9-1-2013

Online Management of Resilient and Power Efficient Multicore Processors

Rance Rodrigues
University of Massachusetts - Amherst, rance.rodrigues@gmail.com

ONLINE MANAGEMENT OF RESILIENT AND POWER EFFICIENT MULTICORE PROCESSORS

A Dissertation Presented

by

RANCE RODRIGUES

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2013

Department of Electrical and Computer Engineering

ONLINE MANAGEMENT OF RESILIENT AND POWER EFFICIENT MULTICORE PROCESSORS

A Dissertation Presented

by

RANCE RODRIGUES

Approved as to style and content by:

To my parents, my brother and my dearest wife.

ACKNOWLEDGMENTS

I am deeply grateful to a number of individuals who have encouraged, supported, mentored and guided me during my pursuit of a M.Sc. and then a Ph.D. Without this, my experience might not have been as memorable as it has been.

Firstly, I would like to offer my gratitude to my Advisor, Professor Sandip Kundu who directed both my M.Sc. thesis and my Ph.D. dissertation from Fall 2007 to Summer 2013. He has been a great inspiration and a wonderful mentor and teacher. He always made himself available, be it during a usual working day or a weekend, especially the one preceding a conference paper submission deadline. He always inspired enthusiasm and led by example. His passion for quality research and publications has left an everlasting impression on me. I still remember the first time I approached a large scale C++ project. I was merely a rookie at the time, yet to master my first program on linked lists. He took the time to sit down with me and explain to me basic coding standards, coding styles and even reviewed my initial attempts at coding. I made several mistakes, but he always encouraged me. Eventually, we ended up coding an optical lithography simulator in C++ spanning more than 10K lines in less than a year. This simulator was part of my M.Sc. thesis and after that, I have never looked back.

I would also like to thank my second Advisor, Professor Israel Koren. Professor Koren was a member on my M.Sc. thesis committee and we got acquainted during the start of my Ph.D. Even Professor Koren always made himself available, weekday or weekend. I still remember the first time when we were trying to submit a paper to a major conference and he came down to my office space in the middle of a Saturday to make sure that the experimental results were as expected. This has happened several times. I also greatly appreciate the times he provided much needed criticism and useful tips to help improve upon my mistakes. I can attribute all my accomplishments (publications, awards etc.) to Professor Sandip Kundu and Professor Israel Koren and will forever be in their debt.

I also offer my thanks to Dr.Susan Bronstein who was the Director of the Learning Resource Center (LRC) here at UMass. It is always an interesting experience working with people with a different expertise and Susan was no exception. I worked at the LRC as a technical assistant and helped develop web pages and managed databases for Susan. The fact that I have no loans to pay for my M.Sc. studies is all attributed to Susan who offered me a Graduate Assistantship during my study. Thank you for everything, for the wonderful working relationship and for the time you offered me a jacket to help me get through my first New England Winter of which I obviously had no clue in Fall 2007. I really appreciate everything.

There have also been many other people that have helped, mentored and supported me during my Ph.D. I will go in chronological order. I would like thank Aswin for his help and mentoring in setting up the optical lithography simulator during my M.Sc. He was instrumental in inspiring me and teaching me several things about computer programming. I also appreciate the help that Omer extended during the brief time he was here at UMass. He was the one who helped me setup the SESC architectural simulator and the benchmarks. My initial learning curve was made very short indeed thanks to the readme's and hints that he left in the code. I would also like to thank him very much for those endless conversations over the phone after he graduated in the effort to come up with that next awe inspiring computer architecture. His motivation and determination continue to inspire me to this day.

I am also very lucky to have met friends like Neha and Hari who were always there for me when I was down mentally or my progress hindered by segmentation faults. Thank you Neha and Hari for those tennis and the coffee sessions. Thank you Neha for lending your car in the pursuit of my first US drivers license. Thank you for those car rides to Hampshire college tennis and swimming classes. Both of you have made a significant contribution in making my experience as a student a memorable and enjoyable one.

I would also like to thank my lab mates Anup, Arunachalam, Sudarshan, Arunkumar, Bharath and Nithesh. I will never forget those long walks to the Computer Science Department for that delicious coffee. Arunachalam was always there to help with code snippets or writing portions of research papers. I still remember the time when we spent the whole night writing a paper that was due the next morning. Your persistence and ideas were of great help. Thank you for the support you provided by sitting through rehearsals for my presentations. Your feedback and timely support was the reason behind the few awards that I bagged and the several publications we authored together.

I would also like to thank my brother and my parents for their support during the last 6 years. I would like to offer special thanks to my brother Richard and his wife Amoli for their support without which I would never have achieved whatever I have to date. Thank you both for persisting with me and believing in me. Thank you for those long talks on the phone regarding the GRE and potential Universities to apply to. Thank you for helping me whenever I needed a code review or advice during the initial stages of my graduate career. This Ph.D. degree is attributed to your support and belief in me. I don't think I will ever be able to thank you enough.

I would like to thank my parents-in-law and sister-in-law who have supported and loved me unconditionally. Thank you for being there to listen to my pitches and calling me before every competition and even making an effort to watch me present my dissertation defense.

I would like to thank Hope Crolius, a wonderful neighbor and a dear friend who has been an angel to my wife and me in the past 2 years. Thank you for letting us use your truck, for the wonderful gifts, plants and so many other things that I cannot say enough. You have truly been part of all our happy moments.

Last but by no means the least, I would like to thank my wife Linzy. She has been with me through the ups and downs and has always been supporting and advising me. Thank you for helping me decide that pursuing a Ph.D. was the way to go for me. I can't tell you enough how important that discussion was. The way things worked out, I am extremely happy and satisfied with what I have done for the last 6 years and this would not have happened without your timely advice and support. Lastly, a special thank you for your continued patience during the time in which this dissertation was written.

I would also like to thank the Computer Science department staff for the numerous times that they provided me with change for the coffee machine. Your role has been indispensable in my pursuit of a Ph.D. I would also like to thank UMass for the wonderful opportunity that it has provided me with. I have come out of Graduate school with a M.Sc. and a Ph.D. and no loans to show for these achievements. UMass has the best opportunities for Graduate student's via research, teaching and project assistantships. Thank you UMass! I will miss you and all the wonderful people here at the visually pleasing Amherst. Here's to the beginning of a new life.

ABSTRACT

ONLINE MANAGEMENT OF RESILIENT AND POWER EFFICIENT MULTICORE PROCESSORS

SEPTEMBER 2013

RANCE RODRIGUES

B.E.E, MUMBAI UNIVERSITY

M.Sc., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Israel Koren and Professor Sandip Kundu

The semiconductor industry has been driven by Moore's law for almost half a century. Miniaturization of device size has allowed more transistors to be packed into a smaller area while the improved transistor performance has resulted in a significant increase in frequency. Increased density of devices and rising frequency led, unfortunately, to a power density problem which became an obstacle to further integration. The processor industry responded to this problem by lowering processor frequency and integrating multiple processor cores on a die, choosing to focus on Thread Level Parallelism (TLP) for performance instead of traditional Instruction Level Parallelism (ILP).

While continued scaling of devices have provided unprecedented integration, it has also unfortunately led to a few serious problems:

The first problem is that of increasing rates of system failures due to soft errors and aging defects. Soft errors are caused by ionizing radiations that originate from radioactive contaminants or secondary release of charged particles from cosmic neutrons. Ionizing radiations may charge/discharge a storage node causing bit flips which may result in a system failure.

In this dissertation, we propose solutions for online detection of such errors in microprocessors. A small and functionally limited core called the Sentry Core (SC) is added to the multicore. It monitors operation of the functional cores in the multicore and whenever deemed necessary, it opportunistically initiates Dual Modular Redundancy (DMR) to test the operation of the cores in the multicore. This scheme thus allows detection of potential core failure and comes at a small hardware overhead. In addition to detection of soft errors, this solution is also capable of detecting errors introduced by device aging that results in failure of operation. The solution is further extended to verify cache coherence transactions.

A second problem we address in this dissertation relate to power concerns. While the multicore solution addresses the power density problem, overall power dissipation is still limited by packaging and cooling technologies. This limits the number of cores that can be integrated for a given package specification. One way to improve performance within this constraint is to reduce power dissipation of individual cores without sacrificing system performance. There have been prior solutions to achieve this objective that involve Dynamic Voltage and Frequency Scaling (DVFS) and the use of sleep states. DVFS and sleep states take advantage of coarse grain variation in demand for computation. In this dissertation, we propose techniques to maximize performance-per-power of multicores at a fine grained time scale. We propose multiple alternative architectures to attain this goal.

One of such architectures we explore is Asymmetric Multicore Processors (AMPs). AMPs have been shown to outperform the symmetric ones in terms of performance and performance-per-Watt for a fixed resource and power budget. However, effectiveness of these architectures depends on accurate thread-to-core scheduling. To address this problem, we propose online thread scheduling solutions responding to changing computational requirements of the threads.

Another solution we consider is for Symmetric Multicore processors (SMPs). Here we target sharing of the large and underutilized resources between pairs of cores. While such architectures have been explored in the past, the evaluations were incomplete. Due to sharing, sometimes the shared resource is a bottleneck resulting in significant performance loss. To mitigate such loss, we propose the Dynamic Voltage and Frequency Boosting (DVFB) of the shared resources. This solution is found to significantly mitigate performance loss in times of contention.

We also explore in this dissertation, performance-per-Watt improvement of individual cores in a multicore. This is based on dynamic reconfiguration of individual cores to run them alternately in out-of-order (OOO) and in-order (InO) modes adapting dynamically to workload characteristics. This solution is found to significantly improve power efficiency without compromising overall performance.

Thus, in this dissertation we propose solutions for several important problems to facilitate continued scaling of processors. Specifically, we address challenges in the area of reliability of computation and propose low power design solutions to address power constraints.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The semiconductor industry has been driven by Moore's law for almost half a century. Miniaturization of device size has allowed more transistors to be packed into a smaller area while the improved transistor performance has resulted in a significant increase in frequency. Increased density of devices and rising frequency led, unfortunately, to a power density problem. The supply voltage has not scaled at par with technology [98]. Higher transistor density and operating frequencies therefore means larger number of transistor switches per unit area per unit time. Hence, power density increases. The trends in CPU characteristics over the years is shown in Figure 1.1. The processor industry responded to the problem of power density by lowering processor frequency and integrating multiple processor cores on a die [35]. This design paradigm focuses more on TLP while traditional ILP is sacrificed. The emergence of multicores is the reason for the right shift in the trend for power and frequency in the Figure.

Even though scaling has enabled many benefits, it has also led to a few problems. Scaling results in increased process variation [8] and soft error rates [5]. The trends in the transistor threshold voltage variability and the soft error rates are shown in Figure 1.2. Larger process variation results in significant variation in characteristics of devices from what was intended at the design stage, which may be viewed upon as defects [8]. Soft errors, where for a brief duration of time, data stored in storage nodes are flipped due to radiation effects, also increase device unreliability. The CMOS wear-out mechanisms such as dielectric breakdown (TDDB) of gate dielectrics, hot carrier

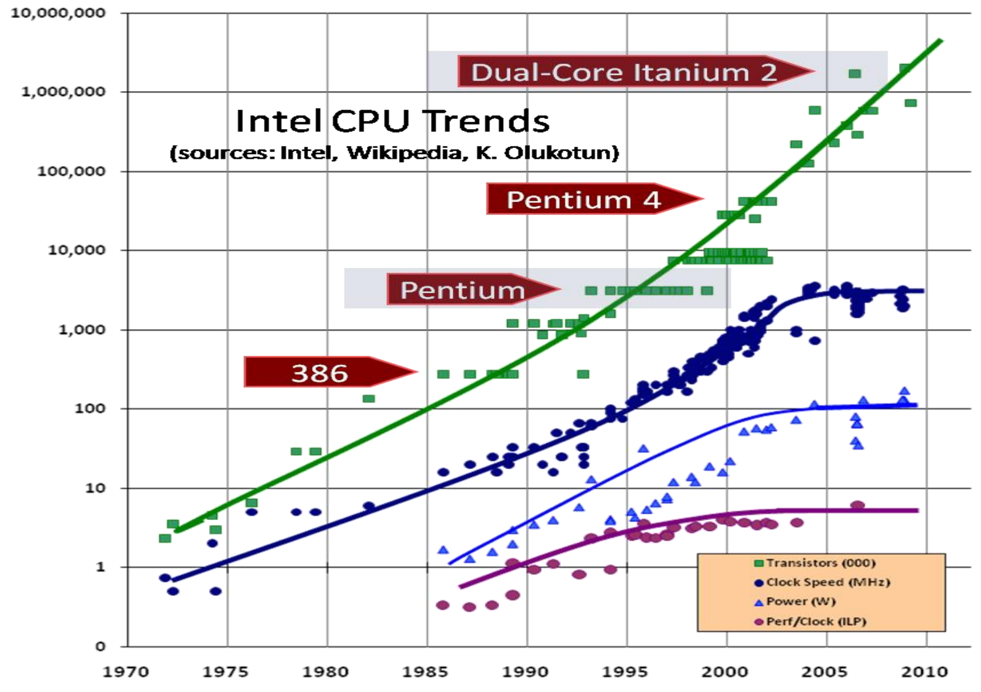
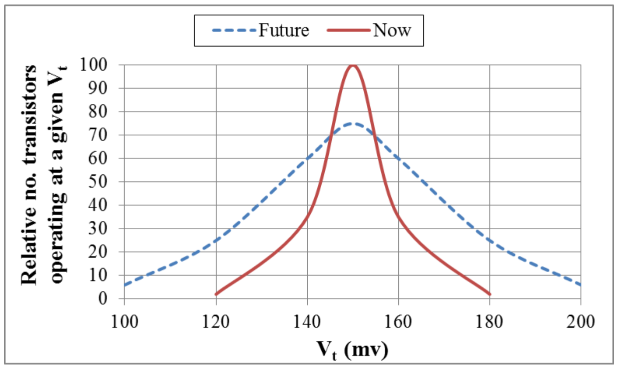
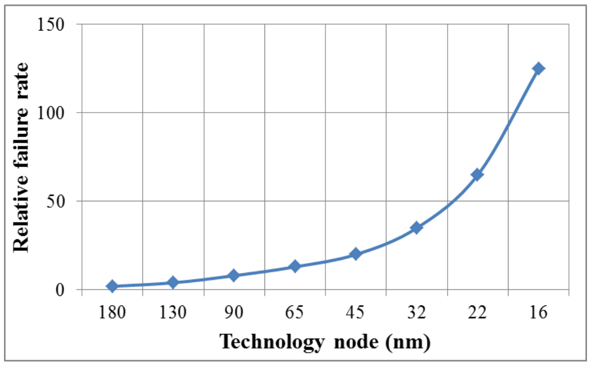


Figure 1.1. Trends in the number of transistors, operating frequency and power dissipation in Intel CPUs. Figure courtesy[4].



(a) Variation in V_t with scaling



(b) Trend in soft error rate

Figure 1.2. Trends in the threshold voltage variability and soft error rates as a function of scaling. Figures have been recreated using data available in [8].

injection (HCI) effect, negative Bias temperature instability (NBTI), electromigration (EM), and stress induced voiding (SIV) etc. have all been documented to worsen with technology scaling [98]. Hence, scaled devices are expected to experience failure in operation while in the field. There is thus a need for mechanisms to detect and correct such occurrences online.

In this dissertation, we propose mechanisms for online testing of multicore processors. Our solution is based on the incorporation of a Sentry Core (SC), the goal of which is to assist fault detection in a Chip Multiprocessor (CMP). The SC is akin to service processors that have been used in the past in main-frame computers[1] [2]. Embedding the SC in the CMP allows the internal states of the CMP cores to be accessed. The SC is a small and simple core with very limited functionality, most of which has something to do with control of other cores. The proposed SC is so simple that it can be assumed to be functionally correct and easily tested to ensure that it is fault-free. Similar assumptions have been used for IBM service processors and the UMich DIVA checker [3]. Whenever cores are found to be idle, the SC initiates test codes on the cores and captures and compares responses to detect faulty behavior. The proposed scheme thus enables low cost online error detection in multicores. Results indicate that a significant proportion of errors can be detected by the proposed scheme.

The SC also enables cache coherence transaction verification. The SC has access to the shared bus, just like the other cores in a shared bus CMP. It monitors and logs all bus transactions, and is aware of the cache coherence protocol being implemented in the system. By observing the source and type of bus transaction, it can predict the expected next coherence state of that line for the requesting core and all other cores that share that line. Whenever the same line appears on the bus again, the SC

[1]http://www.redbooks.ibm.com/abstracts/sg244757.html

[2]http://www1.ibm.com/support/docview.wss?uid=pos1R1003968aid=1

can verify that it transitioned to the correct state. If not, an error is flagged. Our experiments show that a significant fraction of the transactions can be verified by the SC by simply monitoring the shared bus.

Another problem that has come about with aggressive device scaling is increased power density. Even though the multicore design paradigm resulted in lower power density, multicores are always constrained by total power dissipation as allowed by packaging and cooling technologies. Hence, either the number of cores on the chip must be kept in check or the number of simultaneously operational cores must be limited. In general, processors do not typically operate at the maximum possible performance point. For example, Intel's recent processors feature the Turbo Boost feature[3], where depending on certain characteristics (number of threads in the system-per-power dissipation/temperature) frequency and voltage may be boosted for just one or two cores such that thermal limits are not exceeded. However, when all cores in the multicore are active, they operate at well below that in the boosted mode. Hence, in general the operation of the multicores is limited by packaging limits. Better architectures and mechanisms are thus needed such that performance-per-Watt is maximized.

In this dissertation, we propose several solutions for the problem of power efficiency. Different types of architectures are considered.

In AMPs [51, 52, 54, 30], cores of differing capabilities are all included on the same chip. Dynamic thread scheduling is then used to assign threads to cores online such that the objective function (performance, performance-per-Watt) is satisfied. For a given resource and power budget, AMPs have been shown to outperform their symmetric counterparts [51, 31, 38, 76]. However, thread scheduling in AMPs re-

[3]http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html

[4]http://www.gotw.ca/publications/concurrency-ddj.htm

mains a notorious problem. To mitigate this problem, we make use of performance counters available in most modern processors [43, 19, 92]. These counters provide useful information about the resource utilization in a processor. At regular time intervals, they are sampled by a software layer, called the Microvisor which then makes thread scheduling decisions based on this information. The Microvisor is similar to the IBM millicode [36]. In addition to that, we also explore the dynamic morphing of resources between cores in the AMP. The control of morphing is once again carried out by the use of performance counters and the Microvisor. Results indicate that significant performance-per-Watt benefits can be extracted by use of core morphing and dynamic thread scheduling in AMPs.

For SMPs, we revisit resource sharing architectures. While these architectures have been explored in the past [22, 53, 14], the evaluation has been incomplete. Specifically, most previous work only explores the performance impact of such sharing leaving the following questions unanswered.

1. What is the impact of sharing on performance and performance-per-Watt? While sharing clearly results in power savings, for certain workloads, performance loss may be too large.

2. What are the most important parameters influencing performance and performance-per-power in resource sharing architectures? We show that latency and throughput of the shared resources are dominant determinants of performance and performance-per-power, but most previous studies ignore them.

3. How does sharing of resources play out for Big cores or Small cores? Mainstream computing can be broadly classified into performance efficient (Big cores) and power efficient (Small cores). It is thus necessary to study the impact of sharing resources in both such architectures.

4. What is the impact of sharing in Simultaneously Multi-Threaded (SMT) processors? In particular, does sharing in SMT make performance or performance-per-power better or worse? Given that most mainstream cores are SMT capable[4], studying impact of increased resource utilization due to sharing is important.

Our results show that while architectures that share execution units do provide power benefits at a negligible performance penalty (∼5% on average), such benefits hold only when the shared units have low latency and are highly pipelined. Performance and performance-per-Watt loss are observed for workloads that exhibit high contention for the shared execution units. To reduce the performance loss due to contention we propose to increase the throughput of the shared resources via Dynamic Voltage and Frequency Boosting (DVFB) which is controlled dynamically by the occupancy rate. Our results show that such dynamic boosting not only overcomes losses due to contention, but also results in significant increases in both performance (upto 13%) and performance-per-Watt (upto 14%), while realizing considerable savings in area (∼ 7-10% per core).

We also explore the potential for performance-per-Watt improvements in each individual core of a multicore. The observation is that thread swapping in AMPs incurs non-negligible costs. The swapping overhead can vary from a few thousand [81] to millions of cycles [6, 50] depending on the algorithm employed to swap threads and the mechanism to exchange contexts. To amortize the large overhead associated with thread swapping, in most proposals, thread swapping decisions are made at the granularity of hundreds of thousands to millions of instructions [6, 50]. Unfortunately, numerous opportunities to improve performance-per-power and/or energy-delay-squared product (ED^2P) at a more fine grained instruction granularity are missed out by such approaches [65]. Therefore, there is need for a mechanism to realize these opportu-

[4]www.intel.com

nities without incurring large thread swapping penalties. To achieve this we make use of in-built debug mechanisms available in most modern processors to switch the processor operation from out-of-order (OOO) to in-order (InO) at runtime depending on current workload characteristics. Our results indicate that the proposed scheme achieves an ED^2P reduction of as much as 12% at a performance loss of less than 5% when compared to the baseline OOO-core.

Thus, in this dissertation we propose solutions for online error detection and power efficient computing in present day multicore processors.

CHAPTER 2

ONLINE TESTING OF MULTICORES

2.1 Introduction

Errors in processor execution is a growing concern. Errors may result from multiple sources. Untested manufacturing defects [90], transistor and interconnect wear-out [8], variations in operating conditions such as voltage and temperature, and transient errors due to cosmic radiation or α-particles lead to errors in device operations. They must be detected and corrected during runtime. Thus, there is a growing need for online mechanisms to detect and correct such errors in commodity products.

2.1.1 Solving the reliability problem

Traditional solutions for reliability rely on redundancy, either in *hardware* [3] or in *time* [82, 74]. Commodity microprocessors are sensitive to cost and performance, while also constrained by power. Hence, solutions such as triple modular redundancy [91] do not apply to commodity processors. Some processors use error-correcting code (ECC) to protect the cache [83]. Since ECC is relatively low overhead, this is an acceptable solution for protecting all arrays in a processor. Such protection provides coverage for more than half of the transistors in today's processors. However, the remaining half of the transistors found in the pipeline stages are vulnerable. While there have been many solutions proposed to protect the various pipeline stages [90, 79, 12], they tend to be costly and only protect a part of the processor. A low-cost generic reliability solution for the entire processor processor can significantly complement reliability coverage.

Existing solutions may be broadly classified into those that make use of *time* [82, 74] and *structural redundancy* [3, 23, 90, 104, 66, 12]. Redundant Multi Threading (RMT) [82, 74] is an example of *time redundancy*, in which a logical thread is run as two physical threads. A difference in output of the threads indicates the presence of an error. We propose a solution based on DMR, but the control and administration of the mechanism is very different from existing DMR solutions. This scheme has multiple applications and in this dissertation, we have explored online error detection in the execution core as well as cache coherence transaction verification.

2.2 Online Testing of the Execution Core

We propose the addition of a core called the Sentry Core (SC) into the Chip Multiprocessor (CMP). Its goal is to assist in fault detection, debug and diagnosis in a CMP (Figure 2.1). It is similar to previous proposals that feature service processors in main-frame computers or watchdog monitors [7, 3], but also differs in significant ways. The most important differences are: (i) embedding the SC in the CMP allows the internal states of the CMP cores to be accessed, and (ii) the SC is an active participant in dealing with traps and interrupts. The SC is a small and simple core, so simple that it can be assumed to remain functionally correct throughout the life of the CMP. Similar assumptions have been used for IBM service processors and the DIVA checker [3].

Whenever an idle core is detected, the SC initiates test routines on the cores. It then captures and compares the resulting responses to detect faulty behavior. The central idea is to have the SC capture signatures of the program execution that reflect both the control flow and the data execution, which are then compared against responses obtained online from another core in the CMP. More specifically, we collect two signatures for the executing program. The first one is associated with the Program Counter (PC) values of each committed branch instruction. The second is associated

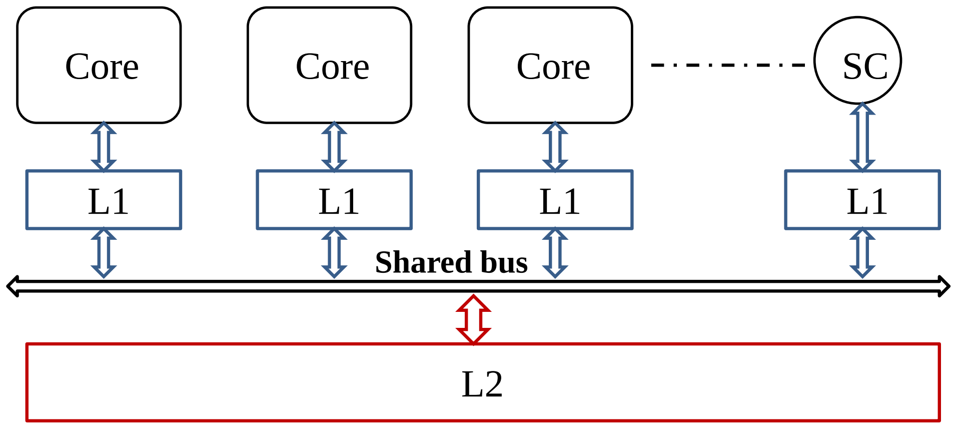


Figure 2.1. The Sentry Core (SC) in a Multicore Processor

with the data and address of store instructions. Both of these are input into a multiple-input signature-register (MISR). The SC performs the tasks of initializing the MISR, collecting the signature of a fixed number of executed branches and store instructions and comparing against the reference obtained dynamically from another core running the same thread. Such a comparison reveals errors in the control flow of the program as well as the data flow. Since the signatures are collected on committed instructions, speculative execution has no effect on these signatures. We use virtual addresses of the branch/store instructions for signature generation and hence, the same signatures will be generated for two fault-free cores running the same code segment. Lastly, while such monitoring is in progress, traps and interrupts will be routed via the SC and any exception will halt the program execution allowing the SC to access the appropriate signatures. When an error occurs, the signatures obtained from the two cores will differ and the error will be detected.

The benefits of the proposed solution are: (i) online testing with minimal overhead, (ii) scalability and (iii) lifecycle testability.

To validate the proposed approach we conducted experiments using the SESC simulator [75] and used eight benchmarks from the SPEC 2000 suite [97]. We chose these benchmarks and not specifically engineered test routines to show the potential benefits of the SC regardless of the executed software. The development of test code to accelerate fault detection is not out focus and is a part of future research. In the experiments, faults were injected to result in a faulty behavior in a 4-core CMP and the resulting fault detection latency and coverage were measured. The relationship between the fault detection latency and the checking interval (time between signature checks) was also analyzed. Our results indicate that even though the SC may add an area overhead of up to 3% for the target system, the rich testing functionality it provides makes it an attractive approach.

2.2.1 Related work

With aggressive technology scaling, aging defects afflict processors with progressively worse delay and catastrophic faults. As a result, fault detection and correction schemes have been a topic of considerable interest. Previous approaches may be classified into those that target certain structures in a processor ([12, 2, 17, 80]) and those that target the entire processor ([3, 82, 74, 90, 68, 62, 91, 93, 95]). Of these a few of them are directly comparable to our approach.

In [12], Bower *et al.* presented a scheme to detect and tolerate faults in array structures of microprocessors. A similar scheme was presented by Rodrigues *et al.* in [80]. Fault detection in integer ALU execution units is proposed by Abella *et al.* in [2]. Self test for register data flow is proposed by Carretero *et al.* in [17]. However, such schemes only protect certain structures of the processor and do not provide chip wide coverage. Chip wide error detection schemes have also been proposed. In [90], Shyam *et al.* protect stages of the pipeline using BIST techniques. Meixner has proposed Argus, a dynamic verification scheme for fault detection in simple cores [68]. Li *et al.* [62] use high level symptoms with system restoration and re-execution on another core to detect faults.

A few of the chip wide error detection approaches have similarities with ours. Austin has proposed the DIVA checker [3] in which a small core is augmented to check for computation correctness of its companion core. Whenever the results from the two cores differ, the checker core commits its result and the pipeline of the larger core is flushed. However, each core requires a DIVA core for error detection which is not the case with our proposal where multiple cores may share a single SC. Replication of pipeline stages [93] and complete replication of core execution for fault tolerance has been explored via DMR/TMR in [91]. These approaches pose very high area and power overhead (200/300%). Our SC based approach reduces these overheads by initiating DMR only when cores are idle. Redundant Multi Threading (RMT) approaches have also been proposed [82, 74], in which a logical thread is run as two physical threads. One of the threads is leading while the other is trailing and the leading thread provides certain inputs to the trailing thread. A difference in execution indicates the presence of a fault. The states of the two threads are compared while our SC based solution only compares the signatures. The amount of state information compared in RMT is important as it determines the overhead. Smolens *et al.* [95] proposed a solution in which the fingerprint of instructions between checkpoints is compared for error detection. Here comparison of states is assumed to be done by an error-free core. Our SC based solution collects signatures of committed branch instructions and then compares them at regular intervals to detect faults. Since the SC is responsible for signature comparison, there is considerably lower probability of an error during this comparison. More recently, lau *et al.* [56], presented the partner cores concept, where each complex core in a CMP is augmented with a small core for reliability and performance improvements. However, just like DIVA pairing a partner core with each complex core increases overhead. Thus, even though there are similarities between our approach and the previous work, our scheme overcomes the drawbacks of other approaches.

2.2.2 The proposed solution

Our scheme is built around the SC which verifies the fault-free operation of the general-purpose processor cores. In this section, we describe the functionality and hardware overhead of incorporating an SC in a CMP. The approach followed to detect faults in the CMP is then presented.

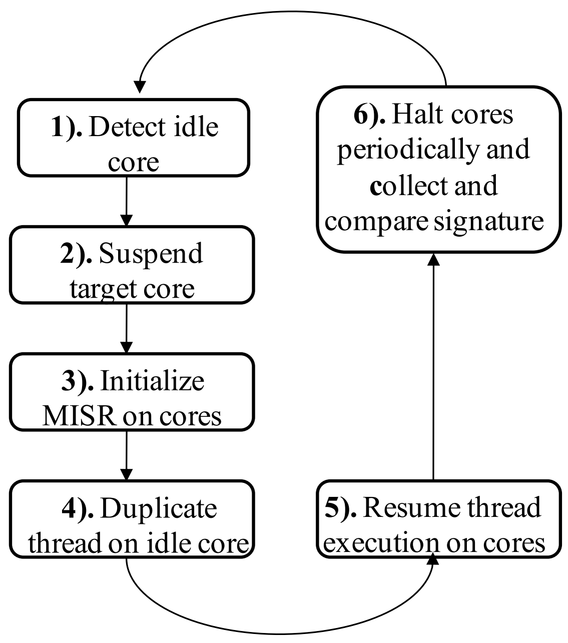
2.2.2.1 The Sentry Core (SC)

The SC is a small and simple core with the objective of enabling quick fault detection in a CMP. For that it requires to be able to test as well as collect responses and then detect faults if any. For that, it is augmented with a variety of features described next.

2.2.2.1.1 Control functions To assist in fault detection, the sentry core supports the following operations:

1. Detect idle cores: SC has the capability to detect if cores are idling.
2. Initialize MISR: initialize the MISR for each core participating in the test.
3. Duplicate: trigger a DMR/TMR (Dual/Triple Modular Redundancy) configuration by replicating a process and executing it on two or three cores.
4. Collect MISR signature: collect and compare signatures periodically.
5. Suspend: halt one or all the processor cores to analyze their state.
6. Resume: resume the operation of a halted core(s).
7. Terminate: terminate a process on a core.

Only a subset of the various functions that the SC can perform have been listed. This functionality can easily be added to any off-the-shelf processor that fits the description of the SC by extending the Instruction Set Architecture (ISA) [58, 59, 108,



1). Detect idle core
2). Suspend target core
3). Initialize MISR on cores
4). Duplicate thread on idle core
5). Resume thread execution on cores
6). Halt cores periodically and collect and compare signature

Figure 2.2. Steps followed for online test by the SC

103]. In most processors, the opcode field size is extensible. The additional instructions will be used to enable the functionality of fault detection and diagnosis. The steps followed by the SC to initialize testing is shown in Figure 2.2. When the cores are halted for signature comparison, the state of the cores must be saved so that execution may later resume. After saving the state, the cores provide the current control/data signatures to the SC (may be done via a write to a shared memory location) and then resume execution when permitted to do so. The routine to save the MISR state to a shared memory location may be the same as that followed for interrupts. During interrupts, the program state is saved; the interrupt handling routine is executed and the core then resumes operation after state restoration. A similar procedure is followed during context switch while running multithreaded applications. The saving of the MISR state to memory may thus be implemented as an interrupt handling routine or as a context switch in the CMP. Thus, the overhead incurred due to SC intervention for signature comparison is similar to that of a context switch.

Note that in the considered CMP, only the L1 caches are not shared. Hence, context switch overhead is very small. We assume 1000 cycles as the context switch overhead for the target system configuration based on inferences made from [61]. A custom design will result in far lower overhead. In this work, we make use of the of the interrupt based scheme due to its simplicity.

2.2.2.1.2 Hardware overheads The SC provides rich set of features for online error detection in the CMP. This comes at a small hardware overhead. To estimate the overhead of incorporating the SC itself in a CMP, we consider for example, the EV4 (Alpha 21064), and EV6 (Alpha 21264) cores. The functionality of the EV4 is sufficient to satisfy the requirements of an SC for a CMP comprising of EV6 cores. The EV4 occupies about 11% of the area of an EV6 core [24]. Hence, for a dual/quad/eight core CMP, the area overhead due to the SC reduces to 5.5/2.75/1.375%. This is an acceptable overhead for the added functionality considering that DIVA [3] adds a 6% overhead. While making this comparison in area, it is important to note that the SC scheme is a more attractive solution when compared to DIVA. DIVA only allows error detection but no additional options to diagnose the cause of the error. The SC has dedicated ISA extensions and enhanced functionality that not only allows online error detection, but also diagnosis. Further, a single SC can service a quad, eight or even a sixteen core CMP. However, as the number of cores serviced by the SC increases, the time slice that each core gets for service reduces. For many core systems, additional SCs may be incorporated such that there is one SC for every m cores in the CMP. We plan to evaluate this in the future. The above is just an example to illustrate the size of the overhead. In reality, a customized (rather than an off-the-shelf) SC design may have far lower area overhead (<1%). Incorporation of an SC results in heterogeneity which increases the design time, but heterogeneity is no longer a novel concept [3, 51] and hence may be considered an acceptable practice.

2.2.2.2 Fault detection strategies

Fault detection is possible if a reference is available for comparison. DMR is the straightforward solution to this. However, as mentioned earlier, this results in large overheads. Hence, the SC only initiates DMR if cores are found to be idle. In [67], Meisner *et al.* note that on an average, cores in servers idle for 70% of the time. The reason for this is that server processors are always designed for peak performance, a situation rarely encountered. We use such idle cores for test. We thus overcome the drawbacks of DMR. Whenever idle cores are found, the SC copies the program state and program data of one core to another core. Both cores execute the same program for a fixed number of instructions as determined by the SC. When the number of branch instructions executed by the core reaches its pre-specified limit (a programmable parameter), an interrupt is generated and the program is vectored into a wait state as described earlier. The SC can then query the signatures generated by the executed branch instructions as well as the memory write operations. If a mismatch is encountered an error is detected. This scheme will be most effective for heterogeneous multicores as the underlying hardware in the cores is different. The scheme will work equally well for homogeneous multicores by ensuring run-time heterogeneity. Most modern processors feature a debug mode, where the core operation mode is switched from OOO (out-of-order) to in-order [50]. This mode is mostly used in the industry for offline testing before the product is released into the market. We propose to use this mode online. On detection of an idle core in a homogeneous CMP, the idle core mode of operation is switched to in-order. The trace generated by the in-order core can then be used to validate that generated from the OOO core. The mode of the in-order core is switched back after testing is complete. The proposed scheme will thus work fine for any type of multicore.

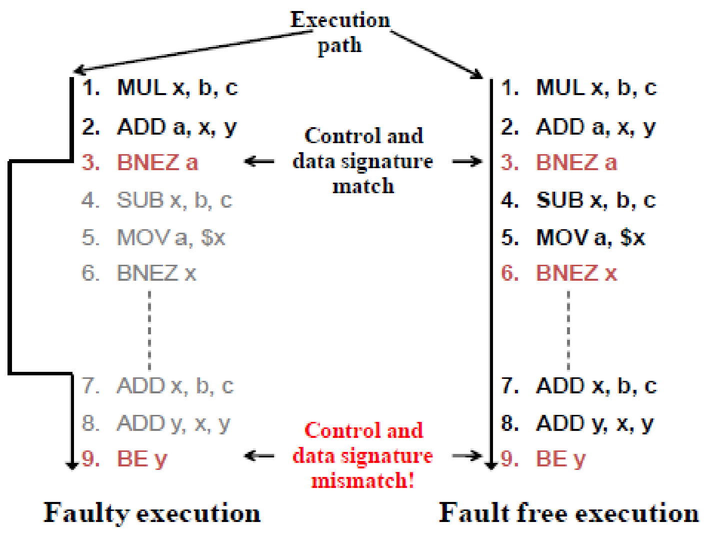


Figure 2.3. An illustration of the fault detection mechanism

2.2.2.3 Fault detection algorithm

Once an idle core is detected, the SC begins the testing process. The SC copies the program state of the core to be tested onto the idle core. The number of branch instructions after which the signatures are to be compared is set (detailed experiments on the choice of this variable to follow) and the cores then begin execution. After committing n branch instructions, the branch counter resets to zero and an interrupt is generated. Both cores then execute the interrupt handling routine and store the MISR signature to the shared memory. The SC then collects and compares the signatures. If they match and there is no job ready to be run on the idle core, execution is resumed on each core for another n branch instructions. If however, the signatures do not match, an error is detected. Figure 2.3 shows an example of running a test code on two cores, the execution of which is faulty on one core and fault-free on the other. The execution path taken is indicated by the solid lines with arrow heads. It can be seen that due to an error in the computation of the value stored in register a, the condition checked by the branch instruction evaluates to true for the faulty core, and false for the fault-free core. As a result, the faulty core now executes along a different path. The two cores then encounter different branch instructions and by comparing the signatures of the execution traces on the two cores the SC can detect the discrepancy in the signature. Similarly, if there was an error while storing a value into register *a*, the store signature generated by the two cores would differ indicating the presence of an error.

2.2.3 Evaluation framework

In our experiments we used the SESC architectural simulator [75] after modifying the code to allow injection of faults to cause an erroneous behavior in the control and data paths. We used eight SPEC CPU 2000 [97] benchmarks as test codes for the sake of demonstrating the effectiveness of our approach. We assume that the SC is incorporated in a symmetric quad core CMP in which one core is faulty. The system parameters of general-purpose cores in the 4-core CMP are shown in Table 2.1. The benchmarks used were *equake, ammp, swim, wupwise, applu, gzip, gcc, mcf* and were chosen so as to be representative of several classes of benchmarks (FP/INT/load/store/Branch intensive) that would exercise different units within the processor. The instruction distribution for each benchmark after executing them for 10 million instructions is shown in Figure 2.4.

Table 2.1. Core parameters.

Parameter	Value
Frequency	2 Ghz
Fetch/Issue/Retire	4/4/4
ROB size	128
ISQ size	80 INT, 40 FP
Branch Prediction	Hybrid: local bits 2, BTB 4096 entries
RAS	size 64, Replacement policy LRU
Functional units	2 FP and 4 INT ALU 1 each of FP/INT MUL, DIV
Registers	104 INT and 80 FP
L1-D/I	cache 64K, 8-way, 1 cycle
L2 cache	2M, 16-way, 10 cycle

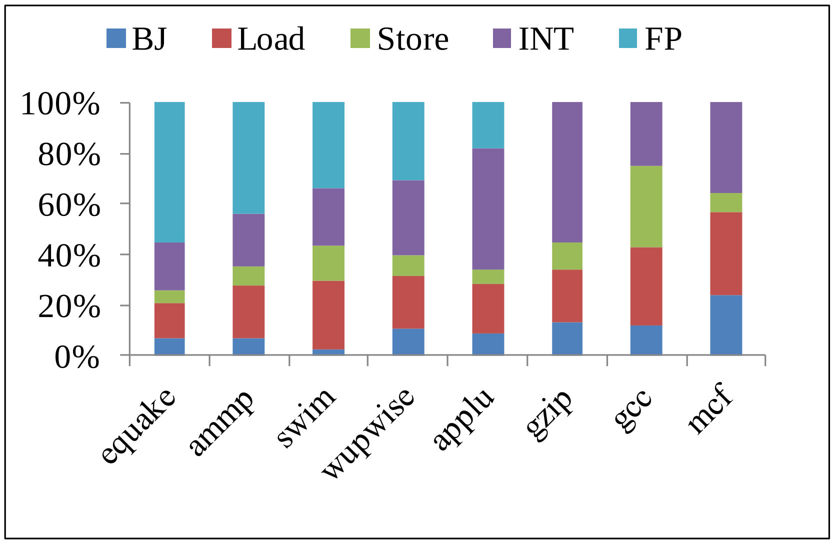


Figure 2.4. The instruction distribution of the workloads used in the experiments. Each workload was run for 10 million instructions.

Since we use a performance simulator, injecting faults into the system is implemented through bit flips in the data structures used to simulate the architectural components. For example, when an instruction is retired, a bit in the Reorder Buffer (ROB) data structure is set to indicate that. If a fault is injected into that bit of the ROB entry, other instructions waiting for this instruction to complete will never resume execution. There are also cases where the injection of a fault may result in multiple faults due to the way the simulator operates. To cause a faulty behavior during a store operation, we either inject a single bit fault in the data register or inject a fault into the address register, resulting in wrong data value or address. In each benchmark run we have injected 100 data and control faults.

2.2.4 Results

Experimental results on the error detection latency and the fault coverage are now presented. The latency is measured as the time elapsed between fault injection and error detection. Since SC intervention incurs an overhead, results are also presented

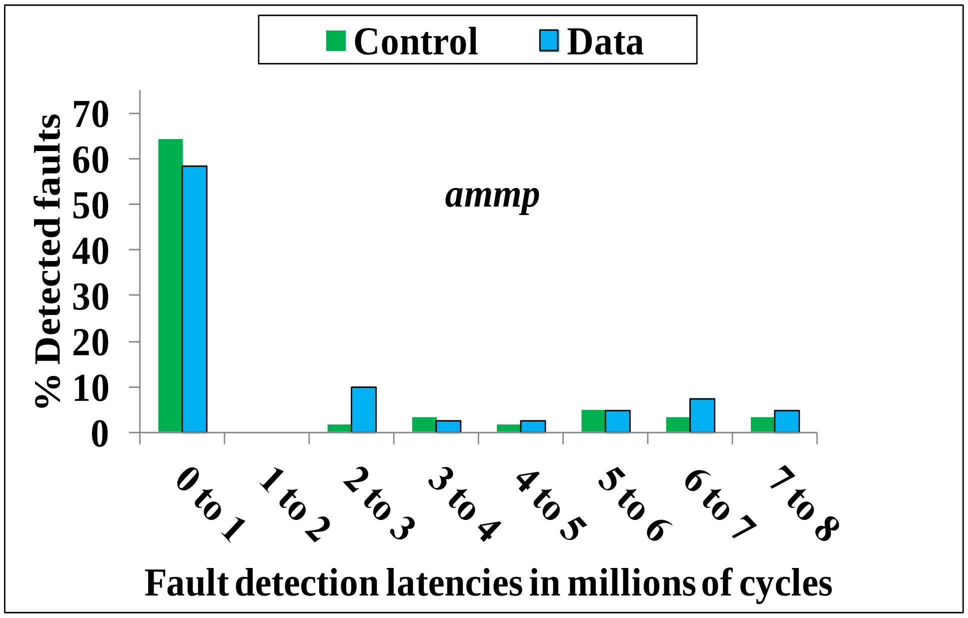


Figure 2.5. The distribution of the detected faults as a function of the detection latency for ammp.

on the effect of checking interval on error detection latency. Based on these experiments, fault coverage results when using an SC checking interval of 100K branches are presented.

2.2.4.1 Fault detection latencies

Results on fault detection latencies are shown in Figures 2.5, 2.6 and 2.8. We have only included results for *ammp, gcc* and *mcf* as these were found to be interesting. These results include both data and control faults for a fixed checking interval of length n = 100K committed branches. For almost all the benchmarks shown, a significant proportion of the injected faults (about 60% on average for control and 55% for data faults) are detected within 1 million cycles of execution. Control related faults manifest themselves faster than data related faults and hence are caught earlier. There are only a few faults which are detected at later stages and a majority of those are data related faults.

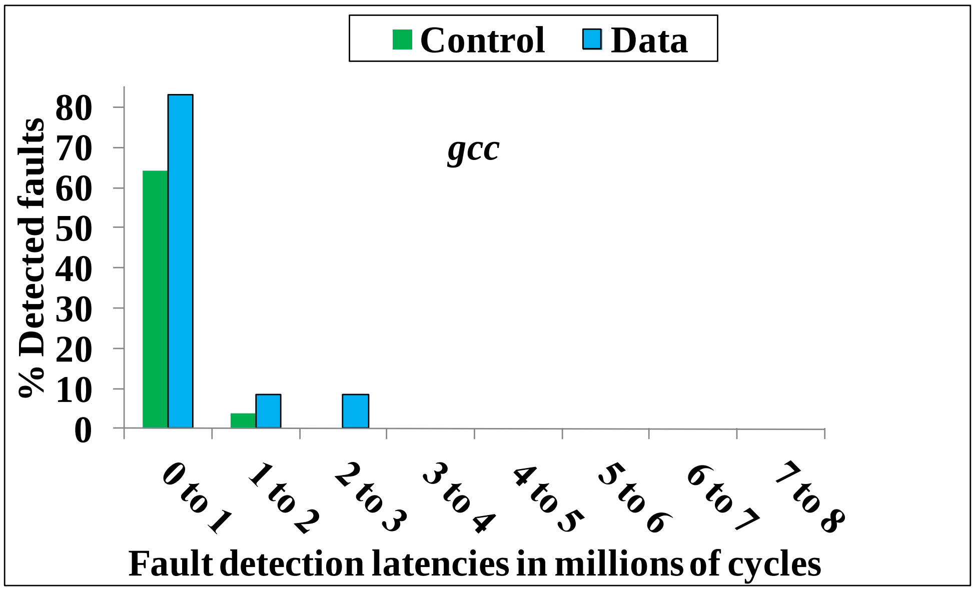


Figure 2.6. The distribution of the detected faults as a function of the detection latency for gcc.

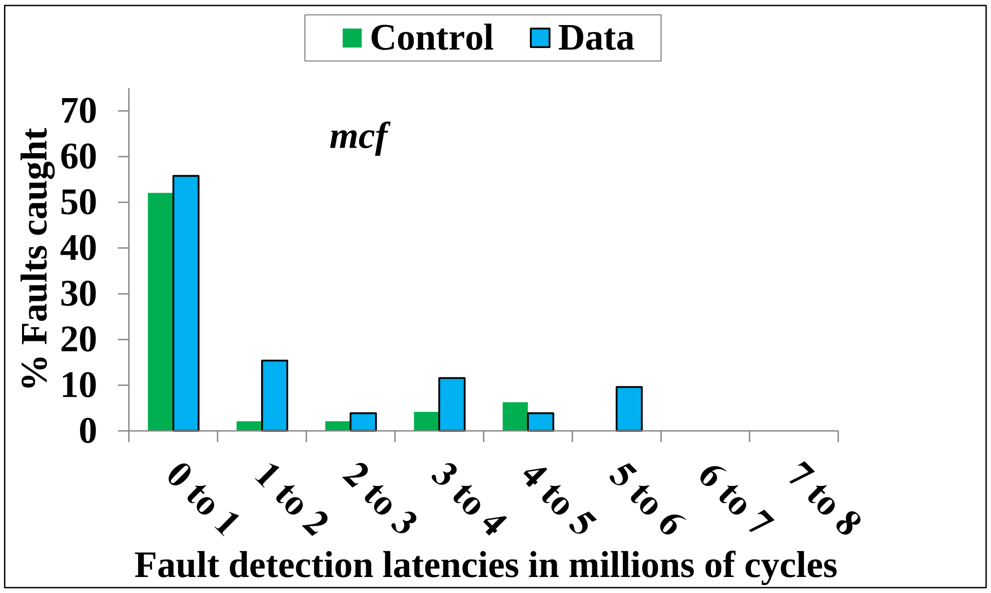


Figure 2.7. The distribution of the detected faults as a function of the detection latency for mcf.

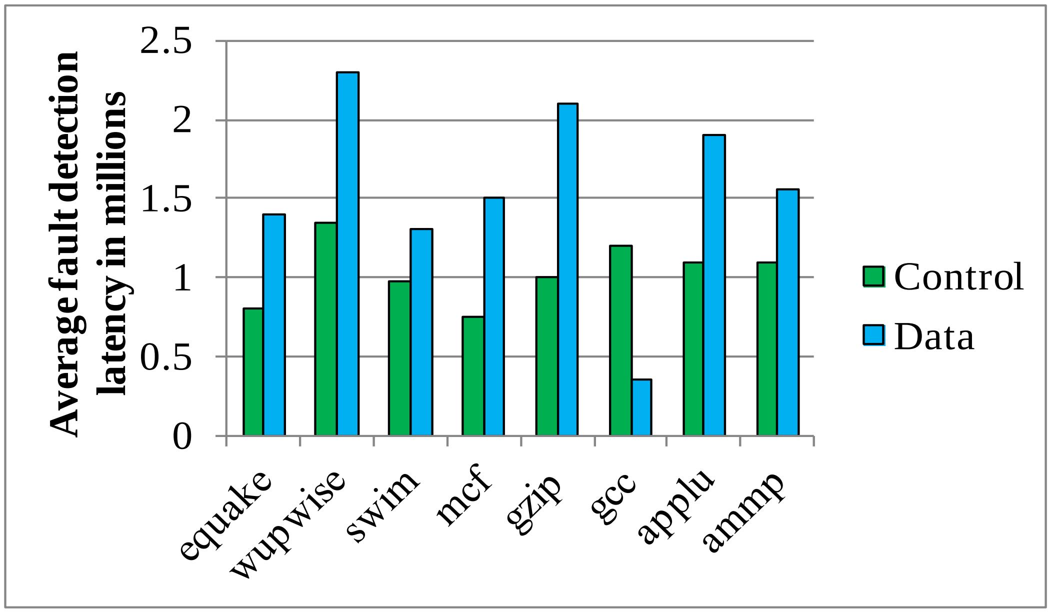


Figure 2.8. Average detection latencies for data and control faults for all eight benchmarks.

There is a relationship between the error detection latency and the instruction distribution (see Figure 2.4). A fault injected into the system will manifest itself only if the faulty unit was used during execution. For example, a fault in an integer issue queue entry will manifest itself only if that entry is used, i.e., the program should have executed a resonable number of integer (INT) instructions. The same holds for floating-point (FP) instructions. The same holds for faults inserted into the load/store queues. We found that on average, control related faults are caught earlier for benchmarks with diverse instruction distribution (e.g., *wupwise*). If a majority of the instructions are of a particular type, faults injected into the system that are used during execution of the other instruction types may not be exercised and hence never be detected. Further, memory intensive benchmarks spend considerable time waiting for the memory operations to complete and as a result, control faults do not manifest themselves fast or not at all. However, diverse instruction distribution does not guarantee error detection. Often programs run loops which may execute

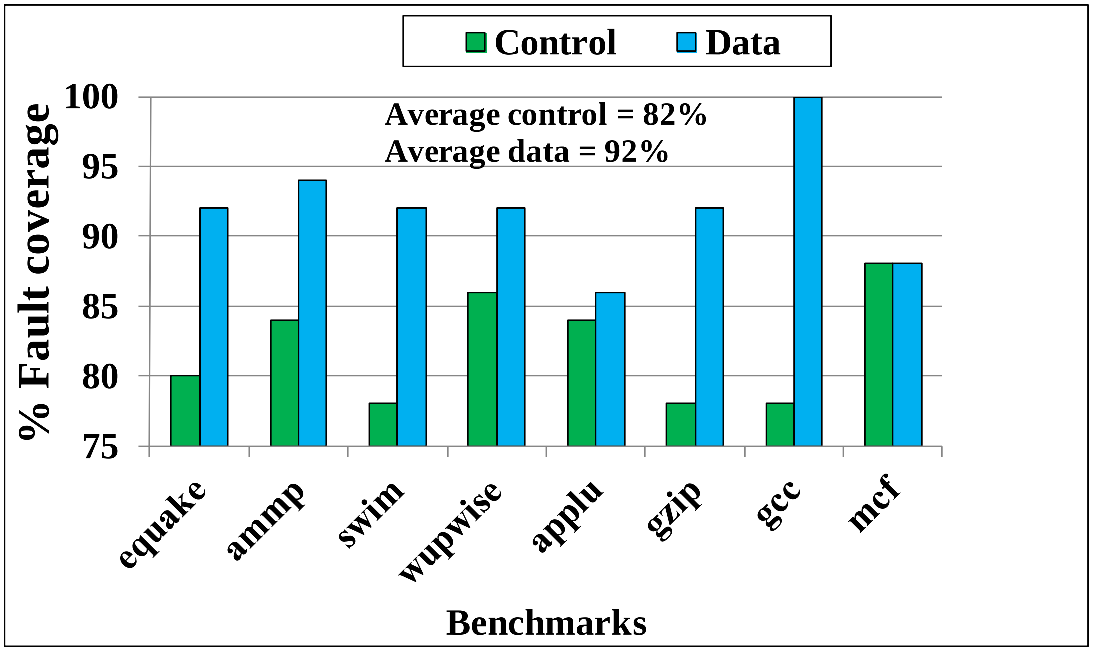


Figure 2.9. Data and control fault coverage for the benchmarks.

the same type of instructions repeatedly thus potentially missing the ones that can exercise the fault. In addition, since our scheme collects signatures only after n branches are committed, a higher frequency of branches in the test routine results in early fault detection (*mcf, gzip, wupwise*). If the branches are sparse, the fault may be detected after a very long latency (*swim*) and the fault may even go undetected. Hence, for control related faults, diverse instruction distribution and the proportion of branch instructions play a major role in the final error detection latency. A similar explanation exists for the data related faults. Since these faults will be exercised only when data is being stored (to the caches), the larger the number of store instructions the higher is the chance to exercise these faults (*gcc*). Here too, having frequent branches may reduce the fault detection latency (*gzip*). The above discussions apply to the plots showing detection latencies. Figure 2.8 shows the average control and data fault detection latencies for all benchmarks.

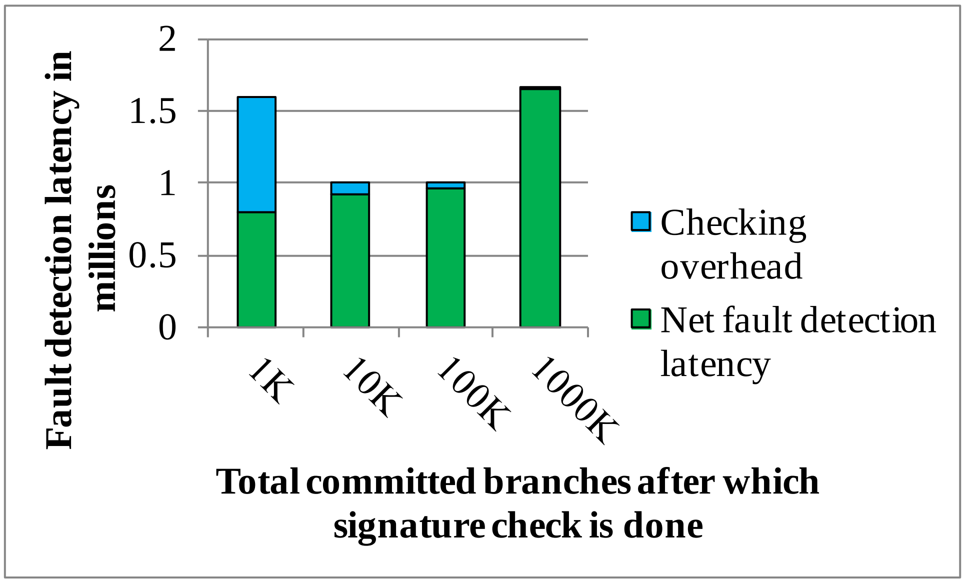


Figure 2.10. Effect of checking interval on fault detection latency. Data shown is averaged for all considered benchmarks.

2.2.4.2 Fault coverage

Instruction distribution also plays an important role in fault coverage. The fault coverage for various benchmarks is shown in Figure 2.9. Larger the utlization of different structures in the processor, higher is the chance of fault detection (e.g. *wupwise, ammp*). The same holds for the coverage of data faults (*gcc*). An important observation is that we were able to catch just 82% of the control faults as compared to 92% of the data faults on average. The reason for this difference is that control faults were mostly caught early or not caught at all. This happens as (i) control faults if exercised, manifest themselves sooner, and (ii) some of the benchmarks are either FP (equake) or INT (gzip) or memory intensive (gcc) and some have very few branches (swim), and hence were not exercising all the injected faults as explained earlier. Data faults were caught more often but with higher latencies. The most notable result is that for gcc where 100% coverage of data faults was achieved due to about 30% store and 10% branch instructions found in the mix. Overall, the scheme was able to detect 87% of the (combined control and data) faults using standard benchmarks rather than specifically engineered test routines, which greatly increases the confidence in the ability of the proposed scheme to detect faults online.

2.2.4.3 Effect of number of branches committed before signature comparison

The frequency of signature check can have a significant impact on the effectiveness of the proposed solution. Smaller the interval, the better is the expected result. However, too frequent a signature check will result in large overhead (1000 cycles for each comparison) for checking. Hence, we experimented with various checking intervals. The average fault detection latency when using checking intervals of 1K, 10K, 100K, and 1000K committed branches is shown in Figure 2.10. Each bar in the figure shows the average for both control and data faults. The net detection latency is the fault detection latency with zero checking overhead. Hence the final detection latency is the sum of the net detection latency and the overhead (total height of the bars in the figure). It can be seen that for the smaller checking intervals even though the net detection latency with no checking overhead is small, the checking overhead is high due to very frequent checks. For larger checking intervals, even though the net detection latency increases, there are fewer checks. Hence the final fault detection latency with checking overhead is small. However, this trend continues only until a 100K interval length. After that, when using for example, a 1000K interval length, even though the checking overhead is small (since there are very few checks) the fault detection latency is very high. We therefore, used in our experiments 100K as the interval length.

2.2.5 Conclusions

We have proposed a new approach for online admistration of testing multicore processors running multithreaded programs. It is based on the incorporation of a small core called the SC to the CMP. It has basic functions of initiating, halting and querying processors. The added sentry core adds less than 3% in area, but enables a rich set of testing features. In theory, the signatures of instruction path and memory writes should provide complete coverage barring masked faults. Simulation results validate that when the faults are triggered by the executing program, they are detected. Further, the latency of detection from trigger to detection is small. This is important because with a small latency a small test code will suffice for fault detection. When using a 100K committed branches as checking interval, our approach was able to detect about 82% of the injected control and 92% of the data faults with an average detection latency of about 1 million cycles for control and 1.5 million cycles for data faults.

2.3 Cache Coherence Protocol Verification using the Sentry Core

So far we have observed the effectiveness of the SC in detecting online errors in the execution core of the processor. This was one of the many applications possible by such an architecture. We now explore the use of the SC to verify cache coherence transactions in multicore systems.

Multicore and many core systems rely on inter-core communication via shared memory. In such systems it is necessary to make sure that data consumed by all the cores is up to date. Cache coherence protocols help ensure this [37]. Functional correctness of shared memory systems thus depends on the correctness of the coherence hardware support. Ensuring correctness of the coherence hardware is difficult as even simple protocols can have multiple states [109]. The state space further increases when considering the state of a cache line shared across cores. Thus, there is need for an online mechanism to verify the operation of cache coherence transactions.

In this dissertation, we propose an online scheme to verify the operation of the cache coherence hardware in a snoopy bus multicore. We use the Sentry Core (SC)

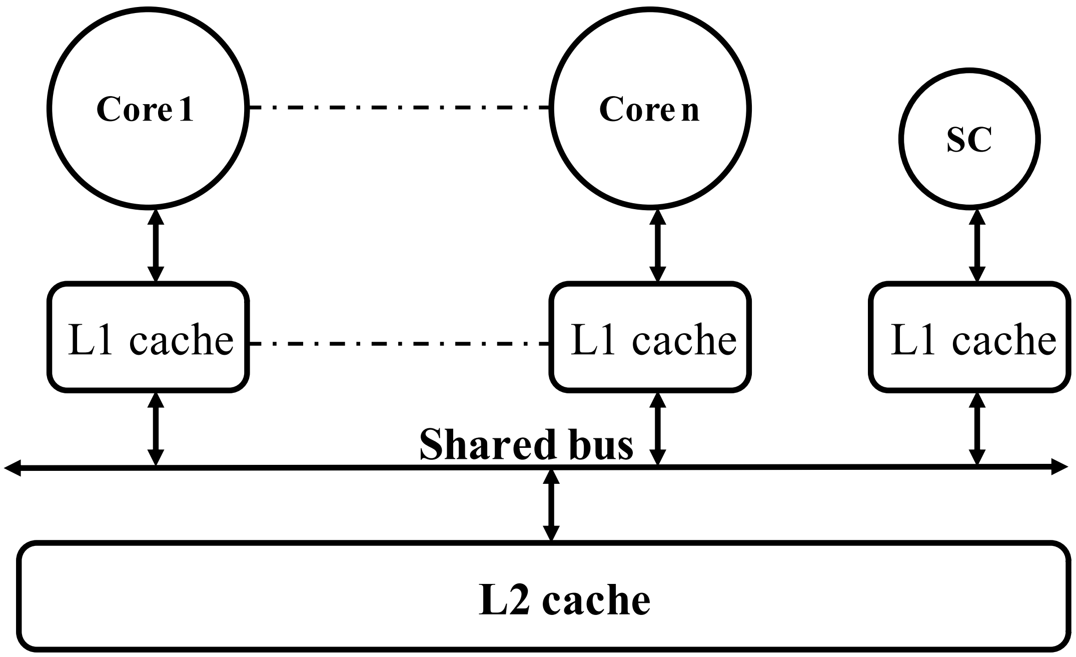


Figure 2.11. A Sentry Core (SC) in a shared memory multicore.

architecture for this purpose. The SC has access to the shared bus, just like the other cores (see Figure 2.11). It monitors and logs all bus transactions, and is aware of the cache coherence protocol being implemented in the system. By observing the source and type of bus transaction, it can predict the expected next coherence state of that line for the requesting core and all other cores that share that line. Whenever the same line appears on the bus again, the SC can verify that it transitioned to the correct state. If not, an error is flagged. Our experiments using the SPLASH-2 [107] benchmarks suggest that a significant fraction of the transactions can be verified by the SC by simply monitoring the shared bus.

2.3.1 Related work

We present in this section, a brief summary of the literature that closely relates to our proposal and point out the key differences.

In [16], Cantin *et al.* presented a variation of the DIVA checker [3] for cache coherence verification. Just like DIVA does for functional correctness of the cores, cache coherence transactions were verified using simpler logic. However, this scheme requires the use of a separate network for global verification of coherence states. In [25], Fernandez-Pascual *et al.* present a scheme for cache coherence verification in the presence of network failures. This scheme cannot be used to ensure correct transition of coherence states. A scheme to verify cache coherence with token coherence was proposed by Meixner *et al.* in [69]. The scheme requires implementation of logical timestamps, signature generation and comparison hardware. In [11], Borodin *et al.* present a distributed system to verify cache coherence. In their solution, each cache that participates in the coherence protocol is assigned a checker that verifies its operation, which enables local verification. Global verification is done by observing the shared bus. This scheme is closest to ours, but its overhead increases linearly with the number of cores in the CMP, unlike ours where a single SC services a number of cores. Furthermore, as will be shown later in this chapter this scheme may be too conservative. In [21], DeOrio *et al.* present an algorithm to verify cache coherence post-silicon. This algorithm, if implemented online, imposes a 26% performance penalty which is unacceptable. Verification of the cache coherence protocol itself was introduced by Zhang *et al.* in [109]. We next present our SC-based cache coherence verification scheme.

2.3.2 The proposed solution

We propose the use of the SC for verifying the coherence protocol in snooping bus multicores. General working of the system and a possible implementation of the system in real hardware are described next. In this work, we assume the use of the MESI protocol [37], but our approach can be applied to any coherence protocol. We refer to the various MESI states as M-Modified, E-Exclusive, S-Shared and I-Invalid throughout this work.

2.3.2.1 General working

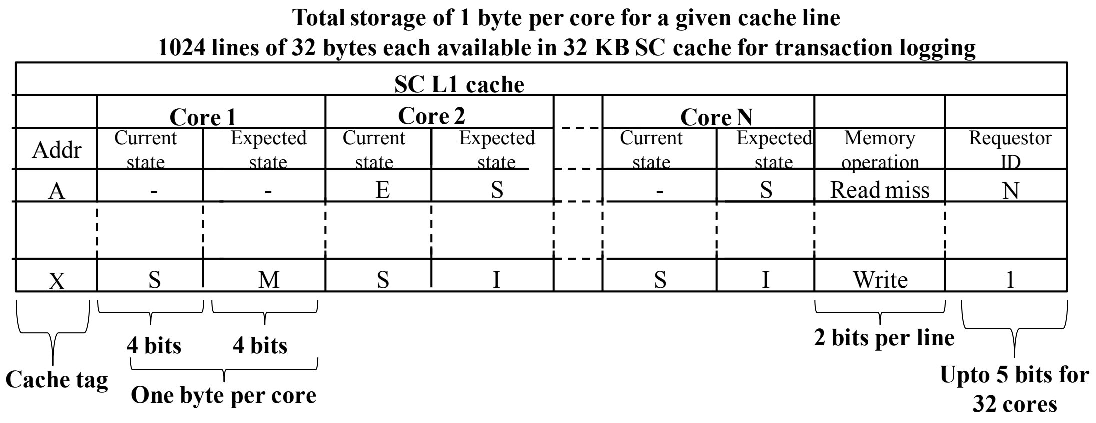
The SC monitors all transactions on the shared bus and makes decisions about the correctness of the transactions. Three steps are involved in the process: (i) Transaction logging, (ii) Verification, and (iii) Retirement.

2.3.2.1.1 Transaction logging This is the first step of the cache coherence verification mechanism. Whenever a cache line is requested due to a read/write miss, it is logged into the L1 cache of the SC. The hardware mapping of each cache access into the SC cache in described in the next sub-section. We assume that along with the address of the memory line being requested, its current coherence state in the sending core is also broadcast. The same assumption has been made by Borodin *et al.* [11]. The SC logs the address of the access, current state of the line and, depending on the transaction, the expected next state of the line. For a given cache line address, entries are maintained for each core in the system. When the line is shared among cores, the corresponding entries are updated, whenever such information is observed on the bus.

2.3.2.1.2 Transaction verification After a request is logged, it is verified once the line appears on the bus again. There are two types of verifications that need to take place, i.e., (i) Local and (ii) Global. Local verification is conducted by computing the expected next state of the transaction for the same core. Whenever the same line appears on the bus, the SC can check if the line transitioned to the expected state. For example, if a core has a read miss and the line was not found in the L1 of any other core, its expected next state should be E (*Exclusive*) since it has exclusive access to the line. Global verification happens by making sure that the state of this line is consistent across cores. For example, a line existing in the S and M states in the L1 caches of two cores is an invalid situation that must be detected. This is done by the SC by comparing the state of the line in each core, that is logged in its own cache.

Verification (local or global) happens whenever the line in question appears on the bus. There are two ways in which this happens. The first is when a core requests a line that is present in the cache of another core. In this case, the owner core will respond to the requesting core with a copy of the line. This information is broadcast on the bus along with the current coherence state of the line. Since this entry must have been logged earlier along with the current state in the logging stage, the SC now has access to the next state of that line. Comparing the current state to the state predicted by the SC enables local verification. If the line is shared by multiple cores, global verification is done by assessing the state of the line across cores. At this stage a new entry is created for the requesting core and the relevant entries are updated in the SC cache. The second verification opportunity arises when the cache of a core is full and lines need to be evicted to make space. If the line that is to be evicted is dirty (M state), a write to memory is initiated so that the main memory is kept up to date. When a write is initiated, the address of the line and its current state are broadcast on the bus and the SC then checks for local and global verification.

2.3.2.1.3 Entry retirement If every line that was accessed is logged but never retired, the SC would need an unlimited cache size to log all the entries and the scheme would not be practical. However, logged cache lines are not needed for an unlimited period of time. Cache lines upon cache conflict have to be evicted from the L1 cache. If the line is in the dirty state, it is written to main memory. Since the cache line is dirty (M), no other core can have a copy of the line and once it has been evicted from the L1 cache, the corresponding entry in the SC L1 cache is retired. Sometimes, cache lines are in states other than M (S or E) and in this case, upon cache conflict, these lines will be overwritten (since the line is consistent with main memory in the S or E states and we do not care about lines in the I state). Whenever this happens, our scheme requires that the SC be notified via the shared bus. The entry is then retired from the SC cache. This event is expected to incur a small


Total storage of 1 byte per core for a given cache line
1024 lines of 32 bytes each available in 32 KB SC cache for transaction logging

SC L1 cache									
	Core 1		Core 2			Core N			
Addr	Current state	Expected state	Current state	Expected state		Current state	Expected state	Memory operation	Requestor ID
A	-	-	E	S		-	S	Read miss	N
X	S	M	S	I		S	I	Write	1

Cache tag
4 bits
4 bits
One byte per core
2 bits per line
Upto 5 bits for 32 cores

Figure 2.12. Mapping cache transactions for each core in the SC cache.

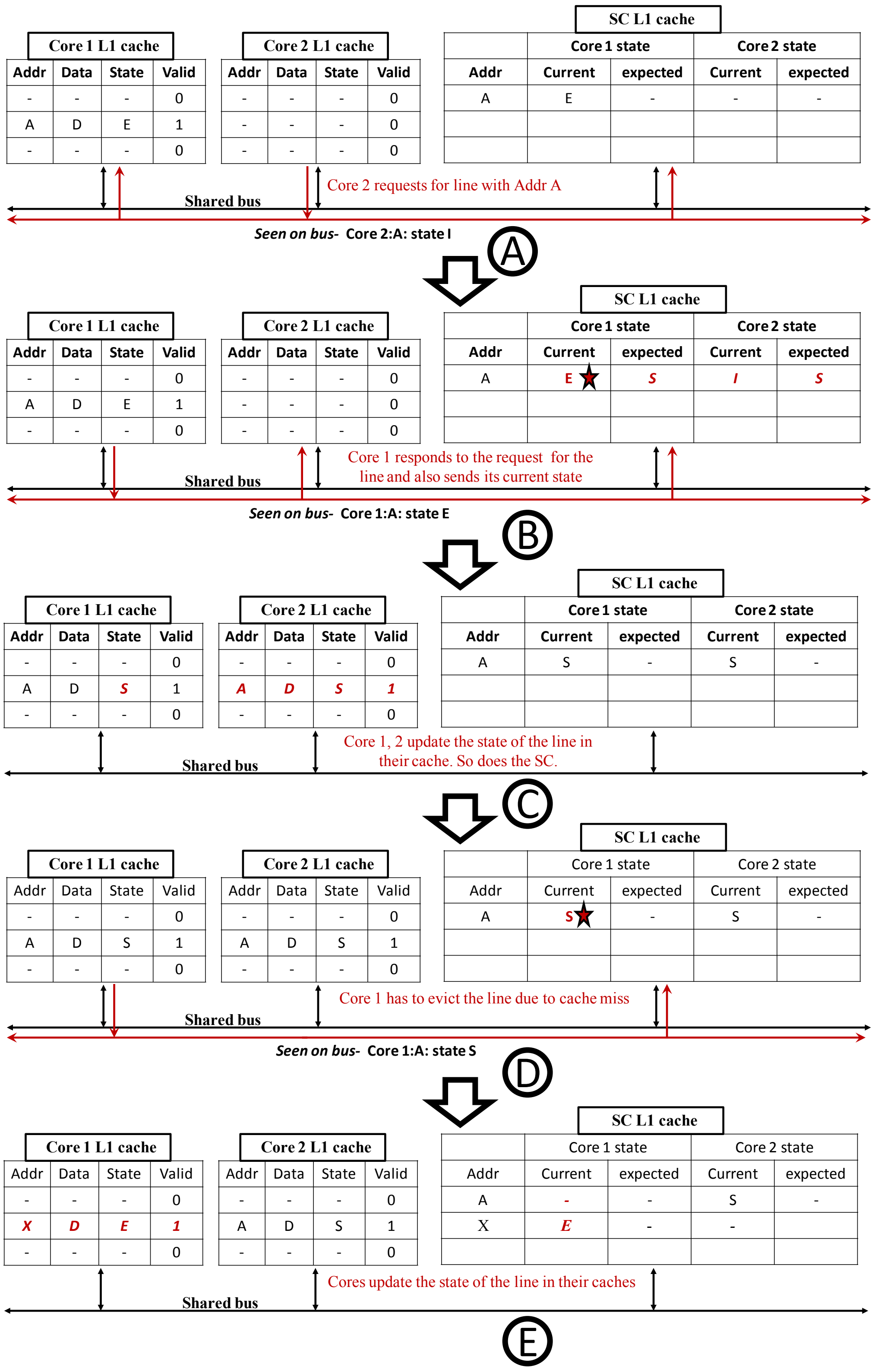
penalty, since it increases traffic on the bus. However, we have observed this penalty to be negligible in our experiments. Entries are also evicted from the SC cache when they are invalidated (I). This also implies that any transaction that appears on the bus with a state other than I, it must be logged in the SC cache, otherwise, an error has occured.

2.3.2.2 Mapping cache access transactions in hardware

We have discussed how the SC logs, verifies and retires transactions from its own cache. In this section we describe how each transaction is mapped into the SC cache in hardware. In the considered system cache lines are assumed to be 32 bytes. The SC cache is assumed to be the same size as that of the general-purpose cores. We have assumed a 32KB L1 cache size and hence the total number of lines available is 1024. The SC cache addressing is done using the same address as that of the operation broadcast on the bus. Since we use the MESI protocol, we assume 4 bits each (13 total states along with transients) for current and expected next states of each core. This requires 1 byte per core. There is also need to log the current memory operation for each line and the requestor ID. Depending on these fields and the current state, the SC can compute the expected next state. Transactions that appear on the bus are either due to a read/write miss, memory push or invalidate. Hence, two bits are reserved for the memory operation and 5 bits for the requestor ID, which allows addressing up to 32 cores. The memory operation and requestor ID fields together occupy a byte, leaving the other 31 bytes to store records for up to 31 cores in the system. The SC cache is implemented as any general-purpose core cache, i.e., with tags, sets and offset. Tags and sets are computed using the address of the memory operation on the bus. Offset is computed using the requestor core ID. Figure 2.12 depicts the SC cache and its entries. Note that for lines exclusively held by a single core, just one entry (1 byte of the available 31) will be used and the rest will be wasted. Also if the number of cores in the system is less than 31, many entries are never used. Instead, if the SC cache was customized such that the number of bytes per line is equal to the number of cores in the multicore, not only would the SC cache be used more effeciently, there would be more entries to store additional cache transactions. However, this would complicate the design of the multicore. To avoid this, we assume that the line size in the SC cache is identical to that in the general-purpose core caches, i.e., 32 bytes.

2.3.2.3 Putting it all together

An example summarizing the above description is presented in Figure 2.13. For simplicity, the memory operation and requestor ID fields in the SC cache have not been shown, but appear in the text in the figure. All state updates are indicated in italic fonts in the caches and any state verifications are indicated by a star alongside the line state. In the example, two cores are considered. The contents of each core cache and SC cache are shown in stages A through E. In stage A, Core 1 has exclusive access to the line at address A. Its state is recorded in the L1 of Core 1 as well as in that of the SC. Core 2 then requests a read for that line. This request is sent on the bus and seen by Core 1 and the SC. The SC logs this request and knows, based on the memory operation and requestor ID, what are the expected next states for both cores. The SC accordingly updates those fields for each core. In stage B,


Core 2 requests for line with Addr A
Shared bus
Seen on bus- Core 2:A: state I
A
Core 1 responds to the request for the line and also sends its current state
Shared bus
Seen on bus- Core 1:A: state E
B
Core 1, 2 update the state of the line in their cache. So does the SC.
Shared bus
C
Core 1 has to evict the line due to cache miss
Shared bus
Seen on bus- Core 1:A: state S
D
Cores update the state of the line in their caches
Shared bus
E

Core 1 L1 cache

Addr	Data	State	Valid
-	-	-	0
A	D	E	1
-	-	-	0

Core 2 L1 cache

Addr	Data	State	Valid
-	-	-	0
-	-	-	0
-	-	-	0

SC L1 cache

	Core 1 state		Core 2 state	
Addr	Current	expected	Current	expected
A	E	-	-	-

Core 1 L1 cache

Addr	Data	State	Valid
-	-	-	0
A	D	E	1
-	-	-	0

Core 2 L1 cache

Addr	Data	State	Valid
-	-	-	0
-	-	-	0
-	-	-	0

SC L1 cache

	Core 1 state		Core 2 state	
Addr	Current	expected	Current	expected
A	*E* ★	*S*	*I*	*S*

Core 1 L1 cache

Addr	Data	State	Valid
-	-	-	0
A	D	*S*	1
-	-	-	0

Core 2 L1 cache

Addr	Data	State	Valid
-	-	-	0
A	*D*	*S*	*1*
-	-	-	0

SC L1 cache

	Core 1 state		Core 2 state	
Addr	Current	expected	Current	expected
A	S	-	S	-

Core 1 L1 cache

Addr	Data	State	Valid
-	-	-	0
A	D	S	1
-	-	-	0

Core 2 L1 cache

Addr	Data	State	Valid
-	-	-	0
A	D	S	1
-	-	-	0

SC L1 cache

	Core 1 state		Core 2 state	
Addr	Current	expected	Current	expected
A	S ★	-	S	-

Core 1 L1 cache

Addr	Data	State	Valid
-	-	-	0
X	*D*	*E*	*1*
-	-	-	0

Core 2 L1 cache

Addr	Data	State	Valid
-	-	-	0
A	D	S	1
-	-	-	0

SC L1 cache

	Core 1 state		Core 2 state	
Addr	Current	expected	Current	expected
A	-	-	S	-
X	*E*	-	-	

Figure 2.13. Working example of transaction logging and retirement. The state verifications are indicated by a star alongside the state in the SC cache and any new state updates are indicated by italics font.

Core 1 responds to the request from Core 2 and broadcasts the state of the line in its cache. This helps the SC to verify the expected state for Core 1. In stage C, a static snapshot of the system with updated states is shown. In stage D, Core 1 has a cache miss and has to evict the line. This is observed on the bus and the SC can once again verify its operation. Stage E shows a static snapshot of the states in the system after the memory eviction for Core 1 is complete and the new line with address X having arrived.

2.3.2.4 Mathematical upper bound on the number of transactions that may be logged and verified

The SC cache is used to log and verify transactions. Hence, the size of the SC cache determines the upper bound on the verification coverage that may be achieved. We now discuss the desired SC cache size that will allow all transactions to be verified. For simplicity, we assume a fully associative cache.

The minimum size of the SC cache required to log all transactions is $\sum linesValid$, where $linesValid$ is the number of valid L1 cache lines, where no two lines have the same address in memory. Note that for two cores sharing a line, only a single entry will be maintained in the SC cache (refer to Section 2.3.2.2). The worst case arises, when every L1 cache line in the multicore is valid and none are shared. In that case, the minimum size of the SC cache is then $n * lines$ where $lines$ is the number of cache lines per L1 cache. In other words, the SC cache must be equal to n times the L1 cache size. It may be noted that this calculation was done using fully associative caches which is not always practical. Considering more realistic set-associative caches this minimum requirement on the cache size may be larger than that just calculated. However, as will be seen in the results, a key observation enables us to keep the required SC cache size realistic.

Table 2.2. Chosen core parameters

Parameter	Value	Parameter	Value
Issue	6	INTREG	96
FPREG	80	INTISQ	36
FPISQ	24	Load/Store units	3
LSQ	32	ROB	128
L1(I/D)	32K	L2	2M
L1 associativity	8	L1 Linesize	32 bytes
L2 associativity	8	L2 Linesize	32 bytes
L1 hit latency	2 cycles	L1 miss latency	10 cycles
L2 hit latency	15 cycles	L2 miss latency	200 cycles
Freq (GHz)	2.4	Operation	Out of order

2.3.3 Experimental setup

The shared memory multicore was simulated using the SESC simulator [75] which was modified considerably to enable cache coherence transaction verification via the SC. We used the SPLASH-2 workloads [107] for our experiments (*cholesky, barnes, fft, fmm, lu, ocean, radix* and *water*). Each core in our multicore represents an Intel Nehalem processor. The specifications of the core parameters that we have used are shown in Table 2.2. We consider 8 cores in the multicore for all our simulations and we simulate the workloads for 100 million instructions.

2.3.4 Results

We now present the results of using the SC for cache coherence verification. The SC can verify transactions once they appear on the shared bus and is unable to verify any transaction until it is seen on the bus. Hence, we present the fraction of cache coherence transactions that can be verified. Note that any unverified transactions will be verified in the near future when they will be seen on the bus, but after a certain number of elapsed cycles. Cache line sharing is a function of the benchmark used and thus, we analyzed the required size of the SC cache for each benchmark. Following this, results are presented when using a realistic SC cache size to evaluate the effectiveness of the system. sizes, the above mentioned experiments are carried out for various cache sizes.

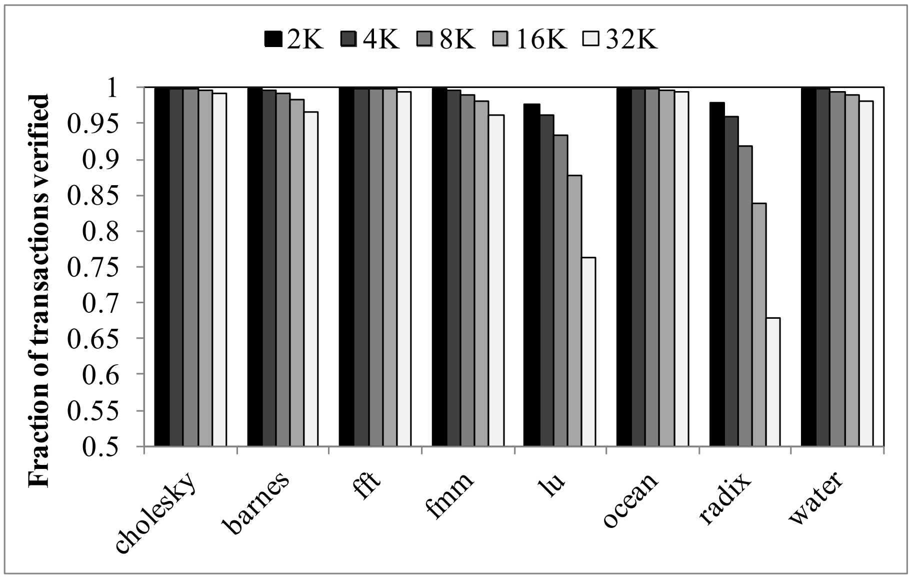


Figure 2.14. Fraction of transactions verified for various cache sizes when using unlimited SC cache size.

2.3.4.1 Unlimited SC cache size

We now present results showing the fraction of transactions verified for unlimited SC cache size and also the upper bound on the required SC cache size such that maximum verification is possible for all benchmarks. The results are plotted in Figures 2.14 and 2.15, respectively, for various general-purpose core cache sizes and various benchmarks. Figure 2.14 shows that in general, a very high coverage is obtained for smaller cache sizes and this reduces with increasing cache size, for the 100 million instructions that we simulated. This is intuitive since the smaller the cache, the larger is the number of cache conflicts and evictions. Also with smaller caches, a small proportion of the lines reside inside the L1 caches as compared to the total transactions seen on the bus. This also increases the fraction of verified transactions. It can be seen that other than *radix* and *lu*, all other workloads show greater than 0.9 coverage even when using a cache size of 32K. The reason for low coverage for *radix* is that it comprises almost 90% floating-point operations and involves very limited sharing of data. Most of the cache lines are exclusive and reside in the local cache for long periods of time. Cache miss rates were also observed to be small for both workloads, leading to fewer transactions on the bus, resulting in low coverage. It may be noted that the fraction of verified transactions asymptotically tends to 1 as the number of instructions executed increases. This is because the unverified transactions are always dependent on the size of the general-purpose caches for reasons just mentioned. But as the time increases, the number of transactions occuring on the bus is very high and the fraction of unverified transactions reduces to zero. Hence, even though some of the values in the plot suggest low verification ratio, it is due to the 100 million instructions that we ran. Increasing this number will increase the verification ratio. The number of cache lines required by the SC to log all entries is shown in Figure 2.15. In the worst case, no cores in the system will share lines and the cumulative sum of the lines occupied by all cores may need to be stored. From the figure, it can be seen that barring the workloads *barnes, lu, water*, the amount of storage required is the cumulative sum of all cache sizes and hence the capacity requirement is very high (almost 256K for 32K cache sizes). This may imply that in order to achieve high verification rates for larger cache sizes, the SC cache size may needs to be prohibitively large. Fortunately, a key insight makes sure that this is not the case.

2.3.4.1.1 Discussion From the results so far, we have seen that for verifying all possible transactions using the SC, the SC cache size may have to be equal to the sum of the cache sizes of all the cores in the system, for certain workloads. However, the transactions considered so far include lines that are exclusively held in the cache of a single core and those that are shared amongst the cores. We have seen in Section 2.3.2.2, that this situation results in the worst storage efficiency for the proposed scheme. However, if something goes wrong in computing the cache line state while the line is held exclusively, it very rarely results in error. For example, faulty change of state of a line from one of the valid states (S or E) to M will result in write back when this line is evicted from the cache, but the data will not be corrupted. The

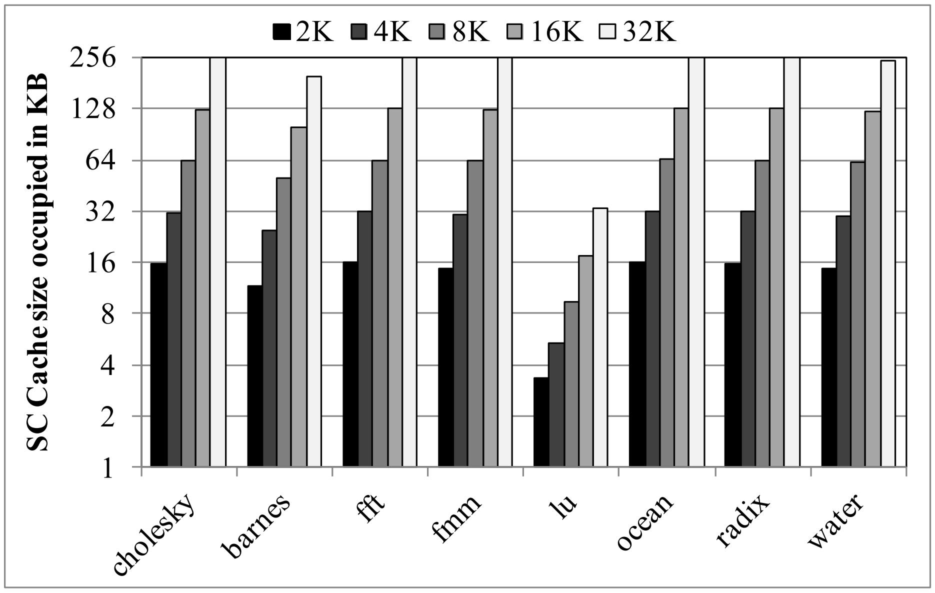


Figure 2.15. Maximum size of the SC cache required such that all possible transactions seen on the bus are verified.

unnecessary write-back will have a small performance penalty, but may be worth it if the trade-off allows all other transactions to be verified. The more malicious case is when a fault causes a line in M state to move to a different state. Here, upon eviction the line will not be written to memory and the memory will no longer be up to date. This situation can be avoided by special encoding of the MESI states. For example, one hot coding for the four MESI states will ensure that no single bit error will ever go unnoticed. Thus, it may not be necessary to verify exclusively held cache line states. The more interesting but challenging case is the verification of line states globally across cores. We have observed that when the verification of the exclusive line states is excluded, the cache size requirement of the SC to log all transactions drops dramatically. Figure 2.16 shows the percentage of transactions that are shared amongst cores for various cache sizes. It can be seen that barring *barnes*, shared transactions for all other workloads account for a very small percentage of the total transactions. Hence, by dropping the verification of exclusive states, the size requirement for the SC cache can be reduced dramatically. In the next sub-section,

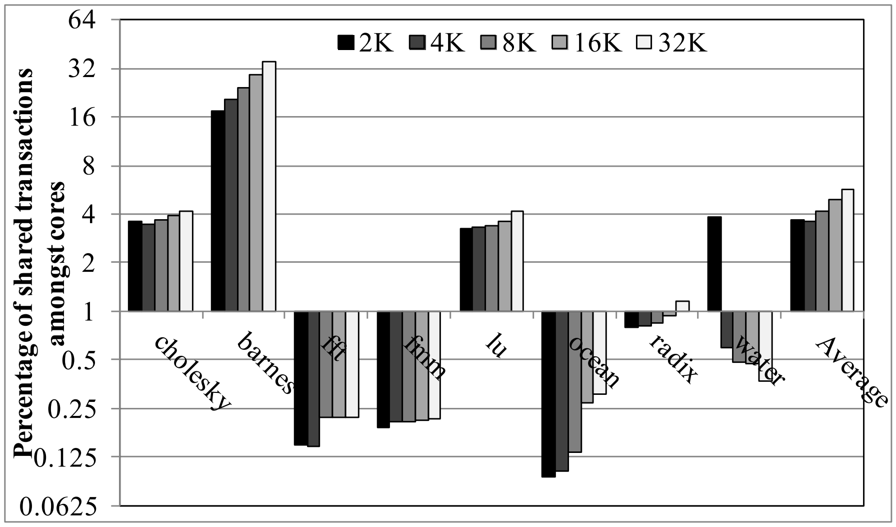


Figure 2.16. Percentage of transactions that involve sharing cache lines amongst cores for each workload and various general-purpose core cache sizes.

we focus therefore, on verifying only the transactions that involve lines shared among cores.

2.3.4.2 Realistic SC cache size

We now present the results of our experiments to determine the capability of the SC to verify memory transactions in a more realistic scenario. Here the SC cache size is not unlimited and its associativity is identical to that of the caches of other cores in the multicore.

2.3.4.2.1 Percentage of transactions verified We varied the size of the SC cache as well as that of the general-purpose cores from 2K to 32K. In total, we ran experiments for each workload for the 25 possible combinations of cache sizes of the SC and the general-purpose cores. We show the results for the workloads which exhibited the worst cases, i.e., *barnes*, *cholesky* and *ocean* in Figures 2.17, 2.18 and 2.19, respectively. For a small SC cache size, it is expected that the cache conflicts in the SC cache during transaction logging will be high and thus, the percentage of transactions verified will be smaller. For a fixed general-purpose core size, it can be seen that the percentage of unverified transactions drops drastically with increasing

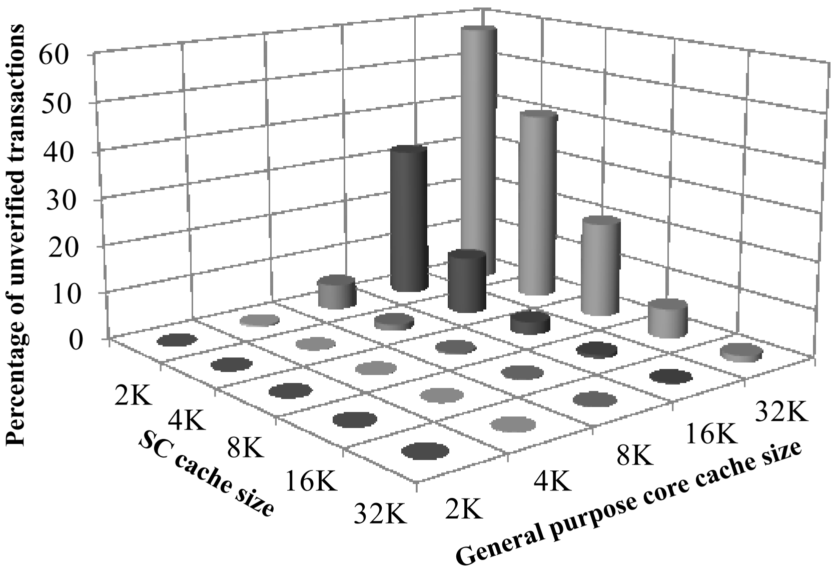


Figure 2.17. Percentage of transactions unverified for the workload *barnes* for various combinations of SC and general-purpose core cache sizes.

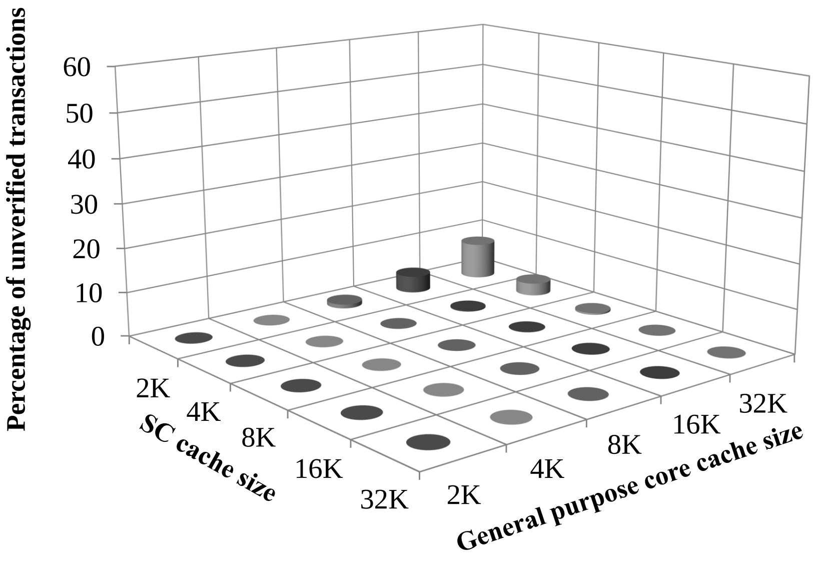


Figure 2.18. Percentage of transactions unverified for the workload *chokesly* for various combinations of SC and general-purpose core cache sizes.

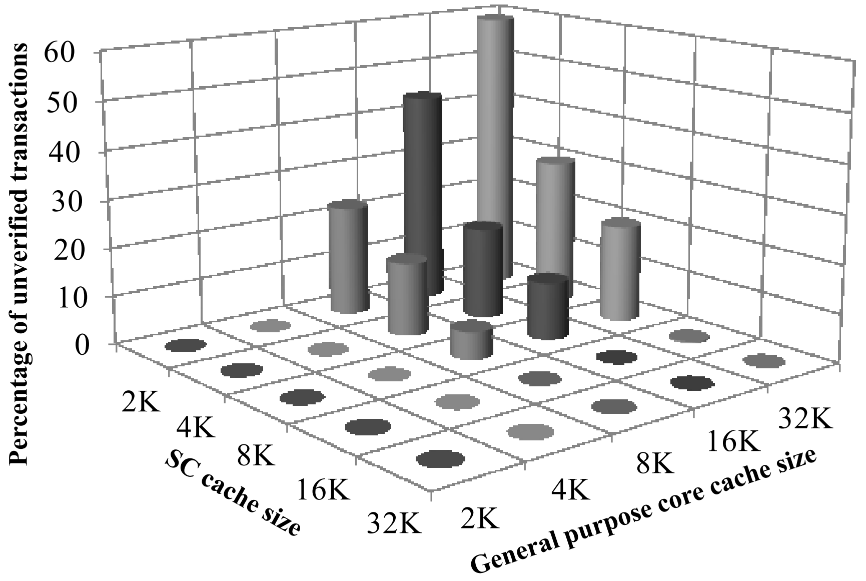


Figure 2.19. Percentage of transactions unverified for the workload *ocean* for various combinations of SC and general-purpose core cache sizes.

SC cache size, which is expected. The worst case of 59% transactions unverified is observed for the workload *barnes* when the SC cache size is set to 2K and that of the general-purpose cores to 32K. However, this is not a realistic scenario as in general, it is expected that the SC cache size will be at least equal to that of the general-purpose cores. Looking at the data points in Figure 2.17 that represent equal SC and general-purpose core cache sizes, it can be seen that in almost all cases 100% transaction verification is possible. The only combination where this is not the case, is when the cache sizes are set to 32K for *barnes*, where 1.3% unverified transactions were observed. This is a very small fraction and this greatly increases our confidence in the capability of the proposed scheme. The other workload that showed missed transactions for same SC and general-purpose cache size is *ocean*, where 5.6% of the transactions were missed for a cache size of 8K. For the rest of the workloads, setting the SC cache size to 16K is enough to verify all transactions even when the general-purpose core caches are set to 32K. In Table 2.3, the minimum size of the SC cache required for 100% verification of shared transactions amongst the 8 cores with L1

Table 2.3. Minimum SC cache size required for ≈100% transaction coverage when using general-purpose cores with L1 cache 32K

Workload	Min SC L1	Workload	Min SC L1
cholesky	16K	lu	2K
barnes	32K	ocean	16K
fft	2K	radix	2K
fmm	2K	water	2K

cache set to 32K, is shown. For a majority of the workloads, an SC cache of just 2K suffices. By excluding the verification of the the exclusive cache states, 100% of the transactions can be verified using realistic SC cache sizes. This result also shows that the proposal made by Borodin *et al.* [11] where a replica is maintained for each cache, is pessimistic, since there for 8 cores and 32K caches, the total checker cache size is 256K, as compared to 32K in the worst case for our scheme. Note that when using 32K cache size, only 1.3% transactions are missed in the worst case of *barnes*.

2.3.4.2.2 Sensitivity to number of cores in the CMP Since the proposed scheme is a centralized mechanism, the central verifier can be a bottleneck. To evaluate the scheme thoroughly, we varied the number of cores in the CMP from 2 to 32. L1 cache size was set at 32KB. It is expected that with an increase in the number of cores, the subset of lines shared by different pairs of cores may increase, which may be too much for the limited SC cache to handle. The obtained transaction coverage for three of the workloads *barnes*, *ocean* and *raytrace* are shown in Figure 2.20. Note that for the other workloads, no drop in coverage was observed due to the low proportion of transaction sharing even in a CMP comprising of 32 cores. In general, increase in the number of cores reduces coverage which is expected. The workload *barnes* shows the worst behavior which is once again attributed to the relatively high number of shared transactions. When considering 32 cores, only 92% of the transactions were verified. For *ocean* and *raytrace*, the worst case was observed was 92.5% and 95% once again for 32 cores. It is interesting to note that even though *ocean* was observed to have a lower proportion of shared transactions when compared to *raytrace*, the

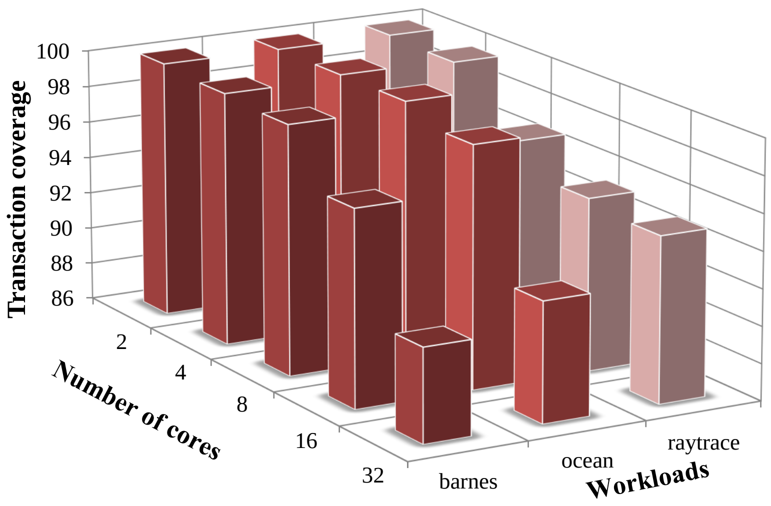


Figure 2.20. Sensitivity of the coverage of the proposed scheme to the number of cores in the CMP for the workloads *barnes*, *ocean* and *raytrace*.

coverage drops faster. The reason for this is the lack of locality in the reference for shared lines produced by it. Hence, even though the number of transactions is rather small, the addresses of lines accessed causes conflict misses in the SC cache which leads to poor transaction coverage. On an average, across all the 8 workloads considered, the transaction coverage observed was 100% , 99.9%, 99.5%, 98.9% and 97.3% considering CMPs with 2, 4, 8, 16 and 32 cores respectively. This shows that even though an increase in the number of cores will result a drop in transaction coverage, it does not drop by much. We also conclude from these numbers that the SC based scheme for cache coherence verification is scalable up to 16 cores without much of a loss in coverage.

2.3.4.2.3 Time to verification time and performance penalty The SC provides error detection. Error recovery is assumed to be in place using a checkpointing scheme [96]. If the error detection latency is larger than the checkpointing interval, the system state will be corrupted. Thus, the latency to error detection is important. In our experiments, we have observed that even in the worst case, the transaction verification latency is a few thousand cycles which is well within reasonable checkpointing intervals. The proposed scheme also results in increased bus traffic in the cases where cache lines in the general-purpose cores are overwritten without write back. We observed a 20% increase in bus traffic in the worst case, but this resulted in performance loss of less than 2%.

2.3.5 Conclusions

We have presented and evaluated a new centralized mechanism to verify the cache coherence transactions in a shared memory snooping bus multicore. The proposed scheme is based on the incorporation of a small and simple Sentry Core that can be assumed to be fault-free. The SC has access to the shared bus and it can log memory requests seen on the bus, in its cache. Since it is aware of the cache coherence protocol, based on the memory operation and the current state of the line requested, the SC knows the expected next state for the line. Whenever the same line is seen again on the bus, the SC compares the state of the line to what it computed and flags an error if a discrepancy is found. As the scheme depends on logging of transactions in a cache, its capabilities are determined by its cache size. Results were presented on the upper bound of the scheme for unlimited SC cache size. A realistic scenario was then presented where the SC cache was assumed to be similar to that of the general-purpose cores. Results were presented for various combinations of SC and general-purpose core cache sizes. These results indicate that in a realistic scenario of equal SC and general-purpose core cache sizes, >94% of the transactions can be verified. The performance penalty arising from the scheme was found to be less than 2% in the worst case. Our analysis also shows that using a centralized checker for cache coherence may result in far lower hardware overhead in terms of additional cache space required for checking.

CHAPTER 3

IMPROVING POWER EFFICIENCY IN ASYMMETRIC MULTICORES

The growing transistor density in microprocessors has enabled very high performance. However, the future generations of processors will be severely limited by energy [40, 9]. Even though transistor dimentions have scaled, the supply voltage scaling has been incremental if at all since the 130nm technology node [40]. The major reason for this is that there is a limit on how much the transistor threshold voltage can be scaled. As a result, transistor switching power has remained more or less constant [65]. Further, during the evolution of the microprocessor over the past decades, a number of performance enhancement techniques were applied at the circuit and architecture level and all of these come at the cost of increased energy consumption and power. The operation of every processor is limited by a power envelope as defined by the packaging thermal limits. Since, the transistor density has increased, but switching energy has not decreased, switching power density has become prohibitive. As a result, we now have a situation where there is abundance of transistors, but not all of them can operate at the same time. As a result, for long now, there has been ongoing research on decreasing energy and power consumption.

Asymmetric multicore processors (AMP) is one of the means explored to achieve power efficiency in multicores [51, 52, 54, 28, 1]. Here cores of differing capabilites comprise the multicore. Each core is well suited to run a subset of the potential applications that will be run. During runtime, dynamic thread scheduling is facilitated such that the best thread to core assignment is achieved at runtime. This results in better resource utilization, lowering of resource idling and hence static power when compared to their symmetric counterparts [54, 38, 106]. Hence power efficiency is increased. However, thread scheduling in AMPs remains a difficult problem [4]. In this chapter, we explore a couple of different strategies to schedule threads in AMPs online.

3.1 Related Work

With the growing popularity of AMPs, a number of proposals involving dynamic thread scheduling have been made on the subject. A brief survey on AMP architectures and dynamic thread scheduling in AMPs is now presented.

3.1.1 AMP architectures

Kumar *et al.* in [51] proposed an architecture consisting of four core types. Each core is designed such that all four of them fall at different points in the performance and power continuum. Dynamic thread scheduling between these cores is made at runtime to improve power efficiency. They later extended this scheme to multithreaded applications in [52]. They also explored the design of an AMP, targeting area and power efficiency in [54]. They use cores that match the resource requirements of certain types of workloads. Ghasi *et al.* [28] have also explored the benefits of AMPs for performance, power energy delay product etc. In [1], Suleman *et al.* propose an AMP consisting of cores of two types, one big and the other small. The big cores are used to accelerate the critical serial portions of the code while the smaller ones are used for the parallel portions. Apart from academia, recently, ARM has introduced an AMP architecture called the big.LITTLE core [30] which consists of big and small cores. The emergence of AMPs in the industry shows the general trend in the industry regarding multicore design.

3.1.2 Thread scheduling in AMPs

With AMPs becoming more common, a number of thread scheduling techniques have been recently proposed. We briefly overview the prior schemes which can be broadly classified into those that employ offline profiling, online learning via sampling and online estimation.

There have been a number of solutions based on offline profiling to determine the best thread to core scheduling in AMPs. Khan *et al.* [45] propose regression analysis along with phase classification to identify thread to core affinity. Shelepov *et al.* [86] profile applications to determine what they call architectural signature of the application. This signature (characterized by L2 cache misses) is unique for each core type and is used to determine the thread scheduling online. In [18], Chen *et al.* use cores in an AMP that differ with respect to issue width, branch predictor size and L1 caches. They use multi-dimensional curve fitting to determine the optimal thread to core assignment offline. All these approaches rely on offline profiling and are not practical, since they require knowledge of the workloads that will be run on the multicore.

Online learning based schemes offer a more practical solution to the AMP scheduling problem. Kumar *et al.* [51] proposed an AMP consisting of cores of various sizes, all running the same ISA. Whenever a new program is run or a new phase [87] is detected, sampling is initiated and the core which provides the best power efficiency is chosen. A similar approach was proposed by Becchi *et al.* [6] for performance maximization of an AMP consisting of two types of cores. Optimal thread scheduling was determined by forcing a thread swap between cores upon detection of phase change. Winter *et al.* [106] study power management techniques in AMPs via thread scheduling. They consider several algorithms, all based on program sampling.

There have also been proposals made on online estimation based schemes. Here, based on the current characteristics of a workload being executed, its performance on other core types of the system is estimated. However, the benefits of the scheme will be determined by the accuracy of the estimation. Saez *et al.* [84] propose a comprehensive scheduler for AMPs consisting of small and big cores using last level miss rates of an application to estimate its performance on each core type. It is, however, unclear whether using L2 misses alone is sufficient to make thread to core assignment decisions such that performance/Watt is optimized. In [100], Srinivasan *et al.* estimate the performance of the thread currently running on one core type, on another core, using a closed form expression. These expressions were developed for specific cores and a general approach was not provided. Koufaty *et al.* [50] determine thread to core mapping in an AMP consisting of big and small cores, using program to core bias which is estimated online using the number of external stalls (proportional to cache requests going to L2 and main memory) and internal stalls (front end not delivering instructions to the back end).

In this dissertation, we have explored the benefits of dynamic thread scheduling in AMPs.

3.2 Improving the Performance/Watt of Asymmetric Multicores via Phase Classification and Adaptive Core Morphing

3.2.1 Overview of the solution

A dual core AMP architecture is considered where each core is moderately sized and have complementary strengths. One of the two cores has strength in executing integer (int) workloads while the other floating-point (fp) workloads. Thus, each core is suited for specific workloads. The baseline architecture is shown in Figure 3.1. The system always executes two threads with one thread on each core. The proposed scheme also involves a novel core morphing functionality, where at runtime, the two cores exchange execution resources. When it is determined at runtime that the work-

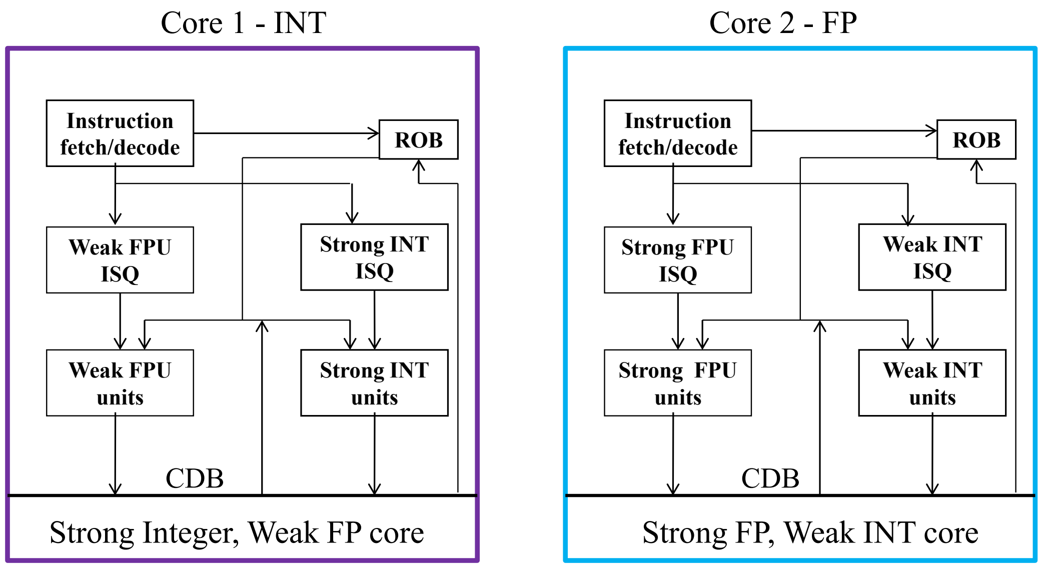


Figure 3.1. Baseline configuration for two heterogeneous cores.

load may fare better with a core strong on both the int and fp fronts, the two cores exchange resources such that in the resulting multicore, one core is strong on all fronts and the other, weak on all fronts (details to soon follow). There are several benefits to this approach.

1. It allows applications to exploit the most suitable core for better performance.
2. Individual cores remain modest in their sizing, therefore allowing the AMP to meet the cost and power targets
3. When operated in the morphed mode, the realigned resources enable higher levels of performance for the applications that can benefit from them

The benefits of AMPs have been stressed upon earlier. Further studies [70, 72, 38, 31, 41, 48, 72, 47, 65] have shown that reconfigurable architectures may increase the benefits of AMPs even further. This provides a strong argument for our target multicore architecture. In this work we use hardware performance monitors to discover thread to core affinity during runtime. Such discovery may trigger a thread swap or core morphing. The trigger to initiate thread swapping or core morphing needs to be determined online. We explore a couple of approaches, one based on offline learning and the other online learning. In the offline scheme (called the Rule based scheme), a subset of the workloads are studied and based on their characteristics, rules to determine core reconfiguration are determined. These rules are then used to make decisions online. In the other scheme no learning is required. Instead, the workload characteristics are learned online using sampling and phase classification. We evaluate and compare both the schemes.

3.2.2 The proposed solution

In this section, we describe in detail our proposed dynamic core morphing (DCM) scheme. The target AMP consists of two cores per tile: a FP core and an INT core where a multicore system may consist of as many such tiles as deemed appropriate. The FP core features strong floating-point execution units but low performance integer execution units, while the INT core features exactly the opposite. Other differences between the cores include the number of virtual rename registers, issue queues (ISQ) and LSQ. The values for these parameters were decided after extensive sizing experiments explained in Section 3.2.3.

In the baseline configuration (Figure 3.1) the cores operate independently providing good performance with each core executing one thread. However, whenever it is determined that a workload requires both floating-point and INT performance, a dynamic morphing of the cores takes place. In this configuration, the INT core takes control of the strong floating-point unit of the FP core to form a strong "Morphed core" while relinquishing control of its own weak floating-point unit to the FP core. The FP core thus becomes a "weak core." Morphing results in two cores: (i) a strong single-threaded core capable of handling both integer and floating-point intensive applications efficiently, and (ii) a weak core which consumes less power and does not provide high performance. Instead of retaining the front end of the FP core as is,

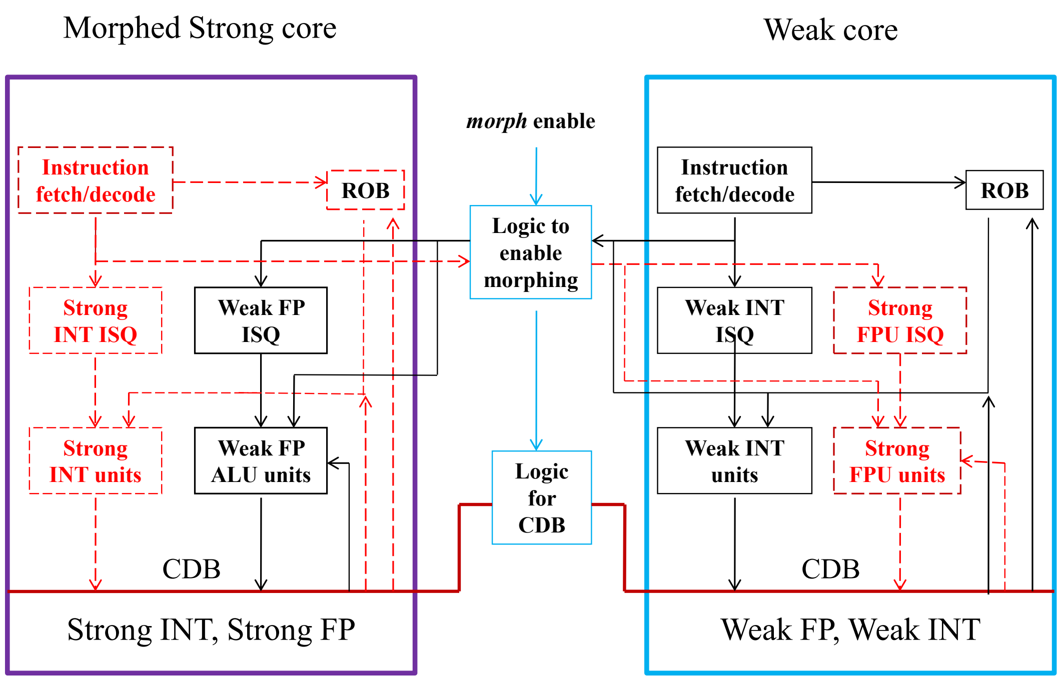


Figure 3.2. Morphed configuration for two heterogeneous cores. The red dotted lines/boxes indicate the connectivity for the strong morphed core configuration and the black solid lines/boxes indicate connectivity for the weak core.

its resources are appropriately sized down, as explained in Section 3.2.3, to suit the application running on it and reduce power. The proposed dynamic morphing of the cores is shown in Figure 3.2. If the morphed mode is no longer beneficial, the system reconfigures itself back to the baseline mode.

Workload behavior tends to change with time. Hence, the ability to swap threads between the two baseline cores could reduce the execution time significantly. Reduced execution time would improve the performance/Watt with less idling and thus more efficient utilization of resources. Therefore, in addition to the baseline and morphed modes of operation, we also allow the two tightly coupled heterogeneous cores to swap their execution contexts.

The proposed DCM scheme is a hardware-only solution that is autonomous and isolated from the Operating System (OS) level scheduler. We assume that only the initial scheduling is done by the OS in the baseline configuration. From then onwards, the thread to core assignment is managed autonomously by our scheme to optimize performance/Watt at fine-grain time slices.

From Figure 3.2, it can be seen that the proposed scheme requires additional hardware such that runtime exchange of execution units is possible. To that end, it must be augmented with multiplexers and logic to forward data. However, we estimate the overhead due to this logic to be very small when compared to the area of a dual core AMP.

3.2.3 Determining the core parameters

We intend to design the two cores such that each core is moderately sized and is capable of running a wide variety of workloads with reasonable performance. We conducted core sizing experiments to determine the sizes of the structures in the two core types. Our goal is to focus on a set of parameters that have the largest impact on the INT and FP cores. If the cores are undersized, the results of core morphing would be biased and misleading.

We used SESC as our architectural performance simulator [75], and CACTI [89] and Wattch [13] to estimate power.

3.2.3.1 Benchmarks

For the experiments, 38 benchmarks were selected from the SPEC suite [97], MiBench suite [33] and the mediabench suite. [57]. The instruction type distribution of the selected benchmarks is depicted in Figure 3.3 showing the diversity of workloads.

3.2.3.2 Core sizing

To determine the architectural parameters for the cores, we have started with a baseline configuration and then upsized the parameter under consideration and recalculated the instructions per cycle (IPC) metric for each core type. Based on the

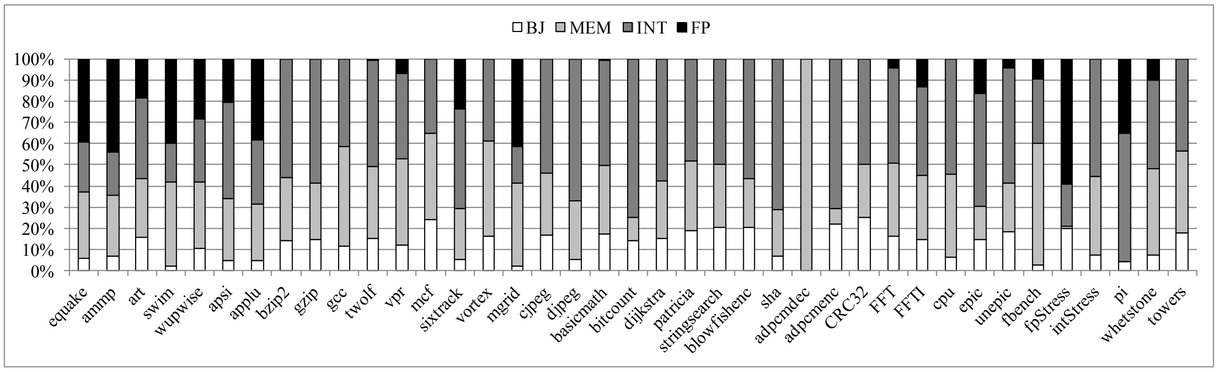


Figure 3.3. Instruction composition of the 38 benchmarks when run for 5 billion instructions.

Table 3.1. Parameter variation steps for the experiments

Parameter	Size	Variation steps
DL1	32K	4-8-16-32
IL1	32K	4-8-16-32
L2	256K	32-64-128-256
LSQ	64 (each LD/SD)	16-32-48-64
ROB	256	32-48-64-128-256
INTREG	128	32-48-64-128
FPREG	80	32-48-64-80
INTISQ	128	16-32-64-128
FPISQ	64	8-16-32-64

Table 3.2. Core configurations after the sizing experiments

Parameter	FP	INT	HMG	Weak
DL1	4K	4K	4K	1K
IL1	4K	4K	4K	1K
L2	128K	128K	128K	64K
LSQ (each LD/SD)	32	32	32	32
ROB	128	128	128	64
INTREG	48	64	56	32
FPREG	64	32	48	32
INTISQ	32	32	32	16
FPISQ	32	16	24	8

Table 3.3. Execution unit specifications for the cores. (P - Pipelined, NP - Not pipelined)

Core	FP DIV	FP MUL	FP ALU
FP	1 unit, 12 cyc, P	1 unit, 4 cyc, P	2 units, 4 cyc, P
INT	1 unit, 120 cyc, NP	1 unit, 30 cyc, NP	1 unit, 10 cyc, NP
HMG	1 unit, 66 cyc, NP	1 unit, 17 cyc, P	2 units, 7 cyc, P
	INT DIV	INT MUL	INT ALU
FP	1 unit, 120 cyc, NP	1 unit, 30 cyc, NP	1 unit, 2 cyc, NP
INT	1 unit, 12 cyc, P	1 unit, 3 cyc, P	2 units, 1 cyc, P
HMG	1 unit, 66 cyc, NP	1 unit, 16 cyc, P	2 units, 1 cyc, P

IPC, the most appropriate value for each parameter was selected. The baseline configuration along with the steps used for the parameter search are shown in Table 3.1.

The parameters that were varied for design space exploration were the L1 and L2 caches, reorder buffer (ROB), load store queue (LSQ), integer issue queue (INTISQ), floating-point issue queue (FPISQ), floating-point registers (FPREG), and integer registers (INTREG). For the sake of brevity only ROB sizing results are shown in Figure 3.4. In the figure, each curve represents the ratio of the performance for the core when going from a smaller to larger ROB. For the FP core, it can be seen that there are several benchmarks that benefit when going from ROB of size 64 to 128 (*equake, swim, applu, twolf, wupwise, fft, ffti and whetstone*) but such benefit is no longer seen when increasing the ROB size to 256. Hence, the size 128 is chosen for the FP ROB. Based on similar observations, the ROB for the INT core was also sized

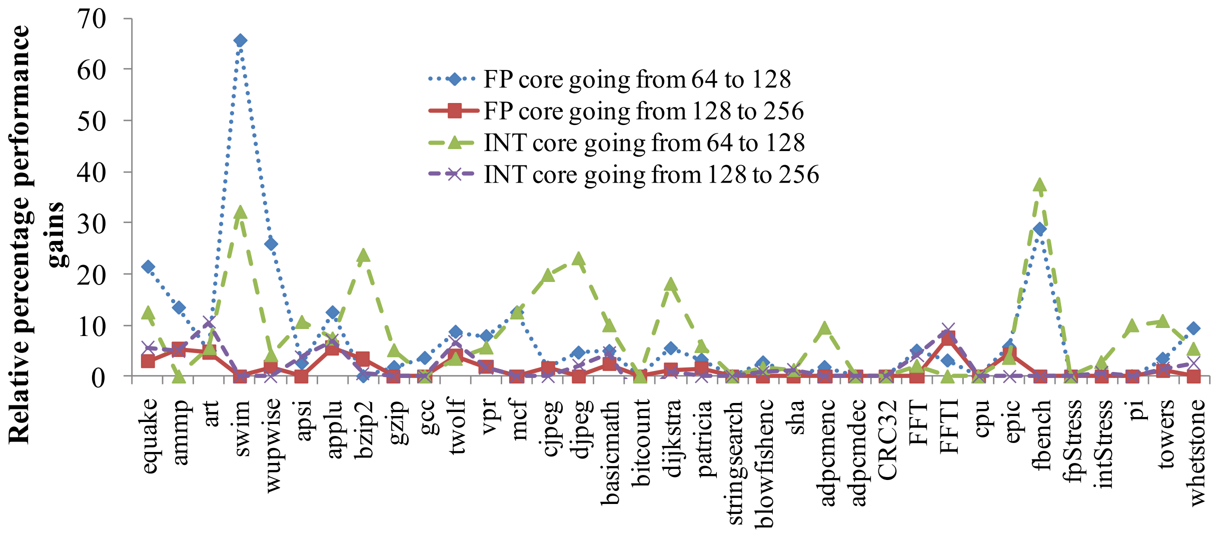


Figure 3.4. Ratio of the IPC for the core configurations when going from lower to higher sizes of ROB.

to 128. Similar sizing experiments were conducted for the rest of the parameters and the resulting core configuration is shown in Table 3.2.

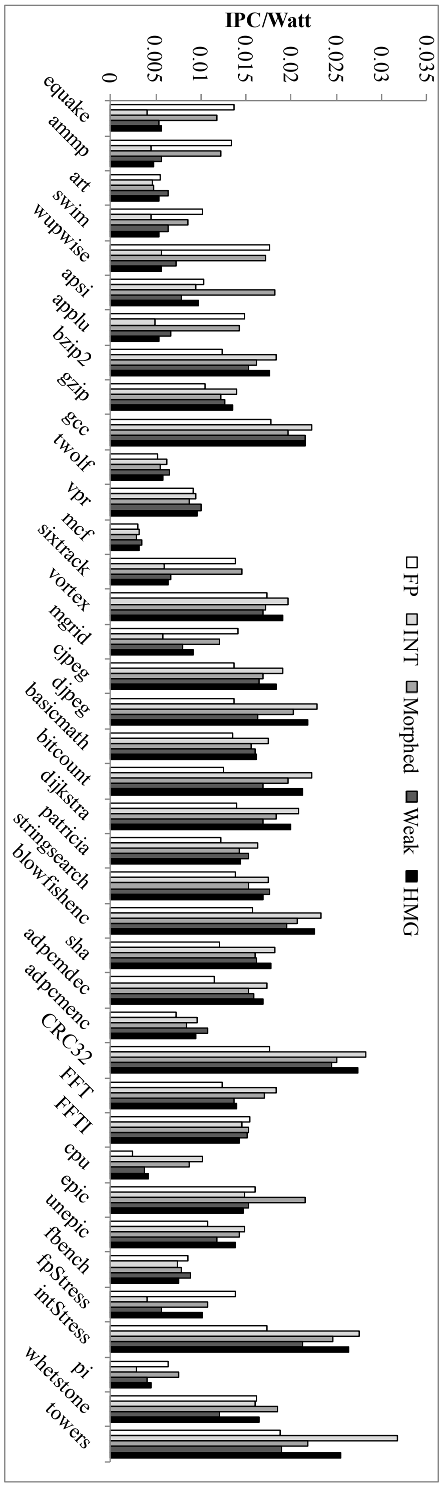
For comparison, we intended to consider a homogeneous (HMG) dual core. For a fair comparison between our dual-core AMP and a HMG design, the area of two HMG cores should match the sum of the areas of the FP and INT cores. Hence, the sizes of the structures for HMG were obtained by averaging those obtained for the INT and FP cores. As mentioned earlier, whenever the multicore enters the Morphed mode of operation, the FP core turns into a Weak core. Since this core is not expected to provide a performance as high as the original FP core, we did similar sizing experiments to try and downsize this core for energy efficiency. This is once again reflected in Table 3.2. We did not include the final configuration of the Morphed mode as it is nothing but a combination of the INT core with the FP units of the FP core. The performance/Watt and performance of these cores are discussed in the next section. The specifications of the execution units is shown in Table 3.3.

3.2.4 Performance/Watt and performance evaluation

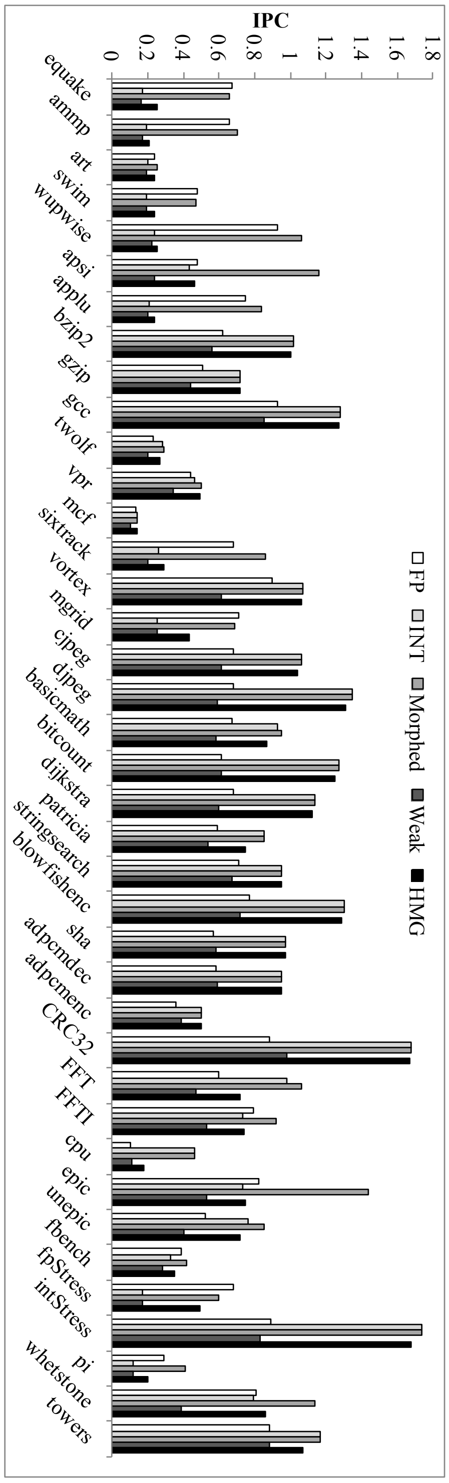
In this section, the performance/Watt and performance of each core is analyzed by running one application at a time on the various core types, i.e., FP, INT, Morphed, HMG and Weak cores.

Each workload was run on each core type for 5 billion instructions and the IPC/Watt and IPC results are plotted in Figure 3.5. With respect to performance/Watt, we observe that 5 benchmarks (*apsi, sixtrack, epic, pi, whetstone*) in the morphed mode show notable gains. Out of these, *apsi* shows 82% improvement over its closest competitor, the FP core. This benefit is more modest for the benchmarks *epic* (35%), *whetstone* (17%), *pi* (12%) and *sixtrack* (5%). The reason why *apsi* shows substantial benefits is related to the temporal distribution of the instruction mix in *apsi*. We have observed that this happens due to the bursty nature of the instruction types encountered when executing *apsi*.

What is depicted in Figure 3.5 represents the average behavior over 5 billion instructions. However, program behavior may change over time. Hence static thread to core scheduling may not be optimal. This is the reason why only 5 out of the 38 benchmarks show benefits when run on the morphed core throughout their execution. In the rest of the 33 cases, the power expended by the morphed core outweighs the obtained performance benefits resulting in poor performance/Watt over the entire run. This is evident from Figure 3.5(b) that shows the IPC for all benchmarks on the four types of cores. As can be seen from this figure, the morphed core performs either equally well or better than the other core configurations when only IPC is considered. Moreover, there is a bigger group of benchmarks (*ammp, wupwise, apsi, applu, sixtrack, FFT, FFTI, epic, unepic, fbench, pi, whetstone*) that show significant benefits from morphing and the gains are even higher (>150% for *apsi*). As we have seen, such performance gain does not always result in a higher power efficiency.



(a) IPC/Watt for all benchmarks



(b) IPC for all benchmarks

Figure 3.5. IPC/Watt and IPC for the 38 benchmarks considered when run on each core configuration for 5 billion instructions.

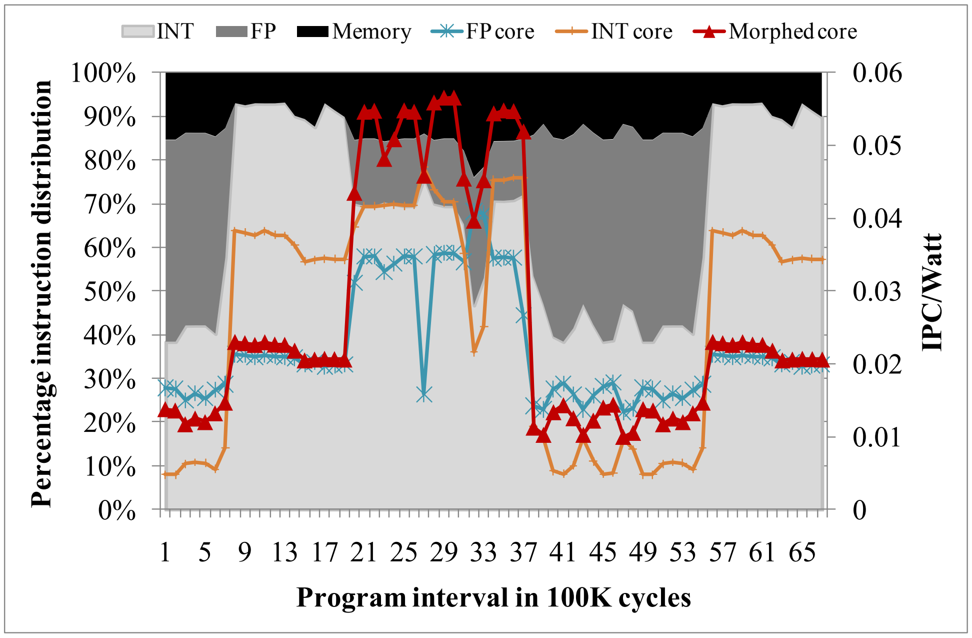


Figure 3.6. Zoomed view of variations in the performance/Watt of *epic* when run on each core configuration.

3.2.4.1 Impact of program phases

We have seen that when static thread-to-core assignment is considered, over the entire run of 5 billion instructions, some benchmarks benefit, some don't, while some others even lose out upon morphing, when performance/Watt is considered. Such analysis does not take into account the effect of program phases [51, 87]. To demonstrate this point, we present a detailed study of the benchmarks *epic* and *fft* that show benefit from morphing. The objective is to investigate performance/Watt that each core type provides when considering fine grained time slices. We also want to study the effect that the varying instruction distribution of a benchmark may have on performance/Watt achieved on each core type in the AMP.

The benchmarks *epic* and *fft* were run for a few million instructions and the results depicting the performance/Watt of the workloads on each core type, as a function of time is shown in Figures 3.6 and 3.7, respectively. The performance/Watt for each core type (FP, INT and Morphed) is represented by the blue, orange and red curves, marked with a $\times$, a dot and a triangle, respectively. The distribution of instruction

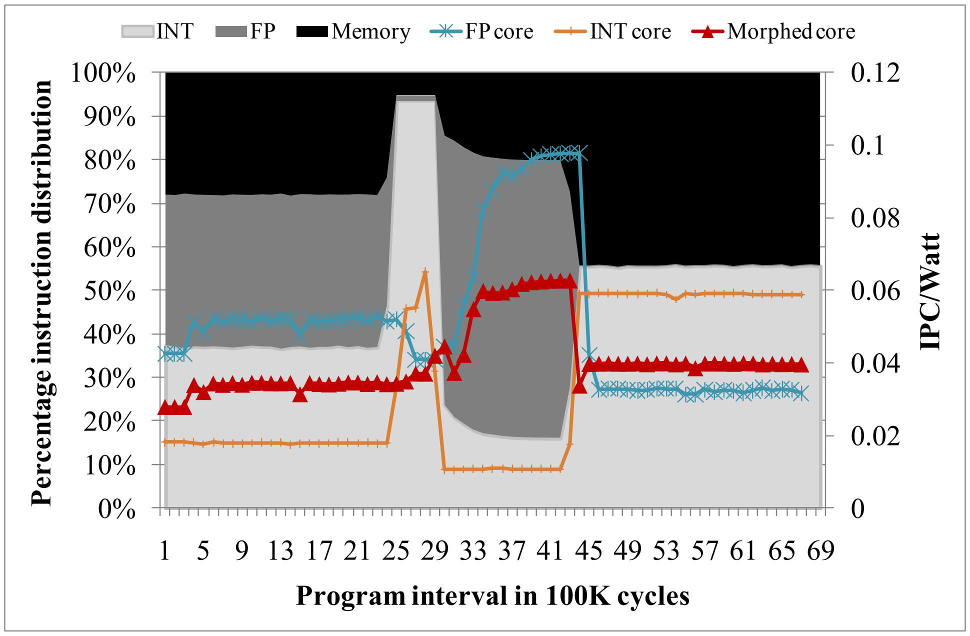


Figure 3.7. Zoomed view of variations in the performance/Watt of *fft* when run on each core configuration.

types at each time instant is represented by the area in the increasingly darker shades (light grey - int, dark grey - fp, black - memory). For the plot showing the behavior for *epic*, for the first 19 data points, the morphed core does not outperform either the FP or the INT core and as a result, staying in the baseline mode of execution is advisable. For the data points 20 to 37, the morphed core performs much better than the other cores (35% on average when compared to the nearest competitor, the FP core). Hence, there is a possibility of considerable performance/Watt gains to be made by morphing. During subsequent stages of execution, the baseline mode of execution again proves beneficial. This shows that by monitoring the program behavior at a more fine-grain level, there are more opportunities for improving the power efficiency by either morphing or exiting the morphed mode. At the same time, even though gains are made for *epic*, careful consideration must be given to the performance/Watt of the second thread running on the AMP which upon morphing gets assigned to the weak core, potentially resulting in a drop in performance/Watt for that thread. A similar in-depth study was carried out for the benchmark *FFT* (see Figure 3.7). It can be seen that even though *FFT* shows a small benefit from morphing over the entire run (see Figure 3.5), it can be seen from Figure 3.7 that thread swapping may provide even better benefits. The study of the above two benchmarks helps infer that the decision to swap or morph should be based on the current behavior (e.g., instruction mix) of the executing workloads. In the next section, we describe in detail our dynamic decision making scheme.

3.2.5 Dynamic Online Reconfiguration

So far it has been established that the expected performance/Watt on each core type is a function of the instruction distribution of the workload being executed. We now describe the mechanism for dynamic decision making. As mentioned earlier, we have explored the use of two types of dynamic decision making schemes. One of them is based on offline analysis while the other, online learning. The offline scheme is called the rule based dynamic core morphing (RDCM) scheme. The online version is called phase classification based dynamic core morphing (PCDCM) scheme. Both these schemes are now described.

3.2.5.1 The rule based dynamic core morphing (RDCM) mechanism

The RDCM scheme consists of two components: an online monitor and a performance predictor. The online monitor continuously and non-invasively profiles certain aspects of the execution characteristics of the committed instructions which is then used to make decisions online.

3.2.5.1.1 Performance prediction at fine grain time slices We have seen that there is certainly a relation between the performance/Watt and the instruction distrubution of the workload to be executed. To detect change in an application's behavior, hardware support is needed. In other words we monitor the instruction distribution and IPC of the workload being executed and accordingly make decisions.

We next describe the process that we have followed in order to make the morph/swap decisions based on the instruction composition and IPC.

For our experiments, twelve benchmarks from the suite of 38 were chosen such that they were diverse in nature. They were then run on each core type, and IPC/Watt as well as the instruction distributions were noted for fixed number of committed instructions, referred to as window. We then ran experiments where two threads were considered. At the end of every window, we analyzed the relation between the instruction types retired and the best thread to core assignment with respect to performance/Watt. For example, at the end of a window, while running a combination of *apsi* and *fft*, if it is noticed that the performance/Watt of running *apsi* on the morphed core and *fft* on the weak core is higher than the baseline mode, this point is marked as a potential switch point from baseline to morphed mode. Similarly, preferred switching points to come out of the morphed mode and to swap threads were identified. In this way, we found potential trigger points for morphing, swapping and reverting to baseline mode. Averaging the values of the percentage of fp instructions, percentage of int instructions and IPC that we have observed for the 132 combinations of two (out of the 12) threads, we set the rules for reconfiguration that are included in the algorithm in Figure 3.8.

It can be seen that in general the rules are intuitive. When a surge in floating-point instructions is observed, it makes sense to move to the FP core. The same holds for the decision to move to INT core. For switching to the morphed mode, there must be an increase in the integer and floating-point instructions. When switching into the morphed mode, we want to make sure that the performance of the thread that will execute on the weak core is not compromised. Hence, morphing takes place when the thread that will be run on the weak core has IPC less than 0.4 which was once again determined during the offline experiments. When the benefits of the morphed mode

```
Algorithm for dynamic reconfiguration:
1.  Threads $T_1$ and $T_2$ assigned randomly to cores
2.  Do Swap if:
     i.   ($\%INT_{FP}$ >= 44) and ($\%INT_{INT}$ <= 30)
                          OR
     ii. ($\%FP_{INT}$ >= 26) and ($\%FP_{FP}$ <=13)
3.  Go from baseline to morphed mode if:
     i.  For $T_1$ ($T_2$ )
          a.   %(FP + INT) >= 50 and
          b.   (17<=%FP<=30) and (26<=%INT<=44)
     ii. And $T_2$ ($T_1$ )
          a.   IPC <= 0.4 and
          b.   %(FP + INT) < 60
4.  Come out of morphed to baseline mode if:
     i.   Thread currently on morphed core shows
          a.  %(FP + INT) < 50
          b.  Use swap rules for thread to core assignment
5.  End
```

- $\%INT_{FP}$ – Integer instruction percentage of thread on FP core
- $\%INT_{INT}$ - Integer instruction percentage of thread on INT core
- $\%FP_{FP}$ – FP instruction percentage of thread on FP core
- $\%FP_{INT}$ – FP instruction percentage of thread on INT core

Figure 3.8. AMP reconfiguration conditions for RDCM scheme

are predicted to have dimished (as indicated by the inequality in the Figure 3.8), the AMP switches back to the baseline mode of operation.

3.2.5.1.2 Accounting for program phase changes We have defined the conditions that determine a switch in the mode of operation for the AMP. However, it is necessary to ensure that the decision is sticky. Otherwise the behavior of the dynamic mechanism may be oscillatory. To avoid this, a reconfiguration decision is made only if the same decision holds the majority for the past n windows, called history depth. The history depth (indicated by n) and the size of the individual window have to be determined experimentally. We have conducted a sensitivity study to determine these parameters.

3.2.5.1.3 Determining the best window size and history depth Choosing too small a window may result in noisy behavior, while too large a window may result in loss of potential opportunities. Hence, we experimented with various window sizes

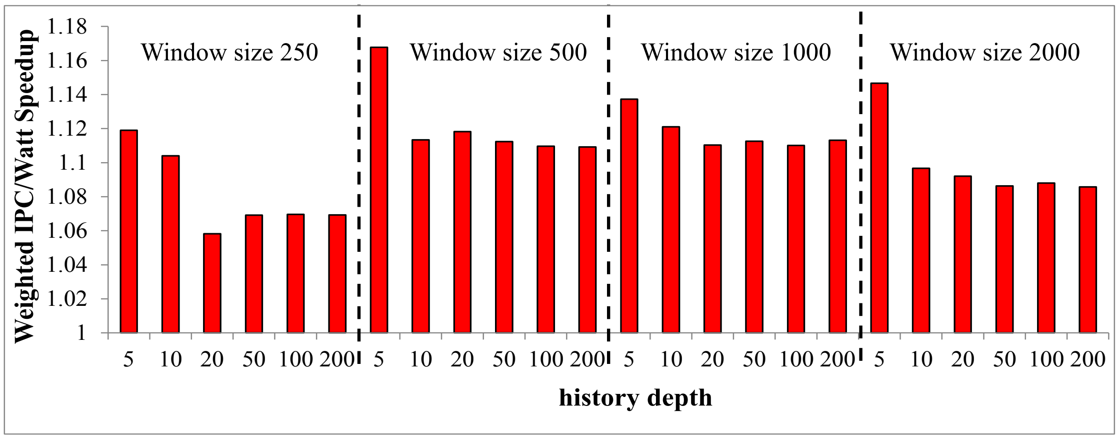


Figure 3.9. Performance sensitivity analysis for determining window size and history depth.

of 250, 500, 1000 and 2000 instructions. The history depth n was varied from 5, 10, 20, 50, 100 and 200 in our experiments. For example, if window size 500 is chosen with history depth 10, the scheme will rely on the behavior of the threads during the 5000 (500×10) recently committed instructions to make the reconfiguration decision. For each combination of window size and history depth, about 140 multiprogrammed workloads were run with a random combination of benchmarks from our set of 38. The weighted speedup in terms of performance/Watt obtained is shown in Figure 3.9. It can be seen that the best speedup (taking into account a certain overhead for reconfiguration) is obtained for a window size of 500 instructions and a history depth of 5. A reconfiguration overhead of 1000 cycles has been considered in these experiments.

3.2.5.2 The phase classification based dynamic core morphing (PCDCM) scheme

The offline decision making mechanism requires offline analysis and as such may not be ideal. This situation may be averted by using a online learning mechanism. We have used the phase classification and sampling technique for online learning.

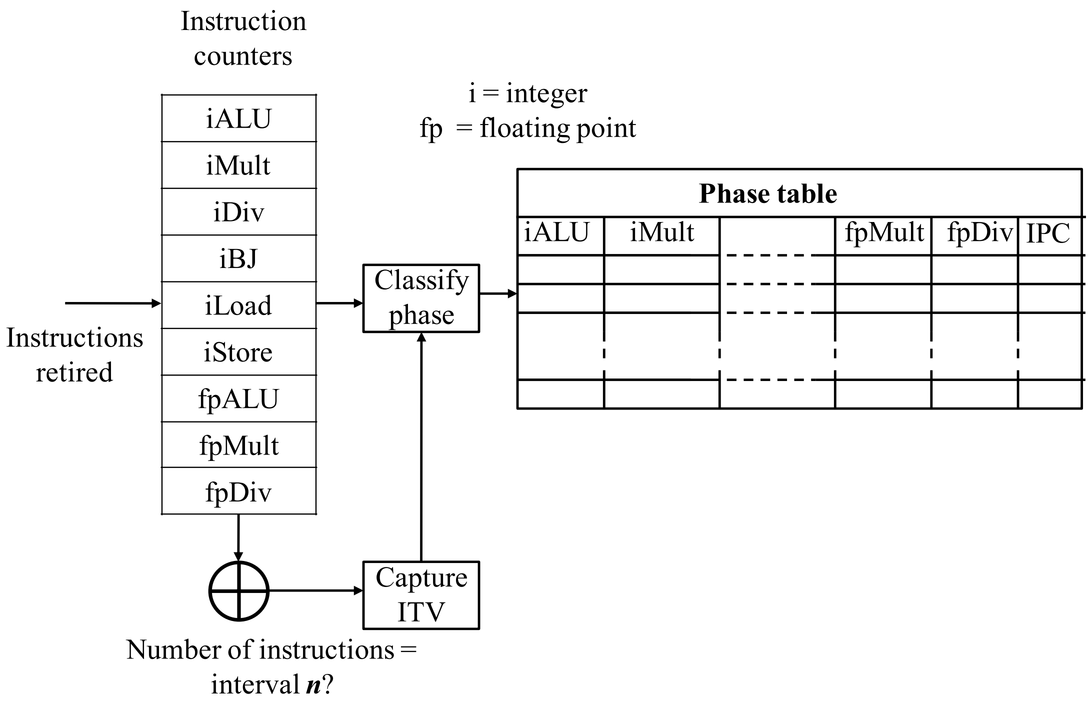


Figure 3.10. Online recording of application behavior via hardware counters and phase table as done by Khan et al. in [46].

3.2.5.2.1 ITV based phase classification Instruction type vectors (ITV) were introduced by Khan *et al.* in [46] for the purpose of program phase classification. We adopt this scheme and modify it to better suit our purposes. The ITVs are created using a circuit similar to that shown in Figure 3.10 where hardware counters are used to count the number of committed instructions of certain types (9 types as shown in the figure) during a specified interval. This interval corresponds to a fixed number n of committed instructions with the value of n to be determined. Whenever an instruction is retired, the appropriate instruction counter is incremented. After n instructions have committed the resulting 9-element vector is captured and compared to previously stored ITVs in the Phase Table. The already stored ITVs correspond to previously encountered stable phases where a phase is classified as stable when at least m consecutive intervals (of n committed instructions each) had almost identical ITVs. The number m is another parameter of the scheme that needs to be determined. The newly captured ITV is compared to each stored ITV by calculating the sum of the absolute differences between their corresponding nine elements. If this sum is smaller than a pre-specified threshold Δ (a parameter that needs to be determined), then the newly encountered phase is assigned the same phase ID as the one it was compared against. This signifies that we expect the current behavior of the program to be very similar to its behavior during the previously encountered phase with likely the same performance and performance/Watt. If however, the sum exceeds the threshold value Δ, it becomes a potentially new phase but it needs to repeat m times before being assigned a new stable phase ID. Every program may exhibit during its execution several short-lived intermittent phases that do not justify actions like thread swapping or core morphing. It is important therefore, to distinguish between stable and unstable phases. The resulting algorithm to detect and classify phases is shown in Figure 3.11. Experiments were carried out to determine the phase classification parameters, i.e., n, m and Δ. Details on these experiments can be found in in section 3.2.7. We found that in general, the phase classification mechanism provides best benefits when the interval range n is between 50K - 200K instructions, the %threshold is between 5 - 15% and the stable phase interval is between 2 - 8. For the rest of our experiments, these parameters have been set as: n = 150K, %threshold = 7.5 and the stable phase interval m=4.

3.2.5.2.2 Extending the phase table to include performance and power entries We made a few changes to Khan's [46] phase classification scheme as shown in Figure 3.12. There are two major differences. (i) The number of instruction types in the ITV vector has been reduced from 9 to 4, and (ii) there are additional entries in the table to indicate the estimated IPC and power for each core type in the AMP. Since the cores in the AMP mainly differ with respect to their capability of processing int and fp instructions, we aggregate all int instructions into a single entry and all fp instructions into another single entry. We also aggregate load and store instructions

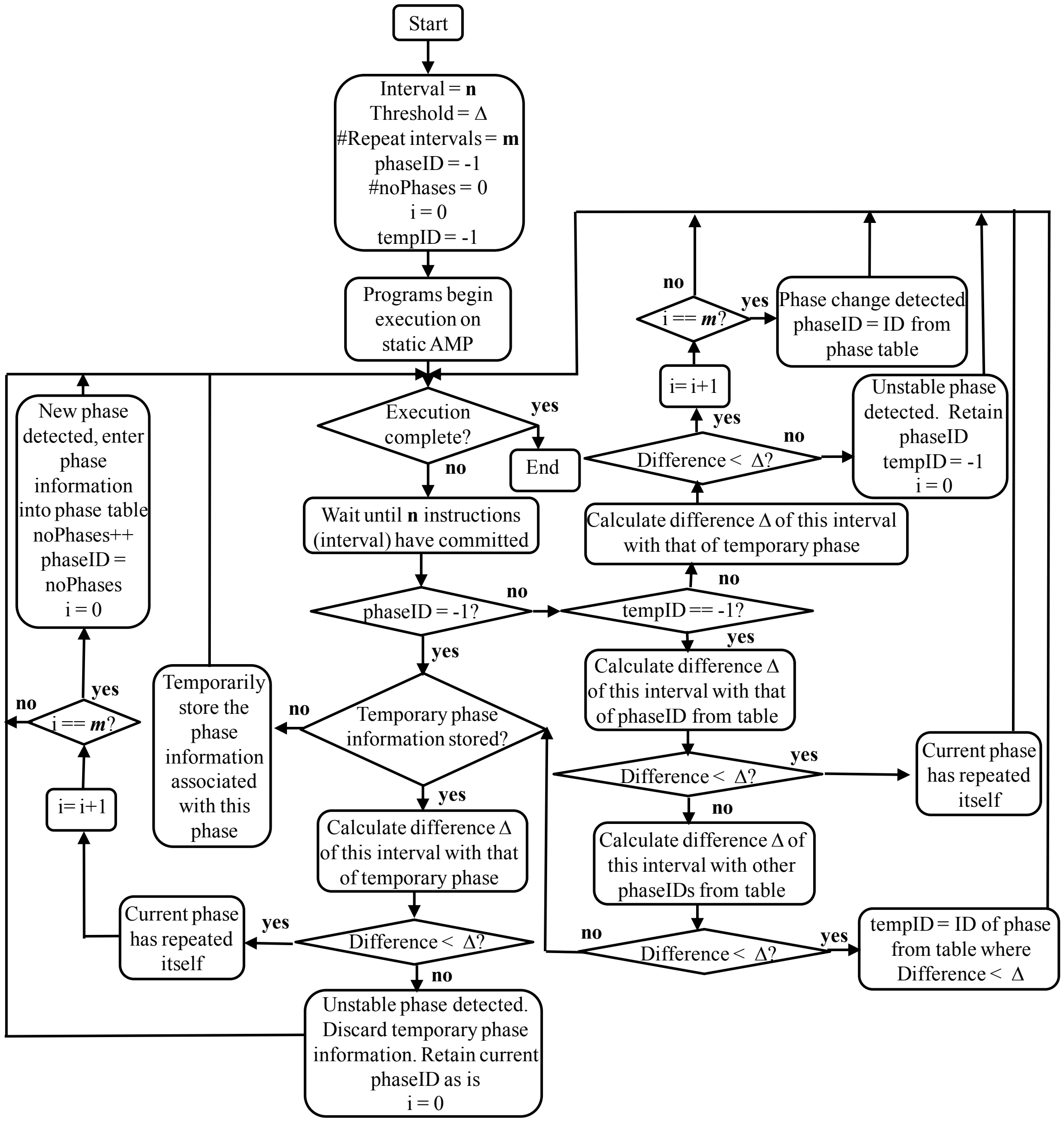

Start
Interval = n
Threshold = Δ
#Repeat intervals = m
phaseID = -1
#noPhases = 0
i = 0
tempID = -1
Programs begin execution on static AMP
Execution complete?
yes
End
no
Wait until n instructions (interval) have committed
phaseID = -1?
yes
no
Temporary phase information stored?
no
Temporarily store the phase information associated with this phase
yes
Calculate difference Δ of this interval with that of temporary phase
Difference < Δ?
yes
Current phase has repeated itself
no
Unstable phase detected. Discard temporary phase information. Retain current phaseID as is
i = 0
i= i+1
i == m?
yes
no
New phase detected, enter phase information into phase table
noPhases++
phaseID = noPhases
i = 0
tempID == -1?
no
yes
Calculate difference Δ of this interval with that of temporary phase
Difference < Δ?
yes
no
i= i+1
i == m?
no
yes
Phase change detected phaseID = ID from phase table
Unstable phase detected. Retain phaseID
tempID = -1
i = 0
Calculate difference Δ of this interval with that of phaseID from table
Difference < Δ?
yes
Current phase has repeated itself
no
Calculate difference Δ of this interval with other phaseIDs from table
Difference < Δ?
no
yes
tempID = ID of phase from table where Difference < Δ

Figure 3.11. Flowchart of Phase Classification algorithm.

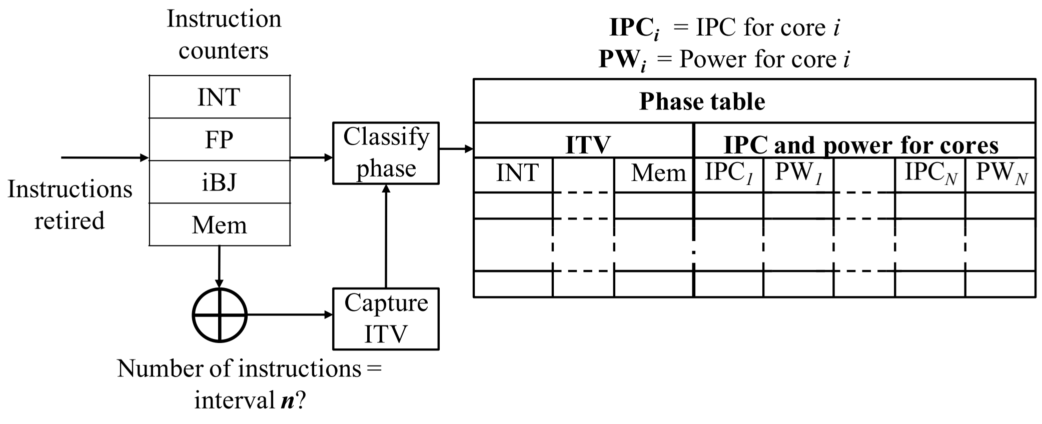


Figure 3.12. Extending the phase table with IPC and Power entries for each core in the AMP. Note that the number of instruction types in the ITV vector has been reduced from 9 to 4.

into a single entry called Mem. As is shown in section 3.2.8, such a reduction does not compromise the benefits of the online mechanism. Further, to be able to use effectively the information about already classified stable phases, there is a need to collect per core type in the AMP, the performance and power for a given phase. We do that by augmenting the phase table with 2 additional entries per core type, one each for IPC and power. Since there are 4 core types in the considered AMP (FP, INT, Morphed and Weak), there are 8 entries corresponding to the estimated IPC and power for the given phase on each core type. Whenever a new stable phase is identified, our scheme will store in the phase table the approximate values of the IPC and the power consumed by each core during that phase.

Online measurement of IPC is straightforward, but the same cannot be said about power measurement. To estimate power, we use performance event counters, available in almost every processor, as a proxy for power. Computer architects have for long used performance monitoring counters as a proxy for estimating the power consumption [43, 19, 92]. The accuracy of such estimates is not high, but still sufficient for comparing the power consumed by different cores executing the same program. We adopted a similar approach to estimate power online using performance counters.

If the approximate IPC and power consumption is available for each phase of an application on each processor, a simple table lookup suffices to determine the best thread to core mapping for future occurrences of the classified phase.

3.2.5.2.3 Online performance and power estimation The phase classification mechanism tracks the current behavior of the workload. Whenever a new phase is detected, or when a previously classified phase is encountered again, it indicates that there is a change in the composition of instructions being executed for the application. Hence, this is the point at which the performance/Watt of alternative thread to core schedules are evaluated. In order to then determine the best thread to core assignment, performance estimation of that phase on each core type in the AMP is needed. As mentioned earlier, we achieve this by dynamic online learning where the newly detected phase is run on each core in the AMP. A similar scheme has been used by Kumar *et al.* [51] and Becchi *et al.* [6]. However, Kumar *et al.* sample the program on each core type in the AMP, each time a new phase is detected even if it has been previously encountered. Becchi *et al.* force a thread swap between cores whenever a new phase is detected to estimate performance of the phase on each core type. Sampling is clearly needed when new phases are detected, but not when a previously encountered phase is detected again, if the information related to the phase is available. Hence, during the proposed online learning process, the program is executed once on each core type and the observed IPC and power information are stored in the phase table. Since the AMP has 4 possible core types, this process must be repeated 4 times.

The overheads of the online scheme stem from the online learning mechanism and context switch on thread swap. We quantify the details of this overhead and its effect on the benefits of our scheme in Section 3.2.6.

3.2.5.3 Putting it all together

We have so far described all the individual components of the RDCM and PCDCM. The working of the entire system is now described next as depicted in Figure 3.13. A software called the microvisor [46] is used to initialize and manage the phase classification mechanism as well as the performance and power estimation mechanisms for both the RDCM and PCDCM schemes. This software is invisible to the OS and is resident in between the OS and hardware. It collects information from cores and makes the best thread to core assignment. It functions the same way as that proposed by Khan *et al.* [46] or IBM's millicode [36].

3.2.5.3.0.1 The microvisor This is the software layer that collects information from the phase table on new phase or phase change detection in case of the PCDCM scheme and the performance counters in case of the RDCM scheme. With the information it has access to, it thus makes decisions. In case of PCDCM, if a new phase is detected, the microvisor controls the process of the sampling mechanism to estimate the IPC and it also collects the counter values that are used to estimate power. The phase table is then updated with the IPC and power information for each core type. If a phase change is detected, phase tables are looked up to fetch the IPC and power values for that phase which are then used to make thread scheduling decisions. In case of RDCM, the microvisor is aware of the rules that determine thread swapping or core morphing. After sampling the performance counters, it applies the data to the inequalities (as described in the Algorithm in Figure 3.8) to determine the best core configuration. Since the microvisor does some computation whenever phases are detected or repeated, it incurs an overhead which will soon be discussed in section 3.2.6.3.

3.2.5.3.0.2 Determining the best thread to core assignment Whenever a phase change is detected or a new phase is classified, a different thread to core assignment or core configuration may be needed to optimize performance/Watt of

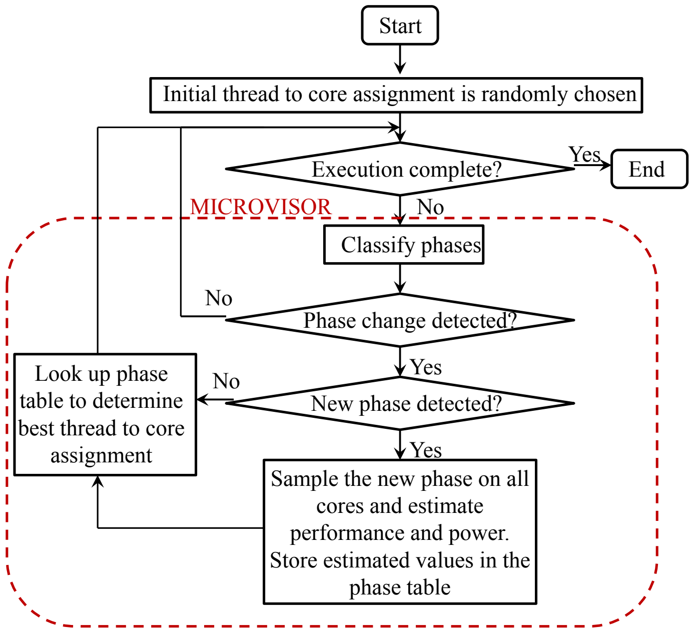

Start
Initial thread to core assignment is randomly chosen
Execution complete?
Yes
End
No
MICROVISOR
Classify phases
No
Phase change detected?
Yes
No
New phase detected?
Look up phase table to determine best thread to core assignment
Yes
Sample the new phase on all cores and estimate performance and power. Store estimated values in the phase table

Figure 3.13. Elements of the proposed PCDCM working together. The part of the algorithm controlled by the software layer (Microvisor) is indicated by the dotted red rectangle.

the applications being executed. This is determined by the microvisor. It collects relevant information from the cores of both the workloads being executed and makes a decision based on the performance/Watt of the various potential configurations. The various thread to core assignments for the proposed AMP are: (i) $thread_0$ running on the FP core and $thread_1$ on the INT core ([FP, INT]) or vice versa ([INT, FP]) (ii) $thread_0$ running on the morphed core and $thread_1$ on the weak core ([MR, WK]) or vice versa ([WK, MR]). Based on the current configuration of the AMP and the information available to the microvisor, the best configuration is chosen such that performance/Watt of the AMP is maximized.

3.2.6 Evaluation

We now present the results on the performance/Watt improvements provided by the proposed PCDCM scheme. In the evaluation, we found that the RDCM scheme performs better on an average than the baselines we considered, but it lost out to the PCDCM scheme. Hence, results are presented on the improvement in performance/Watt of the PCDCM scheme over the baselines and the RDCM scheme. In these experiments, two threaded multi-programmed workloads were run on the AMP. Execution stops when 5 billion instructions of either thread are retired. The phase classification parameters were set to: Interval n = 150K, %threshold $\Delta = 7.5$ and stable phase interval $m = 4$ based on our search described in section 3.2.7.

We now describe the baselines that will be used for comparison. The performance/Watt improvement achieved by PCDCM scheme over each baseline is then evaluated. Since the proposed scheme relies on dynamic online learning in order to determine the affinity of a newly detected phase to a core in the AMP, we present a study on the effect of this overhead on the benefits.

3.2.6.1 Baseline modes considered

We compare the proposed PCDCM scheme to the following baseline configurations:

1. Static: Here the AMP does not feature morphing or swapping of threads, but the thread to core assignment is based on oracular knowledge of the best assignment with respect to performance/Watt.

2. Swap: Here threads are allowed to swap between the cores. The decision to swap is made in the same way as the proposed dynamic online scheme using the ITV phase detection. The only difference is that the cores are not allowed to morph. Comparison with this baseline will allow us to measure the true benefits of the core morphing scheme.

3. HMG: This baseline consists of two homogeneous cores with parameters as described in Section 3.2.3. This dual-core processor is symmetric and occupies the same area as the FP and INT core AMP.

4. RDCM: This is the offline profiling-based DCM scheme. As described earlier, core reconfiguration in this scheme relies on rules derived offline by profiling a subset of the workloads considered. These rules are then applied to trigger either morphing or thread swapping whenever deemed beneficial.

3.2.6.2 Performance/Watt analysis over the baselines

We considered three speedup metrics for comparing the proposed scheme to the baselines. We define the following terms:

$$S_0 = (IPC/Watt_{thread0})_{dynamic}/(IPC/Watt_{thread0})_{baseline}$$
$$S_1 = (IPC/Watt_{thread1})_{dynamic}/(IPC/Watt_{thread1})_{baseline}$$

The various speedups considered are:

1. Weighted:
   $Speedup_{weighted} = (S_0 + S_1)/2$
2. Geometric:
   $Speedup_{geometric} = \sqrt[2]{S_0 \times S_1}$
3. Harmonic:
   $Speedup_{harmonic} = 2/(1/S_0 + 1/S_1)$

From the set of 38 workloads, we randomly selected 100 combinations of two threaded workloads and executed those using the PCDCM scheme and on each of the baselines. A subset (40 of the 100) of the results are plotted in Figures 3.14 and 3.15. These 40 were shortlisted by first sorting the results obtained for the 100 workload combinations in ascending order of weighted IPC/Watt improvement over the baseline and then choosing 10 worst cases, 10 best cases and 20 cases in between. It is clear that in general (see Figures 3.14 and 3.15), a significant IPC/Watt improvement

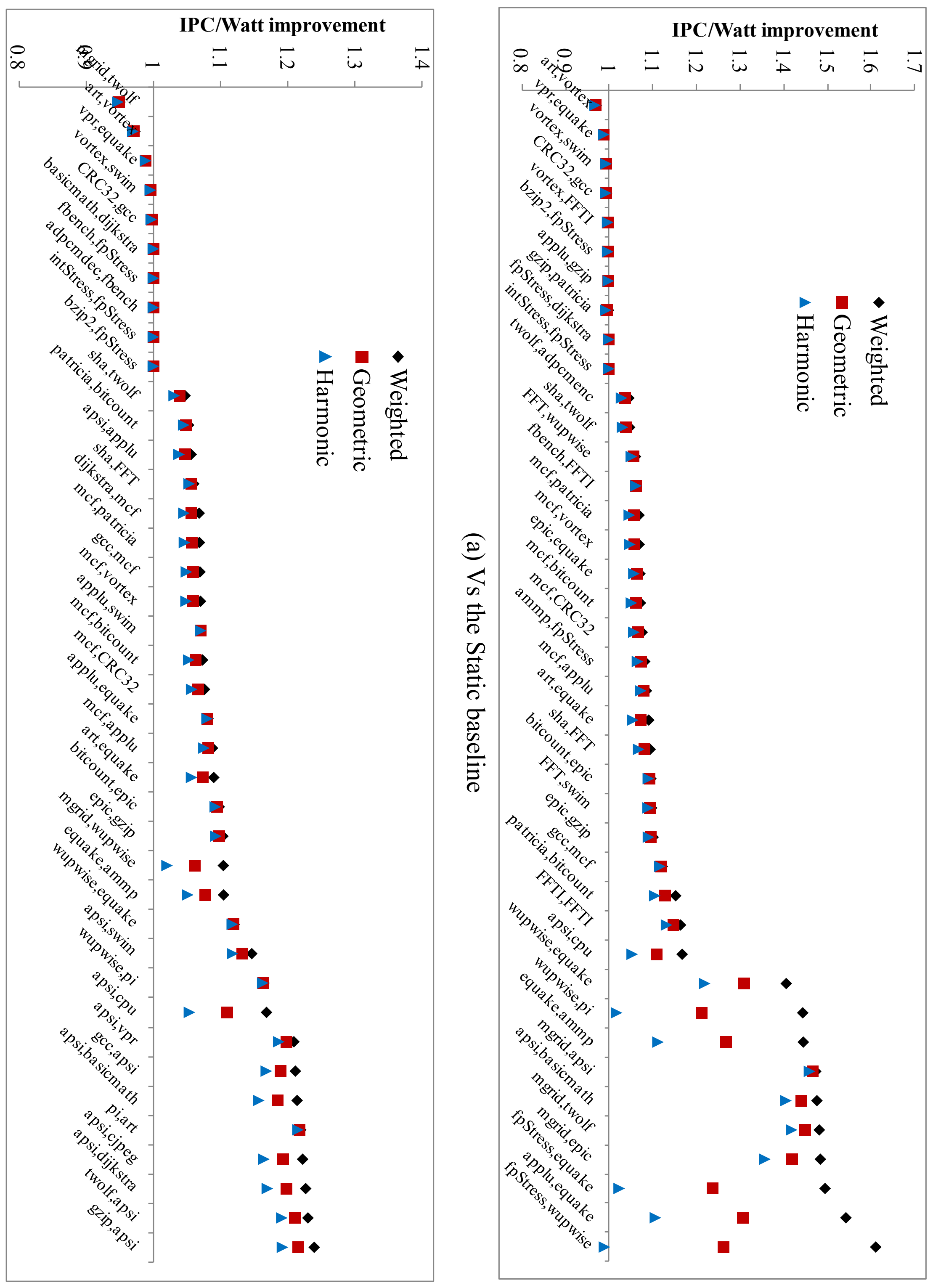
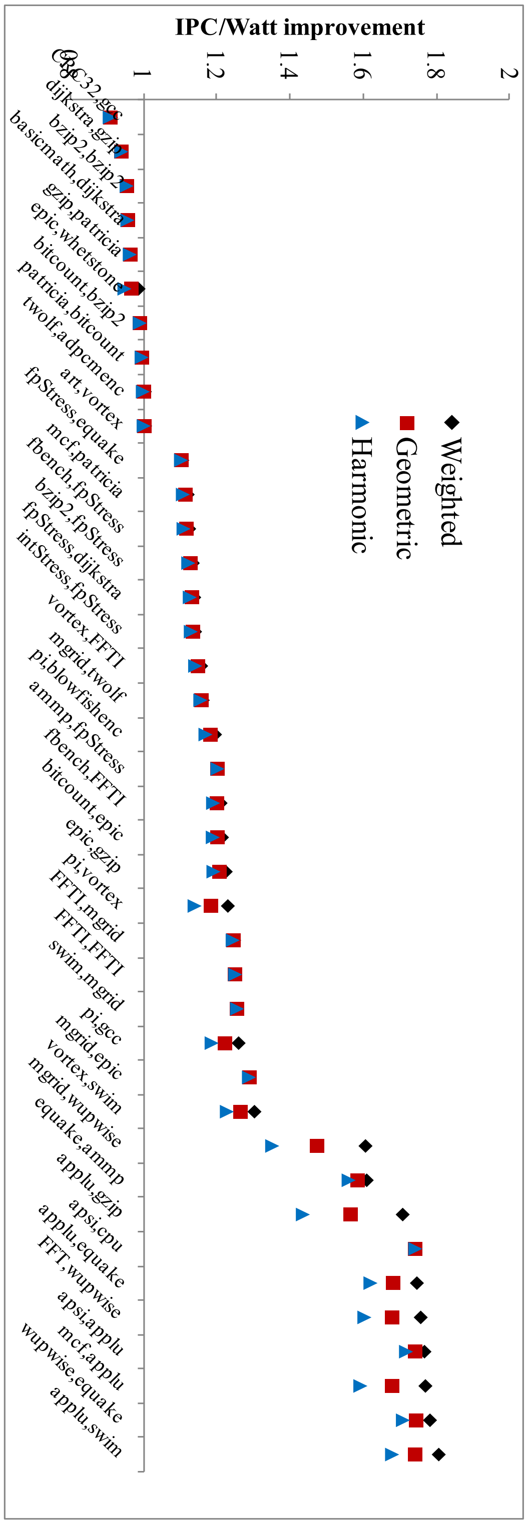
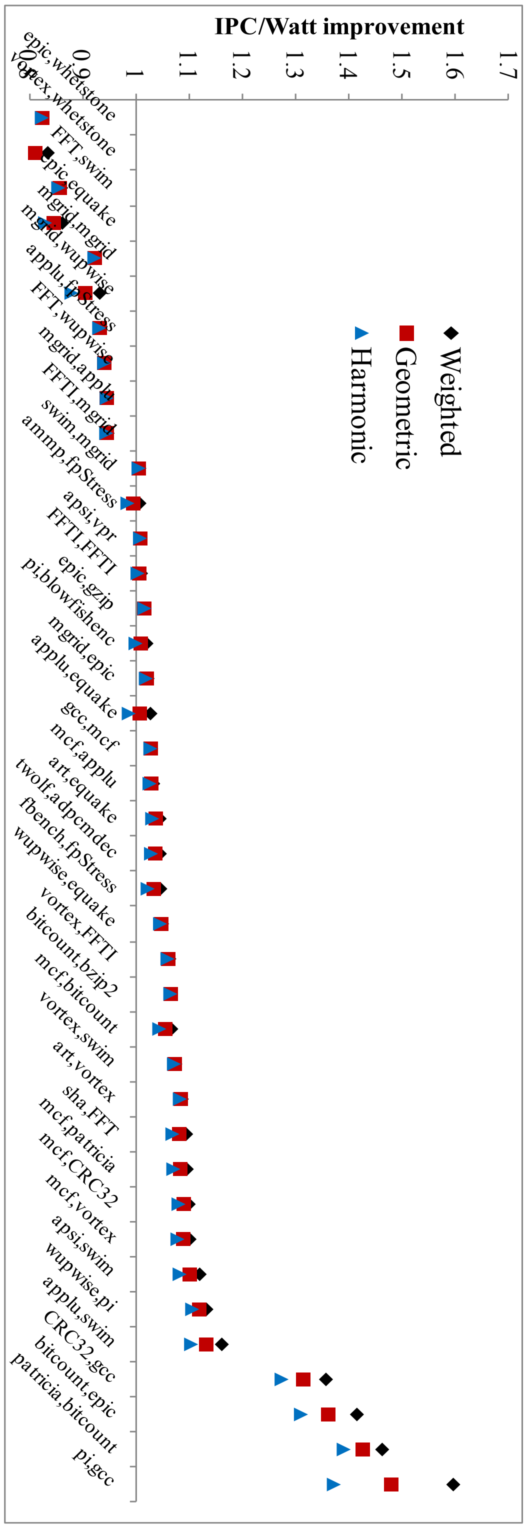


Figure 3.14. IPC/Watt improvement of the PCDCM scheme over the static and swap only baselines for a subset of the workload combinations.



(a) Vs the HMG baseline



(b) Vs the RDCM dynamic baseline

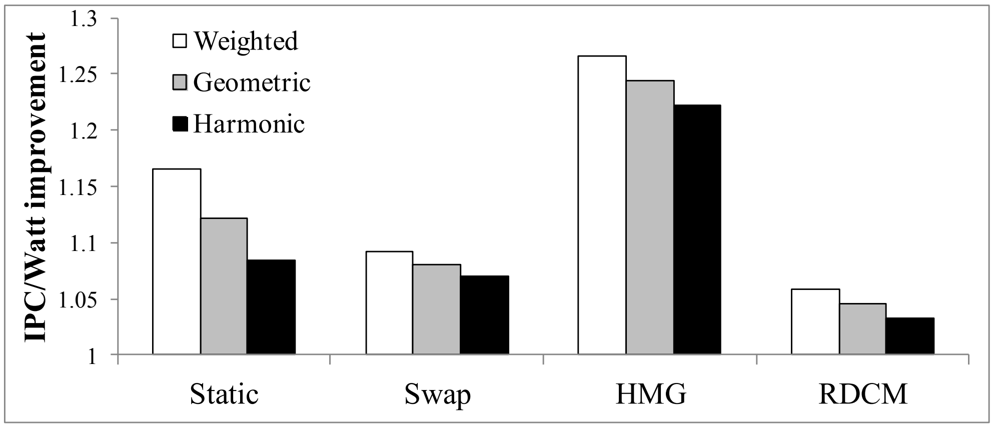
Figure 3.15. IPC/Watt improvement of the PCDCM scheme over the HMG and RDCM baselines for a subset of the workload combinations.

is observed when compared to any baseline. Also, amongst the worst cases for the baselines (static and HMG), it can be seen that the IPC/Watt degradation is not very high (0.86 in the worst case against HMG). Even when compared against the dynamic baselines (Swap and RDCM), it can be seen that significant IPC/Watt improvement is achieved on average.

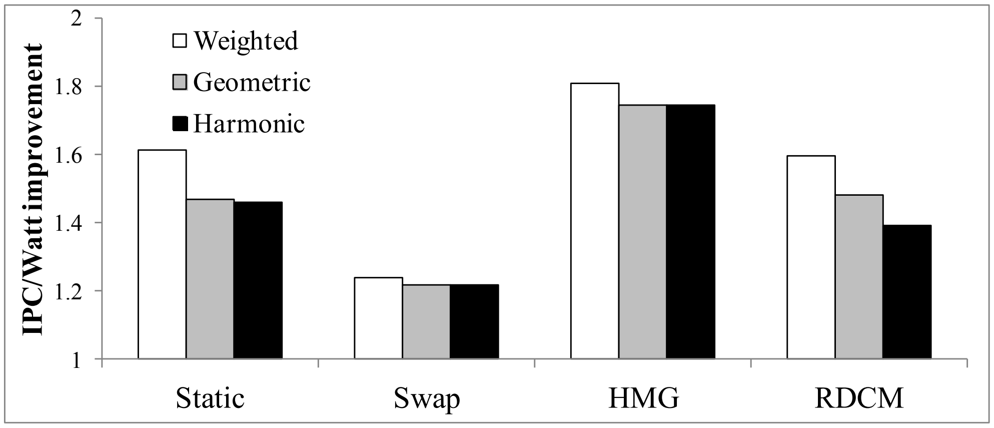
3.2.6.2.1 Analysis of results We now provide detailed analysis and reasons on why the PCDCM scheme performs better on average than both the static as well as the dynamic baselines.

Static: In this baseline, no dynamic thread to core assignment takes place during the run. However, the assignment is assumed to be done by an oracle and as such, cannot be done in practice. It can be seen that significant IPC/Watt improvement is achieved by PCDCM over this baseline. The static scheme cannot take advantage of phase changes or changes in resource demands. We have seen that appliation resource demands change over time and hence, the PCDCM scheme equipped with the phase classification mechanism is better equipped to deal with these changes. For example, over an entire run of 5 billion instructions, the workload *equake* shows an affinity to the FP core (see Figure 3.5). However, during the experimental run, 11 phases were detected for *equake* and affinity for the INT, Morphed or even the Weak core was observed during those phases. The PCDCM scheme detects these phases, re-evaluates the thread to core mapping and hence optimizes IPC/Watt. Hence, the PCDCM scheme achieves significant improvement in IPC/Watt over the static baseline. Still, there are a few workload combinations (3 out of of 100) where the PCDCM scheme performs slightly worse than this baseline (see Figure 3.14(a)). For these workloads, even though phases are detected and classified, at no point did PCDCM trigger a reconfiguration, but phases were detected and the sampling overhead increased the runtime. As a result, the IPC/Watt improvement is less than 1. However, on an average, for all the 100 combinations (see Figure 3.16 where average, maximum and minimum improvements over all baselines are plotted), a significant improvement of 16% is observed with respect to weighted IPC/Watt, which more than justifies the rare cases where no reconfigurations take place.

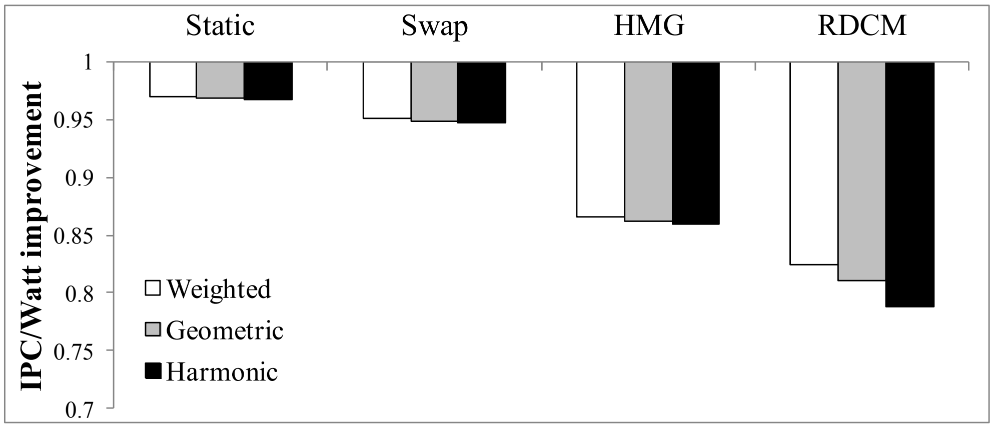
Swap: This is one of the two baselines that are dynamic. Here phase classification is used, but only thread swapping between cores is allowed. Although this scheme is dynamic, it can be seen that the IPC/Watt improvements are significant on average (see Figure 3.16(a)). Also, there are only four cases where IPC/Watt improvement is < 1 (the leftmost workload combinations in Figure 3.14(b)). By allowing cores to morph, the execution of the thread on the Morphed core is accelerated, while that on the Weak core is slowed down. As a result, the phase combinations that are encountered between the two workloads, when the cores have morphing capability and when they do not, are very different. This results in sometimes, different reconfiguration decisions made by the PCDCM and swap-only schemes. For example, when running the workload combination *mgrid,twolf* (leftmost combination in Figure 3.14(b)) where a speedup of 0.97 was observed, the PCDCM scheme performed morphing 10 times, while the swap scheme made no reconfiguration. Since the proposed scheme is greedy in its decision making, thread re-scheduling decisions are made even for the short lived phases. Hence sometimes, the overheads outweigh the benefits which is what led to the PCDCM scheme performing slightly worse. This however, is not a frequent occurrence and it happens only in 4 out of the 100 combinations of workloads. There were also several cases where this baseline performed as well as the PCDCM scheme (14 of the 100 combinations). However for the rest of the workloads the PCDCM scheme significantly outperforms this baseline. There are several instances where a core that is strong on the integer and floating-point fronts was required. This is where the advantage lies for the PCDCM scheme. Further, sometimes there are workloads that are naturally affine to the weak core and hence this increases the benefits of



(a) Average IPC/Watt improvement



(b) Maximum IPC/Watt improvement



(c) Minimum IPC/Watt improvement

Figure 3.16. Average, maximum and minimum IPC/Watt improvement of the PCDCM scheme over the various baselines.

the PCDCM scheme even further. The average IPC/Watt improvement over 100 combinations of workloads was found to be 9% (see Figure 3.16(a)).

It may be noted that the phase classification based "swap" scheme achieves a weighted IPC/Watt improvement of about 8% over the static baseline. Hence, such a dynamic swap scheme may be beneficial for architectures that do not or cannot include the hardware support for morphing.

HMG: This baseline is an area-equivalent symmetric multicore. It can be seen that in general, IPC/Watt is significantly improved by the PCDCM scheme, when compared to this baseline. A thing worth noting is that the number of cases where PCDCM performs worse is on the higher side (9 out of the 100 combinations) when compared to the other baselines. Moreover, the worst case weighted IPC/Watt improvement of 0.86 is one of the worst when compared to all other baselines. This happens because the HMG baseline is well suited to running certain homogeneous workload combinations that exhibit phases, but no difference in execution characteristics on the various core types in the AMP. For example, the left most workload combinations in Figure 3.15(a), i.e. *CRC32, gcc*, *dijkstra, gzip* and *bzip2, bzip2* are all int intensive. In such cases, having a homogeneous multicore may be a better option as both the threads are affine to the same core type in the AMP, the thread assigned to the other core type will suffer with respect to performance. This is evident from Figure 3.15(a). If however, one of the workloads being executed has FP instructions, PCDCM may perform better even if those workloads are similar. As an example, consider the symmetric workload combination *FFTI, FFTI* in Figure 3.15(a) which shows a weighted IPC/Watt improvement of 25% when run on the PCDCM scheme. This happens as *FFTI* shows phases which are FP/INT intensive or have both. Phases that have a reasonable proportion of INT and FP instructions are naturally affine to the Morphed core. PCDCM detects those and makes intelligent thread mapping decisions to improve IPC/Watt. On an average for the 100 combinations, PCDCM scheme achieves

26% IPC/Watt improvement over the HMG baseline (see Figure 3.16(a)).

RDCM: This is the dynamic baseline that uses information learned offline to determine thread scheduling and morphing decisions online. The PCDCM scheme achieves an improvement in IPC/Watt even over this scheme. The major reason for this is that for the RDCM scheme, the rules are dependent on the workloads used for profiling. Hence, when a workload that was not profiled earlier is encountered, the RDCM scheme may not make the most optimal decisions. The PCDCM scheme on the other hand relies on online learning and hence is independent of the incoming workload. It however incurs the phase sampling overhead. Further, the RDCM scheme makes decisions at fine grain instruction granularities while the PCDCM scheme makes those at more coarser gain granularities. Hence, there are workload combinations where the IPC/Watt improvement of PCDCM over RDCM is < 1. Still, on an average, PCDCM achieves 6% IPC/Watt improvement over RDCM as seen in Figure 3.16(a). One of the major benefits of the proposed PCDCM scheme is that it will always make intelligent scheduling decisions irrespective of the incoming workloads, unlike the RDCM scheme, the benefits of which depends on the training set used. Further, the RDCM scheme's rules are only valid for the architecture considered in this chapter. For different architectures, a different set of rules may have to be determined. This is not the case for the proposed scheme which will work just fine for any core types. The PCDCM scheme is therefore, more scalable than the RDCM scheme.

3.2.6.3 Overheads vs. benefits

The PCDCM uses phase classification and phase tables controlled by the microvisor to provide performance/Watt benefits. However, this benefit comes at the cost of an overhead. In this subsection, we quantify the software and hardware overheads of the scheme and also present the effect of the overheads on its benefits.

3.2.6.3.1 Software overheads The software overheads in the proposed scheme arise due to microvisor function, sampling to determine IPC and power information and the times when cores need to swap thread contexts.

The microvisor is invoked whenever a new phase is detected or when a previously detected phase is detected again. Table lookup is then performed and the information is used to determine the weighted speedup metric which is then used to determine the best thread to core mapping based on the newly detected phase. We estimate the overhead of this procedure to be a few hundred cycles every time it happens. We set this number conservatively, as 500 cycles for our experiments. It was observed in our experiments (consisting of 100 combinations) that there were around 5 phases detected on an average and, the maximum number of phases detected was 17. Also, phase changes were detected around 800 times on average and the maximum number of phase changes detected was 2020. Hence, the overhead due to microvisor invocation was found to be (5 + 800) $\times$ 500 cycles which equals 402K cycles on average and, (17 + 2020) $\times$ 500 cycles which equals around 1M cycles overhead for the worst case.

The second source of overhead comes about during the process of sampling. When a new phase is detected, it is sampled on each core type for the defined interval length n (150K instructions). Hence, a total of 600K + 4 $\times$ 150K instructions is executed in total during the sampling phase. On an average, we estimated that it would take around 1.5M cycles to execute this. During this dynamic online learning process, the system continues with one of the core types and hence only 75% of the 1.5M cycles is the actual overhead of sampling. Thus, a significant portion of the overhead is due to online learning. In our experiments, as described earlier, average/maximum phases detected were 5/17 and the corresponding average/maximum overhead due to sampling were 7.5/25.5 million cycles. Considering that each experiment runs for billions of cycles, this overhead in cycles comes to be around 0.2% on an average and, 0.8% in the worst case. It is worth noting that since we use the phase table, we avoid

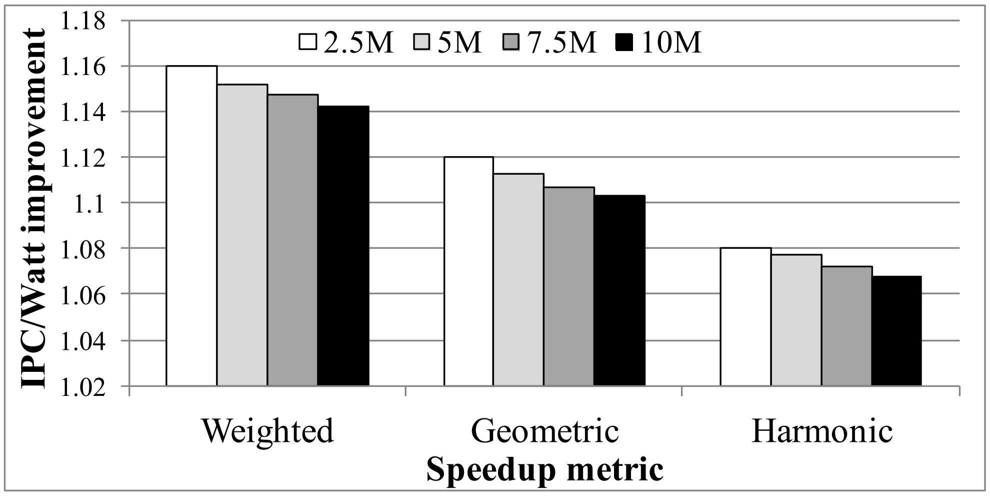


Figure 3.17. Weighted, geometric and harmonic IPC/Watt improvement over the static baseline for increasing overhead for dynamic online learning.

the sampling process whenever the same phase is detected again. If that were not the case, sampling would have to be done 2020 times in the worst case, for every phase change and this would have significantly increased the overheads.

The third source of overhead stems from the context switch whenever the microvisor determines that the cores must swap their contexts or morphing of resources must take place to maximize performance/Watt. It was observed in our experiments that the thread swaps and hardware reconfigurations happened around 90 times on average and around 1000 times in the worst case. This overhead can vary from one architecture to the other depending on dedicated support for thread swapping and context switch. Architectures with support for thread swapping may incur up to a thousand cycles overhead while it may be significantly larger for those without such support. We have assumed this overhead to be 1K cycles and hence the overhead due to thread swapping/morphing may be estimated to be 90K and 1M cycles on an average and in the worst case, respectively. Both of these are negligible considering that we execute the benchmarks for billions of cycles. We experimented with various context switch overheads of 10K, 50K, 100K and 1M cycles and found those to have negligible effect on the benefits of the proposed scheme.

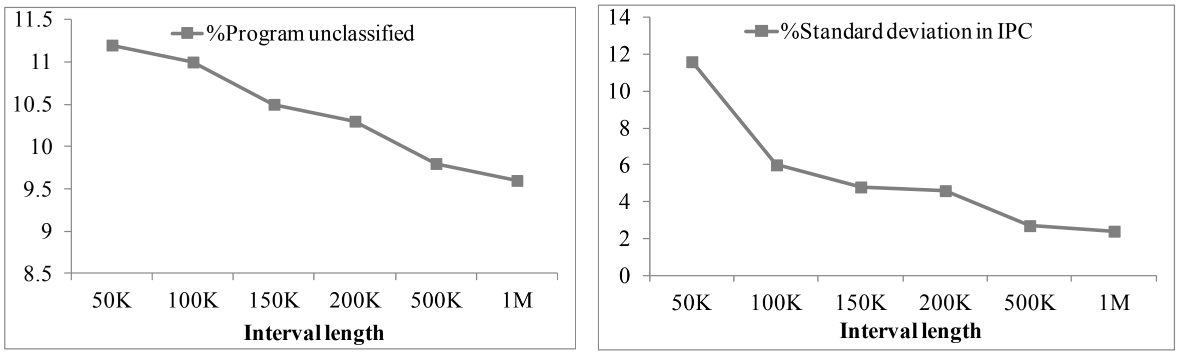
Of the three sources of overhead, the overhead due to sampling dominates the others. To quantify the effect of these overheads on IPC/Watt improvement of our approach, we increased the sampling overhead from 2.5 to 10 million cycles. The result is plotted in Figure 3.17. It can be seen that even with a pessimistic overhead of 10 million cycles per sampling process, the scheme still achieves benefits of 14% weighted IPC/Watt improvement over the static baseline (a drop of 2%), with respect to all three speedup metrics. Hence, we can conclude that the proposed PCDCM scheme has a low sensitivity to the sampling overhead.

For the RDCM scheme, the only overheads are those that arise due to microvisor invocation and context switch, but no overhead due to sampling. Hence, the overheads in the RDCM scheme are negligible.

3.2.6.3.2 Hardware overhead As mentioned earlier, in our experiments we noticed the average number of phases detected to be about 5. For the worst case scenario, this number went up to 17. Therefore, about 20 phase table entries may be sufficient for most cases. Each entry in the phase table captures the ITV, the performance and power information of the phase on each core types. Hence, an entry in the table consists of 12 fields, totaling to about 240 fields (12 fields/entry $\times$ 20 entries) for the entire phase table. Even if each field requires 32 bits, the size of the phase table would be less than 1 KB. Clearly, this is a small overhead considering the total gate count of the dual core processor.

3.2.7 Determination of phase classification parameters

Khan et al. [46] have conducted experiments to determine the parameters of the phase classification mechanism, namely: (i) interval length (n), (ii) phase detection threshold (Δ), and (iii) stable phase interval (m). We repeated these experiments in order to (a) simplify the phase detection hardware and (b) improve the prediction accuracy against a much larger and diverse set of benchmarks. Based on these exper-



(a) %Program unclassified Vs Interval length (b) %Standard deviation in IPC Vs Interval length

Figure 3.18. Sensitivity of the phase classification quality metrics to increasing interval length (n). Note that the results for combinations of phase classification parameters with the same interval length have been averaged.

iments, we reduced the ITV vector length from 9 to just 4, which cuts down the size of phase detection hardware by nearly half. As will be shown in section 3.2.8, such a reduction has little effect on the benefits of the phase chassification mechanism.

In order to determine the parameters for the phase classification mechanism, a number of interval lengths n were experimented with, between 1K to 1M instructions.

In order to measure the quality of the phase classification mechanism, we use the following two quality metrics: (i) percentage of the program that can be classified into stable phases and (ii) standard deviation of the IPC between intervals classified under the same phase ID. Reconfiguration decisions can only be made if the thread under consideration is in a stable phase of execution. The benefits (in terms of the resulting performance/Watt) depend on the standard deviation of IPC between phases classified under the same phase. A high value of the standard deviation may indicate that there is a large disparity between the projected IPC/Watt improvement (due to the reconfiguration) and the real improvement. It is therefore, desirable that most of the program is classified as stable, and that the standard deviation in IPC between phases classified under the same ID is as low as possible.

We found that smaller intervals result in a high proportion of unstable phases and higher standard deviation in IPC between intervals classified under the same phase.

Increasing the interval size results in a reduction of unstable phases and standard deviation in IPC between phases. This happens due to the averaging effect that takes place with an increasing interval size. However, a too large interval may result in the entire program being classified as a single phase eliminating all the potential benefits of our approach.

The ultimate purpose of phase classification is maximizing the IPC/Watt and we need to find the values of the parametrs (n, Δ and m) that will results in the highest IPC/Watt. To reduce the size of the parameter space that needs to be explored we only considered those combinations of phase classification parameters that yielded % unstable phases and % standard deviation in IPC to be below 12%. This resulted in a reduction of more than 65% in the search space size. The remaining combinations of the phase classification parameters were: interval n varying from 50K to 1M instructions, threshold Δ between 7.5 to 12% and the stable phase interval m varying from 2 to 8. The general trends observed in % program classified as unstable and % standard deviation in IPC for increasing interval size are shown in Figure 3.18. Note that we have averaged results obtained for combinations of phase classification parameters with the same interval size, in order to show the results in a single plot.

For each shortlisted combination of the phase classification parameters, we ran 100 random combinations of two threaded workloads from the set of 38 and calculated the weighted IPC/Watt improvement over the static baseline with oracular thread to core assignment. Here too, to show the results in a single plot, we averaged the results observed for the same interval size. Figure 3.19 shows that the IPC/Watt improvement is highest for the interval size of 150K. From the considered combinations of the other phase classification parameters we found the largest speedup when using a %threshold (Δ) of 7.5% and a stable phase interval (m) of 4.

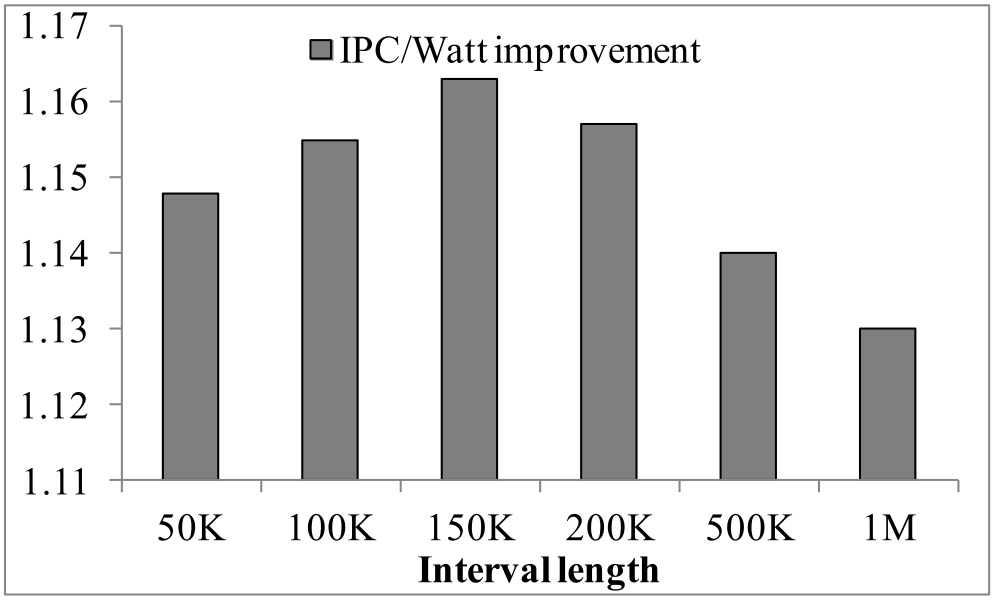


Figure 3.19. IPC/Watt improvement for various interval sizes. Note that the results for combinations of phase classification parameters with the same interval length have been averaged.

3.2.8 ITV vector length vs. performance/Watt benefits

As mentioned earlier, the proposed scheme may not need the details of all the nine types of instructions. To illustrate the effect on the quality of phase classification when reducing the number of ITV entries from 9 to 4, we measured both the quality defining factors for both and the results are plotted in Figure 3.20. In this experiment, the interval length n was kept at 150K instructions, the %threshold Δ was kept at 7.5% and the stable phase interval m was kept at 4.

It can be seen that there is only a small quality degradation with respect to standard deviation of the IPC which is expected. The reduction in ITV length made a difference of less than 1% in the achieved IPC/watt benefits. We used therefore, a 4 entry ITV to save hardware.

3.2.9 Conclusions

In this work we have presented a couple of online mechanisms to determine thread to core assignment online to improve performance/Watt of an asymmetric multipro-

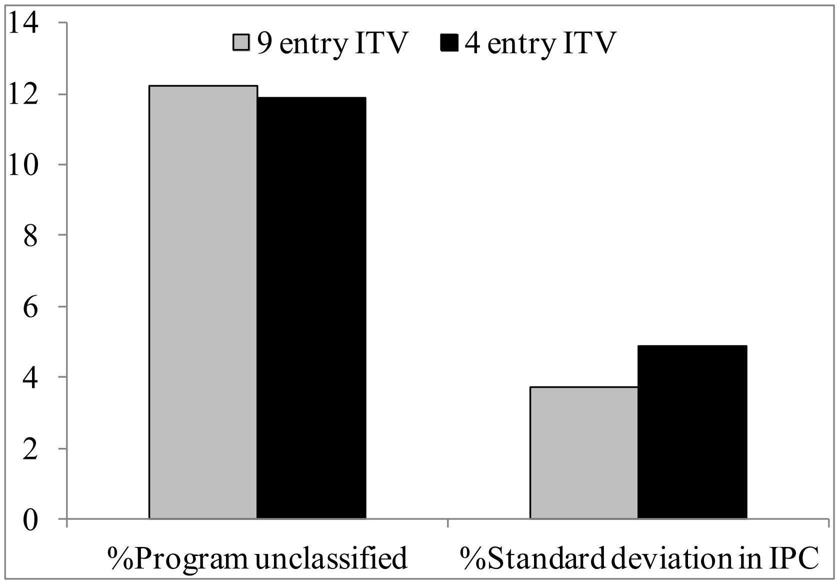


Figure 3.20. %Program unclassified and % standard deviation in IPC when using a 9 entry and 4 entry ITV. It can be seen that quality only degrades a little with respect to standard deviation in IPC.

cessor system. The studied AMP architecture features two cores: one with strength in floating-point computation and the other in integer intensive workloads. By morphing the two cores, we obtained a core that is strong in both integer and floating-point computations, but this resuls in the second core becoming much weaker. We deployed adaptive core morphing alongside thread swapping, at runtime, to reassign threads to cores using the above program phase classification. Two dynamic decision making mechanisms were considered. One of them, called the RDCM scheme relies on information gleaned offline while the other called the PCDCM scheme obtains such information online. Both schemes basically rely on the use of performance monitors to make decisions online. To evaluate the schemes, static and dynamic reconfiguration alternatives were considered. Using the PCDCM scheme, substantial performance/Watt gains are achieved. Our results show that the PCDCM scheme, on an average, outperforms the static heterogeneous baseline by about 16%, the homogeneous baseline by 26% and the best dynamic baseline i.e. RDCM scheme by 6%, with respect to weighted IPC/Watt. It is worth noting that the PCDCM scheme is based on online learning with no prior knowledge regarding the individual capabilities of the individual cores, and hence is not limited to the considered INT, FP dual-core but is applicable to any heterogeneous AMP, unlike the RDCM scheme.

3.3 Scalable Thread Scheduling in Asymmetric Multicores for Power Efficiency

So far, we have seen that rule based and online sampling based schemes can significantly improve the performance-per-power of AMPs. However, the scheme developed was specific to the INT FP AMP considered. In this section, we explore an estimation based scheduling scheme that may be applied to generic AMPs. The key idea is the online estimation of both the performance and power of an application on all the other cores in the AMP, while it is being executed on the current core. This is made possible by using the performance counters of the current core. A relationship is established between the values of these counters in the core executing the application and the expected performance and power of this application if it would run on the other cores in the AMP. By estimating the performance and power on other core types, informed thread scheduling decisions can be made without any of the drawbacks of offline profiling and online learning. To illustrate our approach, we consider an 8-core AMP comprising of two high performance cores (HPerf core) with similar characteristics to an Intel Nehalem or AMD K10 processor, and six low power cores (LP core) similar to an Intel Atom or AMD Bobcat. This choice is in line with recent studies [6, 50, 84]. We present an extensive analysis to determine which hardware performance counters (HPCs) should be used to predict both performance and power. We then formulate expressions using the selected counters for estimating the performance and power on other cores in the AMP. These expressions are used to make real-time thread scheduling decisions in the AMP when eight threads are run. The proposed scheme is compared against the static baseline AMP (the same dual core type AMP with no thread swapping capability) with oracular knowledge of the best thread to core mapping and a previously proposed online learning scheme [6]. We also compare the proposed scheme to a greedy oracle scheduler. Our results indicate that the proposed scheme achieves significant performance/Watt improvements over all the baselines. In particular, on an average, 2X gains are observed when comparing the proposed scheme to that based on online learning.

3.3.1 Methodology

To evaluate our approach (detailed in the next two sections), we selected an 8-core AMP consisting of two core types at the two ends of the performance/power spectrum - a high-performance core (HPerf) and a low-power core (LP). This is one of the worst cases for a scheme for estimating the performance and power of the second core based on the activities observed in the first core. In the considered 8-core AMP, two cores are HPerf cores and 6 are LP cores. The list of core parameters and execution latencies used for both the core types are shown in Tables 3.4 and 3.5, respectively. Most of the core parameters and latencies were taken from [26]. We used SESC as our architectural performance simulator [75] and employed CACTI [89] and Wattch [13] to calculate power with modifications to account for static power. We are aware that Wattch has an error percentage of within 10% when compared to layout-level power estimation tools. Our focus is on estimating instantaneous power and we are mainly interested in detecting changes in the power profile (which may trigger dynamic thread re-scheduling). Hence, comparison of the estimated power (by using different counters) to the power calculated by Wattch is satisfactory. For our experiments, we have selected 38 benchmarks: 16 benchmarks from the SPEC suite [97], 14 from the embedded benchmarks in the MiBench suite [33], one benchmark from the Mediabench suite [57], and 7 additional synthetic benchmarks. These 38 benchmarks encompass most typical workloads, for example, scientific applications,

Table 3.4. Chosen core parameters

Param	LP	HPerf	Param	LP	HPerf
Issue	2	6	INTREG	64	96
FPREG	64	80	INTISQ	NA	36
FPISQ	NA	24	LS units	1	3
LSQ	NA	32	ROB	NA	128
L1(I/D)	32K	32K	L2	512K	2M
Freq (GHz)	2.4	2.4	Type	In-order	OOO

Table 3.5. Execution unit specifications for the cores. (P - Pipelined, NP - Not pipelined, PP - Partially pipelined)

Core	FP DIV	FP MUL	FP ALU
LP	1 unit, 60 cyc, NP	1 unit, 4 cyc, PP	1 unit, 5 cyc, P
HPerf	1 unit, 21 cyc, P	1 unit, 5 cyc, P	2 units, 3 cyc, P
	INT DIV	INT MUL	INT ALU
LP	1 unit, 207 cyc, NP	1 unit, 10 cyc, P	2 unit, 1 cyc, P
HPerf	1 unit, 23 cyc, P	1 unit, 8 cyc, P	8 units, 1 cyc, P

media encoding and decoding and security applications. The instruction distribution of each of the considered workload is plotted in Figure 3.21.

3.3.2 Performance/Watt analysis of the two core types

The two core types that comprise our AMP have very different characteristics with one designed for high performance, while the other for low power. To quantify the difference in the capabilities of the cores, we ran all the 38 benchmarks on both the core types (LP and HPerf cores) for 1 billion instructions, after skipping the initial 5 billion that include the program initialization. The performance/Watt results are shown in Figure 3.21. It can be seen that for some workloads, the HPerf core performs better than the LP core (*ammp, CRC32, pi*) while it is vice-versa for certain other workloads (*equake, bitcount, sha*). The performance per watt is a function of the resource utilization. Efficient resource utilization leads to better figures. In general, for benchmarks which are branch or memory intensive, HPerf core resource utilization is not optimal and hence the performance per watt is lower than that of the LP core.

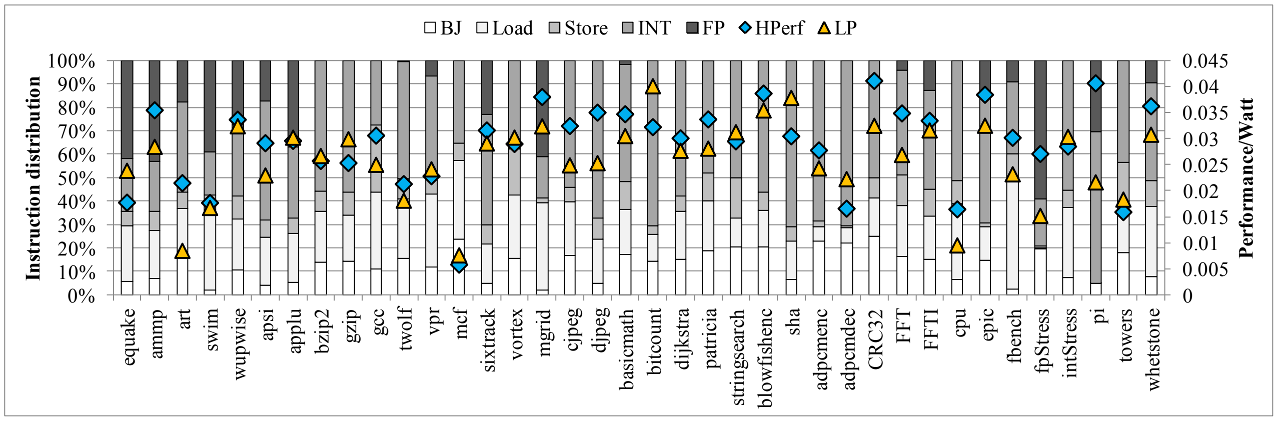


Figure 3.21. Instruction distribution and IPC/Watt for the 38 benchmarks considered when run on each core type for 1 billion instructions.

Clearly, for eight threaded workloads, a correct thread to core scheduling will yield significant benefits, while an incorrect one, will have a much lower performance/Watt.

Figure 3.21 depicts the average behavior over 1 billion instructions and as such only indicates the achieved IPC/Watt due to a fixed thread to core assignment. Many programs exhibit phases with varying computational demands and each core in the AMP may be beneficial for different phases during the program execution. A dynamic thread to core assignment will be able to adapt to the time-dependent program behavior.

3.3.3 Dynamic Thread Scheduling

Determining the affinity of a program phase to a core in the AMP is crucial for establishing a dynamic thread scheduling scheme. Since prior knowledge about the computational needs of the different workload phases is generally unavailable, there is a need to determine them online. Moreover, the dynamic thread scheduling scheme should consider reassignment of a thread only when that thread has moved to a new and stable phase otherwise the scheme's overhead will become prohibitive. Even before determining the affinity of a phase to a core, there is a need to detect and successfully classify stable phases of execution in a program. Only stable phases should be considered since short-lived (unstable) phases do not justify thread reassignment. The phase classification scheme is the same as that used earlier with parameters (*i*) interval length (*i*) n = 150K instructions, (*ii*) threshold $\Delta = 7.5\%$ and, (*iii*) $m = 4$. The online mechanism used to determine the program phase to core affinity is next described.

3.3.3.1 Determining program affinity to a core online

Once a phase classification mechanism is in place, we need to identify the affinity of the current phase to the different cores in the AMP. The objective here is to non-invasively predict program performance on other core types without the drawbacks of online learning based on sampling. Hardware performance counters (HPCs) reveal information about the characteristics of the thread currently being executed. We therefore, decided to develop a scheme to predict power and performance of an executing application on the host core, as well as other cores in the AMP using HPCs. Our scheme is described in detail in the next section.

3.3.4 Using performance counters to determine thread to core affinity

Hardware performance monitoring counters (HPCs) reveal considerable amount of information about the performance and power consumption of a thread [19, 92]. Most prior research dealing with such estimations use HPCs to predict these characteristics on the same core and not on another core in the AMP. To make thread to core assignment decisions, there is a need to estimate the performance and power of the thread on the host core as well as on the potential core where it may be executed. Performance on the host core can be directly collected from the *IPC* counter, but there is a need to estimate the power on the host core, as well as the expected performance and power of the thread if it would be executed on the other cores. Thus, we need to identify a set of counters that will enable prediction of power on the host core as well as performance and power on the other cores. Our objective is to shortlist potential counters with the most impact.

The performance counters studied by us can be grouped as follows:

• **Instructions per Cycle (IPC)**: Power consumption of the processor is dependent on its activity and the IPC counter provides a good measure of program activity.

• **Fetch counters**: The IPC metric considers only the retired instructions, but in a processor, many instructions are executed speculatively and then flushed from the pipeline. To account for these, we considered *# Fetched instructions, Branch correct predictions (BCP)* and, *Branch mispredictions (BMP).*

• **Miss/Hit counters**: Cache hits and misses play a significant role in performance or power consumption of a core. In this regard, the following event counters: *L1 hit, L1 miss, L2 hit, L2 miss, page hit* and, *TLB miss* are considered.

• **Retired instructions counters**: Performance/power consumption can vary significantly depending on the type of the retired instructions (INT, FP, Memory, Branch). Hence we considered retired instructions counters.

• **Stalls**: The activity of the processor will be low when it experiences dependencies (data or resource conflicts) frequently. We consider stalls due to reservation stations, re-order buffer (ROB), load/store queues (LSQ), register renaming and RAT (Register Alias Table). We refer to this counter as *Dispatch Stalls.*

3.3.4.1 Performance / Power Modeling

To shortlist the most influential performance counters, we used correlation between the counters and the metric that is to be estimated. Estimating power on the same core is not difficult and has been done in prior publications using 3 to 4 counters [19, 92]. In contrast, estimation of the metrics on the other core is not straightforward. Our objective is to use the least number of counters to predict all the required metrics. The reason behind this is not just to save hardware, but also to reduce the number of counters that have to be monitored simultaneously. In current processors, the same counters are used for monitoring multiple events and it is not possible to simultaneously obtain the count for two different events from the same hardware counter [19]. We searched for counters that showed high correlation to power and performance of the other core. Since we are interested in swapping threads (between the LP and HPerf cores) at runtime, we need to estimate the performance/Watt of a thread currently running on LP core, on HPerf core and vice-versa. To this end, we need to analyze, offline, the correlation between the performance counters of the LP (HPerf) core to the power and performance of the thread if it would execute on the HPerf (LP) core. To accomplish this, we identified eight representative benchmarks from the set of 38, such that they included: INT intensive (*intStress,bitcount*), FP intensive (*fpStress,equake*), load/store intensive (*gcc*), have high IPC (*apsi*) and low IPC (*mcf,ammp*). The 8 benchmarks were run on both the cores (LP and HPerf) for 1 billion instructions and the value of the above mentioned performance counters for both the cores were sampled periodically after the commit of every 100K instructions (equal to interval length n described earlier). All the counter values obtained were normalized with respect to the number of cycles elapsed during that period. We then computed the correlation between the normalized counter values of one core and the observed power and performance on the other core, and the results are plotted in Figures 3.22 and 3.23. As can be seen from the figures, the observed correlation to both IPC and power is not very high as the counters used to estimate the performance and power are in the other core. From the initial set of 15 counters, we shortlisted *L2 miss, TLB miss, # Fetched instructions, IPC, Power, retired INT, L1 hit* and *Dispatch Stalls* as they showed reasonable correlation to both IPC and power on the other core. To reduce the number of performance counters that are involved in the estimated IPC and power expressions for the other core, we investigated the correla-

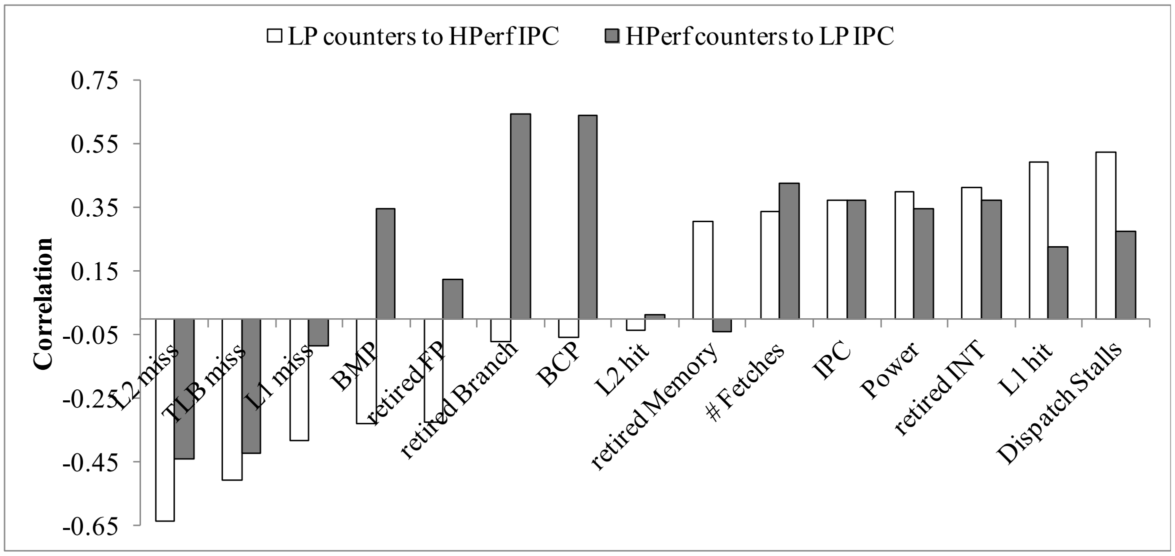


Figure 3.22. Correlation of various performance counters in one core to the observed IPC on the other core.

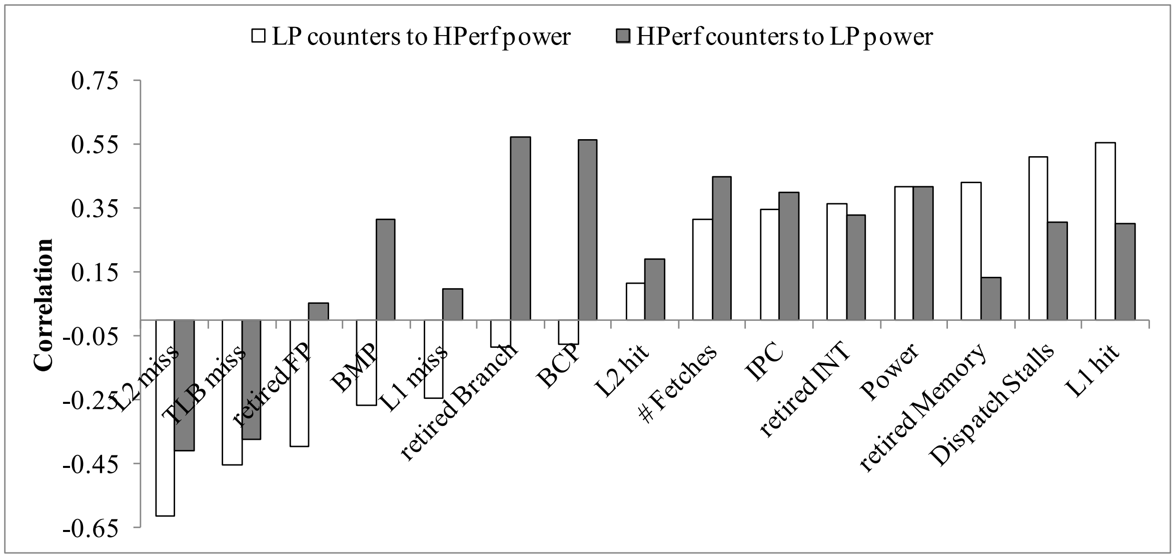


Figure 3.23. Correlation of various performance counters in one core to the power consumed by the other core.

tion of each of the above selected parameters to the rest. The one which correlates well with many other parameters could be used as a proxy for the rest. We found the *# Fetched instructions* to have a high correlation to power, while *IPC* of the current core correlated well with *retired INT* and *L1 hit* counters. Therefore, based on this observation, we chose *L2 miss, TLB miss, # Fetched instructions, IPC* and *Dispatch Stalls* as the main performance counters to be used in our estimation scheme. Having the same set of counters for both the metrics (performance and power on the other core) and for both the core types (LP and HPerf) greatly simplifies the estimation mechanism.

We then used the traces obtained from the 8 selected benchmarks to express the observed performance and power on the other core as a function of the chosen performance counters in the current core. A multi-dimensional curve fitting and regression analysis was performed to obtain expressions for the estimated performance and power for both the core types and these are shown in Table 3.6. A similar procedure was followed to estimate power on the host core using its own counters. We observed that the same set of counters, selected for estimating metrics on the other core, shows a reasonably high correlation to the observed power on the host core too (figure not included due to space constraints). The expression obtained for the online power estimation for the considered dual-core type AMP is shown in Table 3.7.

The accuracy of the expressions obtained was then measured for all 38 workloads. Counter values from the HPerf core were used to estimate its own power as well as the performance and power of the LP core and vice versa. We observed that on an average, the derived expressions estimated power on the host core with a 6.5% error, and IPC and power on the other core with an error of 32% and 9%, respectively. The resulting IPC/Watt average estimation error for the host core was 8%, and was 34.2% for the other core. Even though the errors in estimating metrics for the other core

Table 3.6. Power and performance estimation of the other core using the performance counters of the current core. *L2m - L2 miss, TLBm - TLB miss, S - Dispatch Stalls, F - # Fetched instructions*

Estimating Parameter	Expression
LP IPC	exp(-41.8 × L2m - 30.2 × TLBm - 3.4 × S + 6.5 × IPC - 2.9 × F + 1.44)
HPerf IPC	exp(-389.8 × L2m - 19.6 × TLBm + 3.9 × S + 20.3 × IPC - 22 × F - 3.6)
LP Power	exp(-1.5 × L2m - 2.2 × TLBm - 0.6 × S + 1.2 × IPC - 0.5 × F + 2.9)
HPerf Power	exp(-126.5 × L2m - 4.7 × TLBm + 3.9 × S + 4.2 × IPC - 6.2 × F - 0.4)

Table 3.7. Online power estimation for the host core using its own performance counters. *L2m - L2 miss, TLBm - TLB miss, S - Dispatch Stalls, F - # Fetched instructions*

Estimating Parameter	Expression
LP Power	exp(1.3 × L2m + 1.5 × TLBm + 0.5 × S + 0.5 × IPC + 0.03 × F + 1.7)
HPerf Power	exp(-0.48 × L2m + 4.6 × TLBm - 0.35 × S + 1.3 × IPC - 0.5 × F + 3.3)

are quite high, they proved to be adequate for our purpose of making online thread scheduling decisions. A high estimation error is not important if the right thread to core assignment is made most of the time. We found in our experiments that the proposed estimation based scheme made the right thread scheduling decision 92% of the time, which is acceptable. As will be seen in Section 3.3.5, the 8% erroneous decisions do not have a significant effect on the benefits of the proposed scheme.

3.3.4.2 The complete thread scheduling framework

Having a phase classification mechanism and a scheme to approximately estimate the power and performance of the thread on other cores, we still need a way to govern these two autonomous mechanisms and decide on thread reassignments. The task of managing the phase classification mechanism and the collection of data from the selected performance counters is assumed to be handled by a software layer called the Microvisor. A similar layer has been used by Khan [46] and was previously developed

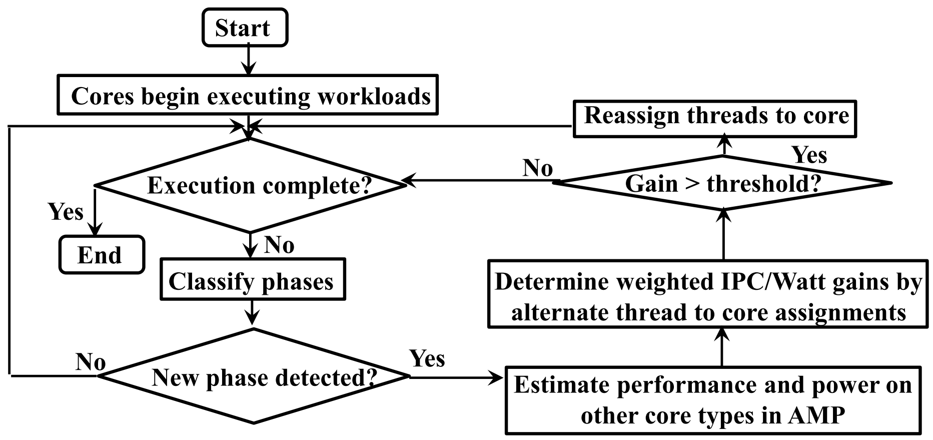

Start
Cores begin executing workloads
Reassign threads to core
Yes
No
Execution complete?
Gain > threshold?
Yes
End
No
Classify phases
Determine weighted IPC/Watt gains by alternate thread to core assignments
No
New phase detected?
Yes
Estimate performance and power on other core types in AMP

Figure 3.24. The thread scheduling flowchart.

by IBM [36]. Additional details on this software layer may be found in those papers. We now describe the working of the entire system, as managed by Microvisor.

The flowchart of the procedure followed in the proposed scheme is shown in Figure 3.24. Eight workloads are run on the dual-core type AMP consisting of six LP and two HPerf cores. Whenever a phase change is detected for any one of the threads by our phase detection mechanism, the power on the host core as well as the power and performance of the thread if executed on the other core are estimated by Microvisor, based on the chosen performance counters (*L2 miss, TLB miss, IPC* and *# Fetched instructions*) of the host core. The performance and power of the other core type running other threads are also collected. The performance/Watt is then calculated for the current and the alternate thread to core assignment. Based on this, the current thread to core assignment may be changed.

The number of potential thread to core assignments to assess increases with the number of simultaneous phase changes for the various workloads. For a single phase change, when the thread on the LP core changes phase, there are two potential threads that it may swap with, i.e. the two threads on the HPerf cores. Similarly, for a phase change in a thread being executed on the HPerf core, there are six threads that it may swap with. Hence, for single phase changes, there are up to six combinations that have to be assessed. We found in our experiments that 92% of the time only a single phase change is detected and the maximum number of simultaneous phase changes detected was 3 (0.2% of the time). Hence, the number of combinations to assess was far lower than the worst case of 8 simultaneous phase changes. Using the estimated performance/Watt of the various threads in an alternate configuration, the weighted performance/Watt improvement (geometric or harmonic speedups may also be used) projected for the new thread to core assignment over the current one is calculated. If the weighted speedup is over 3% (called decision threshold; detailed study was conducted to set this value), the threads are swapped between the two cores. If not, the current thread to core assignment is maintained. Swapping threads between cores incurs an overhead due to context switch and cold cache misses. We assume, conservatively, a swapping overhead of 1K cycles. We observed the system to be not very sensitive to this overhead. Another source of overhead is the invocation of the Microvisor. This was observed to be invoked, on an average, 700 times per run, but this overhead is relatively small as it involves collection of counter statistics and evaluation of the expression. This can be assumed to be at most a few hundred cycles and we found this to have negligible effect on the results. By using phase classification, the proposed scheme needs to make decisions only when stable phase changes are detected, which is not very often. Hence, the overheads associated with decision making are kept at bay. The proposed scheme is evaluated next and compared against various baselines.

3.3.5 Evaluation

In this section, we report the results of our evaluation experiments. Multi-programmed workloads were run on the AMP until one of the threads executed 1 billion instructions. The phase classification parameters were set to: Interval $n =$ 150K, $\Delta = 7.5\%$ and stable phase interval $m = 4$.

We now describe the baselines that will be used for comparison. The performance/Watt improvement achieved by the proposed scheme over each of the baselines is then presented.

3.3.5.1 Baseline configurations considered

We compare our proposed scheme to the following baseline configurations:

• **Static**: Here the thread to core assignment is *static*, i.e., it never changes. This fixed assignment is based on oracular knowledge of the best assignment over the entire run of the workloads and as such is not practical.

• **Online learning-based (*O_Learning*) swapping scheme with sampling overheads**: Threads are dynamically swapped between the cores in this scheme. Detection of phases (based on the ITV scheme) is used as a trigger to initiate a possible swap and the learning is done by sampling the newly detected phase on the other core type of the AMP. This baseline constitutes a modified version of the scheme proposed by Becchi et al [6]. Sampling incurs an overhead and it is assumed to be 1M cycles [6]. Thread swapping overheads are also considered here.

• **Greedy oracle (*G_Oracle*)**: This baseline is capable of swapping threads between the cores. The trigger is once again phase detection, but the thread to core decisions are made based on oracular knowledge at that instant in time, regarding the best current reassignment of threads to cores. No learning overheads are considered for this baseline but thread swapping overheads are taken into account.

3.3.5.2 Performance per watt analysis over the baselines

We considered three speedup metrics to compare our proposed scheme to the baselines. We first define the following terms:

$S_0 = (IPC/Watt_{thread0})_{proposed}/(IPC/Watt_{thread0})_{baseline}$

$S_1 = (IPC/Watt_{thread1})_{proposed}/(IPC/Watt_{thread1})_{baseline}$

The various speedups considered are:

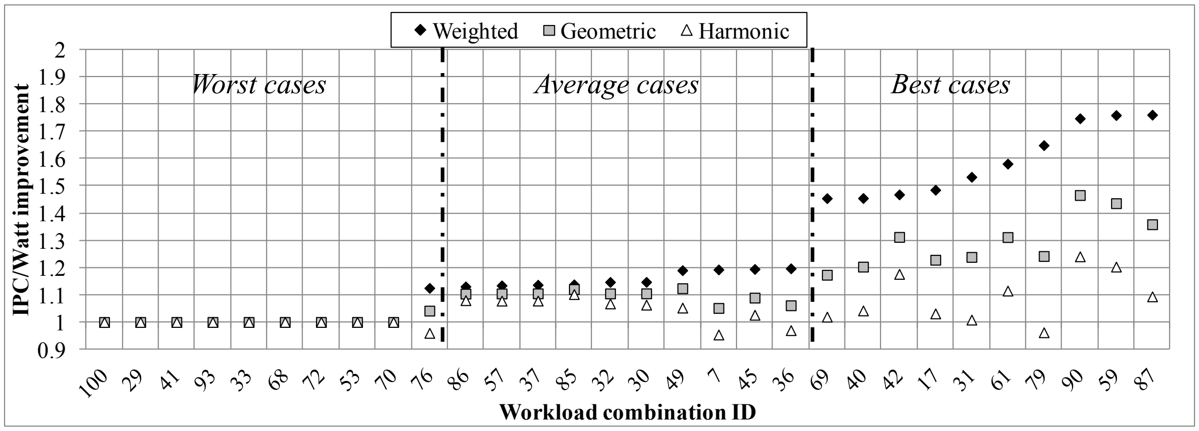


Figure 3.25. IPC/Watt improvement of the proposed scheme against the Static baseline.

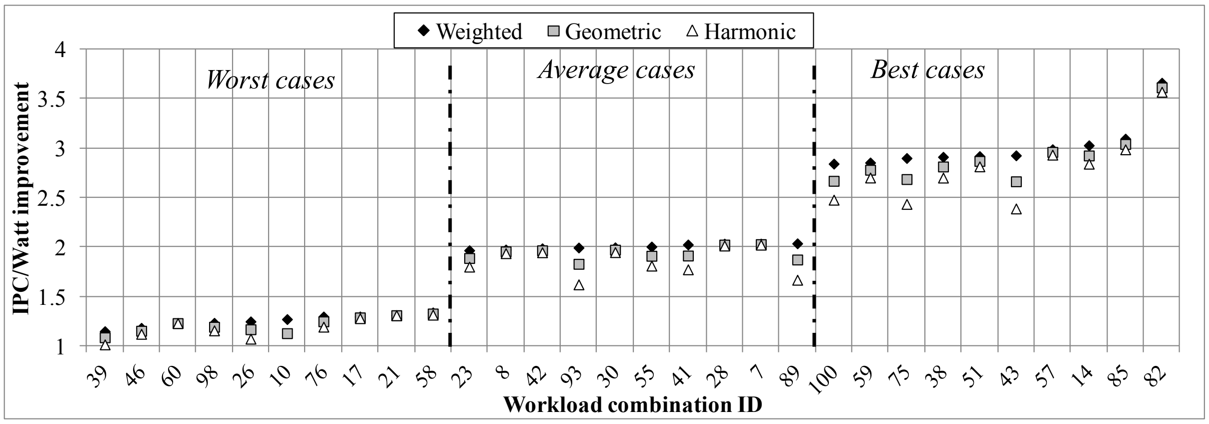


Figure 3.26. IPC/Watt improvement of the proposed scheme against the O_Learning baseline.

1. Weighted: $Speedup_{weighted} = (S_0 + S_1)/2$

2. Geometric: $Speedup_{geometric} = \sqrt[2]{S_0 \times S_1}$

3. Harmonic: $Speedup_{harmonic} = 2/(1/S_0 + 1/S_1)$

From the set of 38 workloads, we randomly selected 100 combinations of eight threaded workloads and had them executed using the proposed as well as each of the baseline schemes. We have plotted a subset (30 of the 100) of those results for various baselines in Figures 3.25, 3.26 and 3.27 for the *Static*, *O_Learning* and *G_Oracle* baselines. The shown 30 combinations include the 10 worst results (out of the 100), the 10 best results and 10 that showed average benefits with respect to

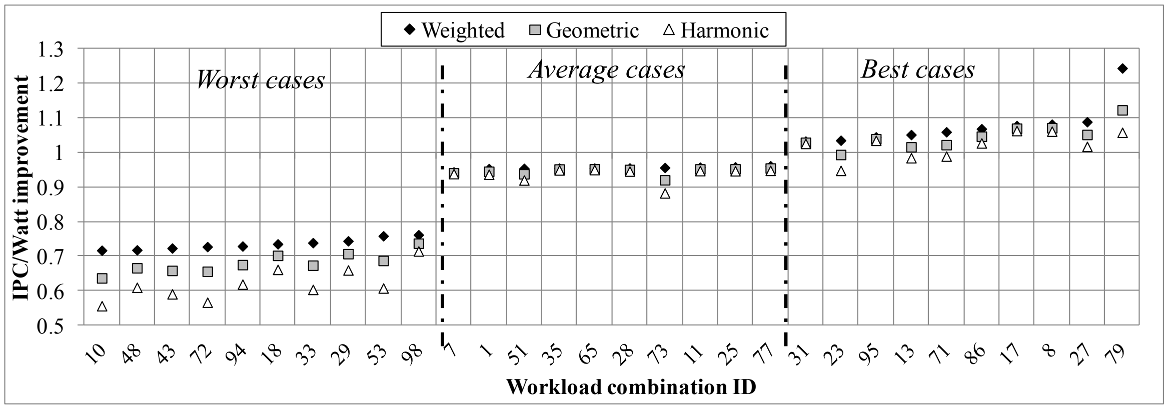


Figure 3.27. IPC/Watt improvement of the proposed scheme against the G_Oracle baseline.

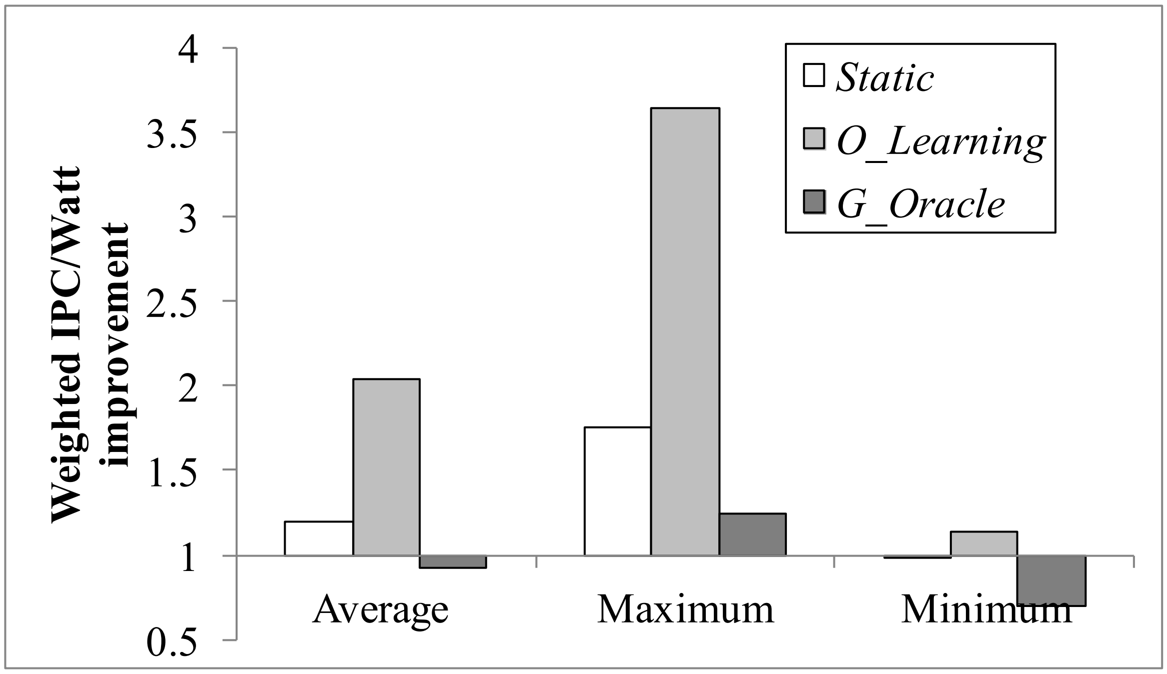


Figure 3.28. Speedup of the proposed scheme against the Static, O_Learning and the G_Oracle schemes.

the weighted IPC/Watt metric. It is clear that in general, considerable IPC/Watt improvement is achieved over the *Static* baseline and in particular, the *O_Learning* baseline, where speedup of up to 3.5X is observed. Amongst the worst cases, it can be seen that an IPC/Watt degradation is observed when comparing against the static baseline (0.99). However, when comparing to the *O_Learning*, even the worst case speedup is 1.14 which shows that the overhead of sampling negates the benefits of the learning-based approach. When compared to the *G_Oracle* baseline, barring a few rare cases, there are no notable gains, as expected. We have also plotted the average, minimum and maximum weighted IPC/Watt gains that the proposed scheme achieves over the baselines in Figure 3.28. It can be seen that on an average, the proposed scheme performs around 20% better than the *Static* baseline with respect to weighted improvement, but what is more noteworthy is that the gain is 200% when compared to the *O_Learning* scheme. The reason for this is the overhead due to sampling (discussed in detail in sub-section 3.3.5.2.1). It can also be seen that the proposed scheme comes to within 92% of what the *G_Oracle* scheme achieves with respect to average weighted gains, which is very encouraging. We provide detailed analysis on these results next.

3.3.5.2.1 Analysis of results

3.3.5.2.1.1 *Static* In this baseline, the thread to core assignment is kept the same throughout the execution. This thread to core assignment is based on an oracle and as such, cannot be done in practice. Still, it can be seen that significant IPC/Watt improvement is achieved by the proposed scheme over this baseline (Figure 3.25). This baseline never takes advantage of phase changes or changes in resource demands. Even if over the entire run, a thread has an affinity for a certain core, there may be periods where this thread would be more affine to another core in the AMP. Hence, the proposed scheme achieves significant improvement in IPC/Watt over the *Static* baseline. Still, there are a few workload combinations where the *Static* baseline performs better. This is mainly due to the mispredictions made by the proposed scheme and the fact that some workloads do not experience many phase changes. However, looking at the average, it is clear that there are only a few mispredictions. The overall benefits (20% on average for weighted gains) more than justify the losses due to mispredictions.

3.3.5.2.1.2 ***O_Learning*** This baseline is dynamic and whenever deemed beneficial, the threads are swapped between cores. The decision to trigger swapping is determined by the same mechanism that is used by the proposed scheme, i.e., phases detected by the phase classification mechanism. Every time a phase change is detected, this scheme initiates an online sampling mechanism. Hence, this scheme is expected to predict thread to core reassignment more accurately than the proposed scheme. However, as mentioned earlier, it suffers from a learning overhead. We found that on an average, there are approximately 700 such events, significantly increasing the overhead of this baseline. This is the reason why the benefits of the proposed scheme over this scheme are higher than even what was obtained against the *Static* scheme (see Figure 3.25 and 3.26, and Figure 3.28). We did not find any case where this scheme performed better than the proposed scheme which is mainly due to the overheads involved during sampling. As the number of core types and workloads increase in the system, the number of phase changes and the number of sampling intervals increase significantly, which nullifies any benefits of this scheme. When ignoring the learning overhead, this scheme performs better than the proposed scheme by 5% on average, due to its more accurate predictions. This shows that even though the proposed scheme is slightly inaccurate in its decision making, the decisions it makes are good enough and they do not incur any learning overheads. These results show that the proposed scheme is a more practical and scalable when compared to the sampling based learning scheme.

3.3.5.2.1.3 ***G_Oracle*** This baseline also has the ability to swap threads between the cores but makes swapping decisions based on oracular knowledge. From Figures 3.27 and 3.28, it can be seen that in general, the proposed scheme performs worse than this baseline. This is expected, as this baseline makes perfect thread to core reassignments without incurring any overheads, which is not practical. What is interesting is that the proposed scheme does better than this oracular scheme in a few rare cases. The reason for this is that sometimes by taking a wrong decision (as is done by the proposed scheme), the opportunities that come up later, as compared to the case where always the right (greedy) decision is made, are different. Sometimes, these additional opportunities may provide even better benefits. Still, on an average, the proposed scheme performs worse than this scheme by 8%.

3.3.6 Conclusions

We have presented a novel technique to assist thread scheduling in AMPs in order to maximize performance/Watt. The key idea is the use of program behavior on one core to predict the power and performance of the application on other cores in the AMP. We leverage the use of performance counters which are available in almost all processors for such a prediction. To illustrate our approach, an eight-core AMP was considered with two core types, one core designed to achieve high performance (HPerf) (two cores) while the other for low power (LP) (six cores). Detailed experiments on the choice of performance counters to estimate the performance and power on the HPerf core while the application executes on the LP core and vice versa have been presented. Approximate expressions based on the values of these counters were formulated to assist in the thread to core assignment so as to maximize performance/Watt. Phase classification was used to trigger the decision making process.

We compared our technique to a static baseline with best thread to core assignment, an online learning based scheme, and an oracular scheme with ability to swap threads between the cores. Our results indicate that the proposed scheme can achieve considerable performance/Watt benefits of about 20% and 200% on an average, over the static and online learning schemes, respectively. Moreover, the proposed scheme performs worse than the oracular scheme by only 8% on average.

CHAPTER 4

IMPROVING THE POWER EFFICIENCY IN SYMMETRIC MULTICORES

Several studies have promoted sharing of large but underutilized resources between cores in a multicore processor [22, 53, 14] to reduce the silicon area at a marginal loss of performance. For example, AMD in its BullDozer architecture [14] has implemented sharing of the entire floating-point unit including reservation stations and execution units. Several research publications, e.g., [22, 53], go beyond FP units and also suggest sharing of caches, crossbars, branch predictors and large latency units. Most previous work only explores the performance impact of such sharing leaving the following questions unanswered.

1. What is the impact of sharing on performance and performance/power? While sharing clearly results in power savings, for certain workloads, performance loss may be too large.

2. What are the most important parameters influencing performance and performance/power in sharing? We show that latency and throughput of the shared resources are dominant determinants of performance and performance/power, but most previous studies ignore them.

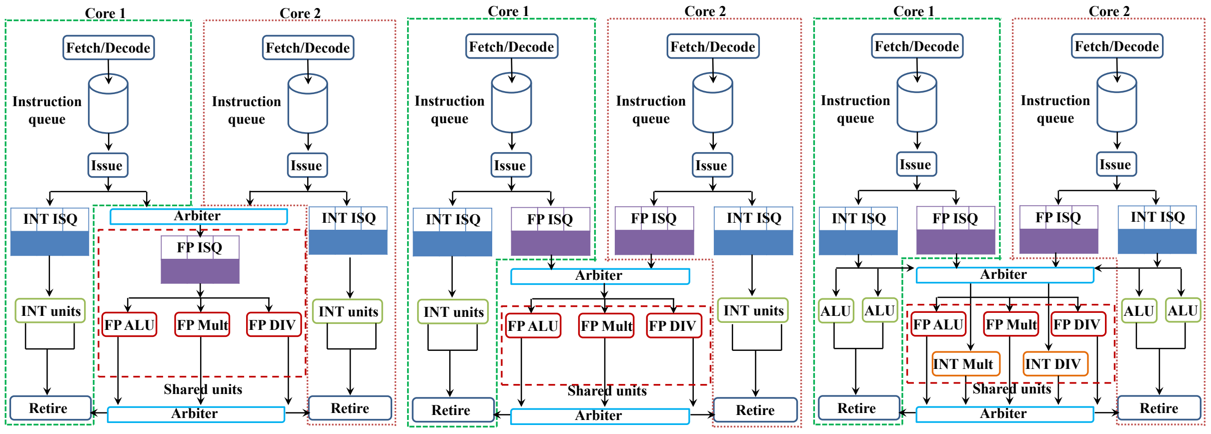
3. How does sharing of resources play out for Big cores or Small cores? Mainstream computing can be broadly classified into performance efficient (Big cores) and power efficient (Small cores). It is thus necessary to study the impact of sharing resources in both such architectures.

4. What is the impact of sharing in Simultaneously Multi-Threaded (SMT) processors? In particular, does sharing in SMT make performance or performance/power better or worse? Given that most mainstream cores are SMT capable[1], studying impact of increased resource utilization due to sharing is important.

In this chapter, we investigate the performance and performance-per-Watt implications of sharing large and underutilized resources between a pair of cores in a multicore processor. At first, we study sharing of the entire floating-point datapath by two cores, similar to AMD's Bulldozer [14], where the issue queue (ISQ) and the FP execution units are shared. Using combination of workloads from various benchmarks, we study both the performance and performance-per-Watt when compared to the baseline architecture that does not involve sharing. Our findings show that while sharing results in considerable power savings, the performance penalty may be high ($\sim$28%) for certain workload combinations.

To mitigate the impact on performance, while still retaining some of the power benefits of sharing, we limit sharing to the underutilized execution units. For most workloads, FP instructions are not frequently encountered. Hence, we first explore sharing of just the FP execution units, while the individual cores retain their reservation stations. This modification yields higher performance compared to previous schemes. Still, a worst case performance loss of 14% is observed. Integer divide and multiply instructions are also encountered infrequently. Therefore, we extend our study to include the corresponding units. We find that sharing the integer divide and multiply units has only a small impact on both performance and performance-per-Watt. A summary of the resource sharing options explored in this section is shown in Figure 4.1.

[1]www.intel.com



(a) Sharing the issue queue and FP execution units (b) Sharing the FP execution units (c) Sharing the FP and integer divide and multiply units

Figure 4.1. Overview of the studied resource sharing. ISQ = issue queue, FP = floating-point, INT = integer.

The utilization of the shared units depends on the width of the fetch and execution path. Accordingly, we target cores at opposite ends of the power/performance spectrum. On the higher end of the performance scale we consider a superscalar processor analogous in resources to Intel Nehalem/AMD K10 architecture (Big core). At the lower end of the power scale, we consider a processor similar in resources to Intel Atom/AMD Bobcat architecture (Small core). Our study includes both single threaded and SMT processor architectures. We also analyze the sensitivity to communication latency between the cores and the shared units. Our results show that while architectures that share execution units do provide power benefits at a negligible performance penalty (∼5% on average), such benefits hold only when the shared units have low latency and are highly pipelined. Performance and performance-per-Watt loss are observed for workloads that exhibit high contention for the shared execution units. To reduce the performance loss due to contention we propose to increase the throughput of the shared resources via Dynamic Voltage and Frequency Boosting (DVFB) which is controlled dynamically by the occupancy rate. Our results show that such dynamic boosting not only overcomes losses due to contention, but also results in significant increases in both performance (upto 13%) and performance-per-Watt (up to 14%), while realizing considerable savings in area ($\sim$ 7-10% per core).

The following are the key contributions of this section:

1. We present a study on the performance and performance-per-Watt implications of three resource sharing alternatives for a dual-core processor.

2. We study the performance and performance-per-Watt implications of resource sharing in SMT cores.

3. We analyze the sensitivity of resource-sharing architectures to latency and performance of the shared resources.

4. We show that while execution unit sharing has negligible impact on performance and positive impact on performance-per-Watt for most benchmark combinations, there are cases where resource contention results in a penalty as high as 22%.

5. We present a dynamic voltage and frequency boosting (DVFB) scheme for the shared resources to mitigate the impact of resource contention, that not only compensates for the loss, but also increases the performance of most workload combinations.

6. Finally, we describe a novel hardware-based feedback control mechanism for DVFB that automates the dynamic control process.

4.1 Related work

The idea of resource sharing has long been in existence [20, 44, 22, 102, 15]. The previous approaches can broadly be classified into those that target improvements in fault tolerance, performance or performance/Watt.

4.1.1 Sharing resources to improve yield and fault tolerance

Sharing resources across cores to improve yield and reliability has been studied by several researchers. [20] and [44] proposed sharing execution units to reduce the die size and thus increase yield. [88] explored the possibility of using multiple execution units already present within a processor to improve manufacturing yield at the cost of performance degradation . A similar approach was followed in [99]. [94] make use of the underutilized execution units for test. Here a small checker core incorporated into the design of the larger core to check its operation. The checker core shares execution units with the host core. [32] propose sharing each stage in pipeline between neighbouring cores in a CMP. When one core experiences a pipeline stage failure, it takes over or shares the healthy stage from the adjacent core. A similar approach but only for fault tolerance of large execution units was adopted by [71]. In their scheme, whenever a core experiences failure of a local large execution unit, it outsources the execution of instructions to the neighboring core via a queue. [78] used integer (INT) ALU sharing between cores for fault tolerance and potential performance mitigation in the presence of failed components.

4.1.2 Sharing resources for improving performance/ performance-per-Watt

The idea of sharing resources for performance or performance/Watt in a multi-core has seen several manifestations. Simultaneous Multi-Threading (SMT) [102, 60] was introduced more than a decade ago to improve the utilization of resources in microprocessors. In SMT, multiple threads are run on the same core and threads share and compete for core resources. Dynamic resource sharing occurs naturally in SMT processors. [22] explore intermediate design points between the CMP and SMT architectures where the sharing of the caches, branch predictor and long latency execution units is explored. A similar study was presented in [53] where the caches, crossbar and floating-point units were shared. Significant area savings at a minor loss of performance were reported. Both these schemes only focus on performance and do not consider performance/Watt. In addition, the impact of the shared resource access latency, or the effects on SMT processors were not studied. [105] explore flexible sharing of a pool of "execution engines" among various processor cores. By ensuring that the producer and immediate consumers are sent to the same engine, efficient usage of the shared units was made possible. Still, each engine requires a queue and other data to keep track of producers and consumers which result in a complex design. In [10], authors propose the sharing of functional units across cores in a 3D stacked die for online testing and/or performance improvement. A similar approach to 3D resource sharing was proposed in [39] where the Reorder Buffer (ROB), register file, instruction queue and the load/store queues were shared. Dynamic exchange of execution units between pairs of cores was investigated in [76, 81]. Here, depending on the current workload characteristics, the cores may exchange execution units to maximize performance/Watt. The major advantage of such an architecture is that resource contention between the two cores does not take place but the design of the two cores is complicated. Further, this scheme will always incur the hardware and power overhead of two sets of execution units compared to the single set in our scheme.

4.1.3 The AMD Bulldozer^{TM}architecture

The first resource sharing architecture we study is similar to the AMD Bulldozer design [14]. In AMD's Bulldozer the fetch, decode and the entire FP execution (reservation stations and execution units) are shared between pairs of cores in a dual-core processor. In our study we also analyze a design that involves the sharing of the FP execution only.

4.2 Shared Resource Multicore Architecture

We now present an overview of the target of our study – the shared resource multicore architecture. Hardware modifications necessary to support such an architecture are also described.

4.2.1 Preliminaries

A high level view of the studied architectures is shown in Figure 4.1. We consider the following three resource sharing alternatives.

4.2.1.1 Sharing the FP ISQ and execution unit (S_FP_QX)

Here the FP issue queue (ISQ) and FP execution unit are shared between two cores. This architecture is similar to AMD's Bulldozer but note that the Bulldozer design also shares the fetch and decode units. Sharing leads to contention for resources and the first point of contention here (in S_FP_QX) is the FP ISQ. Whenever FP instructions are ready to be scheduled, the control logic first checks to see if there is a slot available in the shared ISQ. Since the ISQ is shared, the number of entries available per core is reduced. Hence, whenever both the cores sharing the ISQ run FP intensive applications, the ISQ is expected to become a bottleneck in the design which may lead to pipeline stalls and performance loss. Another source of stalls is the shared execution units. Just like the ISQ, the effective number of execution units available is reduced in the dual-core architecture. Hence, a higher number of stalls is expected when FP intensive applications are run on the two cores that share the FP units.

4.2.1.2 Sharing the FP execution unit only (S_FP_X)

In this instantiation, sharing of FP execution units only is explored. Unlike the previous case, the only source of contention here is the availability of the FP execution units. Hence, we expect a lower performance loss but also lower power savings compared to the previous scheme.

4.2.1.3 Sharing the FP execution units as well as the integer divide and multiply units (S_FP_INT)

In this instantiation, in addition to the FP execution unit, integer divide and multiply units are also shared. The number of stalls for this scheme is expected to be higher than for the S_FP_X architecture but greater power savings is expected.

Since resources are shared in all three architectures, there is a need for a centralized control mechanism that will grant access to the requester core. This is accomplished by means of an arbiter shown in Figure 4.1. The arbiter accepts requests and depending on the availability of the shared resource, grants access. Note that in all three cases, accesses to the shared execution units are independent and hence multiple requests may be sent to them at the same time. Once execution is complete, the execution result must be forwarded to the core that generated the request. This is accomplished by another arbiter that forwards the result to the rightful owner. We do not provide implementation details of the arbiter, which is fairly straightforward.

4.3 Experimental setup

Table 4.1. Chosen core parameters.

Parameter	Small	Big	Parameter	Small	Big	Parameter	Small	Big
Issue	2	4	INTREG	64	96	FPREG	64	80
INTISQ	16	36	FPISQ	16	24	LS units	1	3
LSQ	32	32	ROB	56	128	L1(I/D)	32K	32K
L2	512K	2M	Freq (GHz)	1.5	2.4	Type	OOO	OOO

Table 4.2. Execution unit specifications for the cores. (P - Pipelined, NP - Not pipelined, PP - Partially pipelined)

Core	FP DIV	FP MUL	FP ALU	INT DIV	INT MUL	INT ALU
Small	1 unit, 60 cyc, NP	1 unit, 4 cyc, PP	1 unit, 5 cyc, P	1 unit, 207 cyc, NP	1 unit, 10 cyc, P	2 unit, 1 cyc, P
Big	1 unit, 21 cyc, P	1 unit, 5 cyc, P	2 units, 3 cyc, P	1 unit, 23 cyc, P	1 unit, 8 cyc, P	4 units, 1 cyc, P

Table 4.3. Characteristics of the considered workloads

barnes_barnes	cholesky_cholesky	fmm_fmm	lu_lu	radix_radix	raytrace_raytrace
water_water	flops_fbench	equake_art	gzip_ammp	art_ammp	mcf_gcc

Table 4.4. Workloads considered for the experiments where each core runs two threads. The + sign between workloads indicates that they are run on the same core and the _ is used as separator to indicate what is run on cores 1 and 2.

barnes+barnes_barnes+barnes	cholesky+cholesky_ cholesky+cholesky	fmm+fmm_fmm+fmm
lu+lu_lu+lu	radix+radix_radix+radix	raytrace+raytrace_raytrace+raytrace
water+water_water+water	equake+art_flops+fbench	mcf+gcc_art+ammp
equake+art_gzip+ammp	mcf+gcc_flops+fbench	equake+flops_art+fbench
mcf+art_gcc+ammp	equake+gzip_art+ammp	mcf+flops_gcc+fbench

To evaluate the idea of sharing infrequently used execution units for a wide variety of architectures, we considered processor cores at the two ends of the performance/power spectrum, i.e., a high-performance core (Big) and a low-power core (Small). These cores are representative of the Intel Nehalem/AMD K10 and the Intel Atom/AMD Bobcat architectures, respectively. In the rest of this section, we will refer to them as Big and Small. Note that Big/Small refers to homogeneous dual-core processors.

In Tables 4.1 and 4.2 we describe the resource sizes and execution resource characteristics for the the two core types. The parameters were inspired by commercial architectures [26].

SESC was used for architectural performance simulation [75]. We made significant modifications to the simulator to enable shared resource execution with arbitration. Power was estimated using Wattch [13] and Cacti [89]. In the experiments we targeted 15 benchmarks: 7 from the SPLASH-2 [107] (*barnes, cholesky, fmm, lu, radix, raytrace, water*) and 8 from the SPEC 2000 benchmark suite [97] (*fbench, flops, art, equake, gzip, ammp, mcf, gcc*). These workloads were chosen for their instruction distribution and performance diversity. Several combinations of workloads were considered for the two cores running single threads. We also considered the case of SMT, where each core runs two threads for a four thread combination. Homogeneous workload combinations were created using multiple threads from the SPLASH-2 workloads. We also created heterogeneous workloads by combining threads from the SPEC 2000

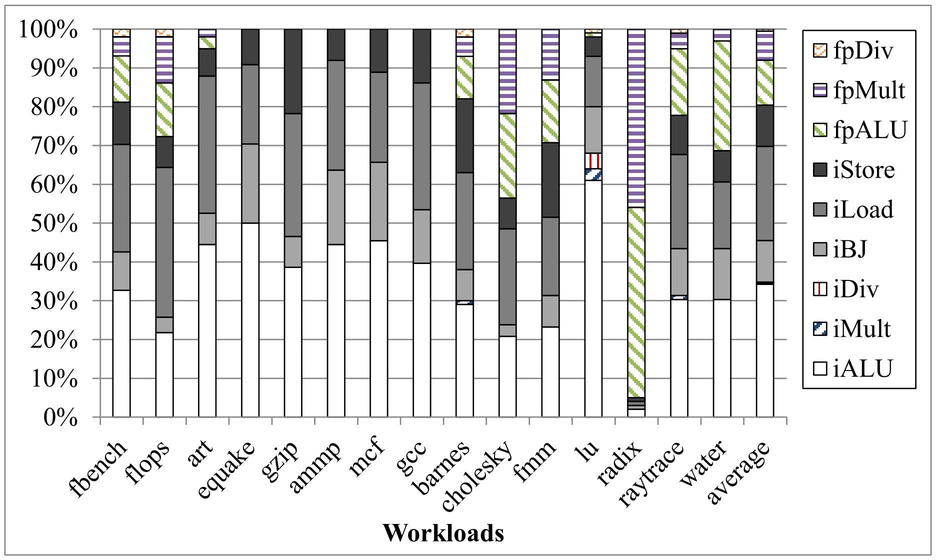


Figure 4.2. The instruction distribution of the various workloads when run for 500 million instructions. The average over all workloads is also shown.

suite. The created workload sets are summarized in Tables 4.3 and 4.4 for the single and SMT experiments, respectively. We thus tried to evaluate the studied architectures over a broad spectrum of potential workloads. Each workload was run until the sum of the instructions retired on the two core types equaled 500 million instructions. The instruction distribution of each individual thread run is shown in Figure 4.2.

4.4 Analysis of resource sharing in single threaded processors

We first present results and analysis for processors running single threads per core. Two cores share resources according to S_FP_QX, S_FP_X and S_FP_INT schemes described in Section 3. The workloads run in these experiments are shown in Table 4.3.

4.4.1 Performance and performance/Watt results

In this section, the performance and performance/Watt of the studied architectures relative to the one where no sharing takes place are presented. Sensitivity to the shared resource access latency is also analyzed. In the next section we study the

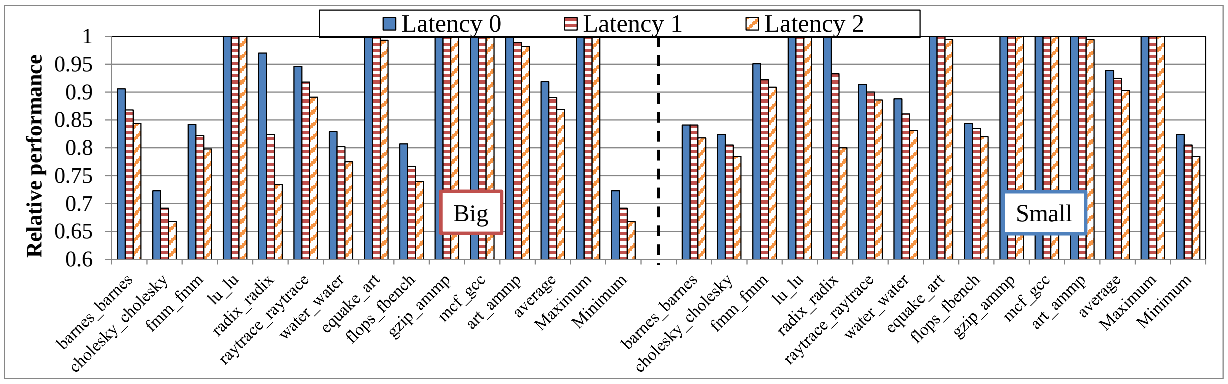


Figure 4.3. Performance of the Big and Small cores resulting from the sharing of the FP ISQ and execution units (S_FP_QX) between the cores relative to a dual-core that does not share them for various communication latencies (between zero to two cycles).

effect of sharing in SMT processors where more than one thread runs on the same core. To compare the resource sharing architectures with the one that does not, three speedup metrics were used including the weighted, geometric and harmonic metrics. For the sake of brevity, only the results using the harmonic metric are presented.

The harmonic speed-up metric is calculated as follows:

$S_0 = (IPC_{thread0})_{new}/(IPC_{thread0})_{baseline}$

$S_1 = (IPC_{thread1})_{new}/(IPC_{thread1})_{baseline}$

$Speedup_{harmonic} = 2/(1/S_0 + 1/S_1)$

Here, baseline refers to the case where the cores do not share any unit. The performance/Watt speedup/slowdown is calculated similarly.

4.4.1.1 Sharing the FP ISQ and execution units (S_FP_QX)

4.4.1.1.1 Performance analysis The performance of the Big and Small cores in the S_FP_QX configuration relative to the non-sharing architecture is shown in Figure 4.3. Shared resource access latencies of zero, one and two cycles were considered. The communication latency of zero cycles represents the ideal case where the design has been optimized to support sharing. It can be seen that even in this scenario, a significant performance loss is observed for both core types. Specifically, a worst case

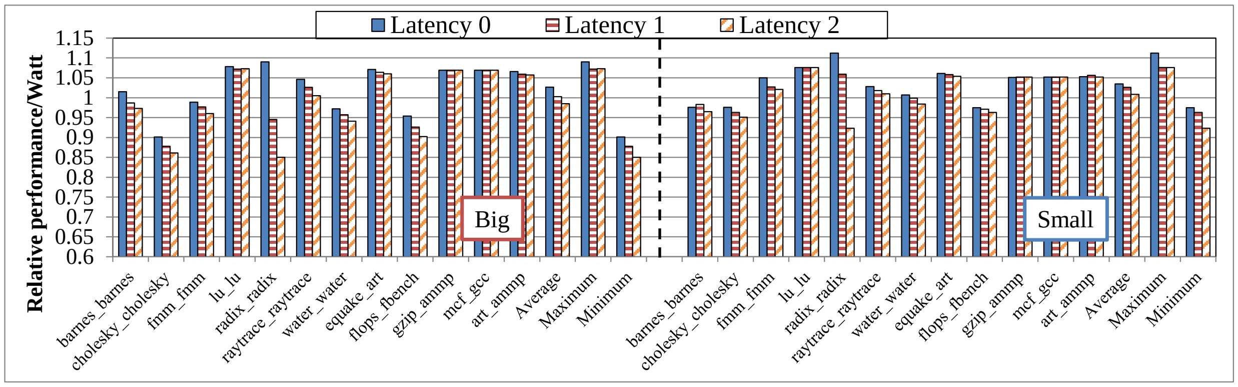


Figure 4.4. Performance/Watt of the Big and Small cores resulting from the sharing of the FP ISQ and execution units (S_FP_QX) between the cores relative to a dual-core that does not share them for different (zero to two cycles) communication latencies between the cores and the shared units.

performance penalty of 28% and 18% (workload *cholesky_cholesky* when run on both the cores) is observed for the Big and Small cores, respectively. This architecture shares the FP ISQ and the FP execution units. Thus, two potential bottlenecks exist in the system yielding a large performance penalty. Increasing the communication latency results in an even larger performance penalty, as expected. This clearly shows the sensitivity of such a resource sharing architecture to communication latency. On an average, ∼5-10% performance penalty is observed for both the core types which increases with communication latency. These results show that when sharing resources between cores, special consideration must be given to the resource access latency. The workloads that do not experience a slowdown are the ones with little or no FP instructions in the mix (e.g., *equake, art, gzip, gcc*). Interestingly, the Small core does not suffer as much as the Big core with respect to performance. The Small core is moderately sized when compared to the Big core and consequently, the experienced bottleneck has a greater effect in the case of the Big core.

4.4.1.1.2 Performance/Watt analysis The performance/Watt resulting from the sharing of the FP ISQ and the FP execution units (S_FP_QX) relative to the non-sharing architecture is shown in Figure 4.4 for both the core types. It can be seen that performance/Watt improvements are achieved for most workloads on both the core types, especially for the ones with no FP instructions. In general, FP instructions are not as frequently encountered as integer ones and hence, for a majority of the workloads this architecture will result in power savings. However, there are workloads where the performance/Watt degrades by as much as 10% (e.g., *cholesky_cholesky* when run on the Big core) even with communication latency of zero cycles. This indicates that even though, in general, this architecture results in power savings, for workloads that contest for the shared resources, the performance/Watt will degrade. On an average, a 2.5% improvement for the Big core and a 3.5% improvement for the Small core were observed when the communication latency was set to zero cycles. Increased latency reduces this improvement.

Even though the S_FP_QX architecture results in power savings in general, the experienced performance penalty can be very large (∼28%). This results in poor performance/Watt and hence, we explored alternative sharing schemes to help mitigate the performance penalty.

4.4.1.2 Sharing only the FP units (S_FP_X)

4.4.1.2.1 Performance analysis The performance of the S_FP_X architecture relative to the one where each core has its own execution units for the Big and Small cores are shown in Figure 4.5 for the various workloads considered. For zero communication latency, it can be seen that for all the workloads, there is no notable performance penalty for the Big core. Even for cases where both threads highly utilize the shared units, no performance penalty was observed (e.g., *cholesky_cholesky, radix_radix, flops_fbench*). This is because the Big core has large and fast execution units that are fully pipelined and unless contention takes place in the same cycle, no performance penalty will be experienced. This indicates that for a high performance core, contention related performance loss will rarely be a problem when the considered execution units are shared even for workloads that include a large proportion of instructions that need the shared units. The worst case performance penalty has dropped to lower than 1% for the Big core, which is a significant improvement when compared to the S_FP_QX architecture (∼28% performance loss in the worst case). This shows that in the Big core, the major bottleneck is the FP ISQ. Increasing its size may help mitigate the performance penalty but may result in power increase. However, such an analysis is out of scope in this thesis. With an increase in communication latency, there is a notable drop in performance. Still, for small latencies (one to two cycles), the performance penalty is well within reasonable limits (within 5% even for communication latency of two). Note that communication latency of one to two cycles is realistic. A similar assumption has been made in [22, 53, 32]. Hence, for cores such as the Big core, for small shared resource communication latencies, the performance loss is acceptable if FP execution units are shared between pairs of cores. This is mainly attributed to the highly pipelined and low latency execution units.

The results obtained for the Small core do show notable performance penalty, even for the ideal case of zero communication latency. This happens due to non-pipelined and relatively higher latency execution units present in the Small core. Since not all the execution units are pipelined, there is greater chance for contention for the shared units. For example, for a non-pipelined multiplier with latency of 10 cycles, the execution unit cannot accept any more requests during the 10 cycles that follow this request. If this unit was pipelined, unless contention takes place in the same cycle, a performance penalty will never be observed. In particular, the performance loss is the worst for *barnes_barnes* and *flops_fbench* (13-14%). In both these cases, the workloads running on each core exhibit significant proportion of FP instructions and as a result contention is very high for the shared resources. The average performance loss is within 8% for a two cycle communication latency. It is thus clear that for cores with non-pipelined and large latency execution units, sharing may result in

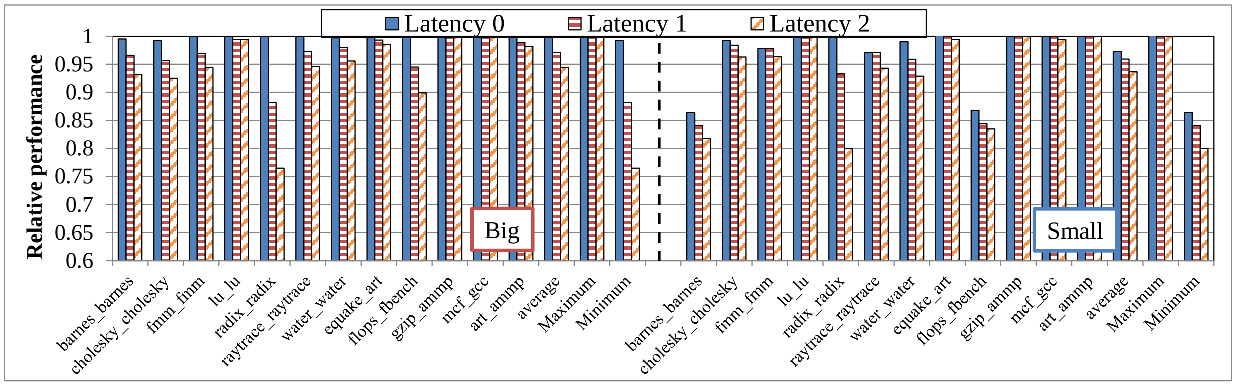


Figure 4.5. Performance of the Big and Small cores due to sharing of the FP execution units (S_FP_X) relative to a dual-core that does not share them, for different communication latencies. The different bars correspond to various round-trip communication latencies (zero to two cycles) between the cores and the shared units.

significant performance loss. When compared to the S_FP_QX architecture, for the Small core the average performance loss drops from the observed 7% (for a zero cycle communication latency) to around 3%. Hence, this architecture certainly results in lower performance penalty.

4.4.1.2.2 Performance/Watt analysis Sharing the large and infrequently used execution units results in static power savings. This is expected to improve performance/Watt especially for the cases where no notable performance penalty is observed. However, power savings are not as large as that observed for the S_FP_QX architecture. The performance/Watt results obtained for both core types are shown in Figure 4.6. We have already seen that for the Big core there is no notable performance loss even for a communication latency of two cycles between the core and the shared units. Performance/Watt improvements of >1 were observed for the Big core with communication latency of one cycle. It can be concluded that for the Big core, sharing of large execution units results in performance/Watt gains when considering realistic communication latencies.

For the Small core, performance loss due to sharing even in idealized conditions (communication latency of zero cycles) results in significant performance loss for

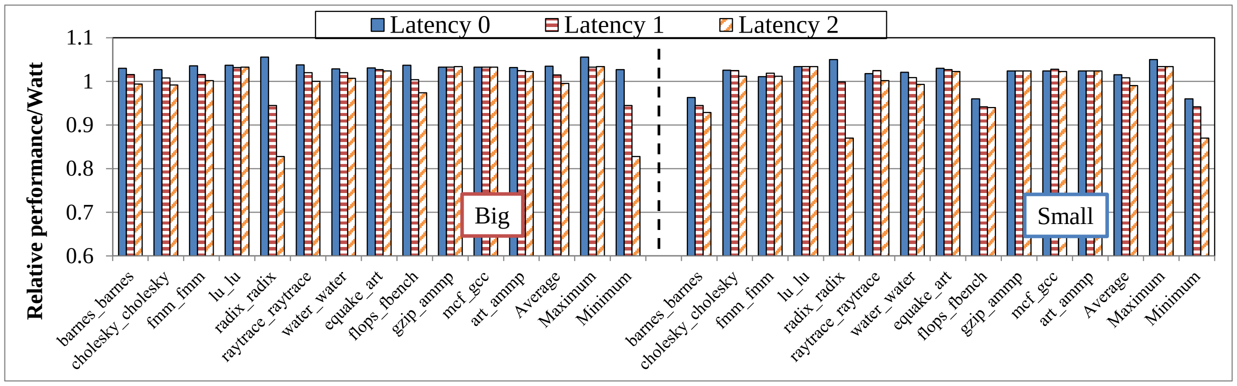


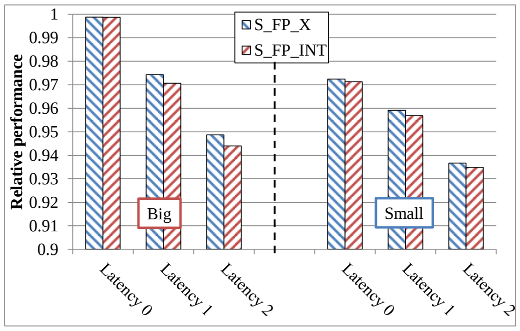
Figure 4.6. Performance/Watt of the Big and Small cores due to sharing of the FP execution units (S_FP_X) relative to a dual-core that does not share them, for different communication latencies. The different bars correspond to various round-trip communication latencies (zero to two cycles) between the cores and the shared units.

several workloads. As a result, performance/Watt results are very modest with a few workloads experiencing performance/Watt loss. Still, on an average the performance/Watt gains are >1 for zero cycle communication latency. Figure 4.6 shows performance/Watt gain of 1.5% on an average. Just like the Big core, increasing this latency to more than one cycle results in overall performance/Watt loss when compared to the baseline architecture. It is important to note that apart from two workloads (*barnes_barnes, flops_fbench*) all other workloads show a small improvement in performance/Watt. From Figure 4.5, it is observed that apart from those two workloads, there were also others such as *fmm_fmm, raytrace_raytrace* that showed performance loss but when considering performance/Watt, show improvements over the baseline. Hence, execution unit sharing architectures do in general improve performance/Watt.

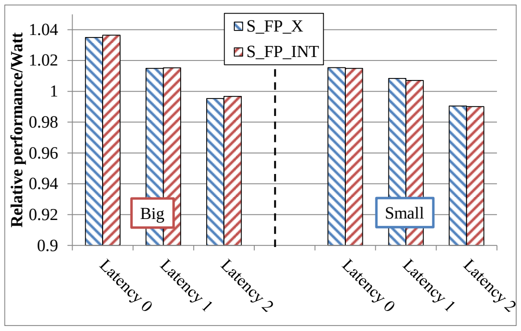
Based on the results presented in this section, we can conclude that for Big cores, sharing FP execution units results in almost no performance loss but may result in small performance/Watt gains. In contrast, for Small cores, even though there is a small performance/Watt gain for low communication latencies (between the core and the shared units), performance and performance/Watt losses observed for a few workload combinations, make the sharing of FP execution units between such cores questionable. This architecture provides slightly lower performance/Watt as the S_FP_QX architecture without considerable performance penalties which is a significant advantage.

4.4.1.3 Extending the sharing to include INT divide and multiply units (S_FP_INT)

Most prior work has explored the sharing of only the FP units between pairs of cores [22, 53]. However, from Figure 4.2, it can be seen that apart from the workload *lu_lu*, no other workload shows any notable INT divide or multiply instructions. Thus, sharing these units in addition to the FP units, is a natural extension. We call the resulting architecture the **S_FP_INT sharing** architecture. We analyzed such additional sharing and the average results obtained over all workloads when run on each core type modelled as the S_FP_X sharing and S_FP_INT sharing architecture with respect to performance and performance/Watt are plotted in Figures 4.7(a) and 4.7(b), respectively. Three possible communication latencies between the core and the shared units are considered. All results are shown relative to the architecture that does not share execution units. In general, it can be seen that for both the core types, with respect to performance, S_FP_X sharing is slightly better than the S_FP_INT sharing architecture and the opposite trend is observed with respect to performance/Watt. However, the differences are too small to prefer one architecture over the other. But since INT divide and multiply are relatively large execution units and sharing them certainly yields area savings (details on area savings to soon follow). Hence, we conclude that S_FP_INT sharing enhances the benefits of S_FP_X sharing architectures.



(a) Performance



(b) Performance/Watt

Figure 4.7. Performance and performance/Watt of the Big core and Small core in S_FP_X and S_FP_INT configurations relative to a dual-core that does not share resources for various communication latencies.

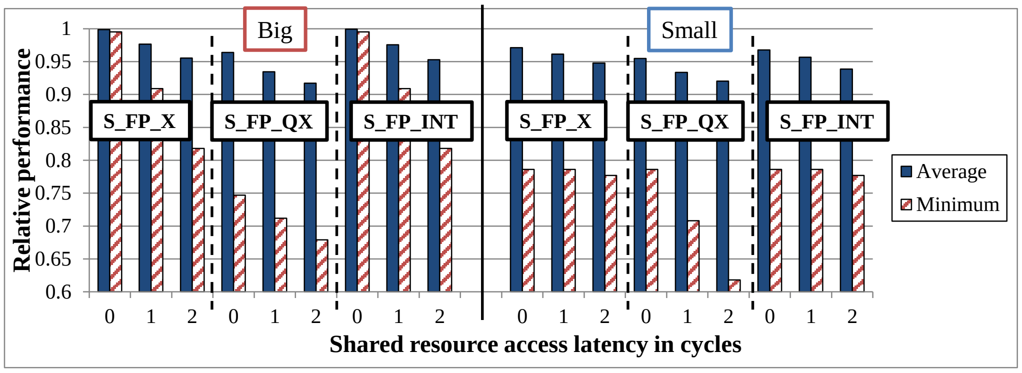


Figure 4.8. Performance of the Big and Small cores in the S_FP_QX, S_FP_X, S_FP_INT configurations relative to the baseline for various communication latencies. Two threads were run on each core.

4.5 Analysis of sharing in SMT processors

We now present results on the effect of sharing resources in SMT processors. In these experiments, each core runs two threads. The various workload combinations considered are shown in Table 4.4. For the sake of brevity, only average and minimum speed-up over all the considered workloads for each of the three resource sharing architectures relative to the baseline (where no sharing is implemented) are presented.

4.5.1 Performance analysis

The average and minimum performance achieved by the three resource sharing architectures relative to the one with no sharing is shown in Figure 4.8.

4.5.1.1 The S_FP_X and S_FP_INT architectures

In general, we found that the architectures that only share execution units result in more or less the same level of performance for both the core types. Hence, we discuss both these architectures in this sub-section. Just as in the case of running one thread per core, for the Small core, a larger performance penalty was observed when compared to the Big core. As mentioned earlier, the window for contention is larger due to the limited capability of the execution units in the Small core yielding a relatively larger penalty. The worst case of 22% performance loss was observed for the workload *barnes+barnes_barnes+barnes* which constitues an increase of 8% over the observed 14% when running the workload *barnes_barnes* in the earlier experiments. There were also some low IPC workloads such as *raytrace+raytrace_raytrace+raytrace* where performance penalty was smaller than that obtained while running *raytrace_raytrace*. On an average, a 3% performance penalty was observed for the Small core.

For the Big core, ignoring communication latency, a 1% performance loss is observed in the worst case and an even smaller penalty is seen on an average. This result is similar to that observed when running only a single thread per core. This indicates that even when up to four threads compete for the execution resources of the Big core, limited performance penalty will be experienced, which is mainly attributed to the large and fully pipelined execution units. In summary, we find that even in SMT processors, sharing execution resources between cores is expected to result in negligible performance penalty in Big cores and sometimes a notable performance penalty in Small cores.

4.5.1.2 S_FP_QX

From Figure 4.8 it is clear that the S_FP_QX architecture results in a larger performance penalty than S_FP_X and S_FP_INT for both core types. Ignoring communication latency, we have observed that an average performance loss of 4% and 5%

and a worst case loss of 22% and 25% were observed for the Big and Small cores, respectively. This performance loss increases with an increase in communication latency as expected. For the Small core, just as for the S_FP_X and S_FP_INT architectures, the worst case was observed for the workload *barnes+barnes_barnes+barnes*. Another workload that exhibited a significant (17%) performance penalty was *radix+radix_radix+radix*. No performance penalty was observed for the same workload when running on the S_FP_X and S_FP_INT architectures. This shows that this workload suffers mainly from stalls in acquiring reservation station slots on the small core. Overall, the performance loss goes up by 2% on an average when compared to the S_FP_X and S_FP_INT for the Small core.

On the Big core, in the single threaded experiments, the workload *cholesky_cholesky* experienced the worst case of 28% performance loss. The loss was reduced to 16% when running the workload *cholesky+cholesky_cholesky+cholesky* in SMT mode. The reason for this penalty drop is that in SMT mode, a multicore IPC of 0.35 was observed, which was a drop from the observed IPC of 0.5 in the single threaded experiments. Thus, additional stalls due to resource sharing do not have a high impact on the performance. A worst case performance loss of 25% was observed for the workload *water+water_water+water*. This constitutes a 8% increase in the observed 17% performance penalty when running the workload *water_water*. Hence, for the workload *water*, increasing the number of thread contexts per core results in an increased penalty for the Big cores. The performance loss is higher by 4% when compared to the S_FP_X and S_FP_INT architectures for the Big core.

In summary, performance is expected to degrade for a few workloads in either of the sharing architectures. For the Big core, performance penalty is expected only in the S_FP_QX design. When compared to the experiments where only single threads were run on each core, performance penalty may sometimes be lower for SMT proces-

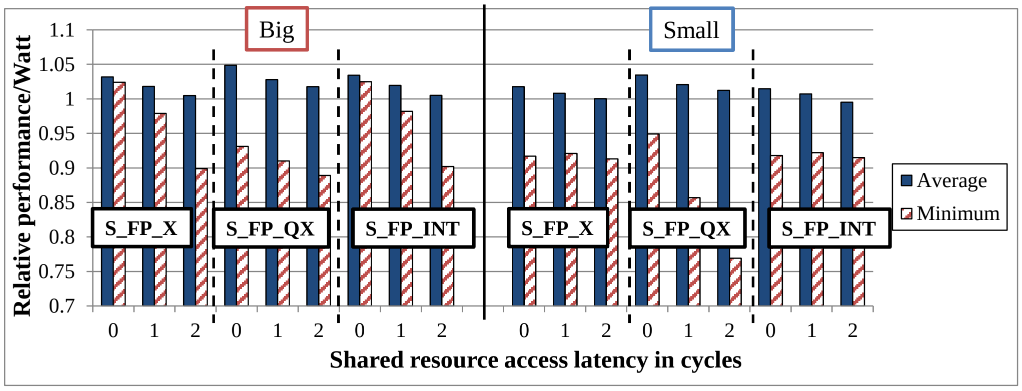


Figure 4.9. Performance/Watt of the Big and Small cores in the S_FP_QX, S_FP_X and S_FP_INT configurations relative to a dual-core that does not share resources for various communication latencies. Two threads were run on each core.

sors. The reason for this is that in SMT mode resource utilization is higher. Hence, if IPC is low, performance penalty due to sharing is also low.

4.5.2 Performance/Watt analysis

The performance/Watt of the various resource sharing architectures relative to the one with no sharing is shown in Figure 4.9.

4.5.2.1 S_FP_X and S_FP_INT

We have seen that for these architectures, little or no performance penalty was observed on the Big core. Consequently, power savings that result from sharing resources lead to performance/Watt gains. Such gains drop with an increase in the shared resource access latency. On an average, a performance/Watt gain of 3.1% and 3.5% were observed for the S_FP_X and S_FP_INT designs on the Big core. On the Small core we observed a significant performance penalty for some workloads. A worst case performance/Watt loss of 8% was observed for the workload *barnes+barnes_barnes+barnes*. However, on an average, a small performance/Watt gain of around 1.7% and 1.4% is observed for the S_FP_X and S_FP_INT architectures, on Small cores. Note that the performance/Watt gain does not drop below 1 for either configuration on both the Big and Small cores, even with a two cycle communication latency.

4.5.2.2 S_FP_QX

In general, performance loss on this architecture was larger than for the S_FP_X and S_FP_INT architectures. However, the power savings were far greater. Hence, even though the worst case performance/Watt loss of 8% was observed on the Big cores, an average gain of 5% and a maximum gain of 11% were observed for the workload *radix+radix_radix+radix*.

A similar result was observed on the Small core, where an average performance/Watt gain of 3.5% and a maximum gain of 7% were observed for the workload *raytrace+raytrace_raytrace+raytrace*.

In summary, this architecture results in better performance/Watt than the other two. We have seen that the performance penalty is smaller in the case of SMT processors. Therefore, in general, the performance/Watt gain also turns out to be greater than for the single thread case.

4.6 Dynamic Frequency Boosting (DFB) and Dynamic Voltage and Frequency Boosting (DVFB)

In the previous experiments we have observed that some workload combinations experience a significant loss of performance in shared architectures with a more pronounced loss for Small cores. As indicated earlier, there are two reasons for this performance degradation. The first one is contention for the shared resources and the second reason is communication latency between the core and the shared resources. Performance loss due to contention can be mitigated if the shared resources run faster. This may be achieved via more powerful and small latency shared execution units [22]. However, as was observed in Figures 4.3 and 4.5, the performance of most workloads does not degrade by sharing resources. Furthermore, increasing the strength of the execution units will result in power inefficiency for these workloads. Therefore, we propose the use of Dunamic Frequency Boosting (DFB) or Dynamic Voltage and Frequency Boosting (DVFB) where, depending on the workload characteristics, the voltage and/or frequency of only the shared execution units is increased. We only consider boosting of the shared execution units and not the shared ISQ in the case of the S_FP_QX configuration as accelerating the ISQ is not expected to yield any benefit.

Selective boosting of the shared execution units is achieved via Voltage and Frequency Islands (VFI) [55, 27, 42, 85]. In VFI, part of the processor core is operated at one voltage and/or frequency, while another part may be operated at a different voltage and/or frequency. For example, Ghosh *et al.* make use of voltage scalable hybrid arithmetic units in [29] for power benefits. Most previous work makes use of this concept for energy savings. Our objective is performance improvement of the shared resources only during periods of resource contention. This may potentially also result in performance/Watt improvement. Given that the shared execution units are already separated from the cores, placing them in an island is relatively simple. We did not consider full-chip voltage and frequency boosting due to its inherent power inefficiency.

Performance boosting may be achieved by increasing the frequency of the shared units. Often, power is the limiting factor that governs operating frequency. The frequency may be increased as long as package thermal limits are not exceeded and the circuit timing margins are not violated. Since the execution units are shared, increasing their frequency results in a much smaller power increase than full-chip boosting. Hence, if the circuits allow increasing the frequency of operation on demand, the implementation is simple. We call this mode the High Frequency Mode (**HFM**). For some circuits voltage may also need to be increased to meet the timing requirements.

We call this mode the High Voltage and Frequency Mode (**HVFM**) and this mode is expected to incur a higher energy penalty. Note that these two modes are mutually exclusive for a given design and are analyzed here for completeness of the evaluation. Either the circuit allows HFM and HVFM is never needed or vice-versa. Thus, in the shared resource VFI, three modes are considered; the Nominal Mode (**NM**) with nominal voltage and frequency, the High Frequency Mode (HFM) and the High Voltage and Frequency Mode (HVFM). The voltage and frequency levels used for both core types in all the three modes are shown in Table 4.5. These values were obtained from [24] and from data available on Intel's turbo boost technology[2][3]. The high frequency modes can potentially mitigate the performance loss due to resource sharing. On the other hand, power overhead is also expected. It is thus necessary to limit the use of these modes to only those instances when the shared resources are overwhelmed.

In order to model the high frequency modes in our experiments, the latency of the shared execution units was reduced proportionally to the gains provided by the increase in frequency. Latencies are set back to the usual values when the system returns to NM. Cycles are always measured in the units of the NM frequency. Hence, we continue to use performance/Watt as the metric to measure relative speedup even though the shared resource island may switch between NM and HFM/HVFM.

Table 4.5. The voltage and frequency levels considered for the two cores.

	High Voltage and Frequency Mode (HVFM)		High Frequency Mode (HFM)		Nominal Mode (NM)	
Core	Voltage	Frequency	Voltage	Frequency	Voltage	Frequency
Big	1.35V	3.4 GHz	1.1V	3.4 GHz	1.1V	2.4 GHz
Small	1.35V	2.13 GHz	1.1V	2.13 GHz	1.1V	1.5 GHz

We first present results on performance and performance/Watt when operating the Small cores in the HFM/HVFM throughout the execution. A dynamic scheme

[2]http://www.intel.com/content/www/us/en/processors/core/core-i5-processor.html

[3]http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html

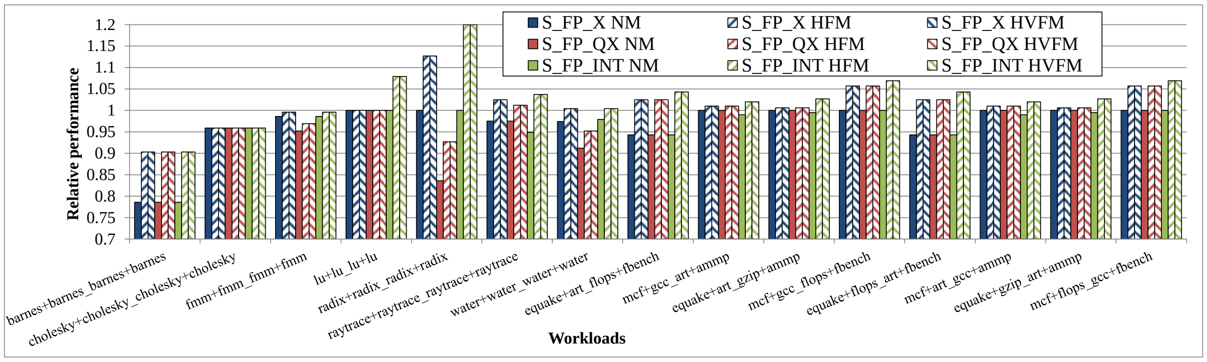


Figure 4.10. The performance of the three resource sharing designs of the Small core relative to the design that does not share resources, for various workloads when operated in the NM, HFM and HVFM. Latency of zero cycles was considered.

to switch between operation modes is then presented. We do not explore boosting the performance of the Big core since the shared execution units are not expected to become a bottleneck.

4.6.1 Static Voltage Frequency Scaling

Here the shared execution units are always operated in the boosted mode (HFM/HVFM) irrespective of the workload characteristics. Such a scheme will result in increased power dissipation but is an interesting case to study as a potential upper bound on the performance mitigation possible by frequency boosting. The calculated harmonic performance and performance/Watt speedups for all the considered workloads when executed on small cores in SMT mode for the NM, HFM and HVFM operating modes are shown in Figures 4.10 and 4.11, respectively. A shared resource communication latency of zero cycles was considered to get a representative picture without loss of generality.

4.6.1.1 Performance analysis

It can be seen that the performance is significantly improved in the boosted modes (HFM/HVFM) for several workloads. In particular, the workloads *barnes+barnes_barnes+barnes*, *radix+radix_radix+radix* and any workload running *flops* and *fbench*, show a consider-

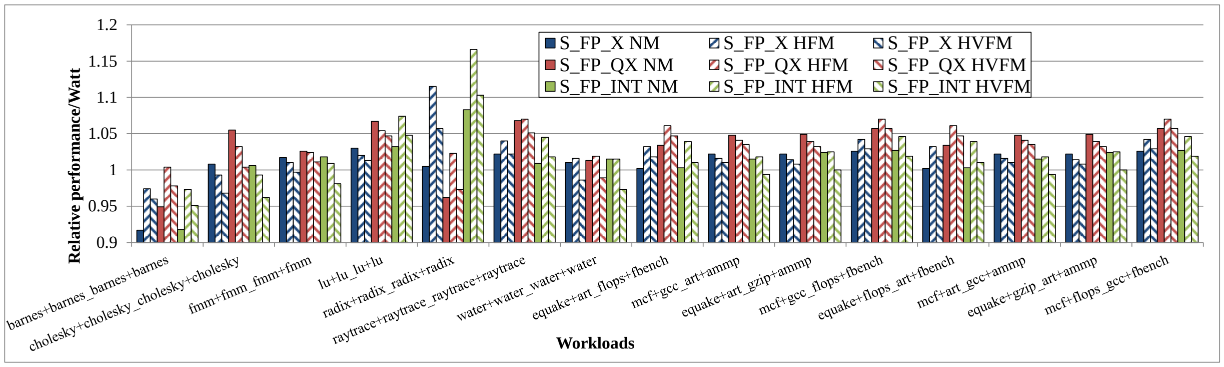


Figure 4.11. The performance/Watt of the three resource sharing designs of the Small core relative to the design that does not share resources, for various workloads when operated in the NM, HFM and HVFM. Latency of zero cycles was considered.

able performance gain (7-20%) in the boosted modes of operation. There are also several workloads such as *cholesky+cholesky_cholesky+cholesky*, *fmm+fmm_fmm+fmm*, *equake+art_gzip+ammp*, and *mcf+gcc_art+ammp* where no notable improvement is observed. There is no difference between the HFM and HVFM modes as is evident from the figures. The boosted modes achieve a 4-5% on an average and a maximum of 20% improvement in performance over the NM mode. Clearly, from a performance stand point operating in the boosted mode is the best option.

4.6.1.2 Performance/Watt analysis

With respect to performance/Watt, it can be seen that there are workloads that benefit from the HFM/HVFM. Workloads such as *barnes+barnes_barnes+barnes*, *radix+radix_radix+radix* show a 6-7% improvement in performance/Watt. However, there are several workloads where performance/Watt in the NM mode is the highest. These workloads are *cholesky+cholesky_cholesky+cholesky*, *fmm+fmm_fmm+fmm* and workloads containing the combination *equake+art_gzip+ammp* and *mcf+gcc_art+ammp*. These were the workloads where no notable performance improvement was observed (see Figure 4.10). Between the HFM and HVFM, the HVFM performs worse which is expected. This mode requires a higher voltage and hence results in larger power r

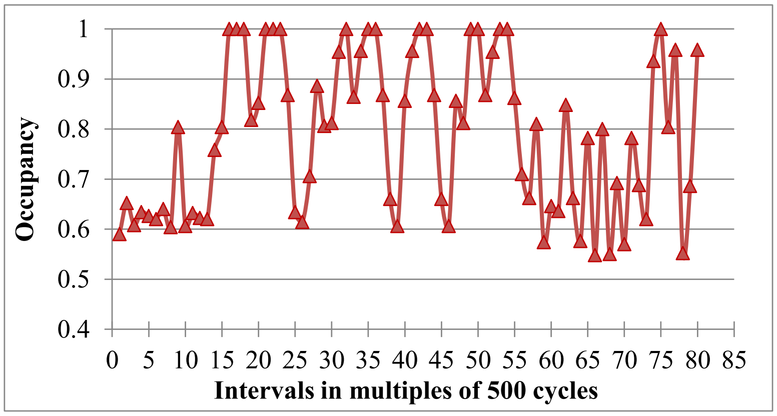


Figure 4.12. The occupancy of the unit with the highest occupancy of all the shared units, over intervals of 500 cycles for the workload *flops_fbench* when running on the Small core in S_FP_INT configuration.

penalty than the HFM. These results clearly show that operating in the HFM or HVFM modes is not desirable with respect to performance/Watt for several workloads. A dynamic scheme may yield better results.

4.6.2 Dynamic Voltage Frequency Scaling

To motivate the need for a dynamic scheme, we show in Figure 4.12 the occupancy of the unit with the highest occupancy of all the shared units over intervals of 500 cycles for the workload *flops_fbench* when run on the Small core. Occupancy is measured as the number of cycles during which the unit is busy within the interval. It can be seen that the worst case occupancy changes over time and some windows show 100% occupancy while others, much less. Clearly a dynamic scheme is needed to optimize both performance and performance/Watt.

4.6.2.1 Switching between NM and HFM/HVFM

We developed a simple hardware scheme to enable switching between the HFM/HVFM and NM. The shared resources will form a bottleneck whenever the contention for any one of the shared units increases. Occupancy or utilization of the shared execution units can potentially provide a good estimate of whether the bottleneck exists. Hence,

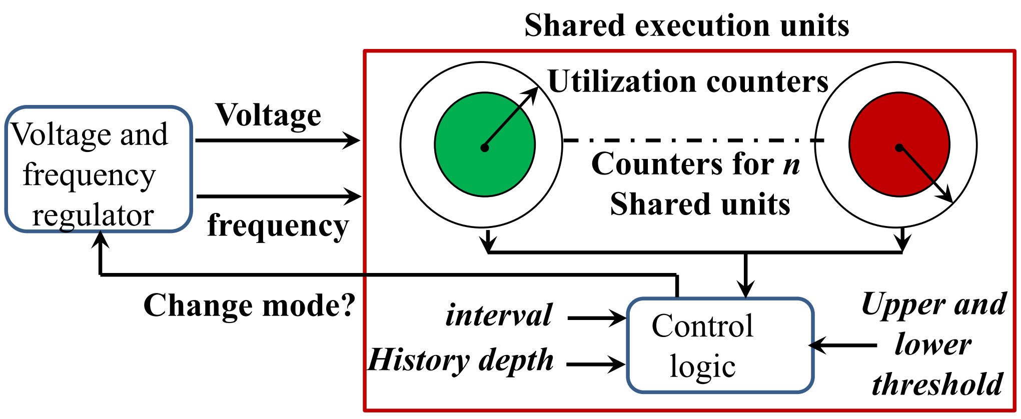


Figure 4.13. A high level view of the feedback control mechanism that may be used to control the voltage and frequency of the VFI containing the shared resources.

we make use of resource occupancy or utilization as a metric to switch between the NM and the boosted modes of execution.

Performance monitoring counters are available in most modern microprocessors [19, 92]. For our purposes, we need as many counters as there are shared units to count the number of busy cycles for each execution unit. Whenever the occupancy for any shared unit exceeds a threshold (upper), the boosted mode is enabled. Switching back to the NM takes place when utilization reduces below a threshold (lower). As the occupancy of the execution units changes over time, it is necessary to keep checking for utilization within small intervals. At the end of each interval, all the counters are set to zero so that counting for the new interval may begin afresh. Furthermore, to avoid too frequent voltage and/or frequency changes, a switch is initiated only if the decision to switch was observed for atleast 90% of the last $HisD$ windows, referred to as history depth. For example, considering $HisD = 10$ a switch in operating mode is affected only of the decision to switch was observed for atleast 9 of the 10 recent windows. In the rest of this section, we refer to the scheme that switches between NM and HFM as Dynamic Frequency Boosting (DFB) and the the scheme that switches between NM and HVFM as Dynamic Voltage Frequency Boosting (DVFB)

A simple illustration of the mechanism to control the mode switching is shown in Figure 4.13. There is a utilization counter for each shared execution unit. The control logic monitors these counters and accepts as input certain parameters that we call interval length, history depth, threshold upper and lower (soon to be introduced). The utilization for that window is then calculated and depending on the current operating mode of the VFI and the values of the input parameters, a change in operating mode signal may be sent to the voltage and/or frequency regulator. Note that utilization (proportion of busy cycles) is always measured with respect to the cycle time of NM. Since the execution units are accelerated in the boosted modes, this effectively reduces the utilization, potentially mitigating the bottleneck. The following four parameters of the dynamic mechanism need to be determined:

1. The **window** or **interval length** (*IntLen*) in cycles after which the utilization counters must be sampled. Choosing too small a value may result in noisy behavior, while too large a value may result in missing potential opportunities.

2. The **number of intervals** to wait until high confidence decisions may be made. This is called the **history depth** (*HisD*). A switch in mode is initiated only if the decision to switch was observed for 90% of the last *HisD* windows. Here as well, choosing too small a depth may result in frequent mode switches while too large a depth may result is missing opportunities to switch mode.

3. The threshold to enter HFM/HVFM from NM. We call this **Threshold Upper** (*ThU*). A mode switch takes place only when the utilization of one of the shared execution units exceeds the *ThU*.

4. The threshold to go back into NM from HFM/HVFM. We call this **Threshold Lower** (*ThL*). This mode switch takes place only when the utilization of **all** shared execution units goes below the *ThL*.

The search space to determine the best optimal combination of parameter values is very large, so reducing its size is necessary. Our objective is to find the set of

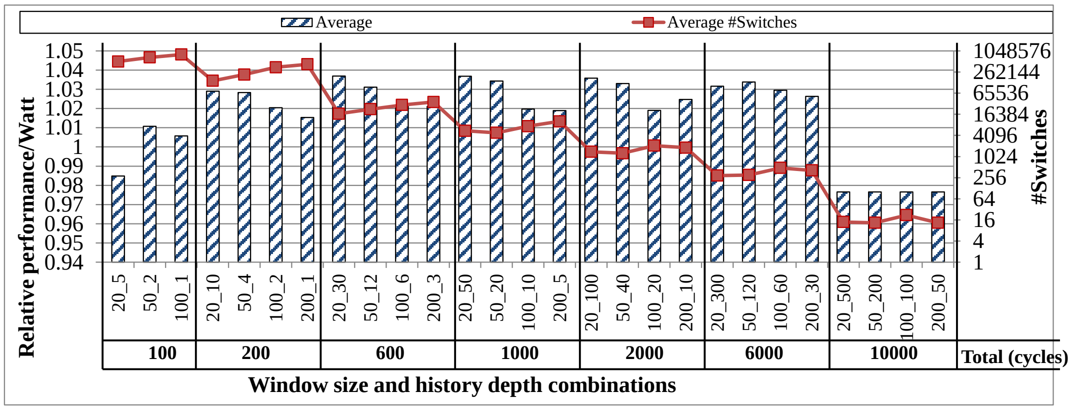


Figure 4.14. Setting *IntLen* and *HisD*. The x-axis is read as *IntLen_HisD*. The thresholds were constant during these experiments and were set to: upper = 85%, lower = 50%. The relative performance/Watt is shown on the primary y-axis while the number of switches in mode is shown on the secondary y-axis.

parameter values that results in performance and performance/Watt improvement for a majority of the workloads.

The choice of parameter values is likely to be a function of the workload currently being executed. The best method may be to learn the behavior of all workloads offline and based on this, set the values of the parameters. However, this method is not practical and is time consuming. In our experiments, the workloads *barnes+barnes_barnes+barnes*, *raytrace+raytrace_raytrace+raytrace* and *equake+art_flops+fbench* were found to result in the worst performance and performance/Watt on the Small core. Hence, we selected these workloads for the training experiments. The parameters values determined in these experiments will be used for all the other workloads. In this experiment, the boosted mode considered was HFM and an overhead of 10 cycles was used as the time to transition between operating modes (details on the overhead to soon follow).

In our experiments we considered various values for the *IntLen*, *HisD*, *ThL* and *ThU*. To reduce the search space, we carried out two different experiments. In the first experiment we set *ThL* and *ThU* as constants and varied the *IntLen* and *HisD*. In the second experiment, *IntLen* and *HisD* were set to constants and the values of

ThL and *ThU* were varied.

• **Determining *IntLen* and *HisD***: The results of the first experiment are plotted in Figure 4.14. Here *ThL* was set to 50% and *ThU* to 85%. *IntLen* was varied in between 20 and 200 and *HisD* between 1 and 500. We found that combinations of small *IntLen* and large *HisD* results in fewer mode switches. The number of mode switches tends to increase with larger *IntLen* and small *HisD*. Small *IntLen* will always result in a noisy behavior. Note that a decision to switch mode is made only if it holds for 90% the last *HisD* windows. With smaller *IntLen* stable decisions are not always expected, reducing opportunities. Even though the number of mode switches increases with larger *IntLen* and small *HisD*, sometimes it may result in thrashing between modes and this results in performance/Watt degradation. It is thus necessary to find the right compromise between the parameters. From the figure it can be seen that th best compromise is achieved for parameter values where 600 cycles $\leq$ $IntLen \times HisD \leq$ 2000 cycles. Based on this observation, we set *IntLen* to 20 and *HisD* to 50.

Similar experiments were conducted to determine *ThU* and *ThL*. In these experiments, we set $IntLen = 20$ and $HisD = 50$ based on the previous experiment. We found that there was not much sensitivity to the thresholds and based on observations set $ThU = 85\%$ and $ThL = 50\%$.

In summary, the selected paramters are: $IntLen = 20$, $HisD = 50$, $ThU = 85\%$, $ThL = 50\%$. In the rest of this section, we refer to the scheme that switches between NM and HFM as Dynamic Frequency Boosting (**DFB**) and the scheme that switches between NM and HVFM as Dynamic Voltage and Frequency Boosting (**DVFB**).

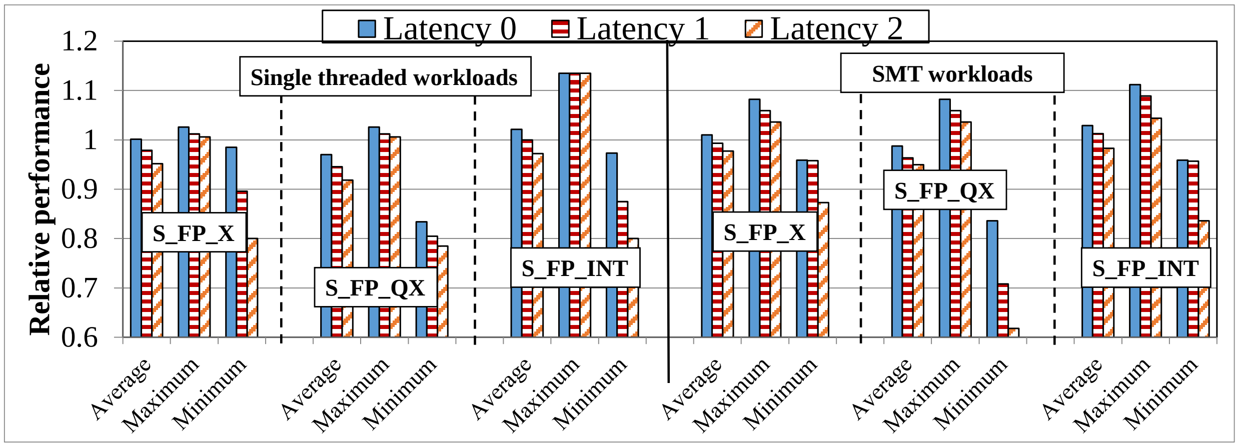


Figure 4.15. Relative performance of the Small core in the S_FP_X, S_FP_QX and S_FP_INT configurations for various communication latencies when run using DFB. Results presented are summarized over all workloads for both the single threaded and SMT workloads.

4.6.3 Performance and performance/Watt analysis when using the proposed DFB or DVFM schemes

We now present the performance and performance/Watt achieved by the resource sharing architectures equipped with DFB and DVFB. Results for the Big core are not shown as the shared execution units were not found to be a bottleneck.

4.6.3.1 Performance analysis

The average, maximum and minimum relative performance of the DFB scheme over the baseline inwhich no sharing takes place in the S_FP_X, S_FP_QX and S_FP_INT configurations are shown in Figure 4.15 for communication latencies of zero to two cycles. Results are shown for both single threaded and SMT workloads. A comparison of the relative performance obtained in NM, DFB and DVFB modes are presented in Figure 4.16.

Considering the single threaded workloads, the worst cases observed in the NM for the S_FP_X and S_FP_INT configurations were for the workloads *barnes_barnes* with relative performance of 0.86 and *flops_fbench* with relative performance of 0.87. The performance of these workloads was significantly increased by 13-15% with an

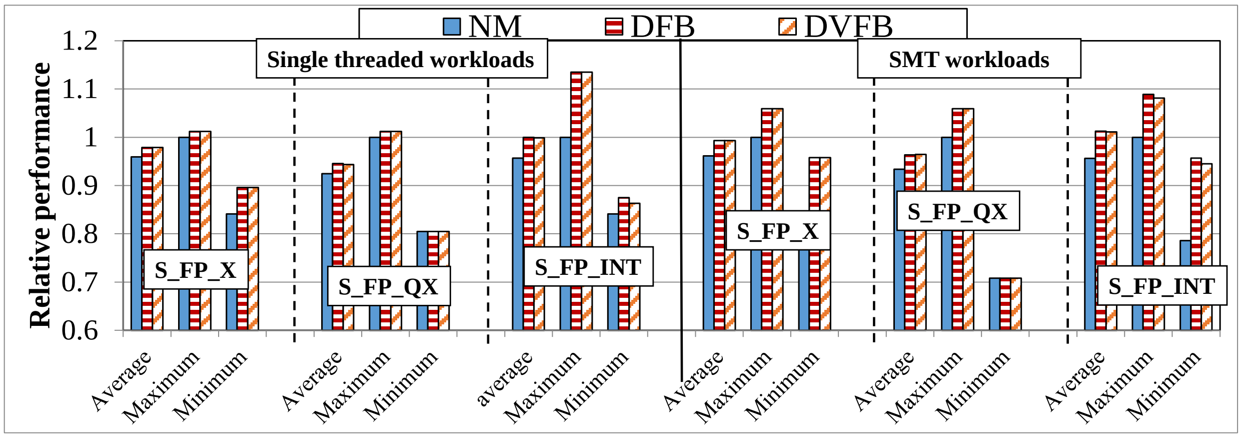


Figure 4.16. Relative performance of the Small core in S_FP_X, S_FP_QX and S_FP_INT configurations in NM, DFB and DVFB for communication latency of one cycle when run using DFB. Results presented are summarized over all workloads for both the single threaded and SMT workloads.

observed relative performance of 0.99 and 1.026 for these two workloads, respectively. On an average, performance was boosted by 3% for the S_FP_X configuration and by 4.5% for the S_FP_INT configuration when compared to the NM. Maximum improvement in performance of 3% and 13% were observed for the S_FP_X and S_FP_INT configurations, respectively over the baseline. There were instances where integer divide and multiply units were bottlenecks for a few workloads (containing *raytrace* or *lu*). Boosting the performance of these units resulted in significant performance gains of as high as 13% for *lu_lu*. For the S_FP_QX configuration, the worst case was observed for *barnes_barnes*, *cholesky_cholesky*, *water_water* and *flops_fbench* with relative performance of 0.84, 0.82, 0.88 and 0.84, respectively. Using DFB, the relative performance of these workloads was increased to 0.96, 0.83, 0.89 and 1.02, respectively, but not all workloads showed such notable improvement. The reason for this is that these workloads suffered more due to stalls in the ISQ and not the execution units. On an average, performance improvement of 4% was observed for the S_FP_QX configuration when compared to the NM. Increasing the latency of the

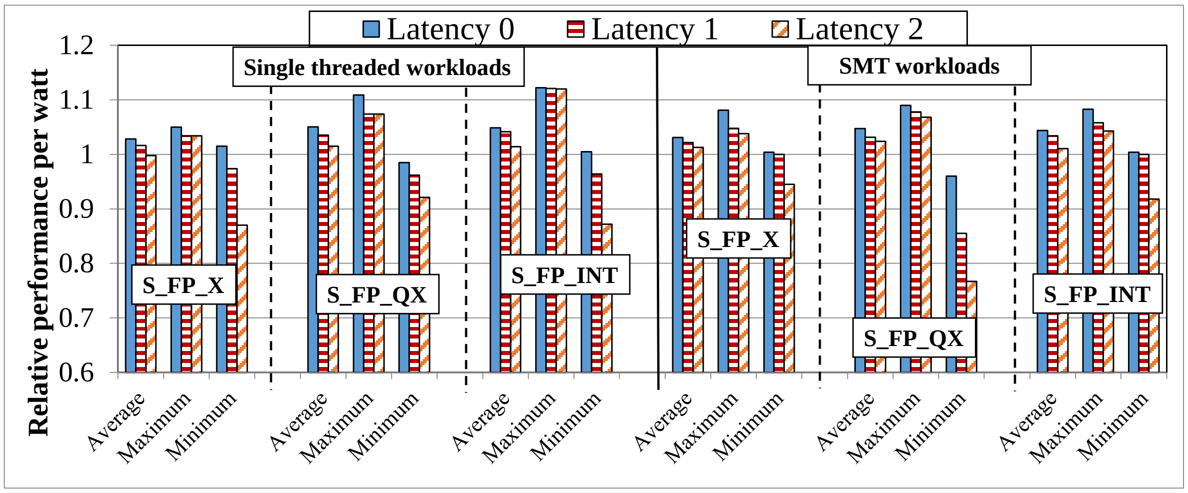


Figure 4.17. Relative performance/Watt of the Small core in S_FP_X, S_FP_QX and S_FP_INT configurations for various communication latencies when run using DFB. Results presented are summarized over all workloads for both the single threaded and SMT workloads.

shared resources results in a 2-3% drop in performance demonstrating the sensitivity of these architectures to the latency.

With respect to the SMT workloads, for all the three configurations, the workload *barnes+barnes_barnes+barnes* showed worst case relative performance of 0.78. This was boosted to 0.96 in all three configurations representing a 23% improvement in performance. On an average, performance was improved by 4%, 3% and 5% for the S_FP_X, S_FP_QX and S_FP_INT configurations, respectively, relative to the NM. These architectures also compare well against the baseline architecture. The S_FP_X, S_FP_QX and S_FP_INT configurations achieve performance of 1.01, 0.98 and 1.029, respectively, relative to the baseline.

From Figure 4.16 we note that the benefits of the DFB and DVFB mechanisms are very similar although they differ in the overhead to switch between operating modes (DFB requiring 10 cycles vs. 20 cycles for DVFB).

4.6.3.2 Performance/Watt analysis

The performance/Watt results are summarized in Figures 4.17 for the DFB scheme, and in 4.18 for the NM, DFB and DVFB schemes. Just as was the case with performance, the DFB scheme significantly improves the performance/Watt.

For the single threaded workloads, the worst case workload combinations for the S_FP_X and S_FP_INT configurations were *barnes_barnes* and *flops_fbench* with relative performance/Watt of 0.96. This loss was mitigated with a 5% improvement in performance/Watt in the DFB mode. For the S_FP_QX configuration, the workloads *barnes_barnes*, *cholesky_cholesky* and *flops_fbench* have a relative performance/Watt of around 0.98. Among these, the relative performance of *barnes_barnes* and *flops_fbench* improved to 1.039 and 1.07, respectively, while that of *cholesky_cholesky* was only improved to 0.985. Once again, stalls in the ISQ was the reason for this. Maximum improvements of 5%, 11% and 12% and average improvements of 3%, 5% and 4.5% were observed for the S_FP_X, S_FP_QX and S_FP_INT, respectively, over the baseline. The corresponding average improvements were 2%, 2% and 3% for the S_FP_X, S_FP_QX and S_FP_INT, respectively, over the NM.

For the SMT workloads, worst case relative performance/Watt of 0.91 was observed when running the workload *barnes+barnes_barnes+barnes* on both S_FP_X and S_FP_INT configurations in the NM. This was improved to 1.01 and 1.005, respectively, by the DFB scheme. The worst case for the S_FP_QX was a relative performance/Watt of 0.94 running the same workload in NM. This was improved to 1.039 by running in DFB. A maximum improvement of 8%, 9% and 8.3% and average improvement of 3.1%, 4.7% and 4.3% in performance/Watt were observed for the S_FP_X, S_FP_QX and S_FP_INT, respectively, over the baseline. This yields an average improvement of 2-3% in performance/Watt when compared to the NM.

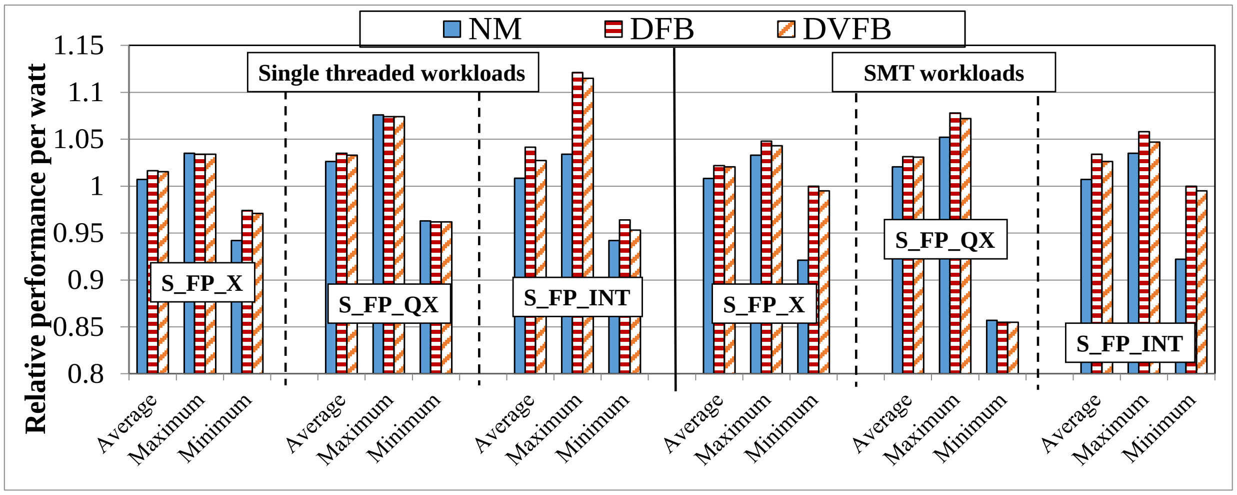


Figure 4.18. Relative performance/Watt of the Small core in S_FP_X, S_FP_QX and S_FP_INT configurations in NM, DFB and DVFB for communication latency of one cycle when run using DFB. Results presented are summarized over all workloads for both the single threaded and SMT workloads.

From Figure 4.18 we note that the benefits of the DFB and DVFB mechanisms are similar with DFB doing a little better (1-2%) since it does not incur the voltage regulator power overhead.

4.6.3.3 Percentage of execution time spent in the boosted modes

The boosted modes should not be used all the time. If this is the case, the processor was not properly sized and the results may be biased and misleading. In Figures 4.19 and 4.20 the percentage of time spent in the boosted mode in the DFB scheme is shown for the single and SMT workloads, respectively. Results are shown for all three sharing configurations for a shared resource communication latency of one cycle.

For the single threaded workload *flops_fbench*, all the three configurations run in the boosted mode for 100% of the time. This shows that for this workload, the shared execution unit was a severe bottleneck. Other workloads that were executed for most of the time (75-80%) in the boosted mode were *lu_lu* and *raytrace_raytrace* when run in the S_FP_INT configuration. These workloads resulted in contention

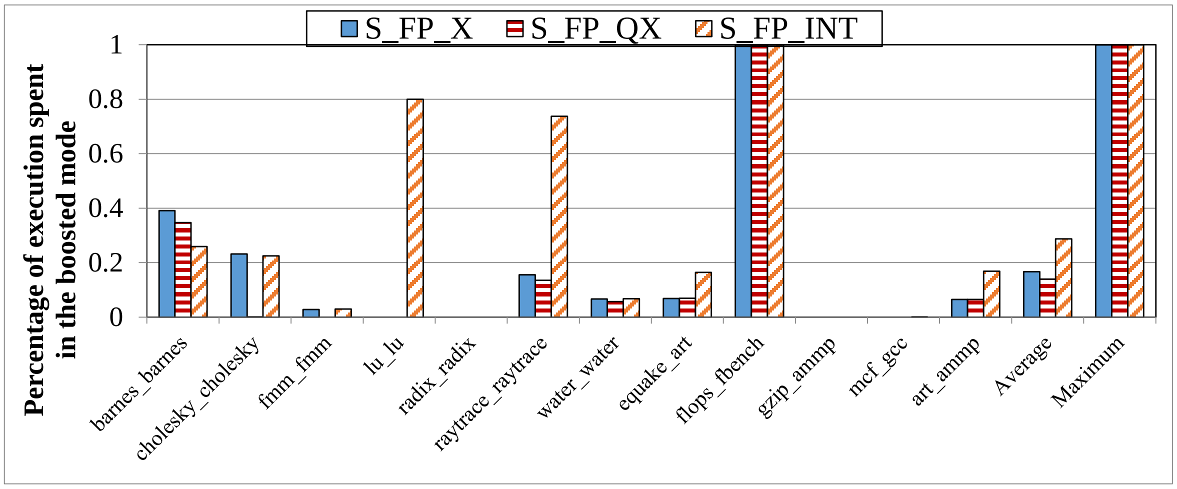


Figure 4.19. Proportion of total execution time spent in boosted mode for the S_FP_X, S_FP_QX and S_FP_INT configurations running single threaded workloads. The Small core was run using DFB and communication latency was set to one cycle. The average is also shown.

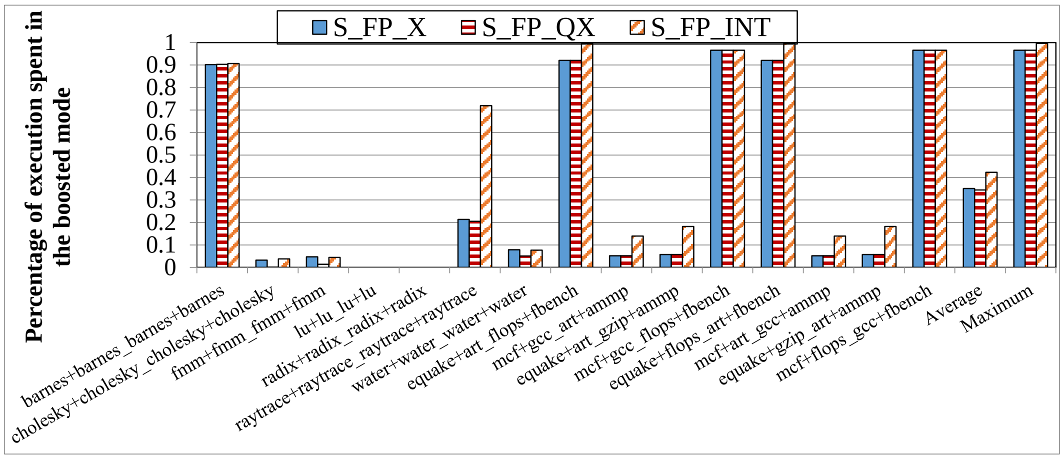


Figure 4.20. Proportion of total execution time spent in boosted mode for the S_FP_X, S_FP_QX and S_FP_INT configurations running SMT workloads. The Small core was run using DFB and communication latency was set to one cycle. The average is also shown.

for the integer multiply and divide operations. The DFB scheme detected this and accordingly operated in the boosted mode. The remaining 9 workloads operate in the boosted mode for 0-40% of the time. On an average, the Small core was operated in the boosted mode for 17-25% of the time for all the three configurations. Similar results were obtained for the DVFB mode.

For the SMT workloads, the number of workloads run in the boosted mode for all three configurations is higher. Nearly 6 of the considered 15 workload combinations run in the boosted mode for 70-100% of the time. In these experiments, contention for the shared resources is higher than that observed when running single threaded workloads. On an average, the Small core was operated in the boosted mode for 32-40% of the time and 9 of the 15 workloads operated in the boosted mode for less than 20%. of the time. These results show that while some workloads prefer to run in the boosted mode for longer duration than others, there are also several workloads for which the NM suffices indicating that the target architecture was sized appropriately.

4.7 Implementing the dynamic boosting mechanisms

The proposed DFB and DVFB schemes have shown significant potential to not only mitigate performance loss, but in some cases result in both performance and performance/Watt improvements over the baseline. However, implementing such mechanisms may result in hardware and performance overheads. We now discuss these overheads and present the resulting area overhead in the next sub-section.

4.7.1 Power overheads

With respect to power, a negligible power overhead is expected for DFB but for DVFB, power is lost during conversion. Assuming that the on-chip voltage regulator has a conversion efficiency of 90% [49], 10% of the power is wasted. We have found this power to be around 1% of the total power expended in the processor and is constitutes therefore, a very small overhead. This should be compared to the 12.5% power consumed by the execution units (measured during simulation) in conventional processors where no sharing takes place. Clearly, the overheads are far lower than the benefits provided by the boosting schemes.

4.7.2 Performance overheads

The dynamic boosting schemes affect a shift in voltage and/or frequency whenever deemed necessary. Two issues arise when employing such a dynamic control: (i) Lost cycles during the transition in voltage and/or frequency, and (ii) synchronization between the VFI's.

4.7.2.1 Cycles lost during operating mode transition

For the DFB scheme, only few cycles are lost during the frequency transition. IBM's PowerTune technology [64] generates multiple frequencies which are selected using multiplexers. The overhead to switch between frequencies was reported to be one cycle. Even if a separate PLL is used to generate the additional frequency, the overhead to transition between the two frequencies is not expected to be significant. We have pessimistically assumed an overhead of 10 cycles for the DFB mode. For the DVFB mode, in addition to frequency transition, a voltage transition is also needed. In [24], it is reported that the dV/dT for on-chip voltage regulators is around 20mV/ns. In our scheme, the cores transition between 1.1 and 1.35V. Hence, the time to transition between the two voltages is around 12.5ns. Considering that the Small core operates at 1.5GHz, the overhead in cycles for voltage transition is about 20 cycles. Note that during this period, the shared execution units are not accessible to avoid loss of signal integrity.

4.7.2.2 Synchronization between the VFI's

Since the VFIs may sometimes operate at different frequencies and/or voltages, this may lead to synchronization problems between the islands possibly leading to loss

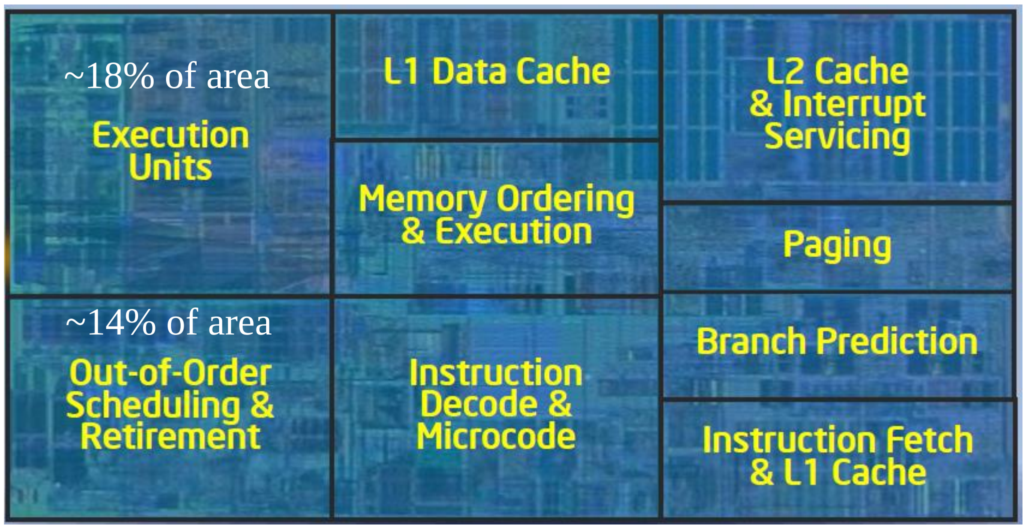

~18% of area
Execution Units
L1 Data Cache
L2 Cache & Interrupt Servicing
Memory Ordering & Execution
Paging
~14% of area
Out-of-Order Scheduling & Retirement
Instruction Decode & Microcode
Branch Prediction
Instruction Fetch & L1 Cache

Figure 4.21. Floorplan of the Intel Nehalem processor. Courtesy Andrew Semin, Intel Corporation. http://www.notur.no/notur2009/files/semin.pdf.

of cycles. Note that synchronization problems will be avoided if buffers are inserted at the boundary of the two VFI's. In all the considered designs, buffers are already present in the design (ISQ). Furthermore, by making use of certain types of FIFO buffers [85] any penalty due to synchronization can be completely avoided. Hence, in our experiments, we do not consider any overhead due to synchronization.

4.8 Area savings

In the target architecture, large and infrequently used resources are shared between cores. This certainly results in area savings. We present the estimated area savings using data available in literature as well as a tool that estimates core area.

4.8.1 Area savings based on literature

Kumar *et al.* in [53] report that the area savings of sharing just the FP units is around 6.1%. Hence, the S_FP_X configuration is expected to result in around 6-7% savings in area per core. In [88], Shivakumar *et al.* specify that the area occupied by the INT and FP execution units is approximately 12-13%. In Figure

4.21, the floorplan of the Intel Nehalem processor is shown[4]. The approximate area occupied by the execution units and the OOO scheduling logic (integer/FP ISQ and ROB) is also shown. The execution units occupy around 18% of the area of the core. Considering that ALUs account for a very small portion of the 18% occupied by the execution units, the S_FP_INT configuration is expected to yield around 8-9% savings in area per core. The OOO logic occupies 14% of the core area and assuming that half of that is occupied by the ROB and the other half by the integer and FP ISQ, the approximate area savings per core for the S_FP_QX configuration is around 9-10%.

4.8.2 Area savings as calculated by McPAT [63]

We have also estimated the area savings due to hardware resource sharing using McPAT [63] for a 45nm technology. This tool takes as input the dual-core configuration and outputs the estimated area for each block in the floorplan. The Small core was estimated to occupy 23mm^2 while the Big core around 35mm^2 excluding the L2 cache. The area of the baseline Big dual-core is thus estimated to be around 70mm^2 and that of the Small core around 46mm^2. For the Big core, floating-point execution units occupy 9.3mm^2, the integer divide and multiply units occupy 0.47mm^2 while the floating-point instruction window is reported to occupy 0.16mm^2. Thus for the Big dual-core the area savings of the S_FP_X, S_FP_QX and the S_FP_INT architectures is around 13.2%, 13.5% and 14%, respectively. In the Small core, the floating-point execution units occupy 4.6mm^2, the integer divide and multiply units occupy 0.47mm^2 while the floating-point instruction window is reported to occupy 0.14mm^2. Thus for the Small dual-core the area savings of the S_FP_X, S_FP_QX and the S_FP_INT architectures is around 10%, 10.3% and 11% respectively.

These savings in area are certainly expected to be considerably larger than the investment in real estate required for controlling access to the shared units.

[4]http://www.notur.no/notur2009/files/semin.pdf

4.8.3 Area and power estimation of the on-chip voltage regulator

Next, we estimate the area requirement for an on-die voltage converter. [34] reports an area of $0.008mm^2$ for an output power of 0.1 Watts in 90nm technology. We therefore, estimate an area of $0.16mm^2$ $(20X)$ for an on-die voltage converter with 2 Watts of output power. Considering that the die area of the Atom processor[5][6] is around 24-$26mm^2$, the area of the on-chip voltage regulator is negligible compared to the execution core area.

4.9 Conclusions

We have investigated the performance and performance/Watt of multicore processors that share infrequently accessed execution resources. Inspired by the AMD BullDozer architecture, we studied the impact of sharing the floating-point (FP) execution unit and issue queue between two cores in a dual-core processor. We then expanded the scope of the study by considering a Big core that is akin to Intel Nehalem processor and a Small core that is akin to Intel Atom processor. A variety of multi-programmed and multi-threaded workload combinations were studied in single-threaded and Simultaneously Multi-threaded (SMT) modes. We found that this architecture can sometimes result in large loss of performance ($\sim$ 28%). To mitigate this performance loss we limited the sharing to just the execution units including FP and integer divide and multiply units. This reduced the performance penalty to 14%. Sensitivity of the performance and performance/Watt of such architectures to shared resource access latency was also investigated. It was found that both performance and performance/Watt are highly sensitive to the communication latency. Our sensitivity study has further indicated that as long as the cores

[5]http://vsevteme.ru/attachments/show?content=7591

[6]http://ark.intel.com/products/35635/Intel-Atom-Processor-230-512K-Cache-1_60-GHz-533-MHz-FSB

share high throughput execution units, for most of the workloads, a small gain in performance/Watt is achieved at the expense of a small loss in performance. In order to mitigate such loss in performance, a dynamic voltage and frequency boosting (DVFB) scheme has been presented to accelerate execution in the shared resources. Such dynamic boosting was found to completely negate the performance losses and resulted in significant performance/Watt gains. The dynamic scheme improves the performance and performance/Watt of resource sharing architectures by as much as 22% and 10%, respectively. We also observed a performance and performance/Watt improvement of 13% and 14%, respectively, over non-sharing cores. Furthermore, the performance/Watt/area improves by as much as 26.2%, increasing the attractiveness of sharing.

CHAPTER 5

IMPROVING POWER EFFICIENCY WITHIN INDIVIDUAL CORES IN MULTICORES

We have seen that AMPs are more suited to cater to the diverse needs of workloads. Often, the explored AMPs employ two kinds of cores: out-of-order (OOO) big cores and in-order (InO) small cores [51, 77, 30]. . The big cores provide higher performance while the in-order small cores are more power efficient. As the benefits of such AMPs are highly dependent on a proper thread-to-core assignment, the threads are swapped between the cores at runtime so that the objective function (performance, performance/power, energy etc.) is improved for the current program phase.

However, thread swapping incurs non-negligible costs. The swapping overhead can vary from a few thousand [81] to millions of cycles [6, 50] depending on the algorithm employed to swap threads and the mechanism to exchange contexts. To amortize the large overhead associated with thread swapping, in most proposals, thread swapping decisions are made at the granularity of hundreds of thousands to millions of instructions [6, 50]. Unfortunately, numerous opportunities to improve performance/power and/or energy-delay-squared product (ED^2P) at a more fine grained instruction granularity are missed out by such approaches [65]. This point is illustrated in Figure 5.1 where the IPC resulting from running the workload *mcf* on the OOO and InO cores is shown. In the Figure, the IPC is sampled at coarse grain instruction granularities of 50K instructions. Here, it can be seen that at no point is the IPC of the InO core comparable to that of the OOO core. However, when considering a more finer instruction granularity of 500 instructions (inset), it can be seen that not only are the IPCs of the two cores comparable, but at some points in the plot, the InO core

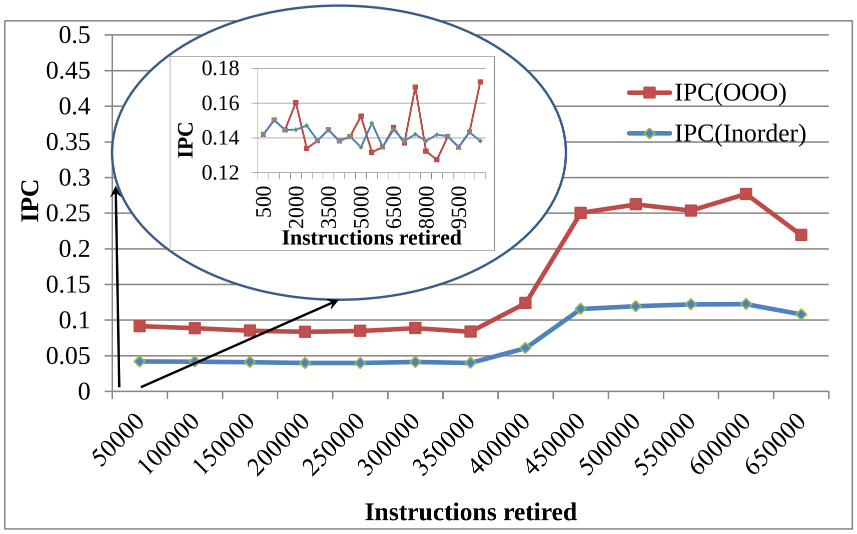


Figure 5.1. IPC comparison between the OOO and InO cores when executing the workload *mcf*. In the main figure, each point on the horizontal axis represents 50K retired instructions. In the inset figure, IPC for the the instructions from 0 - 10K have been sampled at 500 instructions.

outperforms the OOO core. The InO is the power efficient core and from the figure, it is clear, that at smaller instruction granularities, there is even more potential to make gains in performance/power by switching operation from OOO to the InO core. However, swapping threads at such a small granularity in current AMPs, will likely negate all benefits. Hence, there is need for a more fine grain switching mechanism that does not incur large thread swapping penalties. Therefore, there is need for a mechanism to realize these opportunities without incurring large thread swapping penalties.

In this chapter, we propose a novel core morphing mechanism that reaps most of the benefits of AMPs, without incurring the penalty associated with thread swapping. Our proposed mechanism introduces heterogeneity within the same core by morphing it from OOO to InO core and vice-versa. Certain Intel processors feature a special debug mode in which the OOO core turns into an InO core [50]. We extend this mechanism for energy efficiency by opportunistically switching to the InO mode, if deemed beneficial. As the morphing is performed within the same core and the archi-

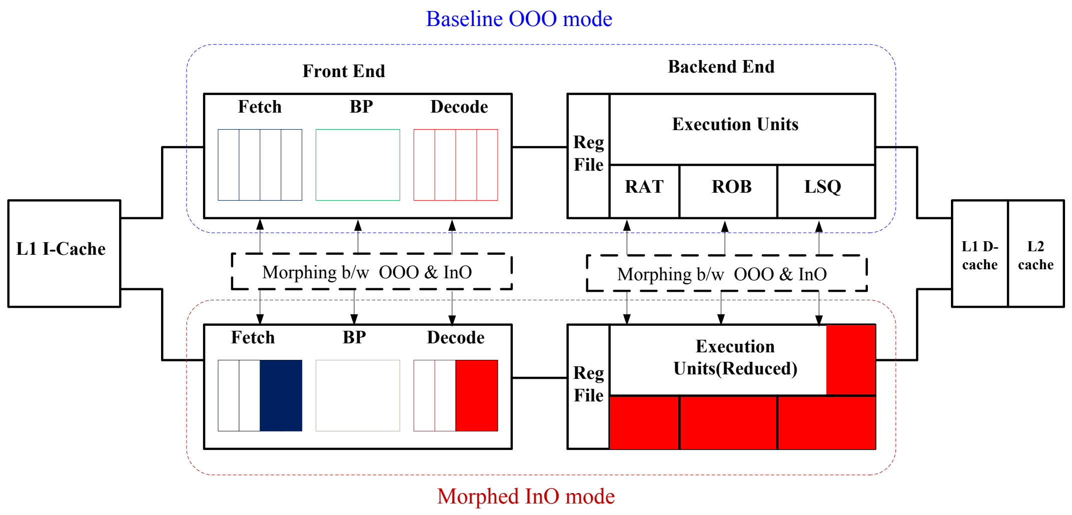


Figure 5.2. High-level view of the proposed core morphing scheme. The baseline OOO mode is shown at the top. The shaded regions indicate the units of the baseline core that are power-gated to facilitate in-order execution in InO mode.

tectural states are retained, the overheads associated with our scheme is negligible, thus enabling fine-grained switching between OOO and InO modes.

At a base level, we consider a single complex superscalar core. In the baseline mode, the core operates in the OOO mode providing high performance. However, during low IPC phases of the program, the operation mode may be switched to the InO mode for energy reduction. A similar switch is made from InO to OOO when these benefits are predicted to have diminished. To achieve energy benefits without impacting performance significantly, we employ energy-delay-squared product (ED^2P) as our optimization metric. The central idea of our proposal is the online estimation of the expected ED^2P of the executing thread in the other mode, while it is being executed in the current mode. Based on such an estimation, the mode that is expected to provide lower ED^2P for the current program phase is then chosen. The estimation is made possible by employing the performance monitoring counters (PMCs) of the baseline core.

Our results indicate that the proposed scheme achieves an ED^2P reduction of as much as 12% at a performance loss of less than 4% when compared to the baseline

OOO-core. Since the proposed scheme makes use of existing facilities in a processor, it has the advantage of being completely designed and verified in silicon and incurs no hardware overheads unlike several comparable schemes [65, 48, 101, 73, 47]. The key contributions of this chapter are:

1. Dynamic morphing within the same core between OOO and InO modes using existing debug mechanisms in current microprocessors.

2. Analyzing the trade-off between performance loss and energy savings when switching between OOO and InO modes of operation on the same core.

5.1 Related work

We now cover some of the recent advances made in literature that closely relate to the proposed architecture.

5.1.1 Morphable or dynamic multicores

There have been several proposals that advocate dynamic morphing of multicores or single cores such that performance and power efficiency is enhanced at run time. In a number of proposals, the starting point is a multicore consisting of small cores which then fuse together into a large OOO core on demand [48, 101, 73]. Such approaches suffer from additional latencies that arise from combining resources from various cores. A different scheme was adopted by Khubaib *et al.* in [47] where they start with a baseline OOO core that morphs itself into a Simultaneously Multithreaded InO core depending on the number of incoming threads. All such schemes require significant changes to the microarchitecture to be designed in practice.

Dynamic sharing of processor resources for power and performance benefits is also a well explored area. Kumar *et al.* [53] explore sharing of various large structures in the multicore for energy and area savings. In [81], we have explored dynamic exchange of execution units such that performance/Watt is improved. All such schemes require extra circuitry that must be designed and verified. In, [65], Lukefahr *et al.* make a proposal that is similar to ours. In their scheme heterogeneity is introduced into the same core by provisioning two execution backends to the same core. One backend is an OOO while the other is InO. Both backends share the same caches and fetch units. The difference between this scheme and ours is explained in detail in the next section.

5.2 Proposed Approach

In this section, we describe in detail both the architectural and implementation details of the proposed core morphing scheme that supports switching between OOO and InO modes at fine-grained time intervals.

5.2.1 Architectural Details

Figure 5.2 shows the considered baseline core which is a 4-way issue OOO superscalar core. The backend of the baseline core is provisioned with register alias table (RAT), load/store queue (LSQ) and Re-Order Buffer (ROB) to facilitate OOO execution and InO commit. The exact sizes of these resources are discussed in Section 5.3. During high-ILP program phases, significant performance benefits are achieved by executing the thread on the OOO baseline core. However, when the processor is waiting for long-latency memory operations to complete or stalls due to dependencies, most of the core resources are idle wasting static power.

For such low-IPC phases, a low-power InO core may be more energy efficient. In order to highlight the difference between the power consumption in the OOO and InO modes of operation, we analyzed the various components of the power spent for each mode of operation when executing the workload *equake*. The results are plotted in Figure 5.3. In general, it can be seen that the OOO mode consumes considerably more power than the InO mode. The OOO mode relies heavily on

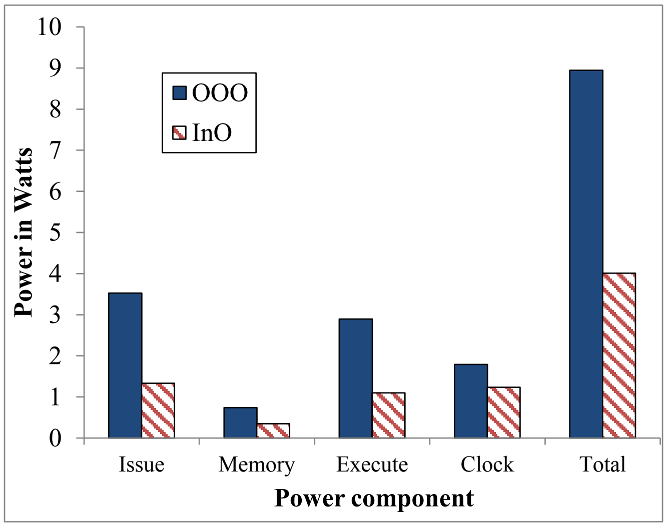


Figure 5.3. The components of the power expended when the workload *equake* is run in OOO and InO modes of operation.

speculative execution by making use of data structures such as the ROB and the reservation stations to ensure OOO execution but in-order commit. Data movements between these structures consume significant power. For some phases of a program, this increase may not commensurate with the performance benefits resulting in poor ED^2P. It can be seen in the figure that the issue and execution stage power for the OOO mode are significantly higher than the InO mode. These are the stages where the data structures are used and accessed the most. This result shows that the power expended in the OOO mode can be significantly higher than the InO mode. When such increase in power is not accompanied with a significant performance gain, a switch in mode from OOO to InO may be beneficial. To this end, during low-ILP/memory intensive phases, we power off the ROB, RAT, and LSQ, enabling only in-order execution/commit. Thus, the baseline OOO core is opportunistically morphed into an InO core providing significant power benefits. In this mode the baseline core supports only in-order execution and retirement of instructions. As the performance of the core in InO mode is expected to be low, we reduce the fetch width of the core from 4 to 2, and further, power off half of the decoders and, shut-down few of the multiple execution units. The InO mode (see Figure 5.2) is thus more power efficient than the baseline OOO mode. The configuration of the processor in the InO mode is discussed in Section 5.3. While in InO mode, if the program moves to a high-ILP phase, the shut down units are powered on, reverting back to the baseline OOO execution. This dynamic morphing of the core is facilitated by the existing debug capability in certain Intel processors that supports switching from OOO to InO modes [50].

Our proposed core morphing scheme is similar to the one proposed by Lukefahr *et al.* [65] but differs in the following ways. Firstly, Lukefahr *et al.* employ two different backend pipelines and decode units while our scheme uses the same for both modes (OOO and InO). The additional units increase the core area and design/verification effort. More importantly, the scheme proposed in [65] requires the architectural states to be transferred across the two pipelines which adds to the overhead. In contrast, the same register file is used by the two modes in our scheme. Finally, our scheme differs in when the mode switch (OOO to InO and vice versa) actually happens. Whenever the scheme decides to switch from OOO to InO mode, the ROB is power gated and the subsequent instructions are re-fetched in InO mode. Unlike [65], our scheme does not delay the OOO to InO mode switch until all the other speculative instructions are drained from the ROB. This way, we fully capitalize on the power benefits of moving to the InO mode while keeping the switching complexity and ROB power overhead at bay. When switching from InO to OOO mode, the ROB is powered back on and, the head and tail pointers of the ROB are re-initialized to point to the same slot. Thus, the ROB is presumed to be completely empty when the core is morphed back to the baseline OOO mode. The fact that we make use of existing facilities in the processor core to enable morphing makes our proposal much more practical and realizable.

Morphing from the OOO to InO modes of operation needs to be done at runtime. This requires a mechanism that makes dynamic decisions depending on the characteristics of the currently executing workload. A description of the mechanism employed in this work is presented next.

5.2.2 Implementation Details

Prior knowledge about the computational resource requirements of different applications is generally not available beforehand. Hence, there is a need for an online mechanism to characterize the time-varying program behavior and determine the appropriate mode (OOO or InO) at runtime such that ED^2P of the executing application is minimized. The proposed core morphing scheme accomplishes this task by estimating the expected ED^2P of the current execution phase of the application in both the modes (OOO and InO).

5.2.2.1 ED^2P prediction mechanism

The current characteristics of the application being executed on a core can reveal considerable information about how suitable the core is to that application. For example, an application phase that results in a significant number of misses in the level-1 cache will result in low performance and high energy consumption. Executing this phase on an InO core would make more sense with respect to ED^2P. In order to assess the current characteristics of the application being executed, we make use of Performance Monitoring Counters (PMC).

In order to estimate the ED^2P, both performance and energy (power) need to be measured or estimated. Performance measurement is straightforward, while real time power or energy measurement is not. PMCs have been used as a proxy to estimate power in the past [19, 92] and we follow a similar approach. Note that most previous work makes use of PMCs to estimate power on the same core while we need to estimate power and performance on the currently active mode (OOO/InO), as well as the other mode (InO/OOO) to make an informed decision.

5.2.2.1.1 PMCs explored in this study There are many events that take place in a modern processor but some of them provide better hints than others about the performance and power of the currently executing application. To this end, we have explored fourteen different performance counters. We considered (i) the number of retired instructions of each type (integer, floating-point etc.), (ii) memory hit and miss counters (level-1, level-2 and TLB misses), (iii) number of mis-predicted and correctly predicted branch instructions, (iv) number of instructions fetched and instructions retired per cycle (IPC), and (v) pipeline stall counters which consist of stalls resulting due to lack of reservation station, load/store queue, RAT and ROB slots.

5.2.2.1.2 Shortlisting the PMCs In general, we expect a higher estimation accuracy using large number of counters. However, there is a limit on the number of counters that may be accessed at the same time. This limit varies from one architecture to another. For example, in the Intel XScale processor [19], only two counters may be accessed while for the AMD Phenom processor, at most five counters may be accessed at the same time [92]. There is, therefore, a need to find a minimal subset of PMCs that have the most impact on power and performance both in the currently active mode, and the other.

To accomplish the task of making the right choice of PMCs, we devised an efficient heurestic that searches the counter space iteratively. During each iteration, our counter selection algorithm picks a new counter that best fits the estimating parameter (performance or power) along with the set of counters already chosen in previous iterations. We tried only linear models for curve-fitting and the best fit is qualified by the R^2 coefficient. During the initial few iterations, the value of the R^2 coefficient increases steeply as more counters are added, but it tends to saturate later. The best set of counters is around the region where the R^2 coefficient tends to saturate.

The result of one such counter selection experiment is shown in Figure 5.4. Here, the expected performance of the application in InO mode is estimated using the

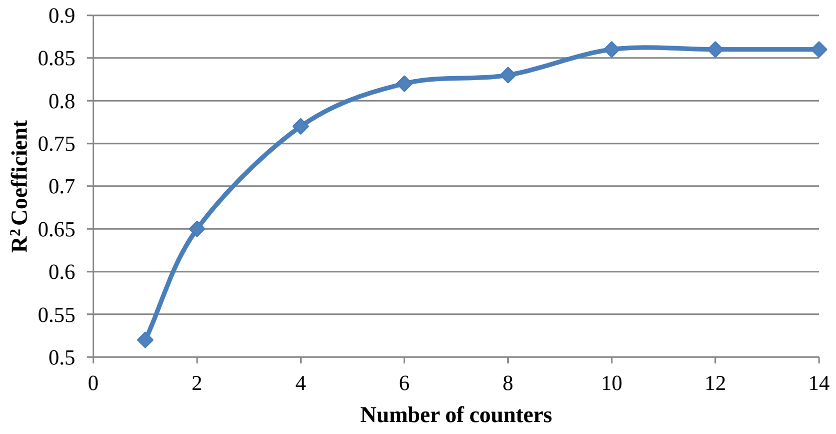


Figure 5.4. Variation in R^2 coefficient while estimating the performance in InO mode using the values of PMCs observed in OOO mode.

values of PMCs observed in the OOO mode. As expected, increasing the number of counters yields better R^2. However, we arrive to the point of diminishing returns after 6 counters. These 6 counters were IPC, number of retired load and store instructions, pipeline stalls, branch mis-predictions and level-1 cache hit rate. Similar experiments were run to obtain expressions that can be used to estimate both performance and power on both modes using PMCs. The final expressions obtained are shown in Table 5.1.

The average error observed when using PMCs on one mode (OOO/InO) to predict power in that mode as well as performance and power on the other mode (InO/OOO) is show in Figure 5.5. While estimating the OOO parameters (IPC and power) from the InO mode using PMCs in the InO mode, average % error in estimating IPC and power is around 16% and 10% respectively. Similarly average % error in computing the InO parameters from OOO core was found to be 15% and 8% respectively. Error in estimating power using counters in the same mode was found to be around 9% for the OOO mode and 8% for the InO mode. Using the estimated power and performance values, ED^2P for both modes is then computed using PMCs from the currently

Table 5.1. Power and performance estimation of the other mode using the performance counters' values in the current mode. *L1h - L1 Hit, Bmp- branch miss prediction, S - Store, L- Load, DS- Dispatch Stall*

Estimating Parameter	Expression
InO $\Rightarrow$ OOO IPC	$4.5 \times 10^{-3} \times$ L1h + $4.417 \times$ IPC - $0.0273 \times$ Bmp - 2.3255
InO $\Rightarrow$ OOO Power	$0.080 \times$ L1h + $71.15 \times$ IPC - $0.4112 \times$ Bmp - 38.46
InO $\Rightarrow$ InO Power	$0.0047 \times$ L1h + $13.062 \times$ IPC - $0.0069 \times$ S - $7.4 \times 10^{-5} \times$ DS + 1.5547
OOO $\Rightarrow$ InO IPC	- $0.00616 \times$ L1h + $0.06671 \times$ IPC - $4.2 \times 10^{-4} \times$ Bmp - $7.5 \times 10^{-5} \times$ DS + 0.2768
OOO $\Rightarrow$ InO Power	$-0.0039 \times$ L1h+ $0.9022 \times$ IPC + $0.0104 \times$ S - $0.0103 \times$ Bmp + 4.4669
OOO $\Rightarrow$ OOO Power	$0.0141 \times$ L1h + $13.81 \times$ IPC + $0.0295 \times$ S - $0.0118 \times$ Bmp - 0.2989

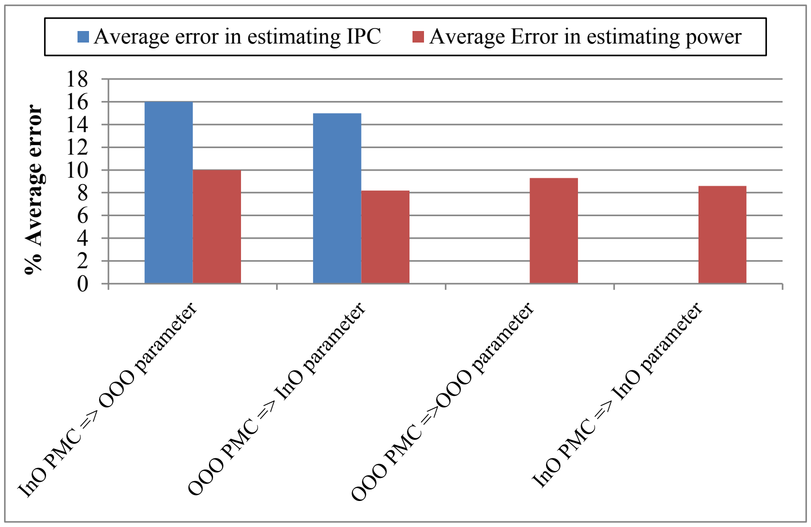


Figure 5.5. % Average error observed in estimating IPC and power of OOO (InO) mode using InO (OOO) counters.

operating mode. The average error in ED^2P estimation for both modes was found to be around 20%, reflecting resonable accuracy of our prediction mechanism.

We have seen how ED^2P can be estimated using PMCs in the proposed scheme. The decision to move to the alternate mode of operation is the one that provides best ED^2P. The decision to switch mode of operation should be one of high confidence. Otherwise, we risk running into oscillations between the two modes. This will likely negate all benefits of the proposed scheme. Hence, it is necessary to ensure that the decision to change operation mode is effected only if it expected to be long term. Determining the confidence of a decision is described next.

5.2.2.2 Capturing Application Phase behaviour

After certain number of retired instructions, referred to as *window*, a tentative morphing decision about the best mode (OOO or InO) is made based on the above ED^2P estimations. To avoid too frequent switching between the modes (InO and OOO), we prefer to wait until the new execution phase of the thread has stabilized. To that end, we base our morphing decision on the most frequent tentative decision made for the past n retired instructions (n (history depth) = *integer* $\times$ window length). For example, if for the past n committed instructions, moving from OOO to InO mode was the frequent decision, it may be predicted that the application has entered a phase where InO mode may provide lower ED^2P. The window size and history depth need to be determined experimentally. We have conducted a sensitivity study to quantify the impact of window length and history depth (indicated by n) on the achieved benefits. The window size and history depth combination that yields the lowest ED^2P for entire program execution would be the best choice.

The window length was varied from 250 to 1000 instructions. Within a particular window, the history depth (n) was varied from 2000, 3000, 4000 and 5000. For example, a history depth (n) of 2000 indicates that for a particular window, we

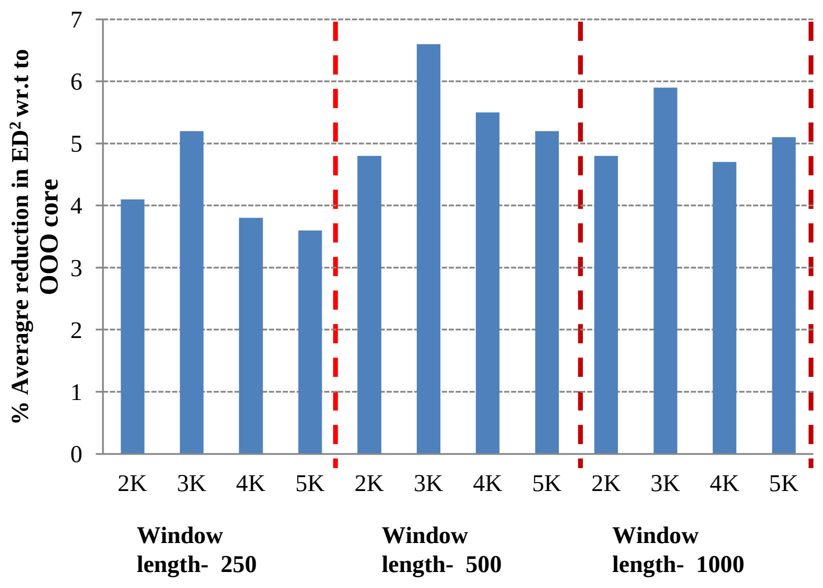


Figure 5.6. % Average reduction in ED^2P of the proposed scheme w.r.t the baseline OOO core for different values of window length and history depth.

make a reconfiguration decision at the end of every 2000 instructions. To determine the optimimum window size and the history depth, we ran a set of 10 benchmarks. After each benchmark was run for 1 billion instructions (after skipping the initial 5 billion instructions), we computed the average reduction in ED^2P of the proposed scheme (that can switch between OOO and InO modes) over the baseline OOO core. The decision to switch between operation modes is determined by the most frequent decision made within the history depth. As shown in Figure 5.6, window length of 500 and history depth of 3K provides the maximum reduction in ED^2P. The above computation of ED^2P reduction takes into account the overhead for switching between modes, as explained in the next section. Thus, in all our future experiments, the window length of 500 and history depth of 3K is used.

5.2.2.3 Switching between OOO and InO modes

Due to the low overhead associated with our morphing scheme, we dynamically morph from one mode to another at a fine-grained instruction granularity. As mentioned earlier, InO mode with reduced architectural units provides better energy efficiency at the cost of performance. It is critical that we move into InO mode only when we expect increased energy benefits without compromising performance significantly. To minimize the performance loss encountered while running in this dynamic configuration (OOO + InO modes), we use ED^2P as the switching metric which assigns higher weight to performance than energy.

At the end of every history depth, we decide to move to InO mode only if the expected ED^2P in InO mode is less than that of the OOO mode by a defined threshold. This defined threshold is referred to as ED^2P threshold (see Section 5.3). Based on the window length and history depth determined previously, the proposed scheme decides the best mode of operation (OOO or InO) to execute the current application phase considering the recent tentative decisions (made for each window). Once the decision to switch to InO mode is made, the ROB, LSQ and the RAT units are powered off and the subsequent instructions are re-fetched for in-order execution.

5.2.2.4 Morphing overheads

Previously proposed schemes for morphing [81, 65] or swapping of threads between asymmetric cores [51, 77, 6] incur large overhead and as a result, thread swapping or morphing were done at a very coarse grain granularity. The overheads for these schemes arise from the transfer of architectural state requiring a warm up the cache and the branch predictor [77] or due to a high communication latency to send or receive data operands [76]. In our proposed scheme, morphing is done within the core and thus it avoids all the above overheads as there is no need to change the state of the register file, caches and branch predictors. The overhead associated with our scheme is due to the power gating/power up of the ROB, RAT and LSQ units and partial power on/off of fetch, decode and execution units while switching between OOO and modes. When power-gating individual units, there is no dynamic energy consumed and the static energy consumed by these idle units is not very high providing us with

Table 5.2. Baseline OOO core parameters considered. The values in parenthesis represent the change while in InO mode.

Param	Value	Param	Value
Issue	4 (2)	INTREG	96 (NA)
FPREG	80 (NA)	INTISQ	36 (NA)
FPISQ	24 (NA)	LS units	3 (1)
LSQ	32 (NA)	ROB	128 (NA)
L1(I/D)	32K	L2	2M
Freq (GHz)	2.4	Type	OOO (InO)

Table 5.3. Execution unit specifications for the baseline core. (P - Pipelined, NP - Not pipelined, PP - Partially pipelined).The values within parenthesis represent the change while in InO mode

FP DIV	FP MUL	FP ALU
1 unit, 21 cyc, P	1 unit, 5 cyc, P	2 (1) units, 3 cyc, P
INT DIV	INT MUL	INT ALU
1 unit, 23 cyc, P	1 unit, 8 cyc, P	4 (2) units, 1 cyc, P

increased power savings. Power gating/power-on of all the blocks simultaneously may lead to a sudden power surge and thus we employ staggered power gating where one block is gated every clock cycle and thus requiring 6 clocks to gate the 6 blocks. Thus, the total overhead when switching between modes is assumed to be 20 clock cycle with additional margin (14 clock cycles) for every switch. The switch between OOO and InO modes is handled in hardware and no changes are required to the operating system.

5.3 Results and Analysis

In this section, we evaluate our proposed core morphing scheme. The core parameters considered in this work are listed in Tables 5.2 and 5.3. Most of these parameters were taken from [26]. As shown in Table 5.2, the OOO core is provisioned with large resources (e.g., integer and floating-point registers, issue queues and L2 cache) which is representative of modern superscalar processors. The changes to the architectural parameters and the execution units in the InO mode are shown in parenthesis in Tables 5.2 and 5.3, respectively. We used SESC [75] as our architectural simulator and employed Wattch [13] and CACTI [89] to measure power. The evaluation was

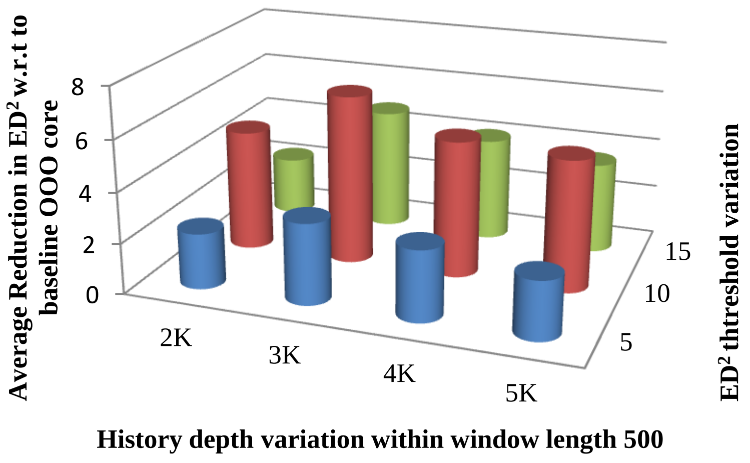


Figure 5.7. % Reduction in ED^2P vs threshold variation for various history depths.

carried out using 10 benchmarks from SPEC [97] and Mediabench suites [57]. Each of the benchmarks were run for 1 billion instructions after skipping the first 5 billion instructions.

5.3.1 Trade-off analysis between energy savings and performance loss

The decision to switch from one configuration mode to another is based on the ED^2P threshold. We now explain the process of determining the ED^2P threshold. The ED^2P threshold was varied from 5% to 15% for different history depths and window lengths. By analyzing the benefits with respect to ED^2P for various combinations of window lengths and history depths over all benchmarks, it was determined that the window length of 500 provided the maximum benefits for all the thresholds. In figure 5.7 we analyze the ED^2P reduction for window length of 500 for various thresholds and history depths. It can be seen that going from ED^2P threshold of 5% to 10% , the average reduction in ED^2P when compared to the baseline OOO core increases by 2-3%. As we further increase the ED^2P threshold, the benefits tend to drop by 1-2%. The reason for improved benefits when the threshold is changed from 5% to 10% is that for low threshold values the number of reconfiguration increases

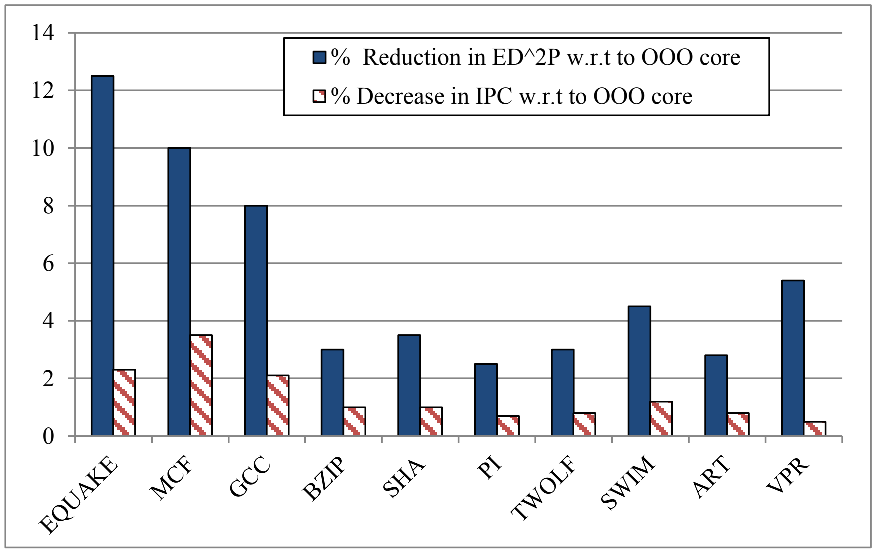


Figure 5.8. % Reduction in ED^2P of proposed scheme w.r.t the baseline OOO core.

significantly. Thus, for a threshold value of 10%, the number of reconfigurations tend to stabilize and for threshold values beyond 10%, there is reduced benefit due to very few reconfigurations. The ED^2P threshold was thus set to 10%.

Having determined the ED^2P threshold, history depth and window length, we are now ready to determine the reduction in ED^2P that can be achieved using the proposed scheme over the baseline OOO core. The results obtained with respect to ED^2P reduction and performance loss is shown in Figure 5.8. Benchmarks such as EQUAKE, MCF, GCC are observed to achieve substantial reductions in ED^2P of 13%, 10% and 8%, respectively, with other benchmarks achieving an ED^2P benefit of within 5%. Since the ED^2P threshold trades more weight for performance than energy, the performance loss of all the benchmarks over the entire run is found to be less than 4%.

So far in the proposed scheme, we assigned priority to performance loss over energy savings and hence considered the ED^2P metric. An alternatve approach would be to prioritize energy savings. This is important in case of laptops and cell phones where the end user is willing to tolerate more performance loss for increased energy

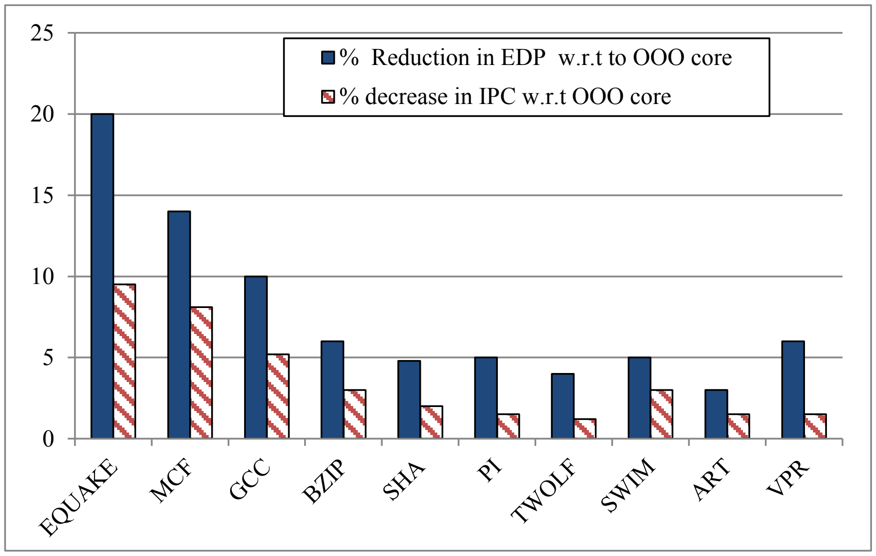


Figure 5.9. % Reduction in EDP of proposed scheme w.r.t the baseline OOO core.
Table 5.4. number of switches per million instructions and percentage time spent by benchmarks in morphed mode

Benchmark	Switches/million instruction	Percentage time spent in Morphed mode
mcf	450	76
equake	250	83
sha	420	30
pi	130	20
swim	180	26
art	80	18
gcc	140	44
bzip	120	23
twolf	180	21
vpr	210	30

savings. For such cases, we propose to use the Energy-Delay Product EDP since this metric gives equal weight to both performance and energy. Results obtained when using this metric is shown in Figure 5.9. The % reduction in EDP when compared to the baseline OOO core was found to be 20% for EQUAKE, 14% for MCF and 10% for GCC with performance loss of 8%, 7% and 5%, respectively, while other benchmarks achieving EDP benefits of up to 5% with a small performance penalty. Thus, the EDP metric provides increased energy savings at the cost of performance when compared to the ED^2P metric.

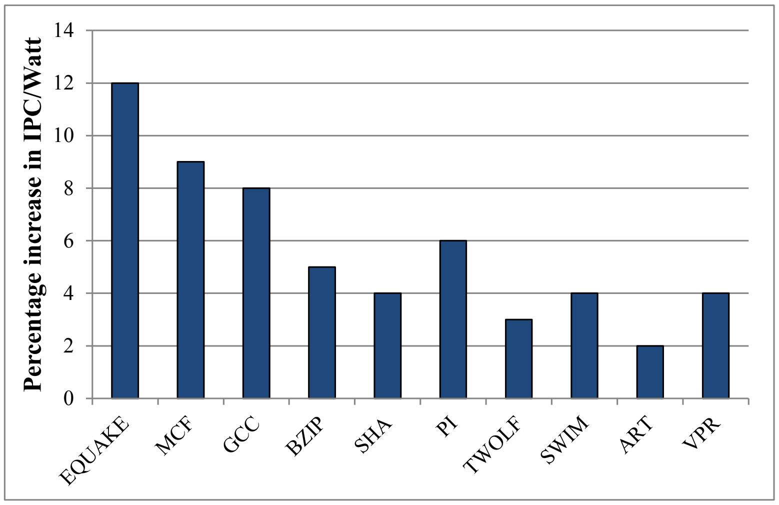


Figure 5.10. % increase in IPC/Watt of proposed scheme w.r.t to the baseline OOO core.

5.3.2 Number of switches and time spent in InO mode

The number of mode switches for the chosen window length and history depth is shown in Table 5.4 for all the benchmarks. As expected, benchmarks which achieve increased energy savings exhibit higher number of switches in mode. But, due to sharing of architectural resources by both the modes, the switching overhead is negligible for our scheme. The overall performance impact due to switching overhead was found to be less than 1%.

Table 5.4 also shows the percentage of time spent by each benchmark in the InO mode. Benchmarks such as EQUAKE, MCF and GCC that achieve increased energy savings tend to stay in the InO mode for longer time periods as they have many low-IPC phases. MCF and EQUAKE spend significantly more time in the InO mode (70% and 78%, respectively). We observe that low IPC phases of application tend to use the InO mode more frequently leading to increased energy savings.

5.3.3 Benefits of core morphing in terms of performance/Watt

Figure 5.10 shows the performance/Watt benefit obtained by the proposed morphing scheme. The ED^2P metric was used here to determine mode switching. Memory Intensive benchmarks such as EQUAKE, MCF and GCC again show a higher

IPC/watt benefit of about 12%, 9% and 8%, respectively, while other benchmarks show an IPC/Watt gain of up to 6%. It is to be noted that the window length, history depth and ED^2P threshold were kept the same even for these experiments. The mentioned IPC/watt benefits were obtained at an average performance loss of about 4.2%.

5.4 Conclusions

Applications experience a change in characteristics over time. Hence, a different core configuration (size of the ROB, number of execution units etc.) may be more suitable with respect to energy and performance at different time instants. Traditionally, Asymmetric Multicores (AMP) have been considered to support the diverse needs of applications. Here, depending on current application characteristics, threads are swapped between the available cores in the AMP such that the objective function (energy, performance etc.) is optimized. Prohibitive thread migration overheads limit the instruction granularity at which such thread swapping decisions may be made, even though many opportunities present themselves at fine grain granularities. In this chapter, we have considered an architecture that is capable of realizing these benefits. Here, depending on application characteristics, a superscalar OOO processor may morph itself into an in-order (InO) core at runtime, if deemed to be beneficial. Such morphing is made possible by using the existing debug feature present in certain Intel processors. The decision to morph between operation modes (OOO/InO) is made using information gathered from performance monitoring counters. The proposed scheme opportunistically morphs into InO mode to minimize Energy Delay square Product. Our results indicate that ED^2P reduction of up to 12% is possible at a very small performance penalty of less than 4%. This result compares favorably against similar proposals that impose far more hardware overhead.

CHAPTER 6
FUTURE DIRECTIONS

In this this dissertation, we have explored a few mechanisms for online management of resilient and power efficient multicore processors. The future work based on this dissertation is now presented.

6.1 Error resilient processors

There are several extensions that are planned for the Sentry Core (SC) architecture.

1. SC for fault diagnosis in the multicore.
2. SC to detect errors in operation of directory coherence.

6.2 Power efficient processors

The following is the future work planned for power efficient computing.

6.2.1 Thread scheduling in AMPs

1. Improving the accuracy of estimation based scheduling.
2. Development of schemes to estimate performance and power on the host and other cores in the presence of DVFS.

6.2.2 Resource sharing in multicores

1. Explore the effect of resource sharing in AMPs. As opposed to SMPs, such sharing may have potential to boost both performance and performance-per-Watt if the small cores in AMPs are given access to the underutilized big core resources.

2. Explore sharing of other structures such as the ROB, reservation stations etc.

3. Explore sharing of structures in 3D architectures.

6.2.3 Polymorphic processors

So far, we have only considered morphing all the way from an OOO core to an InO core. In the future, we plan to explore if there exists a middle ground operating mode between the OOO and InO modes of operation. We will also consider core morphing in the presence of DVFS.

BIBLIOGRAPHY

[1] Aater Suleman, M., Mutlu, O., Qureshi, M.K., and Patt, Y.N. Accelerating critical section execution with asymmetric multicore architectures. *Micro, IEEE 30*, 1 (jan.-feb. 2010), 60 –70.

[2] Abella, J., Vera, X., Unsal, O., Ergin, O., and Gonzalez, A. Fuse: A technique to anticipate failures due to degradation in alus. In *On-Line Testing Symposium, 2007. IOLTS 07. 13th IEEE International* (july 2007), pp. 15 –22.

[3] Austin, T.M. DIVA: a reliable substrate for deep submicron microarchitecture design. In *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on* (1999), pp. 196 –207.

[4] Balakrishnan, Saisanthosh, Rajwar, Ravi, Upton, Mike, and Lai, Konrad. The impact of performance asymmetry in emerging multicore architectures. In *Proceedings of the 32nd annual international symposium on Computer Architecture* (Washington, DC, USA, 2005), ISCA '05, IEEE Computer Society, pp. 506–517.

[5] Baumann, R.C. Radiation-induced soft errors in advanced semiconductor technologies. *Device and Materials Reliability, IEEE Transactions on 5*, 3 (sept. 2005), 305 – 316.

[6] Becchi, Michela, and Crowley, Patrick. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Proceedings of the 3rd conference on Computing frontiers* (2006), CF '06.

[7] Benso, A., Di Carlo, S., Di Natale, G., and Prinetto, P. A watchdog processor to detect data and control flow errors. In *On-Line Testing Symposium, 2003. IOLTS 2003. 9th IEEE* (july 2003), pp. 144 – 148.

[8] Borkar, S. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *Micro, IEEE 25*, 6 (nov.-dec. 2005), 10 – 16.

[9] Borkar, Shekhar, and Chien, Andrew A. The future of microprocessors. *Commun. ACM 54*, 5 (May 2011), 67–77.

[10] Borodin, D., et al. Functional unit sharing between stacked processors in 3d integrated systems. In *Embedded Computer Systems (SAMOS), 2011 International Conference on* (july 2011), pp. 311 –317.

[11] Borodin, D., and Juurlink, B. H H. A low-cost cache coherence verification method for snooping systems. In *Digital System Design Architectures, Methods and Tools, 2008. DSD '08. 11th EUROMICRO Conference on* (2008), pp. 219–227.

[12] Bower, F.A., Sorin, D.J., and Ozev, S. A mechanism for online diagnosis of hard faults in microprocessors. In *Microarchitecture, 2005. MICRO-38. Proceedings. 38th Annual IEEE/ACM International Symposium on* (2005).

[13] Brooks, D., Tiwari, V., and Martonosi, M. Wattch: a framework for architectural-level power analysis and optimizations. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on* (june 2000).

[14] Butler, M., Barnes, L., Sarma, D.D., and Gelinas, B. Bulldozer: An approach to multithreaded compute performance. vol. 31, pp. 6–15.

[15] Cakarevic, V., et al. Characterizing the resource-sharing levels in the ultrasparc t2 processor. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on* (2009), pp. 481–492.

[16] Cantin, Jason F., Lipasti, Mikko H., and Smith, James E. Dynamic verification of cache coherence protocols, 2001.

[17] Carretero, J., Chaparro, P., Vera, X., Abella, J., and Gonzalez, A. Implementing end-to-end register data-flow continuous self-test. *Computers, IEEE Transactions on 60*, 8 (aug. 2011), 1194 –1206.

[18] Chen, Jian, and John, Lizy K. Efficient program scheduling for heterogeneous multi-core processors. In *Proceedings of the 46th Annual Design Automation Conference* (2009), DAC '09.

[19] Contreras, Gilberto, and Martonosi, Margaret. Power prediction for intel xscale processors using performance monitoring unit events. In *Proceedings of the 2005 international symposium on Low power electronics and design* (New York, NY, USA, 2005), ISLPED '05, ACM, pp. 221–226.

[20] Dao, Tuan Q. (Richardson, TX) Steiss Donald E. (Richardson TX). Shared floating-point unit in a single chip multiprocessor, November 2000.

[21] DeOrio, A., Bauserman, A., and Bertacco, V. Post-silicon verification for cache coherence. In *Computer Design, 2008. ICCD 2008. IEEE International Conference on* (2008), pp. 348–355.

[22] Dolbeau, Romain, and Seznec, Andr. Cash: Revisiting hardware sharing in single-chip parallel processor. Tech. rep., 2002.

[23] Ernst, Dan, et al. Razor: A low-power pipeline based on circuit-level timing speculation. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2003), MICRO 36, IEEE Computer Society, pp. 7–.

[24] Eyerman, Stijn, and Eeckhout, Lieven. Fine-grained dvfs using on-chip regulators. *ACM Trans. Archit. Code Optim. 8*, 1 (Feb. 2011), 1:1–1:24.

[25] Fernandez-Pascual, R., Garcia, J.M., Acacio, M.E., and Duato, J. A low overhead fault tolerant coherence protocol for cmp architectures. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on* (2007), pp. 157–168.

[26] Fog, A. The microarchitecture of intel, amd and via cpu. Tech. rep., Copenhagen University College of Engineering, 2012.

[27] Garg, S., et al. Technology-driven limits on dvfs controllability of multiple voltage-frequency island designs: A system-level perspective. In *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE* (july 2009), pp. 818 –821.

[28] Ghiasi, Soraya, and Grunwald, Dirk. Aide de camp: Asymmetric dual core design for power and energy reduction. Tech. rep., IEEE Trans. Inform. Theory, 2003.

[29] Ghosh, Swaroop, et al. Voltage scalable high-speed robust hybrid arithmetic units using adaptive clocking. *IEEE Trans. Very Large Scale Integr. Syst. 18*, 9 (Sept. 2010), 1301–1309.

[30] Greenhalgh, P. Big.little processing with arm cortex-a15 and cortex-a7, sep. 2011.

[31] Grochowski, E., Ronen, R., Shen, J., and Wang, Hong. Best of both latency and throughput. In *Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings. IEEE International Conference on* (oct. 2004).

[32] Gupta, S., et al. The stagenet fabric for constructing resilient multicore systems. In *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on* (nov. 2008), pp. 141 –151.

[33] Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., and Brown, R.B. Mibench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on* (dec. 2001).

[34] Hazucha, P., Karnik, T., Bloechel, B.A., Parsons, C., Finan, D., and Borkar, S. Area-efficient linear regulator with ultra-fast load regulation. *Solid-State Circuits, IEEE Journal of 40*, 4 (2005), 933–940.

[35] Held, Jim, Bautista, Jerry, and Koehl, Sean. White paper from a few cores to many: A tera-scale computing research review, 2006.

[36] Heller, L. C., and Farrell, M. S. Millicode in an ibm zseries processor. *IBM Journal of Research and Development 48*, 3.4 (may 2004), 425 –434.

[37] Hennessy, J.;, and Patterson, D. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2003.

[38] Hill, M.D., and Marty, M.R. Amdahl's law in the multicore era. *Computer 41*, 7 (july 2008), 33 –38.

[39] Homayoun, Houman, et al. Dynamically heterogeneous cores through 3d resource pooling. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture* (2012), HPCA '12, pp. 1–12.

[40] Horowitz, Mark. Scaling, power and the future of cmos. In *Proceedings of the 20th International Conference on VLSI Design held jointly with 6th International Conference: Embedded Systems* (Washington, DC, USA, 2007), VLSID '07, IEEE Computer Society, pp. 23–.

[41] Ipek, Engin, Kirman, Meyrem, Kirman, Nevin, and Martinez, Jose F. Core fusion: accommodating software diversity in chip multiprocessors. vol. 35, ACM, pp. 186–197.

[42] Jang, Wooyoung, et al. Voltage and frequency island optimizations for many-core/networks-on-chip designs. In *Green Circuits and Systems (ICGCS), 2010 International Conference on* (june 2010), pp. 217 –220.

[43] Joseph, Russ, and Martonosi, Margaret. Run-time power estimation in high performance microprocessors. In *Proceedings of the 2001 international symposium on Low power electronics and design* (New York, NY, USA, 2001), ISLPED '01, ACM, pp. 135–140.

[44] Kahle, James Allan (Austin, TX) Moore Charles Roberts (Austin TX). Shared execution unit in a dual core processor, April 2004.

[45] Khan, Omer, and Kundu, Sandip. A self-adaptive scheduler for asymmetric multi-cores. In *Proceedings of the 20th symposium on Great lakes symposium on VLSI* (2010), GLSVLSI '10.

[46] Khan, Omer, and Kundu, Sandip. Microvisor: A runtime architecture for thermal management in chip multiprocessors. *T. HiPEAC 4* (2011), 84–110.

[47] Khubaib, Suleman, M. Aater, Hashemi, Milad, Wilkerson, Chris, and Patt, Yale N. Morphcore: An energy-efficient microarchitecture for high performance ilp and high throughput tlp. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2012), MICRO '12, IEEE Computer Society, pp. 305–316.

[48] Kim, Changkyu, Sethumadhavan, Simha, Govindan, M. S., Ranganathan, Nitya, Gulati, Divya, Burger, Doug, and Keckler, Stephen W. Composable lightweight processors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2007), MICRO 40, IEEE Computer Society, pp. 381–394.

[49] Kim, Wonyoung, Gupta, M.S., Wei, Gu-Yeon, and Brooks, D. System level analysis of fast, per-core dvfs using on-chip switching regulators. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on* (2008), pp. 123–134.

[50] Koufaty, David, Reddy, Dheeraj, and Hahn, Scott. Bias scheduling in heterogeneous multi-core architectures. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10.

[51] Kumar, R., Farkas, K.I., Jouppi, N.P., Ranganathan, P., and Tullsen, D.M. Single-isa heterogeneous multi-core architectures: the potential for processor power reduction. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on* (dec. 2003).

[52] Kumar, R., Tullsen, D.M., Ranganathan, P., Jouppi, N.P., and Farkas, K.I. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on* (june 2004).

[53] Kumar, Rakesh, Jouppi, Norman P., and Tullsen, Dean M. Conjoined-core chip multiprocessing. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2004), MICRO 37, IEEE Computer Society, pp. 195–206.

[54] Kumar, Rakesh, Tullsen, Dean M., and Jouppi, Norman P. Core architecture optimization for heterogeneous chip multiprocessors. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques* (2006), PACT '06.

[55] Lackey, D.E., et al. Managing power and performance for system-on-chip designs using voltage islands. In *Computer Aided Design, 2002. ICCAD 2002. IEEE/ACM International Conference on* (nov. 2002), pp. 195 – 202.

[56] Lau, Eric, Miller, Jason E., Choi, Inseok, Yeung, Donald, Amarasinghe, Saman, and Agarwal, Anant. Multicore performance optimization using partner cores. In *Proceedings of the 3rd USENIX conference on Hot topic in parallelism* (Berkeley, CA, USA, 2011), HotPar'11, USENIX Association, pp. 11–11.

[57] Lee, Chunho, Potkonjak, Miodrag, and Mangione-Smith, William H. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture* (1997), MICRO 30.

[58] Lee, R.B. Multimedia extensions for general-purpose processors. In *Signal Processing Systems, 1997. SIPS 97 - Design and Implementation., 1997 IEEE Workshop on* (nov 1997), pp. 9 –23.

[59] Leibson, S. Reduce soc energy consumption through processor isa extension. In *System-on-Chip, 2007 International Symposium on* (nov. 2007), pp. 1 –4.

[60] Levy, H.M., et al. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Computer Architecture, 1996 23rd Annual International Symposium on* (may 1996), p. 191.

[61] Li, Chuanpeng, Ding, Chen, and Shen, Kai. Quantifying the cost of context switch. In *Proceedings of the 2007 workshop on Experimental computer science* (New York, NY, USA, 2007), ExpCS '07, ACM.

[62] Li, Man-Lap, Ramachandran, P., Sahoo, S.K., Adve, S.V., Adve, V.S., and Zhou, Yuanyuan. Trace-based microarchitecture-level diagnosis of permanent hardware faults. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on* (june 2008), pp. 22 –31.

[63] Li, Sheng, Ahn, Jung-Ho, Strong, R.D., Brockman, J.B., Tullsen, D.M., and Jouppi, N.P. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on* (2009), pp. 469–480.

[64] Lichtenau, C., Ringler, M.I., Pfluger, T., Geissler, S., Hilgendorf, R., Heaslip, J., Weiss, U., Sandon, P., Rohrer, N., Cohen, E., and Canada, M. Powertune: advanced frequency and power scaling on 64b powerpc microprocessor. In *Solid-State Circuits Conference, 2004. Digest of Technical Papers. ISSCC. 2004 IEEE International* (2004), pp. 356–357 Vol.1.

[65] Lukefahr, Andrew, Padmanabha, Shruti, Das, Reetuparna, Sleiman, Faissal M., Dreslinski, Ronald, Wenisch, Thomas F., and Mahlke, Scott. Composite cores: Pushing heterogeneity into a core. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2012), MICRO '12, IEEE Computer Society, pp. 317–328.

[66] M. Yilmaz et al. Self-Checking and Self-Diagnosing 32-bit Microprocessor Multiplier. In *Test Conference, 2006. ITC '06. IEEE International* (2006).

[67] Meisner, David, Gold, Brian T., and Wenisch, Thomas F. Powernap: eliminating server idle power. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2009), ASPLOS XIV, ACM, pp. 205–216.

[68] Meixner, A., Bauer, M.E., and Sorin, D.J. Argus: Low-cost, comprehensive error detection in simple cores. In *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on* (dec. 2007), pp. 210 –222.

[69] Meixner, A., and Sorin, D.J. Error detection via online checking of cache coherence with token coherence signatures. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on* (2007), pp. 145–156.

[70] Najaf-abadi, Hashem Hashemi, Choudhary, Niket Kumar, and Rotenberg, Eric. Core-selectability in chip multiprocessors. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques* (Washington, DC, USA, 2009), PACT '09, IEEE Computer Society, pp. 113–122.

[71] Pan, Abhisek, et al. Improving yield and reliability of chip multiprocessors. In *Proceedings of the Conference on Design, Automation and Test in Europe* (2009), DATE '09, pp. 490–495.

[72] Pericas, Miquel, Cristal, Adrian, Cazorla, Francisco J., Gonzalez, Ruben, Jimenez, Daniel A., and Valero, Mateo. A flexible heterogeneous multi-core architecture. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques* (Washington, DC, USA, 2007), PACT '07, IEEE Computer Society, pp. 13–24.

[73] Pricopi, Mihai, and Mitra, Tulika. Bahurupi: A polymorphic heterogeneous multi-core architecture. *ACM Trans. Archit. Code Optim. 8*, 4 (Jan. 2012), 22:1–22:21.

[74] Reinhardt, Steven K., and Mukherjee, Shubhendu S. Transient fault detection via simultaneous multithreading. In *Proceedings of the 27th annual international symposium on Computer architecture* (New York, NY, USA, 2000), ISCA '00, ACM, pp. 25–36.

[75] Renau, Jose. Sesc: Superescalar simulator, 2005.

[76] Rodrigues, R., Annamalai, A., Koren, I., Kundu, S., and Khan, O. Performance per watt benefits of dynamic core morphing in asymmetric multicores. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on* (oct. 2011), pp. 121 –130.

[77] Rodrigues, R., et al. Scalable thread scheduling in asymmetric multicores for power efficiency. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on* (2012), pp. 59–66.

[78] Rodrigues, R., and Kundu, S. On graceful degradation of chip multiprocessors in presence of faults via flexible pooling of critical execution units. In *On-Line Testing Symposium (IOLTS), 2011 IEEE 17th International* (2011), pp. 67–72.

[79] Rodrigues, R., and Kundu, S. An online mechanism to verify datapath execution using existing resources in chip multiprocessors. In *Test Symposium (ATS), 2011 20th Asian* (nov. 2011), pp. 161 –166.

[80] Rodrigues, R., Kundu, S., and Khan, O. Shadow checker (SC): A low-cost hardware scheme for online detection of faults in small memory structures of a microprocessor. In *Test Conference (ITC), 2010 IEEE International* (nov. 2010), pp. 1 –10.

[81] Rodrigues, Rance, Annamalai, Arunachalam, Koren, Israel, and Kundu, Sandip. Improving performance per watt of asymmetric multi-core processors via online program phase classification and adaptive core morphing. vol. 18, ACM, pp. 5:1–5:23.

[82] Rotenberg, E. AR-SMT: a microarchitectural approach to fault tolerance in microprocessors. In *Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on* (june 1999), pp. 84 –91.

[83] Rusu, S., Tam, S., Muljono, H., Ayers, D., and Chang, J. A dual-core multi-threaded xeon processor with 16mb l3 cache. In *Solid-State Circuits Conference, 2006. ISSCC 2006. Digest of Technical Papers. IEEE International* (feb. 2006), pp. 315 –324.

[84] Saez, Juan Carlos, et al. A comprehensive scheduler for asymmetric multicore systems. In *Proceedings of the 5th European conference on Computer systems* (2010), EuroSys '10.

[85] Semeraro, Greg, et al. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture* (2002), HPCA '02, pp. 29–.

[86] Shelepov, Daniel, Saez Alcaide, Juan Carlos, Jeffery, Stacey, Fedorova, Alexandra, Perez, Nestor, Huang, Zhi Feng, Blagodurov, Sergey, and Kumar, Viren. Hass: a scheduler for heterogeneous multicore systems. *SIGOPS Oper. Syst. Rev. 43* (April 2009).

[87] Sherwood, Timothy, Sair, Suleyman, and Calder, Brad. Phase tracking and prediction. In *Proceedings of the 30th annual international symposium on Computer architecture* (2003), ISCA '03.

[88] Shivakumar, P., et al. Exploiting microarchitectural redundancy for defect tolerance. In *Computer Design, 2003. Proceedings. 21st International Conference on* (oct. 2003), pp. 481 – 488.

[89] Shivakumar, Premkishore, Jouppi, Norman P., and Shivakumar, Premkishore. Cacti 3.0: An integrated cache timing, power, and area model. Tech. rep., 2001.

[90] Shyam, Smitha, Constantinides, Kypros, Phadke, Sujay, Bertacco, Valeria, and Austin, Todd. Ultra low-cost defect protection for microprocessor pipelines. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2006), ASPLOS XII, ACM, pp. 73–82.

[91] Siewiorek, Daniel P., and Swarz, Robert S. *Reliable computer systems (3rd ed.): design and evaluation.* A. K. Peters, Ltd., Natick, MA, USA, 1998.

[92] Singh, Karan, Bhadauria, Major, and McKee, Sally A. Real time power estimation and thread scheduling via performance counters. *SIGARCH Comput. Archit. News 37* (July 2009), 46–55.

[93] Slegel, Timothy J., Averill III, Robert M., Check, Mark A., Giamei, Bruce C., Krumm, Barry W., Krygowski, Christopher A., Li, Wen H., Liptay, John S., MacDougall, John D., McPherson, Thomas J., Navarro, Jennifer A., Schwarz, Eric M., Shum, Kevin, and Webb, Charles F. Ibm's s/390 g5 microprocessor design. *IEEE Micro 19*, 2 (Mar. 1999), 12–23.

[94] Smolens, Jared C., et al. Efficient resource sharing in concurrent error detecting superscalar microarchitectures. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture* (2004), MICRO 37, pp. 257–268.

[95] Smolens, Jared C., Gold, Brian T., Kim, Jangwoo, Falsafi, Babak, Hoe, James C., and Nowatzyk, Andreas G. Fingerprinting: bounding soft-error detection latency and bandwidth. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems* (2004), ASPLOS-XI.

[96] Sorin, Daniel J., Martin, Milo M. K., Hill, Mark D., and Wood, David A. SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Proceedings of the 29th annual international symposium on Computer architecture* (2002), ISCA '02.

[97] SPEC2000. The Standard Performance Evaluation Corporation (Spec CPI2000 suite).

[98] Srinivasan, J., Adve, S.V., Bose, P., and Rivers, J.A. The impact of technology scaling on lifetime reliability. In *Dependable Systems and Networks, 2004 International Conference on* (june-1 july 2004), pp. 177 – 186.

[99] Srinivasan, J., et al. Exploiting structural duplication for lifetime reliability enhancement. In *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on* (june 2005), pp. 520 – 531.

[100] Srinivasan, Sadagopan, Zhao, Li, Illikkal, Ramesh, and Iyer, Ravishankar. Efficient interaction between os and architecture in heterogeneous platforms. *SIGOPS Oper. Syst. Rev. 45*, 1 (Feb. 2011), 62–72.

[101] Tarjan, D., Boyer, M., and Skadron, K. Federation: Repurposing scalar cores for out-of-order instruction issue. In *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE* (june 2008), pp. 772 –775.

[102] Tullsen, Dean M., et al. Simultaneous multithreading: maximizing on-chip parallelism. *SIGARCH Comput. Archit. News 23*, 2 (May 1995), 392–403.

[103] van de Waerdt, Jan-Willem, Vassiliadis, Stamatis, Das, Sanjeev, Mirolo, Sebastian, Yen, Chris, Zhong, Bill, Basto, Carlos, van Itegem, Jean-Paul, Amirtharaj, Dinesh, Kalra, Kulbhushan, Rodriguez, Pedro, and van Antwerpen, Hans. The tm3270 media-processor. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2005), MICRO 38, IEEE Computer Society, pp. 331–342.

[104] Vasudevan, D.P., and Lala, P.K. A technique for modular design of self-checking carry-select adder. In *Defect and Fault Tolerance in VLSI Systems, 2005. DFT 2005. 20th IEEE International Symposium on* (2005).

[105] Watanabe, Yasuko, et al. Widget: Wisconsin decoupled grid execution tiles. In *Proceedings of the 37th annual international symposium on Computer architecture* (2010), ISCA '10, pp. 2–13.

[106] Winter, Jonathan A., Albonesi, David H., and Shoemaker, Christine A. 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* (2010), PACT '10.

[107] Woo, Steven Cameron, et al. The splash-2 programs: characterization and methodological considerations. *SIGARCH Comput. Archit. News 23*, 2 (May 1995), 24–36.

[108] Yu, Pan, and Mitra, Tulika. Characterizing embedded applications for instruction-set extensible processors. In *Proceedings of the 41st annual Design Automation Conference* (New York, NY, USA, 2004), DAC '04, ACM, pp. 723–728.

[109] Zhang, Meng, Lebeck, A.R., and Sorin, D.J. Fractal coherence: Scalably verifiable cache coherence. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on* (2010), pp. 471–482.