduling by providing only some theoretical analysis. However, they did not implement the algorithm.

6.1.1 RL-Based Fine-Grained Scheduler

Having many cores on a processor complicates phase identification methods such as working set signatures because of the tremendous amount of behavior that must be detected on such systems with hundreds of cores. Previous approaches presented in this dissertation shows correlates between the execution phases of programs and program behavior. The performance of an application is also highly correlated to the application's behavior and its phases of execution. Figures 3.4 and 3.5 shown in Chapter 3 demonstrates how *xalan* features such as L2 misses are highly correlated to system performance or IPC. This was also observed for other system features such as L1 cache misses and branch misspredictions. Similar results were observed for all other benchmarks. For many applications running on an HMCP, an even-more compressed representation of program behavior than *Phase_IPC* is desired. RL algorithms can provide a compressed representation of history of applications running on the different types of cores. RL is composed of four main components: A learning agent, an environment, reward and selected features as shown in Figure 6.1. The learning agent is the scheduler, the environment is an HMCP with features are the monitored behavior of applications on the different core types and the reward. Through maximizing long term rewards, RL agents can find a near-global optimization policy. Q-learning is an RL technique that was chosen for an initial study of scheduling in HMPs. Through interaction with the HMCP environment, Q-learning can find the optimal policy for mapping threads to the different types of cores. States of the system and actions are paired such that each state-action pair is assigned a Q-value, which represents the expected reward for that state-action pair. Equation 6.1 shows how this Q-value is updated used the old Q-value, α the learning rate, γ the discount factor and the next maximum expected reward. In each state, the agent compares the Q-value of all the available state-action pairs. In the typical greedy step, the agent chooses the next action, the one that has the highest Q-value for the current. However, to explore different options and speed the learning process, a small percentage of actions consist of exploration steps. This means that, in order for the agent to learn a small fraction of the time an action other than the action with the highest Q-value for the current state, it tries a random action. After each action a reward (r) is assigned based on the number of executed instructions per cycle. The goal of this approach is to maximize the global reward in terms of weighted speedup computed similar to [42] in that the weighted speedup is calculated by dividing on the IPC of the application running separately on the system, requiring prior profiling.

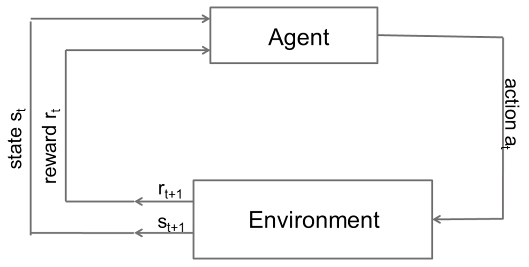


Figure 6.1: Reinforcement learning.

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{6.1}$$

Scheduling Agent

The heterogeneous nature of applications and the different types of cores on a chip causes a gross behavior of the system. Thus, the HMCP behavior is considered continuous. To represent this continuous behavior, artificial neural networks are used (ANNs). ANN is a mathematical model that is used to model complicated relationships between system features and its output [59]. It consists of a set of computational nodes called neurons that receive input and produce output. Artificial neurons are inspired by the biological neurons. ANNs support non-linear relationships between inputs and outputs. A fully connected network, with hidden layers of artificial neurons along with easily obtained system features (behaviors) and appropriate reward (weighted speedup), updates the expected reward for each available action. Thus, the reinforcement learning agent can efficiently assign threads to the dissimilar types of cores in HMCPs.

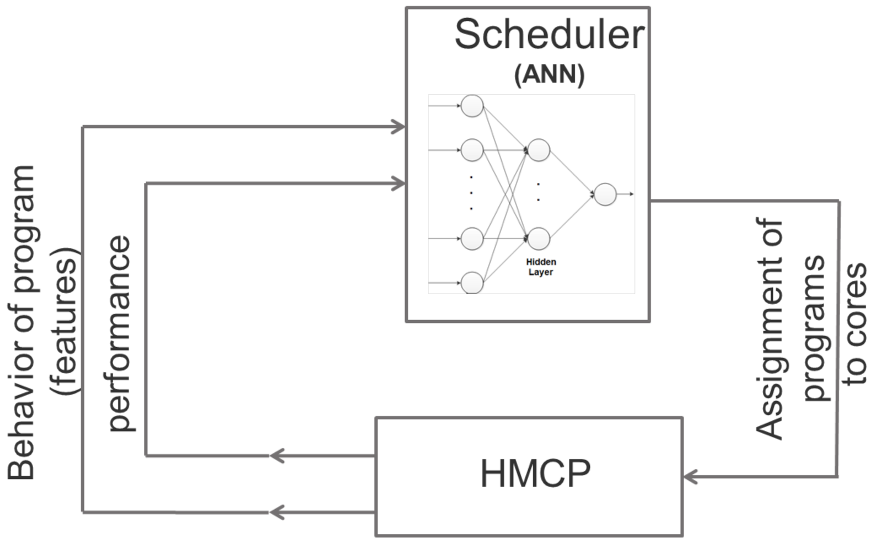

Scheduler
(ANN)
Hidden Layer
Behavior of program (features)
performance
Assignment of programs to cores
HMCP

Figure 6.2: RL-based HMCP scheduler.

HMCP Features and Actions

The HMCP system features represent the states of the environment gathered during execution. Actions in the HMCP scheduling system represent the possible thread-to-core assignments. For an initial study of this approach, nineteen different architectural and performance evaluation features are associated with each core. The nineteen different features are: percentage breakdown of executed instruction types (between load, store, multimedia, basic floating point, floating point multiplication, floating point division, basic integer, integer multiplication and integer division), percentage of total executed instructions that are L1 and L2 cache hits/misses, and percentage of correctly and incorrectly predictor branch instructions over total executed instructions. ANNs use non-linear function approximation to represent relationship between state and expected reward for each action for the current. The number of ANNs that are used in this study depends on the number of actions that are available. In this work, four actions are used for a quad-core system containing one out-of-order (OoO), core 3, and three in-order (IO), core 0, from Table 4.4. Each ANN represents the relationship between state and action pair. In each system, all ANNs have the same structure. In this study, there are three layers in each ANN including one input layer, one hidden layer and one output layer. The first layer, input layer, contains the same number of neurons as the number of system features. The second layer is a hidden layer that contains half the number of neurons as the input layer. The third layer, output layer, contains only one output neuron that represents the Q-value (expected reward) for each ANN. Because Q-value is estimated from system features through an ANN, updating the Q-value during learning is performed through back-propagation in the ANN. In this way, ANNs using non-linear functions are used by reinforcement learning method to update the expected rewards of state-action pairs.

For every state, the available actions are the same: one of the four applications is scheduled to run on the OoO core and all others run on the IO cores. Every 10 000 instruction intervals (episodes), the system sends state features to the RL agent. All ANNs receive these continuous features to estimate the expected reward for each available action. As described earlier, scheduling consists of two steps: greedy steps and exploration steps. In a greedy step, the action with the largest expected weighted speedup, which is the output of ANN, is taken. For a small fraction of the actions, the scheduler agent takes an exploration step in which an action is randomly selected from available actions. In each episode, the weighted speedup is calculated for that window and used as the actual reward. The ANNs are updated using back-propagation according to this actual reward. The RL-based agent thereby learns towards an optimal scheduling policy during the execution of applications.

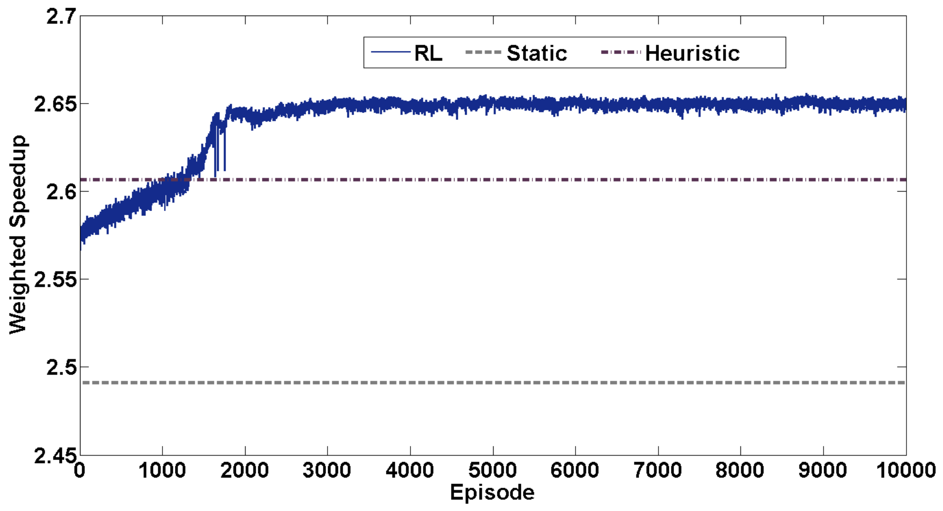
6.2 Evaluation and Results

Initial evaluations are performed using the performance estimation model described in Chapter 3. Two types of evaluations are performed: on-line learning and off-line training. On-line learning is the process of teaching the scheduling agent the best policy to map threads to the different types of cores on-line while applications are running. Off-line training is the process of teaching the agent (training it) off-line how to map threads to cores. The RL-based agent is given ample time to learn a scheduling policy off-line. In Each set of benchmarks was run repeatedly through for 10 000 times to make sure a complete learning curve is generated. The experiments were performed on a quad-core processor for a fast evaluations of RL-based scheduling techniques for HMCPs and can be easily extended.

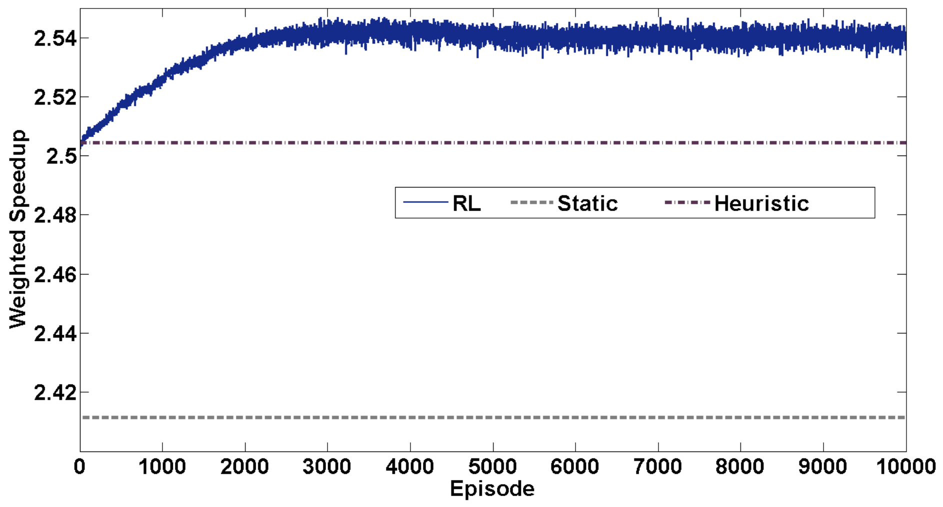
To demonstrate how the scheduling agent's policy improves during the iterative learning process, 300 million instruction regions of each benchmark in each set 10 000 times were ran. The average weighted speedup for all executed windows all of benchmarks are recorded. Iterative learning results are shown in full learning curves shown in Figure 6.3. Figure 6.3(a) shows a full learning curve for four benchmarks on a quad-core HMP. The learning results are better than both fine-tuned heuristic sampling [41] and the average of all static assignments. Figure 6.3(b) shows the average learning curve for fifteen randomly picked benchmark combinations. Figure 6.5 shows a comparison between trained learning results, heuristic results and static results. Trained learning results are taken from the level-off portion of learning curve. For all fifteen benchmark combinations, the trained learning results are significantly better than the other scheduling results. Trained learning results in 1.77% improvement on average over the fine-tuned heuristic sampling results (ignoring the cost of context switching) and 6% improvement over the average of all static assignment.

6.2.1 On-line Learning

Using on-line learning, twelve four-tuple combination of applications were run once and the cumulative mean of the IPC is recorded for each window. Figure 6.4(a) represents on-line learning results for four benchmarks running on the quad-core system for twelve different combinations. The accumulated mean of the weighted speedup is recorded for each window. The weighted speedup results of the on-line learning scheduler is compared to that of the fine-tuned heuristic sampling method. The results shows that the RL agent learns quickly to find a better schedule than heuristic method even at an early stage. Figure 6.4(b) shows the average learning process in one complete run of each of the twelve four-tuple combinations of benchmarks on the quad-core system. This figure shows that RL agent again quickly learns to outperform the fine-



(a) *poveray, soplex, xalan, astar*



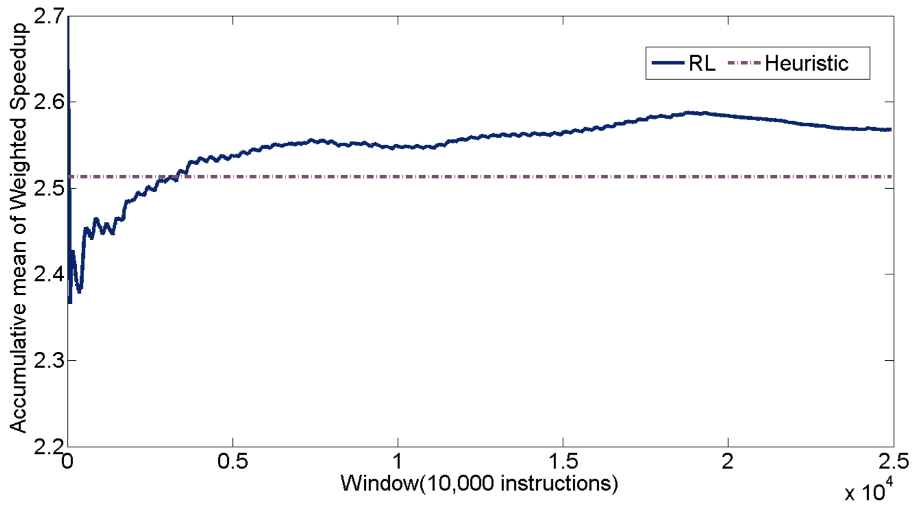
(b) *average*

Figure 6.3: Full learning curves from quad-core system compared with static and heuristic assignments.

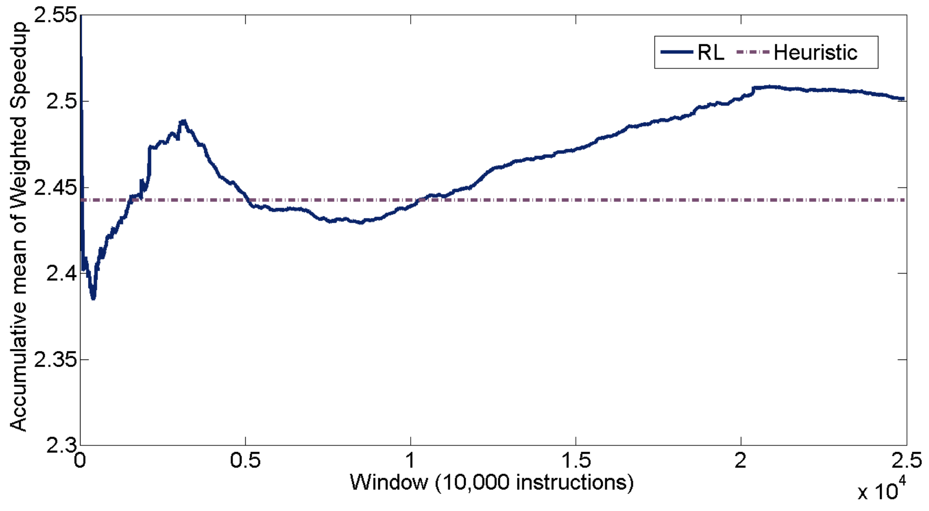
tuned heuristic-sampling-based scheduler in early stage of program execution and eventually learns a good policy to assign applications to the different types of cores. Figure 6.6 represents the comparison between on-line learning results and the fine-tuned heuristic sampling results. on-line learning results are taken from the last point of on-line learning curve. The weighted speedups for on-line learning are better than those for heuristic sampling for most benchmark combinations. On average, the on-line learning result is 2.4% better than heuristic results (ignoring switching cost). Real world applications are usually longer than the portions of benchmarks simulated, thus the reinforcement-learning-based scheduler is capable of optimizing such applications efficiently and transparently.

6.2.2 Off-line learning

Off-line learning is a way for training the scheduler to learn the weight of each feature that is used to decide an action. Off-line learning can be used to set the scheduler to choose the best assignment depending on the previously learned policy. This also reduces the complexity of hardware implementations of such schedulers. During off-line learning, rather than training the RL-based scheduler on individual benchmark combinations, the scheduler was trained using twelve benchmark combinations sequentially. In this case, the scheduler can see more states than what it actually sees during individual benchmarks training. An RL-based agent learns during training a general scheduling policy that can be used for all sets of applications. An RL-based scheduler can be trained off-line and results of training can be applied directly for a global scheduling policy. In off-line-trained systems, the choice of the training set is very im-



(a) *poveray, soplex, xalan, astar*



(b) *average*

Figure 6.4: on-line learning curve from quad-core system compared to heuristic assignments.

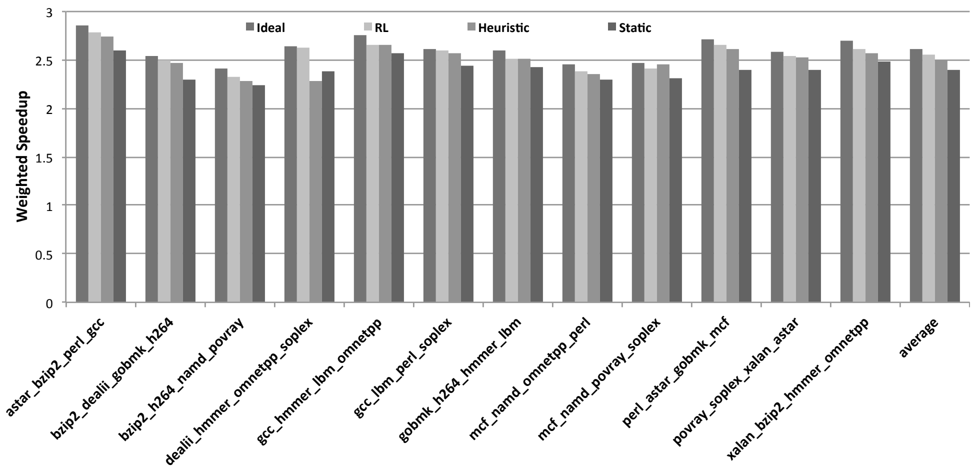


Figure 6.5: Comparison between learning, static, and heuristic results in the quad-core system.

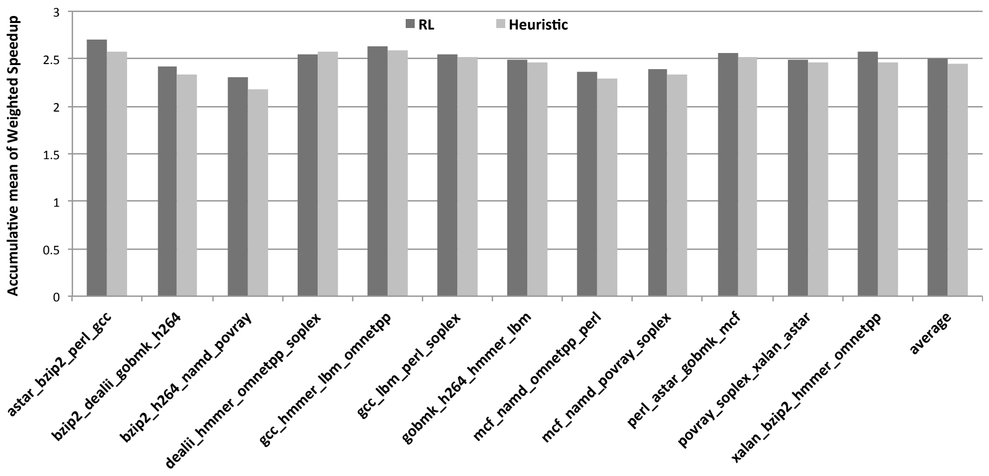
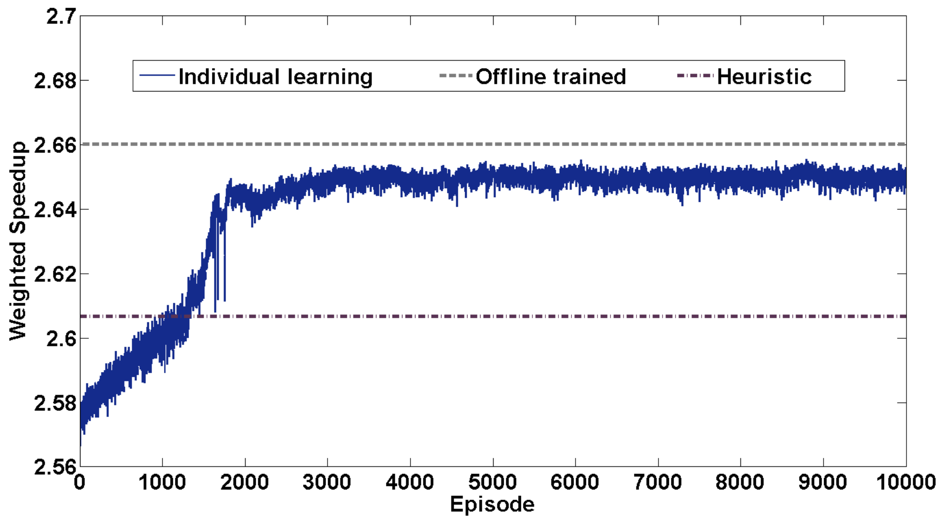
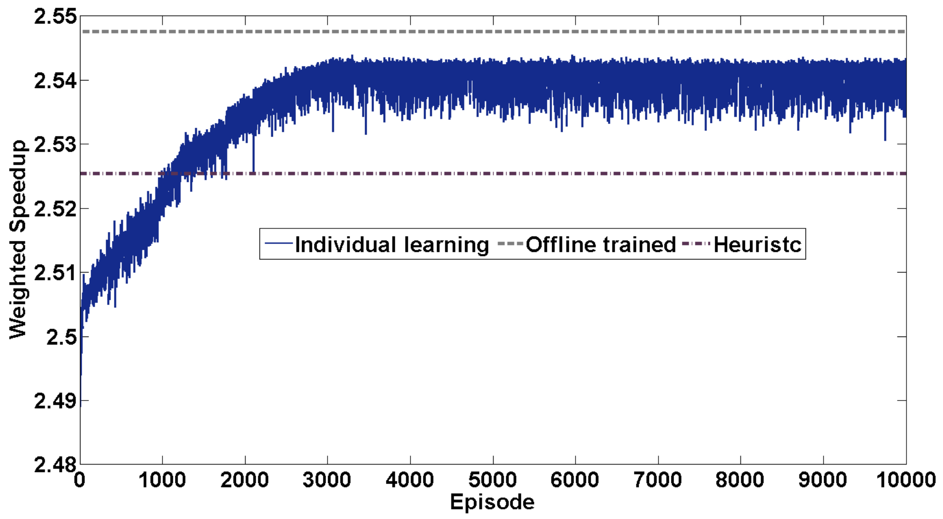


Figure 6.6: Comparison between on-line learning and heuristic results in the quad-core system.

portant to make a global scheduling policy that improves system performance. In this study, the off-line-trained scheduler only works well for some of the benchmark combinations. This is because the training set that was used did not include all the states (behaviors) of applications that other applications encounter. Figure 6.7 shows that the off-line scheduler generate results that are even better than the fully learned individual trained results in the best case. The results indicate that there is a potential for off-line learning, but that for such an off-line-training approach a training set that covers most of the states (features) need to be carefully studied and selected such that the global policy would work for all combination of applications.In all of the above evaluations of RL-based scheduler performed in this study, 19 features were used. However, more careful selection of only the most important features can be performed to reduce the hardware overhead for on-line learning schedulers.



(a) *perl, astar, gobmk, mcf*



(b) *poveray,soplex,xalan,astar*

Figure 6.7: Comparison between individual training, off-line trained, and sampling heuristic results in quad-core system.

Chapter 7

Conclusions and Future Work

This dissertation presented fine-grained thread scheduling combined with low-overhead thread migration for heterogeneous processors to maximize performance and reduce energy consumption. Two phase-based scheduling algorithms were proposed to benefit from the short changes in applications behavior and fully utilize heterogeneous resources on an HMP.

7.1 Summary

In this work, three scheduling approaches targeting performance and one approach targeting reduced energy consumption were proposed. The first approach, *Phase_IPC*, has a significant advantage over existing sampling based scheduling techniques int that it does not require examining the performance of permuting all application threads across each core type. Instead, for each program phase, the performance of a thread is evaluated once on each processor core type. While this results in schedules that achieve a slightly lower weighted speedup compared to more aggressive sampling based approaches, this approach requires far fewer performance evaluation intervals. This approach results in approximately 2% improvement in speedup and 12.5% reduction in execution time over previous sampling heuristic method, and 9.5% improvement in weighted speedup and 37.2% reduction in execution time over a random static assignment. The second approach, *Phase_Sampling* approach outperforms other evaluated approaches, but correspondingly requires the most sampling intervals in my evaluation. Since both of the presented approaches utilize the identification of program phases for the reuse of previously evaluated performance, it is likely that the amount of reuse would be larger over the entire application execution (sometimes trillions of instructions), thus eliminating some of the need for additional sampling. Even for *Phase_Sampling*, there is a good chance that as the length of each of application's run increases, the number of repeated phase sets will increase. By taking advantage of phase behavior, the presented approaches and future approaches have the potential to achieve high throughput and require minimal performance sampling.

Additionally, this dissertation demonstrated that like performance the energy efficiency of application codes running on cores of different types varies along with program execution phases. The (*Phase_EDP*) scheduling technique demonstrates the utility of phase identification for energy-delay-aware scheduling of applications on single-ISA HMPs. Unlike many previous approaches, the *Phase_EDP* technique does not require evaluating energy consumption (just as *Phase_IPC* does not require evaluation of the performance of each thread-to-core mapping) by permuting all application threads across each core type. Simulated evaluation of the *Phase_EDP* approach shows 16% on average and up to 29% reduction in energy-delay product compared to the average of EDP for all possible static assignments. At the same time, while a significant reduction in EDP is achieved, there is only a very slight reduction in program throughput.

To support fine-granularity scheduling a novel thread context switching circuit is also proposed. The circuit supports multitasking through additional context sets beyond the number of cores in the processor. It further supports migrating all executing threads on processor cores simultaneously, and supports fine-granularity thread scheduling by rapidly migrating the contexts of all executing threads simultaneously. Only two cycles are required to switch the executing threads among cores (in addition to the time required to flush processor pipelines). Additionally, the maximum number of cycles required for thread assignment due to multitasking is $m \times l + 2$, where l is the active number of threads and m is the number of CSs. This hardware design results in a large reduction in the latency of context switching among multiple cores, compared to tens of thousands of cycles for the direct cost of context switching by the OS. This fast switching mechanism results in approximately 1 380X improvement for migrating the same running threads between cores and 407X improvement for migrating threads with multitasking for a quad-core system with eight context sets over the traditional operating system switch method. Furthermore, this fast switching technique can support other migration-based systems such as load balancing, thermal and power management, manufacturing-fault tolerance and coherence protocols.

Finally, this dissertation demonstrated that a reinforcement learning algorithm is effective at scheduling for heterogeneous systems, exploiting some differences between fine-grained program phases. In the proposed online-trained approach the scheduler agent improves its decision policy over time and results in an increased performance without any prior offline training. Preliminary evaluations of off-line trained agent showed that such scheduler agent can be capable of choosing near optimal thread-to-core assignments. However, careful choice of the quantity and variety of training sets are crucial to achieve a

Table 7.1: Comparison between the proposed scheduling algorithm and heuristic sampling approach [42].

Metric	Phase_IPC	Phase_Sampling	Heuristic sampling [42]	RL-Based
Reschedule interval	small window size (short intervals)	short intervals	Long intervals	short intervals
Learning mechanism	yes	yes	no	yes
Previous info.	no	no	yes	yes
Phase detection	working set signature	working set signature	IPC change	Behavior change using RL
Thread life time	short, medium or long	Short, medium or long	medium or long	medium or long
Frequency of switching	high	high	low	learns
Number of sampling periods	short	short	long	exploration only
Number of cores on a chip	small	small	small	medium or large
Required training	no	no	no	yes

global optimization policy. Additional training sets need to be investigated to maximize the benefit from this scheduling technique. The RL-based scheduling mechanism, through its greedy policy and exploration of different policies, shows a great potential for scheduling on heterogeneous many-core processors in which combining both sampling and prediction such as the *Phase_IPC* may result in large overheads due to the amount of memory needed to record programs signatures in a hardware history table.

Table 7.1 summarizes four different scheduling techniques proposed in this dissertation. While the *Phase_IPC*, *Phase_EDP* and *Phase_sampling* methods work efficiently for short, medium and long runs, the on-line trained RL-based scheduling technique is more suited for long runs.

7.2 Future Work

- This work utilized multiprogram workloads consisting of single-threaded applications. While studying single-threaded applications behavior on the different types is important, it would also be interesting to consider a fully multithreaded, multiprogram workload. For parallel applications with relatively homogenous threads, it is likely that each of these threads should be mapped to cores of the same particular type. For other applications with functionally different parallel tasks, these differences can be exploited by mapping these tasks to different types of cores, each suited for a particular task. In either case, our approach can be modified to incorporate application throughput instead of single-thread instructions per cycle.

- To support fine-granularity scheduling and further exploit heterogeneous systems, fast and transparent thread migrations are desired. This dissertation provided a hardware solution for the direct cost of context switching that drastically minimized the direct overhead of switching. However, there is still some overhead due to the indirect cost resulted by warming up caches and branch predictors. To maximize the benefit from fine-grained scheduling on HMPs, a reduced indirect cost is desired. More work could be done on reducing the overhead of warming up data and instruction caches.

- With the increased number of transistors on a chip, future microprocessors are expected to include hundreds of cores on a chip. Many-core processors will a challenge for both designers and programmers. Heterogeneous many-core processors have the ability to support an increased number of processor cores on a chip and an increased variety of domain specific applications than multicore processors. Additionally, increased functionality failures and transistor parameter variation will cause different performance and power consumption in different cores of the same design adding heterogeneity that complicates the scheduling problem.

- As heterogeneous multicore processors replace our current homogeneous ones (due to their benefits in performance, power, and adaptability to program requirements and behavior) this will be a sea-change in the field of computing. The recent era of homogeneous multicore processors has already challenged programmers with task parallelization. Having different types of cores will multiply these challenges. ISA specialization will additionally provide a great opportunity for even further enhancement. However, different ISAs will make the job of the programmer that much more difficult. New models of computing are needed to make this feasible for programmers, including novel architectures and hidden dynamic binary (ISA-to-ISA) translation.

Bibliography

[1] Single-chip cloud computer. [Online]. Available: http://www.intel.com/content/www/us/en/research/intel-labs-single-chip-cloud-computer.html

[2] The story of the intel® 4004. [Online]. Available: http://www.intel.com/content/www/us/en/history/museum-story-of-intel-4004.html

[3] G. M. Amdahl, "Validity of the single-processor approach to achieving large-scale computing capabilities," in *Proceedings of the Spring Joint Computer Conference*, April 1967, pp. 483–485.

[4] *The ARM Cortex-A9 Processors*, 2nd ed., ARM, September 2009.

[5] C. Banino, O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert, "Scheduling strategies for master-slave tasking on heterogeneous processor platforms," *IEEE Transactions on On Parallel and Distributed Systems*, vol. 15, no. 4, pp. 319–330, April 2004.

[6] O. Beaumont, A. Legrand, and Y. Robert, "The master-slave paradigm with heterogeneous processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 2003, pp. 897–908, September 14.

[7] M. Becchi and P. Crowly, "Dynamic thread assignment on heterogeneous multiprocessor architectures," *Journal of Instruction-Level Parallelism*, vol. 10, June 2008.

[8] J. L. Berral, I. Goiri, R. Nou, F. Julia, J. Guitart, R. Gavalda, and J. Torres, "Towards energy-aware scheduling in data centers using machine learning," in *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking*, 2010, pp. 215–224.

[9] F. Blagojevic, X. Feng, K. Cameron, and D. Nikolopoulos, "Modeling multi-grain parallelism on heterogeneous multicore processors: A case study of the Cell BE," *High performance embedded architectures and compilers*, vol. 4917, pp. 38–52, 2008.

[10] G. Blake, R. G. Dreslinski, and T. Mudge, "A survey of multicore processors," *IEEE Signal Processing Magazine*, vol. 26, no. 6, pp. 26–37, November 2009.

[11] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: a framework for architectural-level power analysis and optimizations," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000, pp. 83–94.

[12] P. D. Bryan, J. A. Poovey, J. G. Beu, and T. M. Conte, "Accelerating multi-threaded application simulation through barrier-interval time-parallelism," in *IEEE 20th International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, 2012.

[13] B. Burgess, B. Cohen, M. Denman, J. Dundas, D. Kaplan, and J. Rupley, "Bobcat: AMD's low-power x86 processor," *IEEE Mirco*, vol. 31, no. 2, pp. 16–25, April 2011.

[14] G. Chanteperdrix and R. Cochran, "The ARM fast context switch extension for Linux," in *Proceedings of the 11th Real-Time Linux Workshop*, September 2009, pp. 255–262.

[15] J. Chen and L. K. John, "Energy-aware application scheduling on a heterogeneous multi-core system," in *IEEE International Symposium on Workload Characterization*, September 2008, pp. 5–13.

[16] ——, "Efficient program scheduling for heterogeneous multi-core processors," in *Proceedings of the 46th Annual Design Automation Conference*, July 2009, pp. 927–930.

[17] M. H. Cho, K. S. Shim, M. Lis, O. Khan, and S. Devadas, "Deadlock-free fine-grained thread migration," in *Proceedings of the Fifth ACM/IEEE International Symposium on Networks-on-Chip*, May 2011, pp. 33–40.

[18] M. Co and K. Skadron, "The effects of context switching on branch predictor performance," in *IEEE International Symposium on Performance Analysis of Systems and Software*, 2001, pp. 77–84.

[19] A. K. Coşkun, T. Š. Rosing, K. A. Whisnant, and K. G. Gross, "Static and dynamic temperature-aware scheduling for multiprocessor SoCs," *IEEE Transactions on Very Large Scale Intergration (VLSI) Systems*, vol. 16, no. 9, pp. 1127–1140, September 2008.

[20] J. Cong and B. Yuan, “Energy efficient scheduling on heterogeneous multi-core architectures,” in *International Symposium on Low Power Electronics and Design*, July 2012.

[21] T. Constantinou, Y. Sazeides, P. Michaud, D. Fetis, and A. Seznec, “Performance implications of single thread migration on a chip multi-core,” *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 80–91, November 2005.

[22] F. M. David, J. C. Carlyle, and R. H. Campbell, “Context switch overheads for linux on arm platforms,” in *Proceedings of the 2007 workshop on Experimental computer science*, ser. ExpCS ’07. New York, NY, USA: ACM, 2007.

[23] A. S. Dhodapkar and J. E. Smith, “Managing multi-configuration hardware via dynamic working set analysis,” *ACM SIGARCH Computer Architecture News*, vol. 30, no. 2, pp. 233–244, May 2002.

[24] S. Eggers, J. Emer, H. Levy, J. Lo, R. Stamm, and D. Tullsen, “Simultaneous multithreading: A platform for next-generation processors,” *IEEE Micro*, vol. 17, no. 5, pp. 12–19, Sept.-Oct. 1997.

[25] J. Emer, M. D. Hill, Y. N. Patt, J. J. Yi, D. Chiou, and R. Sendag, “Single-threaded vs. multithreaded: Where should we focus?” *IEEE Micro*, vol. 27, no. 6, pp. 14–24, November-December 2007.

[26] A. Fedorova, D. Vengerov, and D. Doucette, “Operating system scheduling on heterogeneous core systems,” in *Proceedings of the First Workshop on Operating System Support for Heterogeneous Multicore Architectures*, 2007.

[27] N. Freeman, “Soonergy: A pluggable, cycle-accurate computer architecture simulator,” Master’s thesis, The University of Oklahoma, Norman, Oklahoma, May 2011.

[28] A. Galstyan, K. Czajkowski, and K. Lerman, “Resource allocation in the grid using reinforcement learning,” in *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, vol. 3, July 2004, pp. 1314–1315.

[29] C. Goh, E. Teoh, and K. Tan, "A hybrid evolutionary approach for heterogeneous multiprocessor scheduling," *Soft Computing*, vol. 13, no. 8-9, pp. 833–846, March 2009.

[30] P. Greenhalgh, "Big.LITTLE processing with ARM $cortex^{TM}$-A15 & cortex-A7," ARM Limited, Tech. Rep., September 2011.

[31] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki, "Synergistic processing in Cell's multicore architecture," *IEEE Micro*, vol. 26, no. 2, pp. 10–24, March 2006.

[32] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, July 2008.

[33] K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, 1st ed. McGraw-Hill, 1993.

[34] W. W. Hwu and Y. N. Patt, "Checkpoint repair for high-performance out-of-order execution," *Transactions on Computers*, vol. C-36, no. 12, pp. 1496–1514, Dec. 1987.

[35] *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2B: Instruction Set Reference, N-Z*, Intel, November 2008.

[36] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," in *Proceedings of the 35th Annual International Symposium on Computer Architechture*, June 2008, pp. 39–50.

[37] A. Jooya, A. Baniasadi, and M. Analoui, "History-aware, resource-based dynamic scheduling for heterogeneous multi-core processors," in *CMP-MSI: 3rd Workshop on Chip Multiprocessor Memory Systems and Interconnects in conjunction with ISCA'09*, June 2009.

[38] O. Khan, M. Lis, Y. Sinangil, and S. Devadas, "DCC: A dependable cache coherence multicore architecture," *Computer Architecture Letters*, vol. 10, no. 1, pp. 12–15, 2011.

[39] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagra: A 32-way multithreaded SPARC processor," *IEEE Micro Magazine*, vol. 25, no. 2, pp. 21–29, March 2005.

[40] D. Koufaty, D. Reddy, and S. Hahn, "Bias scheduling in heterogeneous multi-core architectures," in *Proceedings of the 5th European Conference on Computer Systems*, April 2010, pp. 125–138.

[41] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, June 2003, pp. 81–92.

[42] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas, "Single-ISA heterogeneous multi-core architectures for multithreaded workload performance," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, June 2004, pp. 64–75.

[43] N. Lakshminarayana, J. Lee, and H. Kim, "Age based scheduling for asymmetric multiprocessors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, November 2009.

[44] D. C. Lee, P. J. Crowley, J. Baer, T. E. Anderson, and B. N. Bershad, "Execution characteristics of desktop applications on windows NT," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, July 1998, pp. 27–38.

[45] C. Li, C. Ding, and K. Shen, "Quantifying the cost of context switch," in *Proceedings of the 2007 Workshop on Experimental Computer Science*, June 2007.

[46] C. Luk, S. Hong, and H. Kim, "Qilin: Exploiting parallelism on heterogeneos multiprocessors with adaptive mapping," in *42nd Annual IEEE/ACM International Symposium on Microarchitecture*, December 2009, pp. 45–55.

[47] A. McGovern, E. Moss, and A. G. Barto, "Building a basic block instruction scheduler with reinforcement learning and rollouts," *Machine Learning*, vol. 49, no. 2-3, pp. 141–160, November-December 2002.

[48] L. McVoy and C. Staelin, "lmbench: portable tools for performance analysis," in *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, January 1996, pp. 23–40.

[49] M. C. Merten, A. R. Trick, R. D. Barnes, E. M. Nystrom, C. N. George, J. C. Gyllenhaal, and W. W. Hwu, “An architectural framework for runtime optimization,” *IEEE Transactions on Computers*, vol. 50, no. 6, pp. 567–589, June 2001.

[50] J. C. Mogul and A. Borg, “The effect of context switches on cache performance,” *ACM SIGARCH Computer Architecture News*, vol. 26, no. 4, pp. 75–84, April 1991.

[51] G. E. Moore, “Cramming more components onto integrated circuits,” *Electronics*, vol. 38, no. 8, pp. 114–117, April 1965.

[52] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, “Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0,” in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, December 2007, pp. 3–14.

[53] D. Nellans, R. Balasubramonian, and E. Brunvand, “A case for increased operating system support in chip multiprocessors,” in *In Proc. of 2nd IBM Watson P=AC*^{2}, September 2005.

[54] J. Nie and S. Haykin, “A Q-learning-based dynamic channel assignment technique for mobile communication systems,” *IEEE Transactions on Vehicular Technology*, vol. 48, no. 5, pp. 1676–1687, 1999.

[55] H. Oh and S. Ha, “A static scheduling heuristic for heterogeneous processors,” in *Proceedings of EuroPar’96*, August 1996.

[56] J. K. Ousterhout, “Why aren’t operating systems getting faster as fast as hardware?” in *USENIX Summer Conference*, June 1990, pp. 247–256.

[57] M. Pericàs, A. Cristal, F. J. Cazorla, R. González, D. A. Jiménez, and M. Valero, “A flexible heterogeneous multi-core architecture,” in *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, September 2007, pp. 13–24.

[58] M. D. Powell, A. Biswas, S. Gupta, and S. S. Mukherjee, “Architectural core salvaging in a multi-core processor for hard-error tolerance,” *SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 93–104, June 2009.

[59] S. J. Russell and P. Norvig, Eds., *Artificial Intelligence: A Modern Approach*, 3rd ed. Prentice Hall, 2010.

[60] J. C. Saez, A. Fedorova, M. Prieto, and S. Blagodurov, "A comprehensive scheduler for asymmetric multicore systems," in *Proceedings of the 5th European Conference on Computer Systems*, April 2010, pp. 139–152.

[61] J. C. Saez, D. Shelepov, A. Federova, and M. Prieto, "Leveraging workload diversity through OS scheduling to maximize performance on single-ISA heterogeneous multicore systems," *Journal of Parallel and Distributed Computing*, vol. 71, no. 1, pp. 114–131, January 2011.

[62] A. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal*, vol. 3, no. 3, pp. 210–229, 1959.

[63] D. Sanchez, "Flexible architectural support for fine-grain scheduling," in *IEEE/ACM ASPLOS*, March 2010.

[64] L. Sawalha and R. D. Barnes, "Energy-efficient phase-aware scheduling for heterogneous multicore processors," in *IEEE Green Technologies Conference*, April 2012.

[65] ——, "Phase-based scheduling and thread migration for heterogeneous multicore processors," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, September 2012, pp. 493–494.

[66] L. Sawalha, M. P. Tull, and R. D. Barnes, "Thread scheduling for heterogeneous multicore processors using phase identification," *SIGMETRICS Performance Evaluation Review*, vol. 39, no. 3, pp. 125–127, Dec. 2011.

[67] L. Sawalha, S. Wolff, M. P. Tull, and R. D. Barnes, "Phase-guided scheduling on single-ISA heterogeneous multicore processors," in *Proceedings of the 14th Euromicro Conference on Digital System Design Architecture, Methods and Tools*, August-September 2011, pp. 736–745.

[68] R. R. Schaller, "Moore's law: past, present, and future," *IEEE Spectrum*, vol. 34, no. 6, pp. 52–59, June 1997.

[69] D. Shelepov, J. Carlos, S. Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar, "HASS: a scheduler for heterogeneous multicore systems," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 2, pp. 66–75, April 2009.

[70] T. Sherwood, E. Perelman, and B. Calder, “Basic block distribution analysis to find periodic behavior and simulation points in applications,” in *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques.*, September 2001, pp. 3–14.

[71] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, “Automatically characterizing large scale program behavior,” in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002, pp. 45–57.

[72] T. Sherwood, S. Sair, and B. Calder, “Phase tracking and prediction,” in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, June 2003, pp. 336–347.

[73] A. Snavely and D. M. Tullsen, “Symbiotic jobscheduling for a simultaneous multithreaded processor,” in *roceedings of the ninth international conference on Architectural support for programming languages and operating systems*, December 200, pp. 234–244.

[74] T. Sondag and H. Rajan, “Phase-based tuning for better utilization of performance-asymmetric multicore processors,” in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, April 2011, pp. 11–20.

[75] J. Stärner and L. Asplung, “Measuring the cache interference cost in preemptive real-time systems,” *ACM SIGPLAN Notices*, vol. 39, no. 7, pp. 146–154, June 2004.

[76] R. Strong, J. Mudigonda, J. C. Mogul, N. Binkert, and D. Tullsen, “Fast switching of threads between cores,” *ACM SIGOPS Operating Systems Review*, vol. 43, no. 2, pp. 35–15, April 2009.

[77] M. A. Suleman, O. Mutlu, J. A. Joao, Khubaib, and Y. N. Patt, “Data marshaling for multi-core architectures,” in *Proceedings of the 37th annual international symposium on Computer architecture*, June 2010, pp. 441–450.

[78] G. Tesauro, “Online resource allocation using decompositional reinforcement learning,” in *Proceedings of the 20th National Conference on Artificial Intelligence*, 2005, pp. 886–891.

[79] D. Tsafrir, "The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops)," in *Proceedings of the 2007 workshop on Experimental computer science*, June 2007.

[80] D. Vengerov, "A reinforcement learning framework for utility-based scheduling in resource- constrained systems," *Future Generation Computer Systems Journal*, vol. 25, pp. 728–736, 2009.

[81] ——, "A reinforcement learning approach to dynamic resource allocation," *Engineering Applications of Artificial Intelligence Journal*, vol. 20, no. 3, pp. 383–390, April 2007.

[82] D. Vengerov, L. Mastroleon, D. Murphy, and N. Bambos, "Adaptive data-aware utility-based scheduling in resource-constrained systems," *Journal of Parallel and Distributed Computing*, vol. 70, no. 9, pp. 871–879, September 2010.

[83] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal, "Baring it all to software: Raw machines," *IEEE Computer*, vol. 30, no. 9, pp. 86–93, September 1997.

[84] A. Wiggins, H. Tuch, V. Uhlig, and G. Heiser, "Implementation of fast address-space switching and TLB sharing on the strong arm processor," in *Proceedings of the 8th Asia-Pacific Computer Systems Architecture Conference*, September 2003.

[85] J. A. Winter, D. H. Albonesi, and C. A. Shoemaker, "Scalable thread scheduling and global power management for heterogeneous many-core architectures," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, September 2012.

[86] Y. Wu, S. Hu, E. Borin, and C. Wang, "A HW/SW co-designed heterogeneous multi-core virtual machine for energy-efficient general purpose computing," in *9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, April 2011, pp. 236–245.

[87] X. Yan, L. Sawalha, A. McGovern, and R. Barnes, "Supporting transparent thread assignment in heterogeneous multicore processors using reinforcement learning," in *Proceedings of the 3rd Workshop on SoCs, Heterogeneous Architectures and Workloads*, February 2012.