NATIONAL TECHNICAL UNIVERSITY OF ATHENS
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

COMPUTER SCIENCE DIVISION
COMPUTING SYSTEMS LABORATORY

Characterizing Thread Placement and Thread Priorities in the IBM POWER7 Processor

DIPLOMA THESIS

of

Stylianos-Filippos A. Manousopoulos

Supervisors: Nectarios Koziris, Associate Professor of N.T.U.A.
Miquel Moreto Planas, Barcelona Supercomputing Center.
Francisco J. Cazorla, Barcelona Supercomputing Center.

Athens, September 2012

NATIONAL TECHNICAL UNIVERSITY OF ATHENS
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
COMPUTER SCIENCE DIVISION
COMPUTING SYSTEMS LABORATORY

Characterizing Thread Placement and Thread Priorities in the IBM POWER7 Processor

DIPLOMA THESIS

of

Stylianos-Filippos A. Manousopoulos

Supervisors: Nectarios Koziris, Associate Professor of N.T.U.A.
Miquel Moreto Planas, Barcelona Supercomputing Center.
Francisco J. Cazorla, Barcelona Supercomputing Center.

Approved by the committee on the 21st of September 2012.

.....	.....	.....
Nectarios Koziris	Nikolaos Papaspyrou	Dimitrios Soudris
Associate Professor N.T.U.A.	Assistant Professor N.T.U.A.	Assistant Professor N.T.U.A.

Athens, September 2012

.....

Stylianos-Filippos A. Manousopoulos
Electrical and Computer Engineer

Abstract

The current trend in processor design is towards the combination of several thread level parallelism paradigms on the same chip. A popular combination is Chip Multiprocessing with Simultaneous Multithreading (CMP+SMT), implemented in processors such as the IBM POWER7. In such complex multithreaded designs, resource sharing between threads has a great impact on final performance. There are different ways of interfering with resource sharing, including *thread placement*, which involves assigning software threads to the available hardware contexts, and *thread priorities*, assuming a processor in the IBM POWER family, that features a mechanism allowing the user to alter the instruction fetch rate of active threads.

We have analyzed thread placement and thread priorities in the IBM POWER7 processor. Under each placement and priorities setup we analyze in detail how hardware resources are shared among running threads. We show to which extent a software designer can characterize an application on the specific processor and based on that characterization, select the best thread placement and thread priorities configuration to improve a target metric. Our results show that a 54% reduction in execution time can be obtained (11.2% on average) when running pairs of desktop parallel applications under the appropriate thread placement. On top of that, our study has shown that up to an extra 12.7% of execution time improvement can be achieved with the use of priorities on parallel applications.

Keywords

Resource sharing; thread placement; thread priorities; SMT; CMP; IBM POWER7; Micro-benchmarks; PARSEC;

Περίληψη

Η σύγχρονη τάση στη σχεδίαση επεξεργαστών τείνει προς τον συνδυασμό διάφορων παραδειγμάτων παραλληλοποίησης επιπέδου νήματος. Ένας δημοφιλής συνδυασμός είναι η Πολυεπεξεργασία Επιπέδου Chip μαζί με Ταυτόχρονο Πολυνηματισμό (SMT + CMP), που υλοποιείται σε επεξεργαστές όπως ο IBM POWER7. Σε τέτοιες περίπλοκες πολυνηματικές αρχιτεκτονικές ο διαμοιρασμός πόρων μεταξύ των νημάτων έχει μεγάλη επίδραση στην τελική απόδοση. Υπάρχουν διάφοροι τρόποι επέμβασης στο διαμοιρασμό πόρων, συμπεριλαμβανομένης της *τοποθέτησης νημάτων*, που αφορά την αντιστοίχιση νημάτων λογισμικού σε θέσεις νημάτων υλικού, και της *προτεραιότητας νημάτων*, χρησιμοποιώντας ένα μηχανισμό που υπάρχει στην οικογένεια επεξεργαστών POWER της IBM ο οποίος επιτρέπει στον χρήστη να αλλάζει τον ρυθμό προσκόμισης εντολών των ενεργών νημάτων.

Χαρακτηρίζουμε τους μηχανισμούς τοποθέτησης νημάτων και προτεραιότητας νημάτων στον επεξεργαστή IBM POWER7. Κάτω από διάφορες συνθέσεις τοποθέτησης και προτεραιότητας νημάτων αναλύουμε με λεπτομέρεια τον τρόπο διαμοιρασμού του υλικού ανάμεσα στα ενεργά νήματα. Δείχνουμε σε ποιό βαθμό ένας σχεδιαστής λογισμικού μπορεί να χαρακτηρίσει μια εφαρμογή στον συγκεκριμένο επεξεργαστή και βασισμένος σε αυτόν τον χαρακτηρισμό, να διαλέξει την ιδανική τοποθέτηση και προτεραιότητα νημάτων, ώστε να βελτιώσει έναν επιθυμητό στόχο. Τα αποτελέσματά μας δείχνουν ότι μπορεί να επιτευχθεί μέχρι 54% μείωση στο χρόνο εκτέλεσης (11.2% κατά μέσο όρο) κατα την εκτέλεση ζευγών παράλληλων εφαρμογών κάτω από την κατάλληλη τοποθέτηση νημάτων. Επιπροσθέτως, η μελέτη μας έδειξε οτι μέχρι και 12.7% επιπλέον βελτίωση μπορεί να επιτευχθεί με τη χρήση προτεραιοτήτων νημάτων.

Λέξεις - κλειδιά

Διαμοιρασμός πόρων; τοποθέτηση νημάτων; προτεραιότητα νημάτων; Παραλληλοποίηση Επιπέδου Chip; Ταυτόχρονος Πολυνηματισμός ; IBM POWER7; Micro-benchmarks; PARSEC;

Acknowledgements

This work was conducted at the Barcelona Supercomputing Center (BSC) in the Polytechnic University of Catalonia (UPC). Collaboration with the National Technical University of Athens (NTUA) was achieved through the Erasmus student exchange program.

First of all, I would like to thank my supervisors in Barcelona, whose constant attention was invaluable for the progress and completion of this thesis. Miquel Moreto, as my closest supervisor, for helping me in everything and guiding me throughout this period on an everyday basis. Fran Cazorla for being extremely willing to assist me from the first moment I contacted him right until the thesis was complete. Also, Roberto Gioiosa for being constantly available to answer all my questions and help me in any other way I needed.

Of course, none of this would have taken place without the help and support of my supervisors in Athens. Kostis Nikas for his help throughout this period, from choosing the subject and arranging the exchange to the final touches of writing this thesis. Also, Nikos Anastopoulos for his constructive comments and ideas that helped a lot in its improvement. I would especially like to thank my professor, Nectarios Koziris, for introducing me to the subject of computer architecture and for his willingness to let me conduct my thesis abroad.

In any way, I want to express my appreciation to my family for their support all these years and their advice on eating healthy that I never followed. Finally, my gratitude to all my friends in Barcelona and Athens for all the experiences meanwhile working on this thesis.

Contents

List of Figures

List of Tables

Chapter 1

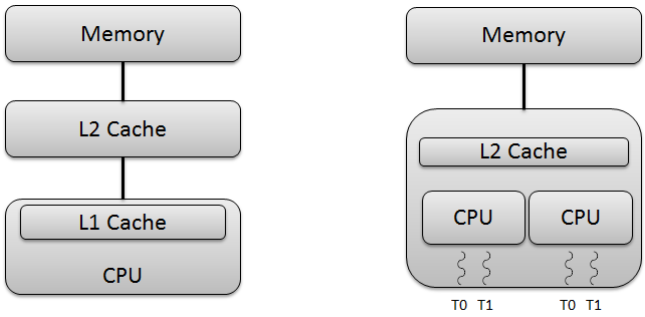
Introduction

1.1 Uniprocessor systems

Formerly, processor manufacturers competed in the race to follow Moore's law on the number of transistors in integrated circuits [22]. The main strategies for increasing processing performance in uniprocessor systems was the increase of clock frequency and the extraction of higher Instruction Level Parallelism (ILP) from a single instruction stream. Processor pipelining was the first step in this direction, since it allowed the fragmentation of resources in order to simultaneously serve multiple program instructions at different stages of execution. The increasing transistor numbers allowed processor designers to implement more and more complex ILP extraction techniques. Some of these techniques included superscalar pipelining, which allowed the parallel issuing of more than one subsequent instructions simultaneously, out-of-order completion, which allowed the completion of instructions that would otherwise have to wait for previous instruction to finish, and branch prediction, which decreased the amount of flushed instructions during commonly used program loops.

However, spatial and thermal barriers led to a physical inability to keep increasing the number of transistors indefinitely. Also, a single executing thread makes it difficult to extract parallelism beyond a certain level, because of limiting factors such as instruction dependencies, or long latency events like cache misses. As a result, manufacturers had to turn to other solutions.

Figure 1.1(a) shows a typical uniprocessor system. It involves a single processor with its on-core L1 cache and is connected via buses to lower cache levels and the main system memory.



(a) (b)

Figure 1.1: Indicative (a) Uniprocessor and (b) Multiprocessor system schematic

1.2 Multiprocessor Systems

The inability to extract parallelism beyond a certain threshold from a single instruction stream led to the observation that most of the core resources are idle while waiting to execute instructions [37]. The idea of maximizing the utilization of processor resources brought about another form of parallelization, Thread Level Parallelism (TLP). Since processor resources could not be fully utilized by a single instruction stream, they could be taken advantage of by multiple independent instruction streams, called threads. The idea of TLP is that if many threads run in parallel, performance can be improved through an increase in throughput, rather than focusing on latency that was the goal anteriorly. For this reason multiprocessor systems with support to multiple threads have become the standard direction in processor design.

TLP is expressed in two main ways, first by sharing available processor resources between threads (Simultaneous Multithreading - SMT) and also by replicating processor pipelines in order to support multiple threads at the core level while also sharing some global resources (Chip Multiprocessing - CMP). Both TLP expressions manage to increase processor resources utilization thus increasing total system throughput. However, they also introduce the problem of resource sharing between threads. In multiprocessor systems many resources are shared between threads in order to maintain high utilization at all times. This opens up a new area of study in computer architecture, involving ways of dealing with resource sharing between threads.

Figure 1.1(b) shows a typical multiprocessor system featuring 2 cores, with each supporting 2 threads. Each of the two cores has its private L1 cache, while they both share the on-chip L2 cache and access the out-of-chip memory.

1.3 Motivation

The current trend in multithreaded systems design leads to increasingly more complex resource sharing. Co-running threads share processor resources in various ways, which significantly affects their performance. Two ways of affecting resource allocation per thread and dealing with the problem of resource sharing are *thread placement* and *thread priorities*. Thread placement involves binding software threads to the available hardware contexts, while thread priorities are supported by the IBM POWER family processors and allow the user to intervene in the distribution of processor resources between threads.

It becomes clear that a characterization of the resource sharing on current complex architectures with regard to two ways of affecting it, like thread placement and thread priorities, can lead to useful conclusions on multiple levels.

1.4 Organization

Following the introduction, this thesis is structured as follows: In Chapter 2 the issue of Multithreading is discussed, with a detailed presentation of various TLP paradigms and some significant market implementations, while introducing the issues of thread placement and thread priorities. In continuation, Chapter 3 presents the IBM POWER7 processor that is used in our study, while focusing on the resource sharing of the processor. The following Chapter 4 describes the subsystem used in our study, including the performance counters, the modifications on the linux kernel, the micro-benchmarks and parallel benchmarks used in our study. Subsequently, Chapter 5 presents a study with micro-benchmarks, that are used to draw useful conclusions, which are then in turn applied to parallel benchmark applications in our presented Case Study in Chapter 6. Finally, Chapter 7 concludes our findings and lists possible steps that could be taken following this study.

Chapter 2

Multithreading

2.1 Introduction to Multithreading

Exploitation of Instruction Level Parallelism (ILP) improves performance for a single instruction stream. However, inherent ILP limitations lead to limited resource usage, since the available issue slots are not always filled at a given cycle. The term 'issuing waste' [37] is used to describe the amount of empty issue slots and can be divided into horizontal and vertical waste, as shown in Figure 2.1. Vertical waste occurs where no instructions are issued at all in a cycle, e.g. due to cache misses or branch mispredictions, and horizontal waste refers to cases where there are some empty issue slots in a cycle, for example due to inter-dependent instructions. Both of them are inherent to ILP and its expressions such as superscalar issuing and out-of-order execution and therefore cannot be alleviated on the instruction stream level.

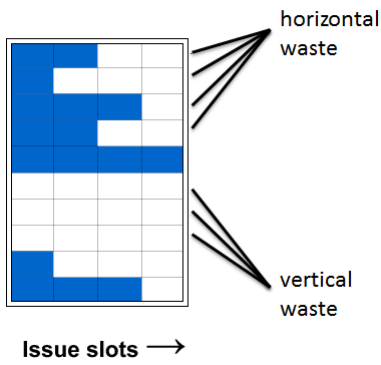


Figure 2.1: Superscalar processor issuing.

This motivates the extraction of parallelism from multiple instruction streams (Thread

Level Parallelism -TLP) in order to achieve higher processor resource utilization. With the term multithreaded system we refer to any system design that implements Thread Level Parallelism. In order to minimize resource waste, multithreaded processors allocate its various resources to more than one threads of execution. There are several multithreading paradigms which deal with issuing multiple threads in different ways, including Chip Multiprocessing (CMP) [24], Simultaneous Multithreading (SMT) [37], Fine Grain Multithreading (FGMT) [12][28], Coarse Grain Multithreading (CGMT) [2][31] or even combinations of the above. Figure 2.2 shows how each case deals with the problems of issuing waste and the following section addresses each of them in detail.

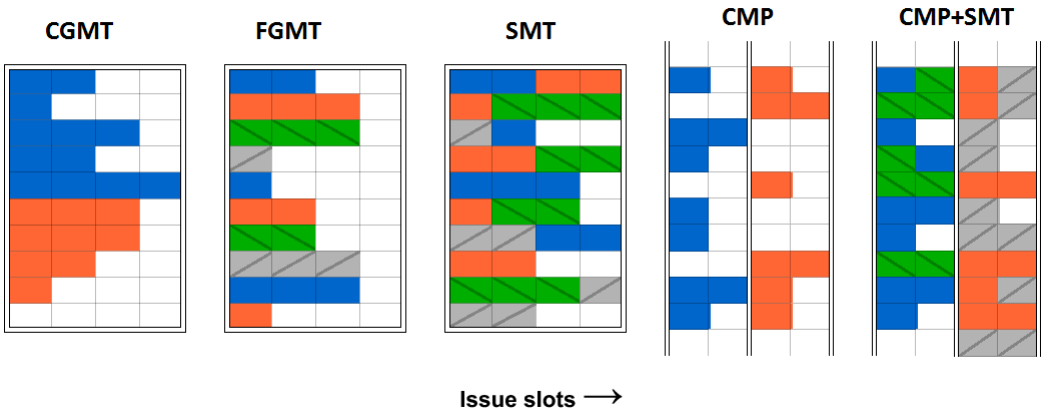


Figure 2.2: Multithreading processors issuing.

2.2 Multithreading Paradigms

In this section we present the basic TLP paradigms, including CMP, SMT, FGMT and the popular combination of SMT & CMP.

2.2.1 Chip MultiProcessing (CMP)

In 1996 Olokotun et al discussed the necessity to shift focus from expanding complex out-of-order issuing mechanisms towards architectures with more but simpler processors [24]. According to them, the expansion of the register files would lead to an increase in implementation complexity and a decrease in clock frequency. They concluded that an implementation of smaller, simpler processors could be realized in the same chip area and give the possibility for higher clock rates, in comparison to more complex single-core designs.

Chip Multiprocessors are a logical development of the idea of Symmetric Multiprocessors (SMP). SMP involved placing two or more identical CPUs in a shared-memory system, allowing parallel processing. CMP advanced this idea by providing multithreading at the chip level with the inclusion of two or more processors on the same chip.

CMP technology does not deal with the issues of resource waste during execution, but instead increases the on-core hardware to provide multiple separate instruction pipelines. In this way, multiple processing cores are provided on the same chip, while improving latency and energy consumption compared to out-of-chip SMPs.

2.2.2 Simultaneous MultiThreading (SMT)

The foundations of SMT technology were laid by Dean Tullsen et al in [37]. They observed that processor resources remained idle for long periods of time during execution so they proposed that instructions from more than one threads should be issued simultaneously. Generally SMT architecture involves partitioning key resources statically while sharing the rest dynamically. Statically allocated resources ensure fairness between threads, so that no thread clogs resources, while dynamically shared resources provide maximum resource utilization.

SMT succeeds in eliminating both horizontal and vertical waste thus achieving maximum resource usage at all times. Vertical waste is removed since a thread that is stalled by a long-latency event can be bypassed and the rest of the threads can use the available core resources. Horizontal waste is effectively reduced due to the fact that instructions from multiple threads can be issued simultaneously, so empty issue slots caused by instruction inter-dependencies can be filled by instructions from other threads.

2.2.3 FGMT

Fine Grain MultiThreading (FGMT) is another alternative to SMT that supports issuing instructions from multiple threads. However, while SMT supports instruction issuing from multiple threads at a cycle, FGMT only supports issuing from just one thread at any instance. Thread switching is performed in a round-robin fashion, while also overriding threads that are stalled on long-latency events. Due to its constant thread switching philosophy, FGMT implementations try to keep instruction pipelines relatively short and simple, in order to increase throughput.

FGMT manages to eliminate vertical waste since stalled threads are overridden, but it does not affect horizontal waste because only instructions from one thread can be issued at any time.

Chip	CMP	FGMT	SMT
IBM POWER4	✓		
IBM POWER5	✓		✓
IBM POWER7	✓		✓
Intel Pentium 4			✓
Intel Core Duo	✓		
Intel i7	✓		✓
Sun UltraSPARC IV+	✓		
Sun UltraSPARC T1	✓	✓	
Sun UltraSPARC T2	✓	✓	

Table 2.1: Different TLP paradigms.

2.2.4 SMT + CMP

A hybrid architecture combining the technologies of SMT and CMP was first introduced by IBM with the POWER5 processor. Such a processor is equipped with duplicate identical processing pipelines (CMP), while each pipeline supports issuing from multiple threads at the same time (SMT). CMP allows replicating of cores with better energy efficiency, while SMT reduces fragmentation in on-chip resources.

So, SMT effectively deals with the issues of both horizontal and vertical waste since instructions from multiple threads are being issued simultaneously, while CMP increases the on-chip resources to support multiple pipelines. CMP + SMT proves to be a very popular combination as it combines the advantages of both paradigms.

2.3 Resource Sharing

The various multithreading architectures share their resources between running threads. Depending on the specific paradigm and implementation, the complexity of resource-sharing varies from case to case. We can characterize the complexity of the multithreading architecture based on the levels on which its hardware resources are shared by executing threads.

In CMP architectures processor resources, such as the interconnection network or an on-chip L2 cache, are only shared on a single level, between threads in different cores (**inter-core**). In SMT and FGMT architectural resources are also shared on one level, between threads in the same core (**intra-core**). On the intra-core level resources are shared on a much larger extent compared to the inter-core level. In both CMPs and SMT pro-

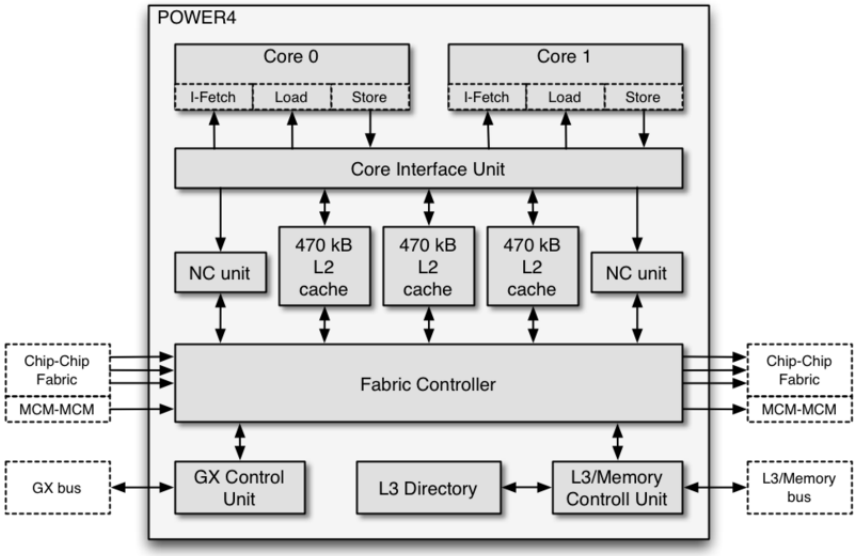


Figure 2.3: POWER4 Processor schematic.

cessors all threads share processor resources equally, regardless of where they are placed. Nevertheless, in hybrid architectures, such as SMT+CMP, thread interaction differs based on which level of resources they have in common: threads in the same core (intra-core level) share many more resources, and hence interact much more than threads in different cores (inter-core level). So, hybrid TLP systems provide 2 levels of resource sharing, both the inter-core and intra-core levels. Some hybrid SMT+CMP implementations increase resource sharing complexity even further, as they divide thread contexts inside each core into two clusters. This provides an added level of resource sharing (**intra-cluster**) with threads in the same cluster sharing more resources than threads in different clusters.

Since conflicts between threads that are fighting for common resources can severely affect performance [35], it becomes evident that resource sharing in multithreaded architectures becomes an important part of any study.

We present the development of multithreading through different market implementations. We focus on how resources are shared and the increasing complexity of the resource sharing on these processors. Table 2.1 presents a selection of representative chips.

2.3.1 IBM

POWER4

The first CMP processor was IBM's POWER4 that was released in 2001. POWER4 features two cores on a single chip [34]; each core has its private pipeline and L1 Instruction

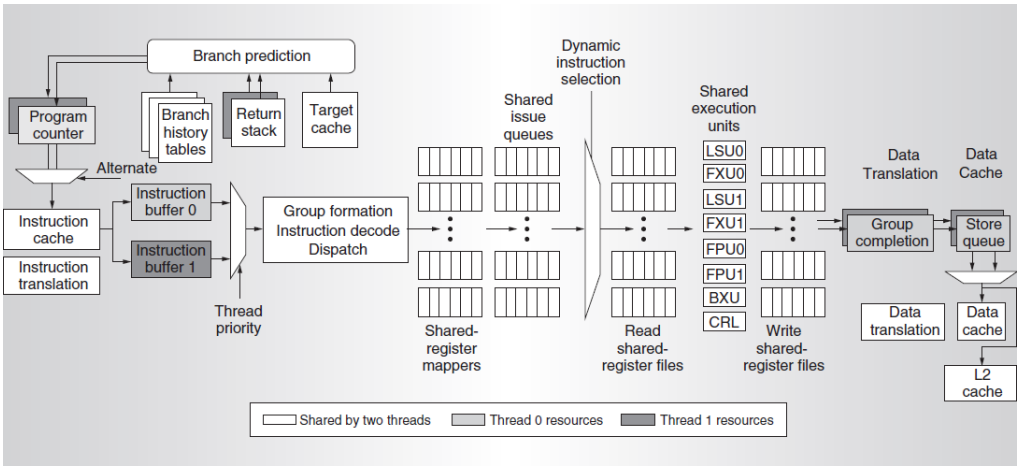


Figure 2.4: Power5 Processor pipeline.

(128 KB) and Data (64 KB) Caches, but the two cores share a unified 1.5 MB L2 on-chip Cache. The Core Interface Unit (CIU) is responsible for the interconnection between the two cores and the L2 cache, which is divided in three individual parts. The CIU is made of a crossbar switch and connects each core with all three cache controllers. Figure 2.3 shows the POWER4 schematic with the main units and the interconnections between them.

POWER5

The first processor to combine SMT and CMP technologies was IBM's POWER5 [17] released in 2004. POWER5 kept its predecessor's dual-core design and expanded each core to support 2-way multithreading, offering a total of four active threads at any single time. The resources on the POWER5 processor are shared between running threads on two different levels:

- Intra-core: all on-core resources are shared between threads on the same core. Figure 2.4 presents all resources shared on this level and shows how some core resources are doubled, while others are shared dynamically.

  Each core has two modes of execution, ST mode and SMT mode. When the core runs in ST mode, all core resources are allocated to the executing thread, whereas in SMT mode, the core uses two separate Program Counters and fetches instructions from the two threads, while switching between them. When running in SMT mode, the POWER5 core ensures fairness between executing threads with dynamic resource balancing. By monitoring the GCT and load-miss queues dynamic resource balancing detects if one thread is clogging resources. According to POWER5 design a thread is considered to cause resource congestion when it overcomes a threshold

of L2 cache or TLB misses, or when it uses too many Global Completion Table (GCT) entries. In that case it prohibits further instructions to be dispatched or decoded (depending on the type of congestion) and so allows the other thread to take advantage of the available resources. Additionally, POWER5 brings another innovation in SMT thread handling: software-controlled hardware thread priorities that determine the ratio of fetched instructions between the two threads.

- Inter-core: global resources are available to all threads on the chip, whether they share a core or not. These resources consist of the L2, and L3 cache and the on-chip memory controller. The unified L2 cache with a size of 1.875 MB is placed on the POWER5 chip. It is implemented as three separate slices, each with their own controller. The 32 MB L3 cache is placed off chip, but its directory is designed on the chip in order to minimize L3 request latency.

2.3.2 Intel

Pentium 4

Intel was the first to implement the Simultaneous MultiThreading TLP paradigm, which it named Hyper-Threading [35][21]. It was first released on the Xeon server processor in 2002 and later in the same year Intel released its desktop version of a Hyper-Threading processor, the Intel Pentium 4.

Hyper-threading involves maintaining dual architectural state copies on the processor, thus forming two logical processors. As a result the Operating System can see two processors and schedule tasks on each one of them. Whereas the architectural state is duplicated, most other resources such as caches, execution units, branch predictors, control logic and buses are shared.

Pentium 4 is a deeply out-of-order multi-stage pipeline processor. Figure 2.5 shows Pentium 4's pipeline, depicting both the parts that are statically allocated to each thread, including the Uop (micro-operation) queue, the rename, issue queue and retire stages, as well as those that are dynamically shared between them, such as the schedule, register read, execute, L1 cache and register write stages. The allocator will select instructions from each thread in turns. In case one of the logical processors has reached its limit of a resource occupancy, the allocator will only select instructions from the other thread. Also, if instructions from only one thread are available in the Uop queue the allocator will try to assign resources for that thread every cycle. The upper limit of resource usage by a thread in different important buffers, is a way to reassure fairness between threads and avoid cases where one thread clogs all core resources.

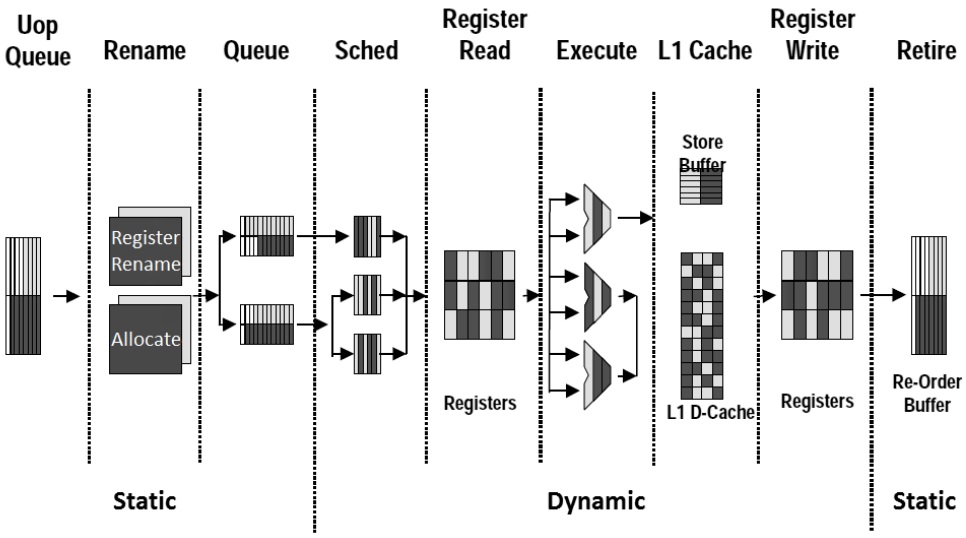


Figure 2.5: Pentium 4 Processor pipeline.

Pentium 4 supports two modes of operation: single task (ST) and multi-task (MT). In the case of a single task, all resources that were split for multi-tasking are combined and become available to it. However in MT mode there are two active threads and the resources each one can access, are divided as described above.

Core i7

Intel's latest i7 processor [19][20] combines the Chip Multiprocessing and Hyper-Threading paradigms. It features up to six cores, which share an L3 last level cache, an integrated memory controller (IMC) and an Intel QuickPath Interconnect (QPI).

The Intel i7 processor was based on the core design of the Core 2 Processor, with the difference that each core of the i7 supports Hyper-Threading, which allows the co-execution of instructions from two threads. Figure 2.6 shows two interconnected quad-core i7 sockets. Each core has a 32-KB data and instruction cache, a 256 KB unified L2 cache and an inclusive last level cache, that is usually 8MB.

The pipeline of the i7 is out-of-order with five main stages: fetch, decode, dispatch, execute and retirement/writeback. The i7 pipeline is also superscalar with a maximum of four instructions issued and decoded per cycle.

The resources are shared between the two running threads in a way that ensures fairness and avoids thread blocking. The reservation station entries are shared between the active threads in Hyper-Threading mode, with some entries reserved for each thread to avoid locking. Otherwise, all entries are available to the single running thread. The

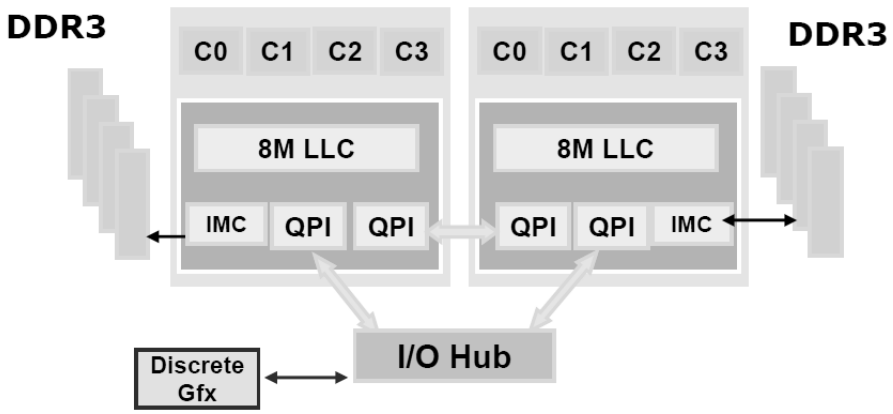


Figure 2.6: 2 quad-core sockets of the Intel i7 Processor schematic.

Reorder Buffer slots are divided if Hyper-Threading is enabled or entirely available to the single thread otherwise. The Reservation Station dispatches the instructions to one of 6 dispatch ports, for a maximum of 6 dispatched instructions for execution.

2.3.3 Sun

UltraSparc T1

The UltraSparc T1, codenamed Niagara [18] was released in 2005. It was the first processor to implement round-robin 4 thread co-execution (FGMT), while at the same time integrating 8 cores on the same chip (CMP). While other companies focused on faster and more complex processors, Sun made efforts to fit 8 simpler processors on the same chip. The T1 integrates fairly simple pipelines on each of the cores and emphasizes on the interconnection between them. Resources on the UltraSparc T1 are shared on an intra-core and and inter-core level.

- On the *inter-core* resource sharing level, the 8 cores connect to all four L2 cache banks through a crossbar switch, called CPU-Cache Crossbar (CCX). The CCX is a three stage pipeline itself, including request, arbitrate and transmit. Figure 2.7 shows this resource-sharing level, in a high-level layout of the UltraSparc T1 processor.

- At the same time each core supports up to 4 threads, in an *intra-core* resource sharing level. Each pipeline maintains a simple 6 stage (Fetch, Thread Selection, Decode, Execute, Memory and Write Back) in-order single-issue pipeline with a

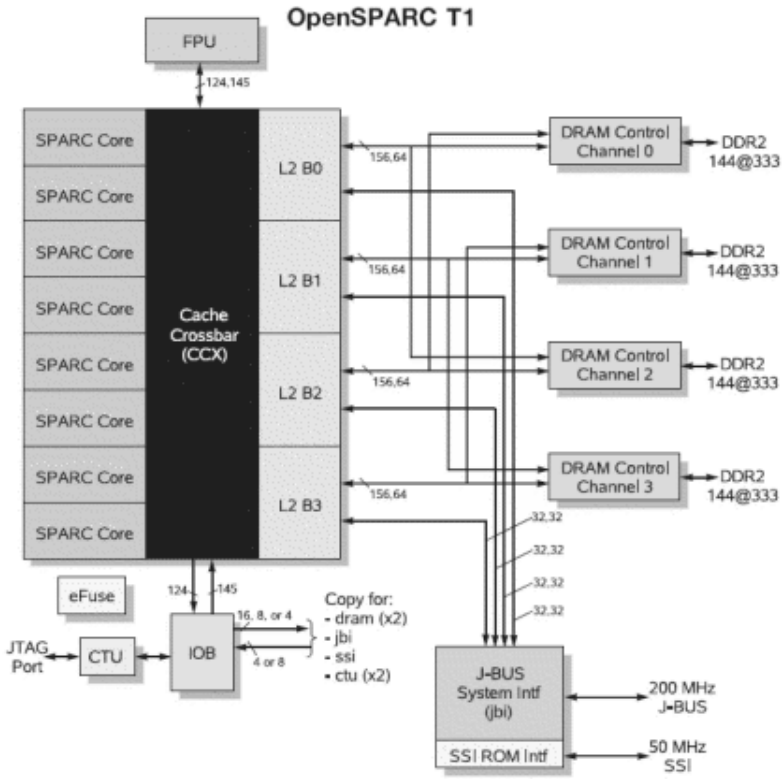


Figure 2.7: UltraSPARC T1 Processor schematic.

small private cache. The idea is to improve performance by maximizing throughput through higher resource usage and not by extracting maximum thread performance.

UltraSparc T2

The UltraSparc T2 processor (2007), or Niagara 2 [32][7] is a CMP/FGMT processor that builds upon the design of the first Niagara, where overall system performance is more important than single-task performance.

Figure 2.8 shows the eight cores of the UltraSPARC T2 connected through a crossbar switch to a shared L2 cache. Each core also supports eight hardware contexts (strands) for a maximum of 64 executing threads. Inside each core strands are divided into two groups of four strands, thus forming two hardware execution pipelines. Tasks running simultaneously on T2 share resources depending on how they are scheduled among strands. The resources of the processor are shared on three different levels: Intra-Pipe -among threads running in the same hardware pipeline, Intra-Core -among threads running on the same core and Inter-Core -among threads executing on different cores. Figure 2.8 shows the 8 cores and their interconnection, as well as the main units inside each core.

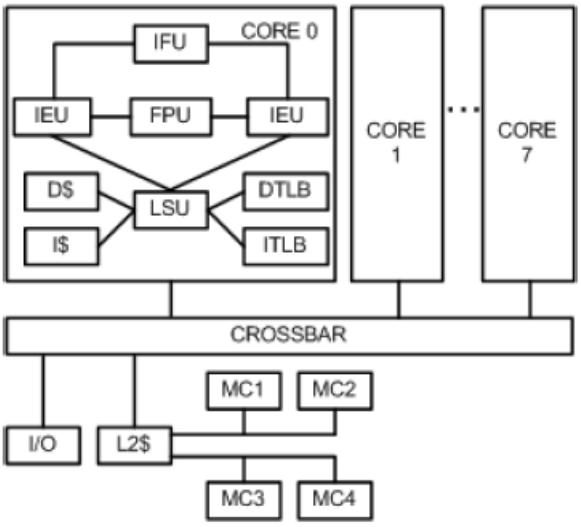


Figure 2.8: UltraSPARC T2 Processor schematic.

- Intra-Pipe: At this level shared resources include the Instruction Fetch Unit (IFU) and the Integer Execution Units (IEU). Even though the IFU is physically shared among all processes that run on the same hardware core, the instruction fetch policy prevents any interaction between threads in different hardware pipes in the IFU. Thus, the IFU behaves as two private IFUs, one for each hardware pipe. When more threads running on the same pipe are able to fetch an instruction in the same cycle, the conflict is solved using a Least Recently Fetched (LRF) fetch policy.

- Intra-Core: Threads that run on the same core share the following resources: the 16 KB L1 instruction cache, the 8 KB L1 data cache, the Load Store Unit (LSU), the Floating Point and Graphic Unit (FPU) and the Cryptographic Processing Unit.

- Inter-Core: At the global inter-core level, the main shared resources are: the L2 cache, the on-chip interconnection network, the memory controllers, and the interface to off-chip resources. The 4MB 16-way associative L2 cache has eight banks that operate independently. The L2 cache connects to four on-chip DRAM controllers, which directly interface to a pair of fully buffered DIMM (FBD) channels.

In the T2 processor, two threads running in the same pipe conflict in all resource-sharing levels: Intra-Pipe, Intra-Core and Inter-Core. Threads running in two different pipes of the same core conflict only at the Intra-Core and Inter-Core levels. Finally, threads in different cores only interact at Inter-Core level.

2.3.4 Related work

Many researchers have previously tried to investigate resource sharing and its effects on multithreaded performance. Tuck et al. analyze the performance of a real SMT processor [35], concluding that SMT architectures provide an average speedup over single-thread architectures of about 20% and that, even if the processor is designed to isolate threads, performance is still affected by resource conflicts.

Furthermore, other authors have characterized resource sharing on different processors. Jung et al [16] and Tang et al [33] have both characterized the resource sharing on the Intel Xeon processor in their works. Also Cakarevic et al [7] have characterized the three-level resource sharing of the Sun UltraSPARC T2 processor, while focusing on how such complex resource sharing can affect OS design. These works do not study the effect of thread placement on these processors since the amount of shared resources does not vary as much as in the POWER7 processor.

Thread Placement

Thread placement involves binding software threads to specific contexts within the range of available hardware thread contexts.

Previous works show that SMT performance heavily depends on the nature of the concurrently running applications [10][29]. They both propose different job scheduling policies for identifying the most suitable approach when co-scheduling different workloads. Snavely et al [29] present a job scheduler that identifies good co-executing applications during a short sampling phase. They manage to achieve a significant improvement of 17% in response time over a scheduler that does not consider co-scheduling. In the same direction DeVuyst et al [10] explore job scheduling on hybrid CMP and SMT architectures. Their proposition is that unbalanced schedules need to be considered additionally to balanced ones, and they propose adaptive policies that improve a random scheduler that only considers balanced schedules by 6-11% in energy-delay product.

Some other works conclude that parallel application threads suffer from being co scheduled in the same core due to intra-core resource contention and data cache conflicts. Jung et al [16] investigate the optimal number of threads when running a parallel application on an SMT processor and propose an adaptive technique to determine it. By using 4 Intel Xeon processors in SMP with 8 logical cores, they achieved 2 and 18 times faster execution time with their modified code w.r.t. the original code running on 4 and 8 logical cores, respectively.

Zhang et al [39] study the effect of thread placement of PARSEC applications on CMP architecture systems with regard to cache sharing. They find that contrary to prior

knowledge, shared cache has little effect on the performance of most PARSEC benchmarks. However, they also show that there is ample room for improvement as by modifying the programs in a cache-sharing-aware manner, they achieve up to 36% performance increase when placing threads appropriately. On the other hand, Tang et al [33] find that, when using Google datacenter applications, thread co-scheduling on the same core results in resource conflicts and data cache contention. Also, they are led to the conclusion that when co-executing parallel applications the optimal thread scheduling is different than when they are running alone.

Software-Controlled Hardware Thread Priorities

There are several previous works that propose the use of hardware thread priorities to control thread execution in SMT processors.

Many of these proposals implement fetch policies to maximize throughput and fairness by reducing the priority, stalling, or flushing threads that experience long latency. Tullsen et al [36] research several SMT architectures and conclude that when a thread is stalled due to a long latency operation, it is preferable to release the resources associated with it, instead of ensuring that the associated resources are at its disposal as soon as it recovers from the stall. They achieve an average of 15% and over 100% speedup when executing four threads and two threads respectively. Moreover, Cazorla et al [8] introduce the concept of dynamic resource control in SMT architectures and propose a dynamic resource allocation policy that ensures a better balance between fairness and throughput. Simulations performed showed an improvement of 8% on average over a static resource allocation policy and of 4% over the best previously proposed dynamic resource allocation policies, such as FLUSH++.

Boneti et al. use hardware priorities to balance resources in SMT processors and analyze the effect of hardware priorities on the IBM POWER5 [5]. They come to the conclusion that prioritization greatly depends on the type of workloads that are co-executing. With the use of priorities in two case studies they manage to improve overall throughput by 23.7% and additionally to reduce total execution time by 9.3%.

Morari et al. also considered the software-controllable hardware-thread priorities mechanism that controls SMT performance on POWER5 and POWER6 and provide a characterization of the differences in the application of priorities between the two microprocessors [23]. They state that the application of priorities has different effect on the two architectures, with POWER6 being less sensitive to priorities because of its in-order design. They show that the use of the mechanism has room for improvement when targeting specific metrics and in that direction they provide several performance models.

Finally, Jimenez et al. also used hardware-thread priorities to perform a power and thermal characterization and reduce power consumption of the POWER6 [15] at the application, operating system and hardware level. Based on the characterization they propose a counters-based model that allows the prediction of total power consumption on the POWER6 with an average error of under 3% for CMP and 5% for SMT.

None of these works involves the use of software-controlled hardware thread priorities with 4 executing threads.

Chapter 3

POWER7

3.1 General Architecture

The IBM POWER7 processor [27] is an 8-core CMP processor in which each core is 4-way SMT, providing 32 threads in total. Every core has its private 32-KB data and instruction L1 caches, a 256-KB private L2 cache and a 4-MB local L3 region that can also be shared between all cores, forming a 32-MB global L3 cache. Placed on the chip are two double-data-rate-three (DDR3) memory controllers, which provide a total of 100-GB/s of memory bandwidth. The POWER7 cache hierarchy is presented in detail in table 3.1. The main processor schematic can be seen in Figure 3.1.

3.2 Pipeline Description

At core level, POWER7 implements out-of-order logic with advanced branch prediction and data prefetching, optimized for enhanced single-thread performance. The main stages of the pipeline are: instruction fetching, decoding and dispatching, register renaming, instruction issuing, execution and completion. In a given cycle the core can fetch up to

Table 3.1: POWER7 Cache hierarchy

Cache level	**Capacity**	**Array**	**Policy**
L1 Data	32K	Fast SRAM	Store-through
Private L2	256K	Fast SRAM	Store-In
Fast L3 Region	Up to 4M	eDRAM	Partial Victim
Shared L3	32M	eDRAM	Adaptive

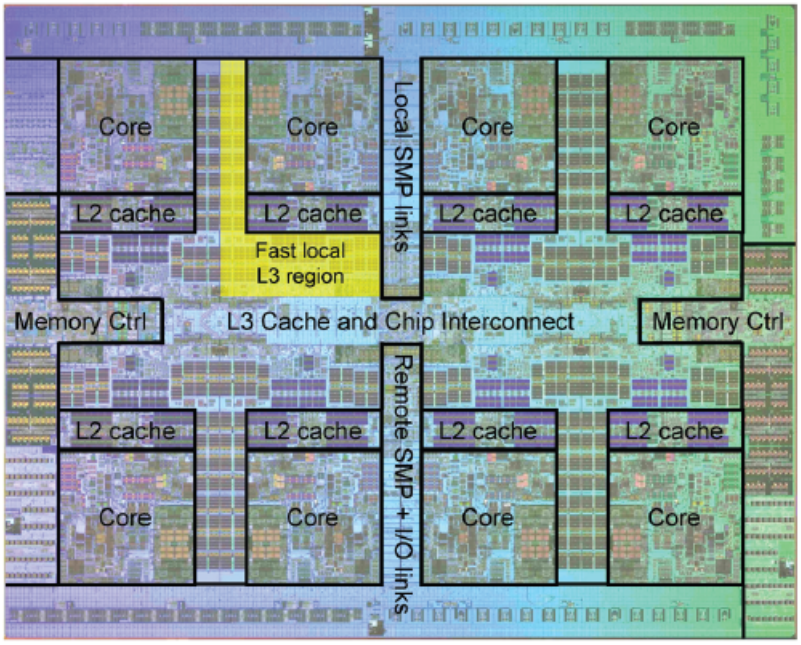


Figure 3.1: POWER7 die.

eight instructions, decode and dispatch up to six instructions and issue and execute up to eight instructions. The main pipeline stages in the POWER7 core are shown in Figure 3.2.

Initially, program instructions are fetched from the Instruction cache. The 4-way set associative Instruction cache is highly banked (16-ways) in order to allow multiple simultaneous reads and writes. After instructions are fetched from the cache, they are sent to the instruction buffers (IBUFs), which support a total of 20 entries of 4 instructions each. Thread priority, pending cache misses, IBUF occupancy and thread balancing are used to determine which thread is selected for fetching in a given cycle. Thread balancing at the instruction fetch stage ensures fairness between all active threads. Thread priority is also used to select instructions from one thread in the IBUFs and send to group formation and decode logic. A maximum of four nonbranches and two branches from one thread can form one group of instructions.

Subsequently, register renaming is done using mappers before instructions are placed in the issue queues. General purpose register (GPR) and vector/scalar register (VSR) are mapped onto 80 rename registers, matching the maximum number of nonbranch instructions between dispatch and completion. The GPR file is implemented as two physical copies of 80 entries each. POWER7 employs a single issue queue for floating-point, fixed-point and load/store instructions, the 48-entry Unified Issue Queue (UQ). It is implemented as two halves for area and power conservation reasons.

The Global Completion Table (GCT) is responsible for tracking all in-flight instruc-

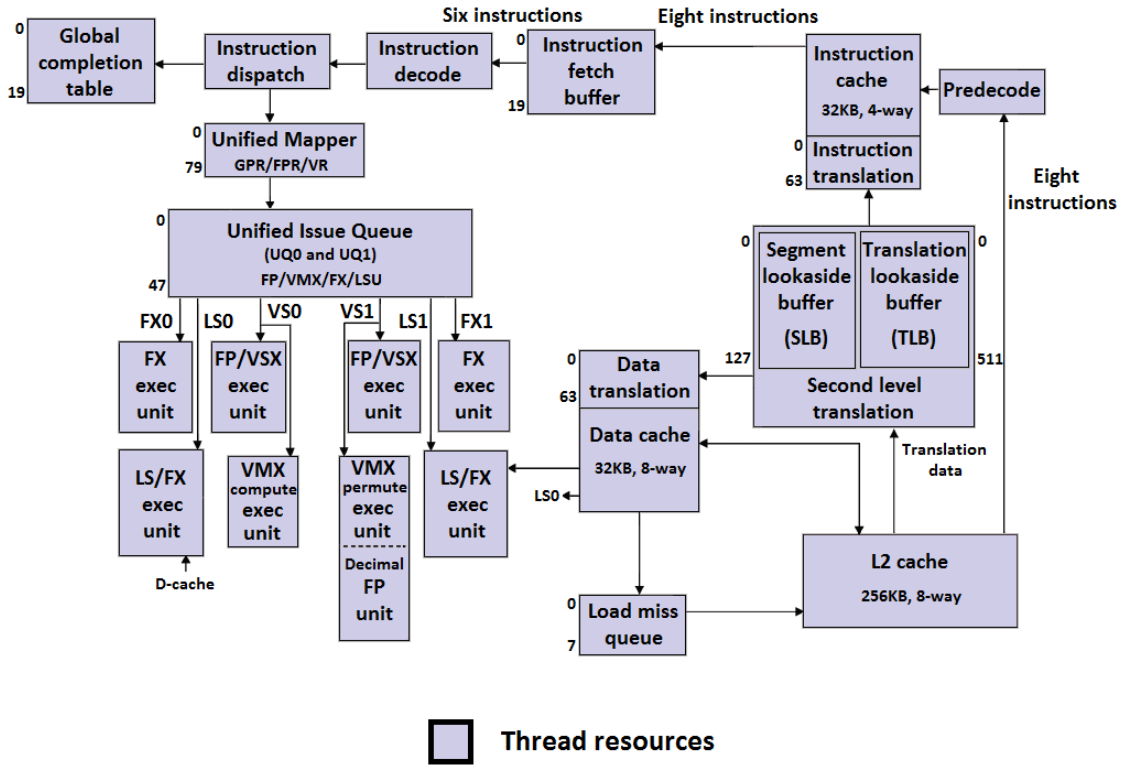


Figure 3.2: Core resource sharing in ST mode [27].

tions after dispatch. Instructions in the core are tracked as groups of instructions and, thus, will dispatch and complete as a group. The GCT has 20 entries and each entry corresponds to a group of instructions. So the maximum capacity of the GCT at any cycle is 120 instructions.

Finally, instructions are executed in the functional units, which include a Fixed-Point Unit (FXU), a Load-Store Unit (LSU) and a Vector and Scalar Unit (VSU). The Fixed-Point Unit is made of two identical pipelines (FX0 and FX1) and is responsible for executing all fixed-point instructions, including adds, subtractions, compares, multiplications, divides, etc. The Load-Store Unit is integrated as two symmetric pipelines (LS0 and LS1), with each capable of executing a load or store operation in a cycle. The LSU has the added capability of executing simple fixed-point operations. The Vector and Scalar Unit (VSU) incorporates both the floating-point unit (FPU), and the vector media extension (VMX) unit, in order to save area and power. The VSU is also made of two pipes (VS0 and VS1) with most VSU operations being able to execute on any of the two pipelines with some exceptions, as seen in Figure 3.2.

In the PowerPC architecture, programs are written using 64-bit effective addresses,

while 46-bit real addresses are used for addressing in the memory hierarchy system. The address translation is performed on two levels in the POWER7. On the first level two 64-entry Data effective-to-real-address translation (D-ERAT) and one 64-entry Instruction effective-to-real-address Translation (IERAT) caches are provided. If translation misses on the first level, a second level of translation is invoked to compute the desired address. The second level of translation involves a 32 entry-per-thread Segment Lookaside Buffer (SLB) cache and a 512 entry Translation Lookaside Buffer (TLB) cache. The SLB keeps entries of translation from effective to virtual addresses, while the TLB is responsible for translation from virtual to real addresses.

3.3 Resource Sharing Levels

POWER7 as a hybrid CMP and SMT architecture shares resources on multiple levels. A major characteristic of the POWER7 is that it is an architecture with two clustered execution pipelines in each core, from now on referred to as clusters. Contexts 0 and 1 are found in the first cluster, while Contexts 2 and 3 belong to the second cluster. Similarly to Sun's UltraSparc T2, we observe that on POWER7 more resources are shared between threads in the same cluster than threads in different clusters. However in POWER7 some resources are shared between threads in different clusters as well, providing an added level of resource sharing. Thus it is the first processor to share core resources on 4 different levels:

- *Inter-Core*: resources are shared by all threads even in different cores, including the global L3 cache and the memory controllers.
- *Intra-Core*: this resource-sharing level features the resources shared by all threads within a core, such as the L1 data and instruction caches, the L2 cache, the local L3 cache, the GCT and the TLB.
- *Intra-Cluster*: this level consists of all the resources shared between threads in the same Cluster (i.e. Contexts 0-1 and Contexts 2-3), and includes the GPR files, the two UQ halves, and the functional units (FXU/LSU).
- *Inter-Cluster*: formed by the resources that are only shared between two threads in different clusters and specifically between Contexts 0-2 and Contexts 1-3, which are the IERAT and the GCT completion bandwidth.

All in all, two threads running in the same cluster conflict in the Intra-Cluster, Intra-Core and Inter-Core levels. Two threads running in different clusters and Contexts 0-2

or Contexts 1-3 conflict in the Inter-Cluster, Intra-Core and Inter-Core levels, while two threads running in different clusters and Contexts 0-3 or Contexts 1-2 conflict only in the Intra-Core and Inter-Core levels. Finally, two threads running in different cores conflict only on the Inter-Core level.

3.4 Core SMT Modes

Depending on how running threads are bound to the four available contexts, the processor operates in different modes. The core mode changes the way resources are allocated to the executing threads. Every core supports three different SMT modes, depending on the placement of threads within it:

- When a single thread runs and it is bound to Context 0, the processor executes in *ST mode*.
- When Context 1 is active and Contexts 2 and 3 are not active (regardless of Context 0) the core runs in *SMT2 mode*.
- When Contexts 2 or 3 are active (regardless of Contexts 0 and 1) the processor runs in *SMT4 mode*.

Because of that distinction, hardware threads are named as following: primary thread (Context 0), secondary thread (Context 1) and tertiary threads (Contexts 2, 3) [1]. POWER7 allows dynamic SMT mode switches with low overhead between them.

Figure 3.2 shows the basic units of the POWER7 core pipeline, when all resources are available to a single thread (ST mode). Next, Figure 3.3 depicts the distribution of resources to two running threads when the core is switched to SMT2 mode. Finally, Figure 3.4 presents the sharing of core resources by 4 threads when SMT4 mode is activated. All three figures are modifications of a figure appearing in [27]. Since SMT2 and SMT4 core modes can be activated with varying numbers of running threads, in these figures we assume the default thread numbers, i.e. 2 threads in SMT2 and 4 threads in SMT4 mode. Table 3.3 complements the figures, as it shows resource distribution in all possible thread numbers and placements.

In POWER7, rename registers other than the GPRs are shared between all threads. The same happens with the TLB. Whereas there are several microarchitectural resources shared between threads, we focus on those mentioned in [27]:

1. Front-end: In ST and SMT2 modes, each thread uses a 16-entry link stack. In the SMT4 mode, each thread uses an 8-entry link stack. The IBUF holds up to 20

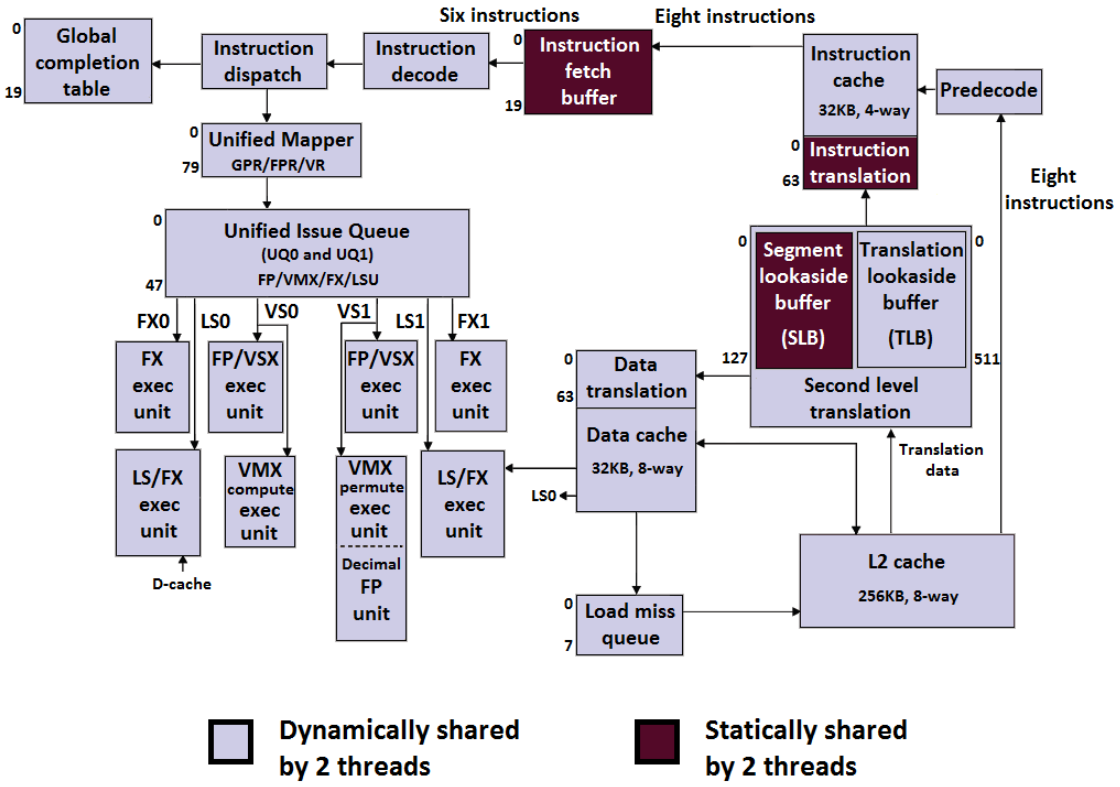


Figure 3.3: Core resource sharing between 2 threads in SMT2 mode [27].

entries, each 4-instructions wide. In SMT4 mode, each thread can have 5 entries, whereas in ST and SMT2 modes, a thread has ten entries.

2. Execution Pipelines: In ST and SMT2 modes, the two GPR files maintain identical contents, so there are a total of 80 rename registers; instructions can be issued to any of the two UQ halves and subsequently be executed in any of the functional units. In SMT4 mode the two GPR files have different contents and threads in Contexts 0-1 can execute in the upper pipeline (UQ0, FXU0/LSU0), while threads in Contexts 2-3 execute in the lower pipeline (UQ1, FXU1/LSU1). The majority of vector and scalar operations can be issued to any of the two UQ partitions and therefore be executed in any of the two VSU pipes, with the following exceptions: a) VMX floating point and simple and complex integer instructions are only dispatched to UQ0 and execute in VS0, b) permute (PM), decimal floating point and 128 bit store operations are only dispatched to UQ1 and execute in VS1.

3. Address translation: In the instruction-fetch stage the IERAT table supports Contexts 0 and 2 in the first 32 entries, and Contexts 1 and 3 in the last 32 entries. The

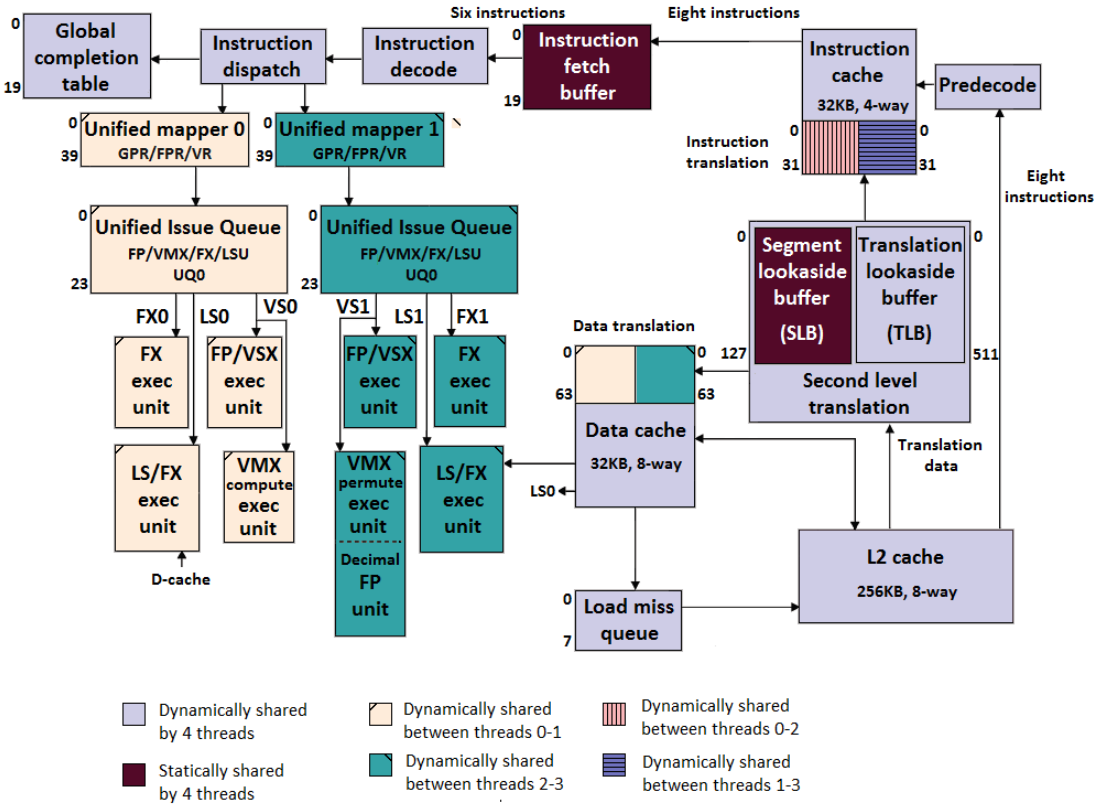


Figure 3.4: Core resource sharing between 4 threads in SMT4 mode [27].

D-ERAT consists of two 64-entry caches. In ST and SMT2 modes the two halves have identical contents, but in SMT4 mode they have different contents with one half dynamically shared between Contexts 0-1 and the other between Contexts 2-3. The TLB is dynamically shared by all four threads.

4. GCT: the 20 entries of the GCT that track groups of 8 instructions are dynamically shared between all four threads. The processor can complete one group per thread pair per cycle, with Contexts 0 and 2 forming one pair, and Contexts 1 and 3 forming the other one.

3.5 Software-controlled Hardware Thread Priorities in the IBM POWER Family Processors

In Power7, just like in previous PowerPC processors, there is a prioritization mechanism that sets the number of decode cycles assigned to each thread through software

control. The enforcement of these software-controlled priorities is carried by hardware in the decode stage. In POWER5 and POWER6, which were only capable of running 2 threads in a core, priorities were applied by altering the decode rate of the primary thread at the expense of the secondary thread. The application of priorities was only based on the priority difference between the two threads [11]. The decode slots of the two threads were allocated exponentially to the difference of priorities between the threads, using the following formula:

$$R = 2^{|PrioP - PrioS| + 1}$$

So the primary thread would receive R-1 decode slots and the secondary thread would receive the remaining slot. For example, for a priority difference of one, the primary thread would receive 3 decode slots, for a priority difference of two, the primary thread would receive 7 decode slots and so forth with the secondary thread receiving the remaining slot in all cases. Since there is no published information about how the priority mechanism in POWER7 works, we treat it as a black box.

Table 3.2: Thread priorities in Power7

Priority	**Priority level**	**Privilege level**	**or-nop inst.**
0	Thread shut off	Hypervisor	-
1	Very low	Supervisor	or 31,31,31
2	Low	User/Supervisor	or 1,1,1
3	Medium-Low	User/Supervisor	or 6,6,6
4	Medium	User/Supervisor	or 2,2,2
5	Medium-High	Supervisor	or 5,5,5
6	High	Supervisor	or 3,3,3
7	Very High	Hypervisor	or 7,7,7

The software-controlled priorities range from 0 to 7, where 0 means the thread is switched off and 7 means the thread is running in Single Thread mode. Also, priority 1 has the effect of executing the thread in low-power mode. The supervisor or OS can set six of the eight priorities ranging from 1 to 6, while user software can only set priority 2, 3 and 4. The Hypervisor can use the whole range of priorities. Priorities can be set by issuing a pseudo `or` instruction in the form of `or X,X,X` where `X` is a specific register number [11][14]. This operation only changes the thread priority and performs no other operation. If it is not supported or not permitted the instruction is simply treated as a `nop`. Table 3.2 shows the priorities, the privilege level required to set each priority and the corresponding instruction. The behavior of the prioritization mechanism changes when

priorities 0 or 1 are applied to a thread [11][14]. When both threads have priority one the processor runs in low-power mode and it decodes only one instruction every 32 cycles.

Table 3.3: Resource distribution under different core modes

thread binding	running threads	core mode	Main available Resources				
			GPR ren.regs.	UQ	Execution Units	IBUF	I-ERAT/D-ERAT
[AX][XX]	1	ST	40	48	FX0/LS0/VS0 FX1/LS1/VS1	10	32 / 64
[XA][XX]	1	SMT2	40	48	FX0/LS0/VS0 FX1/LS1/VS1	10	32 / 64
[XX][AX]	1	SMT4	40	24	FX1/LS1/VS0,1	5	32 / 64
[XX][XA]	1	SMT4	40	24	FX1/LS1/VS0,1	5	32 / 64
[AB][XX]	2	SMT2	40 (shAB)	48 (shAB)	FX0/LS0/VS0 (shAB) FX1/LS1/VS1 (shAB)	10 each	32 each / 64(shAB)
[AX][BX]	2	SMT4	40 each	24 each	FX0/LS0/VS0,1 (A) FX1/LS1/VS0,1 (B)	5 each	32(shAB) / 64 each
[AX][XB]	2	SMT4	40 each	24 each	FX0/LS0/VS0,1 (A) FX1/LS1/VS0,1 (B)	5 each	32 each / 64 each
[AB][CX]	3	SMT4	40 (shAB) 40 (C)	24 (shAB) 24 (C)	FX0/LS0/VS0,1 (shAB) FX1/LS1/VS0,1 (C)	5 each	32(shAC) - 32(B)/ 64(shAB) - 64(C)
[AB][XC]	3	SMT4	40 (shAB) 40 (C)	24 (shAB) 24 (C)	FX0/LS0/VS0,1 (shAB) FX1/LS1/VS0,1 (C)	5 each	32(A) - 32(shBC)/ 64(shAB) - 64(C)
[AB][CD]	4	SMT4	40 (shAB) 40 (shCD)	24 (shAB) 24 (shCD)	FX0/LS0/VS0,1 (shAB) FX1/LS1/VS0,1 (shCD)	5 each	32(shAC) - 32(shBD)/ 64(shAB) - 64(shCD)

Chapter 4

Experimental Setup

We use an IBM PS701 BladeCenter to run our experiments, which contains a single IBM POWER7 processor. The system runs SUSE Linux Enterprise 10 SP2 and the kernel we use is version 3.0.9.

4.1 Performance Counters

In most modern high performance processor systems special hardware registers are provided for accurate CPU measurements without affecting the executing workload or slowing down the CPU [26][30]. Initially, they were used for hardware debugging but later they became used widespread for performance monitoring. Usually these counters are programmable, but other times they support a fixed-function. Programmable counters can be enabled or disabled and they can be configured to count different types of events. Usual countable events include the number of committed instructions, clock cycles, cache misses, or branch mispredictions. Fixed-function counters provide limited programmability, since they always count the same event, or they cannot be disabled. The type and number of countable events depends greatly on the different processors micro-architectures. In case there are more events to count than there are counters, the kernel uses time multiplexing to allow each event to access the monitoring unit. With multiplexing the final result is scaled at the end of execution based on total time counted with respect to execution time.

4.1.1 Performance Counter types

There are five groups in which processor events can be grouped into: program characterization, memory accesses, pipeline stalls, branch prediction, and resource utilization. Firstly, program characterization events are used to define the attributes of a program

or the OS and therefore are independent of the processor"s micro-architecture. Common examples of such events are the number and type of instructions executed by the program, such as loads, stores, floating point, branches, etc. Secondly, memory access events facilitate the study of the processor's memory hierarchy and often constitute the largest event category. Some examples in this category are references and misses to the various levels of cache and traffic on the processor memory bus. Thirdly, the information coming from pipeline stall events aids the analysis of a program's instruction flow through the pipeline and help in locating bottlenecks. Fourthly, branch prediction events assist users in analyzing the performance of branch prediction hardware, like counting mispredicted branches. Finally, resource utilization events provides useful information on how often a processor uses certain resources, such as the number of cycles spent using a fixed-point multiplier, etc.

4.1.2 Software Support for Hardware Counters

The fact that some of the counter configuration or access instructions require kernel mode privileges, and the need to provide per-thread counts, have led to the development of kernel extensions that allow applications to access the counters in user mode. For Linux, some frequently used kernel extensions are perfmon2 and perf. Perf is included in all recent linux kernel versions.

Since these kernel extensions are specific to an operating system, measurement code using these extensions becomes platform dependent. For that reason Application Programming Interfaces (APIs) are provided to facilitate the access of low-level platform-dependent system calls that are needed to access performance counters. Libpfm is such a user-level API library.

4.1.3 Power7 Counters

The POWER7 processor has an integrated performance monitoring unit (PMU) for each hardware thread, which enables performance monitoring, workload characterization, system characterization and code analysis [3]. There are 6 thread-level Performance Monitor Counters (PMC) in a PMU. PMC1 to PMC4 are programmable, PMC5 counts non idle completed instructions and PMC6 counts non idle cycles. On a thread level and core level, the PMU provides access to a numerous set of performance events (close to 550) that cover essential statistics such as miss rates, unit utilization, thread balance, hazard conditions, translation related misses, stall analysis, instruction mix, L1 Instruction and Data cache reload source, effective cache counts and memory latency counts.

4.2 The Linux Kernel

By default, only three of the eight priority values are available in user mode. The other five can only be accessed by the Hypervisor or the Operating System. Modern kernel versions (after 2.6.23) use the software-controlled priorities in order to reduce performance of a process that does not perform any useful computations. In detail the cases where the priorities mechanism is used are the following:

- When a thread in spinlock is waiting in an infinite loop until the lock becomes available its priority is reduced.
- When the kernel is idle waiting for operations to be completed, such as when it requests a specific CPU to perform an operation through a `smp_call_function()` and it cannot proceed until the operation completes, its priority is reduced.
- In a specific hardware context the kernel is executing the idle thread because there are no useful computations to be scheduled. When the priority of the idle thread is reduced and eventually disabled (priority 0) in order not to waste unused core resources by putting the core in an unwanted core mode.

In all of the above cases the kernel reduces the priority of a hardware thread to medium (4) as soon as a useful job is available for scheduling. Also it does not store the actual priority it resets the thread priority to medium each time it enters a kernel service routine, e.g. an interrupt, an exception handler or a system call. That consists a conservative choice.

The kernel patch we use, provides an interface to the user to set all available priorities available in kernel mode. First of all, since the described usage of priorities could unpredictably affect experiments, we have disabled the use of software-controlled priorities inside the kernel. The main contribution is that priorities 1 to 6 are made available in user mode, by an interface provided through the `/sys` pseudo file system.

4.3 METbench Micro-benchmarks

We use a set of micro-benchmarks to perform an in-depth study of resource sharing in POWER7 under different thread placement setups. Micro-benchmarks are simple and repetitive tasks that stress specific processor parts. Each of the micro-benchmarks features a loop body with specific instructions depending on the behavior we want to achieve. For example some tasks continuously perform integer additions, while others continuously hit

in a specific cache level. The loop body is repeated enough times so that the micro-benchmark runs for at least one second. The micro-benchmarks are organized in three categories based on the type of instructions they execute: integer, floating-point and memory.

- Integer micro-benchmarks include `cpu_int_add`, `cpu_int`, `cpu_int_mul` and `lng_chain_cpuint`. Their loop bodies can be seen in Table 4.1. While `cpu_int` contains mixed integer instructions -specifically one multiplication every two additions-, `cpu_int_add` only contains additions and `cpu_int_mul` only multiplications. `Lng_chain_cpuint`, to which we will refer as `lng_chain` for short, includes mixed integer instructions (also one multiplication every two additions) but is specifically designed to limit ILP with a long dependency chain of instructions.

- In the floating-point micro-benchmarks category we include `cpu_fp_asm` benchmark, which we will refer to as `cpu_fp` for short. It is written in POWER assembly to get more precision over its behavior and it includes floating-point subtractions, additions and multiplications.

- Memory micro-benchmarks include all cache or memory related micro-benchmarks: `ldint_l1`, `ldint_l2`, `ldint_l3` and `ldint_mem`. The standard loop body structure of all `ldint_X` micro-benchmarks can be seen in Table 4.1, where the size of the array M varies to match the desired behavior. A pointer chasing technique is used to implement the desired amount of loads on the relative memory hierarchy level. Specifically, an array is initialized with pointers, so that each element has the address of the next element. The last element contains the address of an element in the beginning of the array in order to execute the loop multiple times. So, `ldint_l1` benchmark uses approximately 25% of the L1 cache, `ldint_l2` benchmark fills the first level and hits in the second level and so on. Since POWER7 includes 8 core-local L3 regions that form a larger global L3 region, we have created two micro-benchmarks to target the L3. `Ldint_l3` fills the first and second level of cache and hits in the local L3, while `ldint_mem` fills all levels of cache and hits in the system memory.

4.3.1 Validation of Micro-benchmarks Behavior

We have used performance counters in single-thread mode to validate the behavior of METbench micro-benchmarks.

First we use resource utilization counters to check which core resources are used by each micro-benchmark and show the results in Figure 4.1. `Cpu_fp` executes 99,7% of its

Table 4.1: Source code of some micro-benchmarks

(a) `cpu_int_add`

```
for (it=0; it<M; it++)
  {
    LOOP_UNROL_20(
    a = a+a+it;
    b = b+b+it;
    c = c+c+it;
    d = d+d+it;
    e = e+e+it;
    f = f+f+it;
    g = g+g+it;
    h = h+h+it;
    i = i+i+it;
   )
  }
```

(b) `cpu_int_mul`

```
for (it=0; it<M; it++)
  {
    LOOP_UNROL_20(
    a = a*a*it;
    b = b*b*it;
    c = c*c*it;
    d = d*d*it;
    e = e*e*it;
    f = f*f*it;
    g = g*g*it;
    h = h*h*it;
    i = i*i*it;
   )
  }
```

(c) `cpu_int`

```
for (it=0; it<M; it++)
  {
    LOOP_UNROL_20(
    a = a+a+it;
    b = b+b+it;
    c = c*c*it;
    d = d+d+it;
    e = e+e+it;
    f = f*f*it;
    g = g+g+it;
    h = h+h+it;
    i = i*i*it;
   )
  }
```

(d) `lng_chain`

```
for (it=0; it<M; it++)
  {
    LOOP_UNROL_20(
    a = a+i;
    b = b+a;
    c = c*b;
    d = d+c;
    e = e+d;
    f = f*e;
    g = g+f;
    h = h+g;
    i = i*h;
   )
  }
```

(e) `ldint_X`

```
for (it=M; it>0; it--)
  {
    LOOP_UNROL_20(
    p = *p;
   )
  }
```

instructions in the VSU. `Cpu_int`, `cpu_int_add`, `cpu_int_mul` and `lng_chain` use the FXU or the LSU in a percentage of at least 99,7%. The Load-Store pipelines in POWER7 can complete simple fixed-point operations, like adds and logical instructions [27], which explains why 27,8% of `cpu_int_add` and 16,8% of `cpu_int` instructions are executed in the LSU. Micro-benchmarks in the memory category mostly execute in the LSU with a percentage of at least 94%. The remaining percentage of instructions are branches and fixed-point operations, which are necessary for their function.

Additionally, we check memory access counters to ensure that memory micro-benchmarks fetch data from the correct level of memory hierarchy. Indeed `ldint_l1`, `ldint_l2` and `ldint_l3` fetch at least 99% of their data from L1, L2 and L3 local respectively. `Ldint_mem` fetches 87% of its data from memory and the rest from L1.

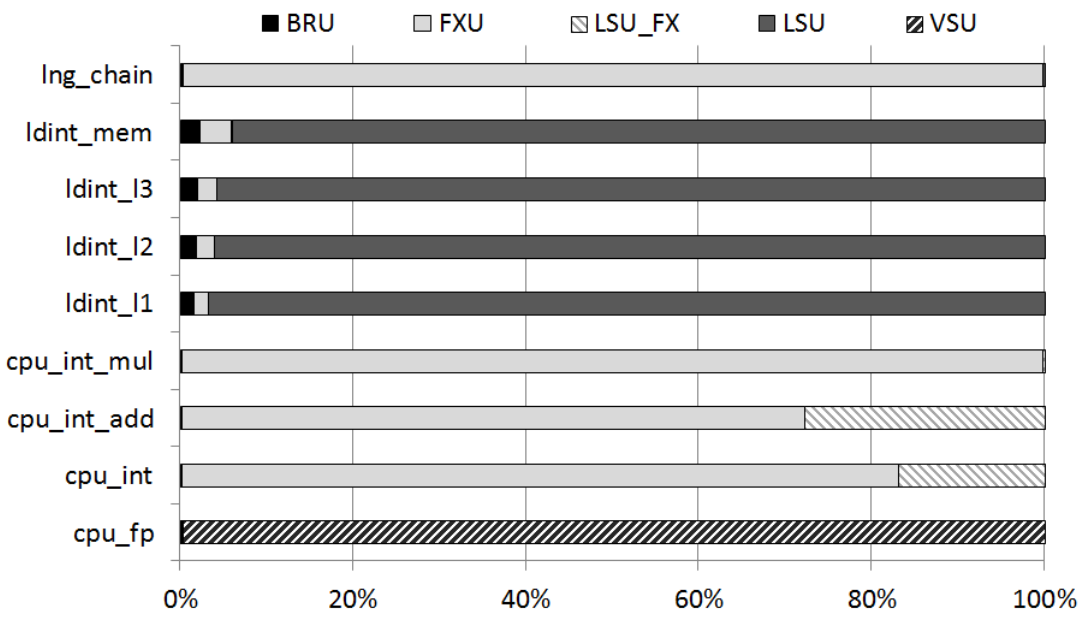


Figure 4.1: Resource-usage for METbench micro-benchmarks.

4.3.2 FAME Methodology

To obtain reliable measurements in the characterization of the POWER7 processor, we use the FAME (FAirly MEasuring Multithreaded Architectures) methodology [38]. This methodology ensures that every program in a multiprogrammed workload is completely represented in the measurements. For that reason, the methodology advises to re-execute once and again one program in the workload until the average accumulated IPC of that program is similar to the IPC of that program when the workload reaches a steady state. FAME determines how many times each benchmark in a multi-threaded workload has to be executed so that the difference between the obtained average IPC and the steady state IPC is below a particular threshold. This threshold is called MAIV (Maximum Allowable IPC Variation). The execution of the entire workload stops when all benchmarks have executed as many times as needed to accomplish a given MAIV value. For the benchmarks used in this paper, in order to accomplish a MAIV of 1%, each benchmark must be repeated at least 5 times. METbench applies the FAME methodology in micro-benchmark executions.

4.3.3 METbench Modifications

Anteriorly, METbench only supported the perfmon2 counter interface to draw information on counters. However perfmon2 has become obsolete and instead perf has taken its place as the default linux kernel extension. Perf is included in all new linux kernel ver-

sions. As part of this study perf was integrated into the METbench benchmark suite with the help of libpfm4. Libpfm is an API in the form of a library that provides functions to access the desired system calls that are necessary for accessing performance counters. So METbench was enhanced and given the potential to get performance counter information for each thread and for the whole system.

4.4 PARSEC Benchmark Suite

With the prevalence of CMP processors and the continuous trend to parallel application programming, comes the need for benchmark programs that are representative of the current and future real-world applications. Such a benchmark suite is PARSEC [4]. It features state-of-the-art, computationally intensive algorithms and very diverse workloads from different areas of computing, such as computational finance, computer vision, real-time animation or media processing.

4.4.1 Benchmarks Presentation

PARSEC is comprised of 13 benchmark programs, from which we use ten[1]: `blackscholes`, `bodytrack`, `dedup`, `ferret`, `fluidanimate`, `freqmine`, `streamcluster`, `swaptions`, `vips`, and `x264`.

`Blackscholes`: This application is an Intel RMS benchmark. It calculates the prices for a portfolio of European options analytically with the Black-Scholes partial differential equation (PDE). There is no closed-form expression for the Black-Scholes equation and as such it must be computed numerically.

`Bodytrack`: This computer vision application is an Intel RMS workload which tracks a human body with multiple cameras through an image sequence. This benchmark was included due to the increasing significance of computer vision algorithms in areas such as video surveillance, character animation and computer interfaces.

`Dedup`: This kernel was developed by Princeton University. It compresses a data stream with a combination of global and local compression that is called 'deduplication'. The kernel uses a pipelined programming model to mimic real-world implementations. The reason for the inclusion of this kernel is that deduplication has become a mainstream method for new generation backup storage systems.

`Ferret`: This application is based on the Ferret toolkit which is used for content-based similarity search. It was developed by Princeton University. The reason for the inclusion

[1]We encountered compilation or execution errors with `canneal`, `facesim`, `raytrace`

in the benchmark suite is that it represents emerging next-generation search engines for non-text document data types. In the benchmark, we have configured the Ferret toolkit for image similarity search. Ferret is parallelized using the pipeline model.

`Fluidanimate`: This Intel RMS application uses an extension of the Smoothed Particle Hydrodynamics (SPH) method to simulate an incompressible fluid for interactive animation purposes. It was included in the PARSEC benchmark suite because of the increasing significance of physics simulations for animations.

`Freqmine`: This application employs an array-based version of the FP-growth (Frequent Pattern-growth) method for Frequent Itemset Mining (FIMI). It is an Intel RMS benchmark which was originally developed by Concordia University. Freqmine was included in the PARSEC benchmark suite because of the increasing use of data mining techniques.

`Streamcluster`: This RMS kernel was developed by Princeton University and solves the online clustering problem. Streamcluster was included in the PARSEC benchmark suite because of the importance of data mining algorithms and the prevalence of problems with streaming characteristics.

`Swaptions`: The application is an Intel RMS workload which uses the Heath-Jarrow-Morton (HJM) framework to price a portfolio of swaptions. Swaptions employs Monte Carlo (MC) simulation to compute the prices.

`Vips`: This application is based on the VASARI Image Processing System (VIPS) which was originally developed through several projects funded by European Union (EU) grants. The benchmark version is derived from a print on demand service that is offered at the National Gallery of London, which is also the current maintainer of the system. The benchmark includes fundamental image operations such as an affine transformation and a convolution.

`X264`: This application is an H.264/AVC (Advanced Video Coding) video encoder. H.264 describes the lossy compression of a video stream and is also part of ISO/IEC MPEG-4. The flexibility and wide range of application of the H.264 standard and its ubiquity in next-generation video systems are the reasons for the inclusion of x264 in the PARSEC benchmark suite.

4.4.2 Applications Characterization

We use performance counters to characterize the core resource usage of PARSEC applications and present the results in Figure 4.2. We notice that real world applications like PARSEC benchmarks, have balanced instruction mixes in general. We notice that `dedup` and `freqmine` do not include any vector or floating point operations. Also, we can

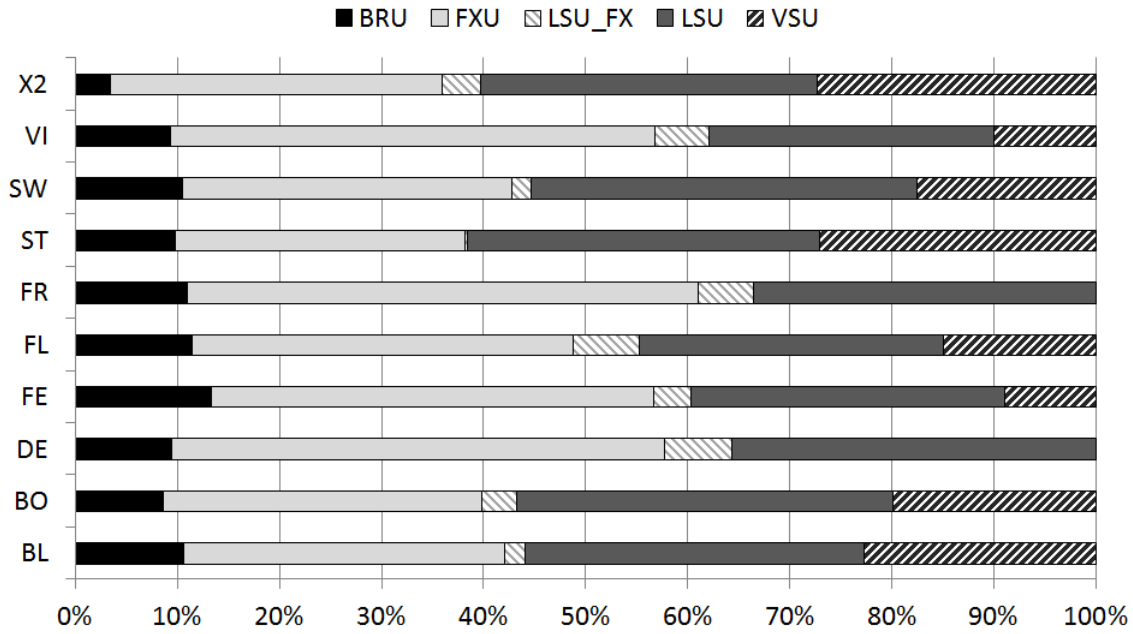


Figure 4.2: Resource usage for PARSEC applications.

see that `vips`, `freqmine` and `dedup` have more than 50% fixed-point instructions in their instruction mix (executing in the FXU or the LSU).

Subsequently, we checked memory access counters to gain information about the memory hierarchy usage of each of the PARSEC applications. We found that `streamcluster`, which is a streaming application, executes the highest amount of loads than any other application, especially from memory.

4.4.3 Threading Models

PARSEC 2.0 supports three different parallelization models: Pthreads, OpenMP and TBB. Firstly, POSIX threads (Pthreads) [6] is one of the most commonly used parallelization models in shared memory machines that lets programmers handle all thread creation, management and synchronization issues. It is supported by all PARSEC workloads except `freqmine`. Secondly, OpenMP [9] is a shared memory compiler-based parallelization model, with which the programmer can give directives to the compiler to execute sections of code in parallel and all the details of the thread management and synchronization are handled in runtime. It is supported by `bodytrack`, `freqmine` and `blackscholes`. Finally, the Intel Thread Building Blocks (TBB) [25] is a high-level alternative to pthreads which provides functions that express task-based parallelism while hiding the details of the platform and the threading mechanism. PARSEC gives TBB support to `blackscholes`,

`bodytrack`, `fluidanimate`, `streamcluster` and `swaptions`. We make use of Pthreads parallelization model for all these benchmarks, except for the case of `freqmine`, which is only available in OpenMP. (connection pthreads, tbb to task-parallel data parallel)

PARSEC benchmarks use various parallelization approaches, including data parallelization and task parallelization. Data parallelization is common for many benchmark applications for years [13], however task parallelization has seen limited benchmarking support. Task parallelization and especially the pipelining programming model are extensively supported by PARSEC. Pipelining involves breaking the work into stages and executing them concurrently. Not only does pipelining simplify large-scale application programming, but can also increase throughput with exploiting parallelism in a higher level due to the concurrently executing nature of the various stages. Data-parallelism is used within the different pipeline levels to further exploit parallelism. The benchmarks that are programmed with the pipeline data parallelization model are `dedup`, `ferret` and `x264`. The rest of the benchmarks are programmed solely with a data-parallelization approach, with the exception of `canneal` that follows an unstructured approach.

4.4.4 Input Sizes

There are several input sizes for all benchmarks, including Test, Simdev, Simsmall, Simmedium, Simlarge and Native. The test size has a minimal input size and only serves to verify that programs are executable, with an execution time of less than 1 second. Simdev to Simlarge input sizes are provided for microarchitectural simulators, with execution times ranging from 1 second up to 20 seconds. Native inputs are large-scale experiments intended for real machines, with execution times of up to 30 minutes. We used native input sets for all the experiments, as they are the most representative for real-world applications.

Chapter 5

Performance Evaluation with Micro-Benchmarks

We use METbench micro-benchmarks to gain insight on the resource-sharing of the IBM POWER7 and draw conclusions regarding the issues of thread placement and thread priorities.

5.1 Thread Placement Performance Characterization

5.1.1 Thread Placement of a Single-Thread Micro-Benchmark

In previous multicore and multithreaded processors, when a task is executed in isolation, its performance is independent of the particular context to which it is bound. For example, for the IBM POWER5 and POWER6 that are 2-core, 2-thread SMT processors, the performance of a task is the same regardless of which of the four contexts it runs in. This is not the case for the IBM POWER7. The binding of a single thread in one of the four core contexts dynamically switches the core mode between ST, SMT2 and SMT4 and leads to different resource allocation. We expect that the more resources a thread has at its disposal the better it performs.

Figure 5.1 shows the performance of the different micro-benchmarks when they run in isolation in a core and are bound to different hardware threads, while disabling the unused contexts. 'X' means that the context is disabled and 'A' denotes the micro-benchmark under consideration runs in that context. Hence, 'XAXX' means that the task is bound

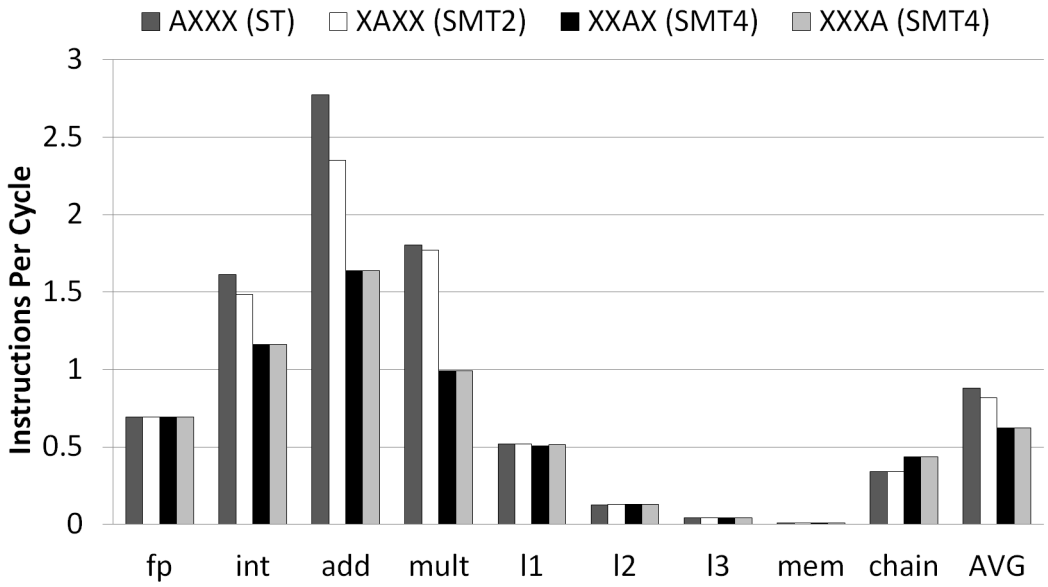


Figure 5.1: IPC of single-thread micro-benchmarks when they are bound to a given context, while the other contexts are disabled.

to Context 1 (i.e. secondary thread) while Contexts 0, 2 and 3 are disabled.

We observe that when executing a single thread, binding it to different core contexts can have an important role in its performance. For some benchmarks the variation is small while for others it is significant (70% for the case of the `cpu_int_add` benchmark). On average the maximum performance is obtained when the task is bound to Context 0, with the second highest being when the task is bound to Context 1 and the lowest performance equally seen when the task is bound to Context 2 or 3.

When the benchmark is bound to Context 0 and the other hardware threads are disabled, the core runs in ST mode. Under this mode, the contents of the register files in each cluster are duplicated. This allows the benchmark to access both partitions of the Unified Queue (UQ) and subsequently issue instructions to any of the functional units. A thread has access to 10 out of the 20 entries of the Instruction Buffer, 112 physical general purpose registers -from which 32 are architectural registers and the remaining 80 are used as rename registers, all 48 UQ entries and 2/2/2 functional units, meaning 2 fixed-point pipelines (FX), 2 load-store pipelines (LS) and 2 vector-scalar pipelines (VS).

When a task is bound to Context 1, with Context 2 and 3 inactive, the core runs in SMT2 mode. We notice in Table 3.3 that under this mode the resources that the task can use are in general equal to the resources available in ST mode. However, we observe that some benchmarks obtain less performance in this mode than in ST mode. We further observe that the longer the latency of the executed instructions in a given task the lower

the performance degradation that task suffers in SMT2 mode: 15%, 8% and 2% for `cpu_int_add`, `cpu_int` and `cpu_int_mul` benchmarks, respectively. From the description of the POWER7 [27] it was not clear why the pipeline of the POWER7 core is causing this behavior. We have two possible explanations for this behavior. It could be the case that in ST mode the task in Context 0 benefits from some type of prioritization that the task does not suffer when it runs in Context 1 under mode SMT2. It could be also possible that a number of resources are partitioned in the cluster so that when the processor runs in SMT2 mode, some of them are reserved for Context 0 even if there is no task running.

When the task is bound to Context 2 or 3, the core runs in SMT4 mode. In this mode the GPR files have different contents and a thread can only access one UQ half (24 entries) and issue instructions to one fixed-point pipeline (FX1), one load-store pipeline (LS1), but two vector-scalar pipes (VS0,1). Also, the thread in this mode can only use 5 Instruction Buffer entries, whereas in the other modes it could use 10 entries.

The `cpu_fp` benchmark does not show any degradation in performance between the different modes. This is because in all SMT modes most Vector and Scalar operations including floating-point can be dispatched to any of the two UQ halves and then executed on any of the two vector-scalar units, as cited in [27]. Therefore it is not affected by the placement of the thread and consequently the core mode.

The `lng_chain` benchmark has a counter-intuitive behavior as it improves performance in SMT4 mode in comparison to ST and SMT2 modes. In SMT4 mode, each thread is confined to a UQ half and in that case data dependent operations can be issued back-to-back [27]. Meanwhile, in ST and SMT2 modes, each thread is given access to both UQ partitions and so dependent operations can be issued to different partitions, suffering one cycle delay to bypass data from on cluster to the other. Table 4.1 shows that `lng_chain` is comprised of many inter-dependent instructions that greatly limit its ILP, thus limiting its performance to 3.7x slower than `cpu_int`.

All single-thread versions of cache and memory bound benchmarks including `ldint_l1`, `ldint_l2`, `ldint_l3` and `ldint_mem` are not affected by the thread placement. Consecutive instructions of the `ldint_X` benchmark cannot be run in parallel since the pointer chasing technique is used, as explained in Section 4.3. The performance of this category of micro-benchmarks is dominated by the latency of the relative level of memory hierarchy that they access.

To sum up, when a single thread runs in a POWER7 core, binding it to Context 0 provides the best results, unless it includes significant data dependencies, in which case Context 2 or 3 provides the best performance.

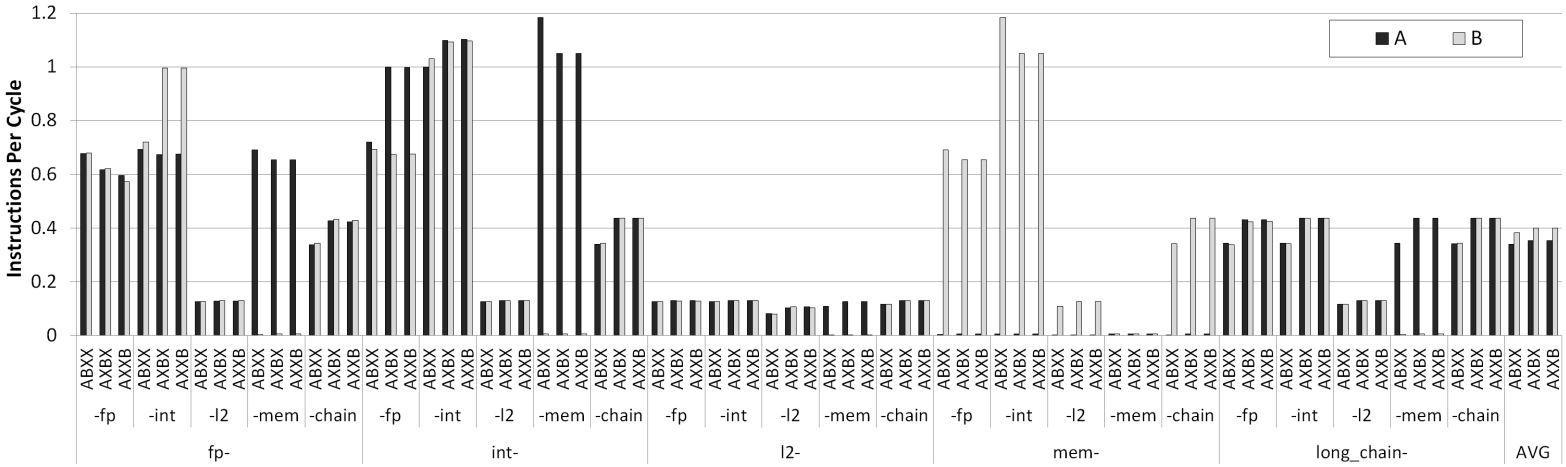


Figure 5.2: IPC of different pairs of single-thread micro-benchmarks when bound to two given contexts.

5.1.2 Thread Placement of 2 Single-Thread Micro-Benchmarks

In this case we run two different threads under different thread assignments. We investigate how the POWER7 core performs with 2 running threads in SMT2 and SMT4 mode. Figure 5.2 shows the results of five representative micro-benchmarks, as well as the average values for the whole METbench suite. In each of the 5 sets of bars of the graph we show one of the five micro-benchmarks against all the other. For example if we take the bars for fp- in the graph and then focus on the -int subset of bars we get the results for the `cpu_fp-cpu_int` pair, where `cpu_fp` is A and `cpu_int` is B in [AB][XX] (SMT2), [AX][BX] (SMT4) and [AX][XB] (SMT4) thread placements.

We can clearly see the symmetry in performance between the various thread pairs, for example the pair `cpu_int-cpu_fp` results in the same performance for both micro-benchmarks, as that of `cpu_fp-cpu_int` in all three configurations. We conclude that *in a specific core mode* with 2 running threads the POWER7 core has symmetric behavior. In any case, configurations that change the core mode cannot possibly achieve similar performance, for instance [AB][XX] (SMT2) and [XX][AB] (SMT4 with two threads sharing resources in one cluster).

We notice that the two SMT4 configurations have similar performance in all cases, which leads us to the conclusion that the micro-benchmarks do not cause significant contention in the IERAT or the GCT completion bandwidth which are shared between hardware threads 0-2 and 1-3.

We observe on Table 3.3 that in SMT2 mode both threads *dynamically* share all core resources, while in SMT4 mode each of the 2 running threads is statically assigned half of most resources, like UQ entries and functional units. However, in SMT4 mode each pair has access to more rename registers. Whereas some pairs can benefit from dynamic resource allocation to more resources, others gain from the extra rename registers and would rather be confined in a cluster with half the core resources to themselves. Micro-benchmarks with low-latency instructions and a constant need for rename registers, such as `cpu_int`, suffer from the lack of rename registers in SMT2 and so generally perform better in SMT4 mode.

Moreover, there are some additional parameters to consider. We can clearly see how some low-IPC benchmarks, like `ldint_l2` and `lng_chain` limit the IPC of their co-runner. Counter-intuitively, the behavior of `ldint_mem` when co-executing with another task is entirely different to `ldint_l2`, even though its IPC is significantly lower. This is due to the existence of a hardware mechanism that automatically detects LLC (Last Level Cache) misses and reduces the rate at which instructions from the LLC-missing thread

are fetched. As a result, the co-runners of `ldint_mem` manage to reach an IPC close to their peak while the same is barely affected since its performance depends primarily on the latency of memory access. This mechanism is enabled both in SMT4 and SMT2 mode, but in the latter the co-runners of `ldint_mem` have access to twice the resources and in that case they have more room for improvement when this prioritization is applied.

The only benchmark that performs better against `ldint_mem` in SMT4 rather than SMT2 is `lng_chain`. In fact `lng_chain` performs better in SMT4 mode against all other micro-benchmarks since as we have already seen in single-thread placement that it favors being limited to a UQ half.

To conclude, based on our characterization we can choose the optimal configuration between SMT2 and SMT4 modes. Two different threads perform better in SMT2 mode ([AB][XX]) than SMT4 mode ([AX][BX] or equally [AX][XB]), assuming none of them executes a high percentage of data-dependent or low-latency operations (e.g. integer adds).

5.1.3 Thread Placement of 2 Two-Thread Micro-Benchmarks

In this experiment we co-execute 2 two-thread micro-benchmarks in a POWER7 core. We expect the best performance in the placement that causes the least amount of resource sharing between threads of the same type.

We have tried all possible pairs and in Figure 5.3 we show a representative selection of five of those pairs, in addition to the average values for all pairs. With a total of four active threads, the only possible core mode is SMT4. If we consider the core symmetry in a specific core mode, the possible placements are the following three: [AA][BB], [AB][AB] and [AB][BA]. We confirm that the symmetric placements [BB][AA], [AB][AB] and [BA][AB] do not show any difference in performance to the three previous configurations. In each of the five sections of the graph we show the IPC of each micro-benchmark against some of the other four as well as against itself, marked in the horizontal bar as 'AAAA'.

For the observed micro-benchmarks, we confirm that the placement [AB][BA] provides the best overall results (12% performance improvement w.r.t. the worst case), followed closely by [AB][AB]. The small differences between the two configurations could be explained by conflicts in the I-ERAT and the GCT, which are shared between Contexts 0-2 and Contexts 1-3 and since in [AB][AB] these contexts run threads of the same kind. Also, in the [AA][BB] placement -for which we notice the worst results for all three cases, by putting two threads of the same kind in each of the two clusters we are forcing them to contend for the same intra-cluster resources, including rename registers, UQ entries and functional units.

The instruction fetch unit in the POWER7 tries to balance instruction fetch rates

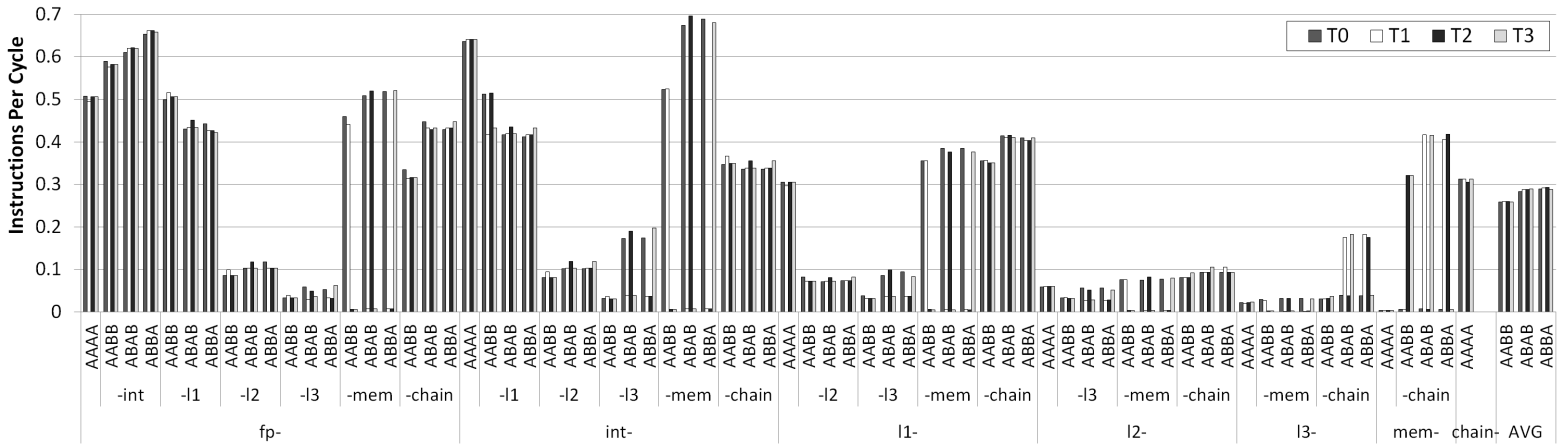


Figure 5.3: IPC of different pairs of two-thread micro-benchmarks when bound to four given contexts.

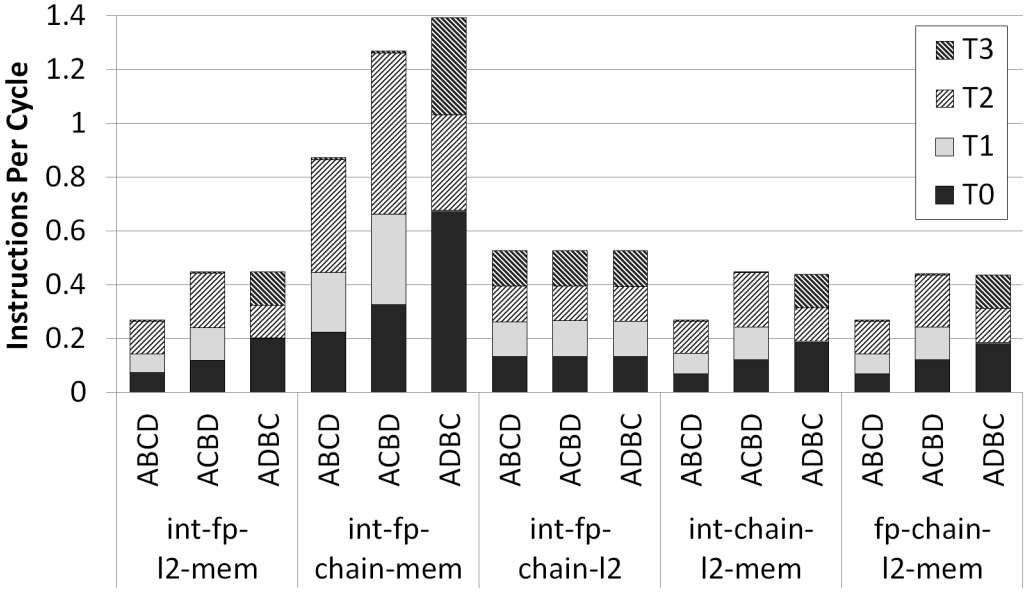


Figure 5.4: IPC of several combinations of 4 single-thread micro-benchmarks.

between threads. According to our results, this happens even between those in different clusters. Consequently, applications with very low-IPC, such as `ldint_l2`, significantly deteriorate the performance of all their co-runners in all three configurations. However, the placement of two cache-bound threads in the same cluster results in further slowdown caused by resource conflicts. Similarly to the results for 2 threads, we notice that `ldint_mem` is a good co-runner due to the hardware mechanism that prevents LLC-missing threads from clogging shared resources.

Overall, with two pair of threads in a core we conclude that the best thread placement is [AB][BA] as in this configuration most of the resources are shared between threads with different resource profiling.

5.1.4 Thread Placement of 4 Single-Thread Micro-Benchmarks

We perform a case study with 4 workloads consisting of different threads. We try different placements to investigate the cases which provide the optimal improvement with regard to core throughput. Like in previous experiments we expect maximum performance when we minimize resource conflicts in the core.

In Figure 5.4 we see various combinations of the 5 representative micro-benchmarks we used in our previous experiments. The IPC bars of each micro-benchmark are stacked in order to add up to the total throughput. We have seen that symmetric pairs (or in

this case quartets) lead to similar behavior *in a specific core mode*. Also, according to our previous results with 2 pairs of threads, inter-cluster resource sharing causes small performance differences, so configurations such as [AB][CD] and [AB][DC] lead to similar performance. We confirm this in our experiments and reduce the total number of possible placements to 3: [AB][CD], [AC][BD] and [AD][BC]. In a given quartet we symbolize with [AB][CD] the presented order of threads, for example for int-fp-l2-mem, A is int, B is fp, C is l2 and D is mem and so for the same quartet [AC][BD] is the order int-l2-fp-mem.

First of all, we notice that the placement of the four executing threads plays a great role in the final performance. We can see that different thread placements lead to impressive speedups of up to 1.66x in throughput for a quartet of threads. In order to get the maximum performance we need to follow a similar approach as to that when running 2 pairs of threads. This can be done by taking advantage of the automatic fetch prioritization mechanism that is activated with `ldint_mem` and avoiding resource conflicts as much as possible. Also, in accordance with our previous findings, it is clear that the presence of a cache-bound application like `ldint_l2` in the quartet considerably lowers the performance of all other co-runners. By combining tasks of different profiles in a cluster we get the maximum possible performance.

Therefore, we should place the threads with highest IPCs in the same cluster as a memory-bound thread -if available, and make sure cache and memory-bound applications are put in different clusters.

5.1.5 Optimal Thread Count Evaluation

In this section we show the throughput of one core of the IBM POWER7 when several copies of each micro-benchmark are executing. In Figure 5.5 we compare the cases of one thread in isolation, two copies of a thread in SMT2 and SMT4 (we use [AX][XA]) and four copies of a thread. The results vary from benchmark to benchmark but we can draw some general conclusions. As a rule of thumb, the lowest throughput is taken when executing a single copy of a micro-benchmark and the highest when executing four copies of the micro-benchmark in the same core. Some benchmarks almost reach their peak with one or two running copies, whereas some others noticeably improve their throughput from a single thread to four threads simultaneously. Indicatively for `cpu_fp` core throughput improves by 190% and for `lng_chain` by 263% when using all four contexts w.r.t. running one thread.

The throughput of `cpu_fp` scales almost linearly up to 4 threads. This is expected if we consider that besides the fact that it is comprised of long latency floating-point instructions, the Vector Scalar Units (VSU) in POWER7 are highly pipelined and capable

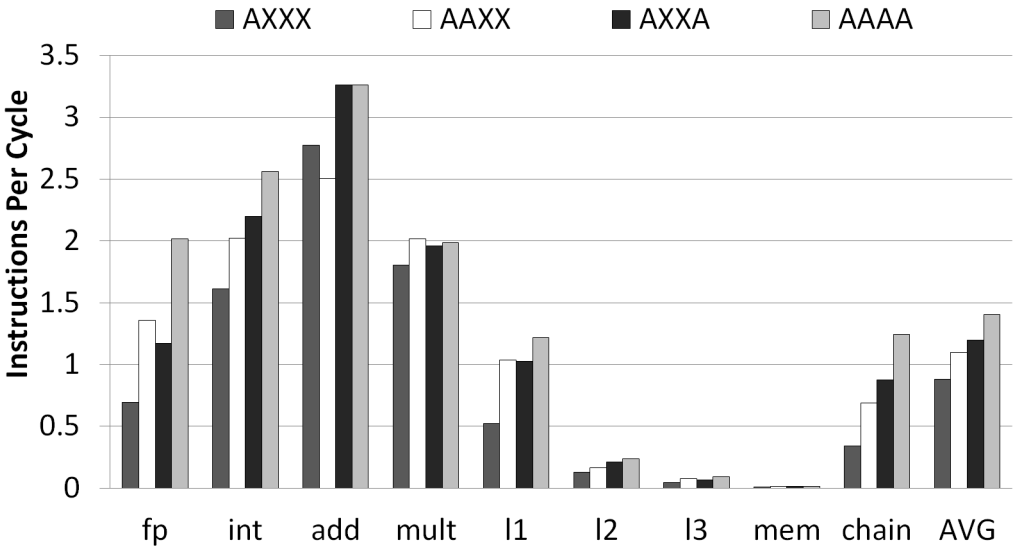


Figure 5.5: Core throughput when different numbers of copies of the micro-benchmarks are executed.

for dual instruction issue, with each pipe supporting different types of instructions [27]. `Cpu_int_add` saturates with 2 threads, since then the peak of execution to issuing ratio is reached in the FXU and LSU (that also performs simple fixed-point instructions). On the other hand, `cpu_int_mul` almost saturates with 1 thread. Compared to `cpu_int_add`, it has a higher instruction latency (integer multiplications), that bottleneck the two FX units and do not leave much space for throughput improvement with more threads. `Cpu_int` and `lng_chain` are comprised of a mix of instructions of the two previous micro-benchmarks and their behavior is more variable from one to four executing threads in the core. The various cache and memory-bound benchmarks improve their throughput by as high as 133% when increasing the threads from one to four, but with the main improvement noted when moving to 2 threads. As we mentioned before, consecutive instructions from these micro-benchmarks cannot be run in parallel due to the pointer-chasing technique. But load instructions *from different threads* can be executed in parallel by the two LS units. Of course, the ability to fetch data in parallel from a certain level of memory hierarchy depends on the amount of cache banks that level has and the ability to access them in parallel. It is known that the L1 cache is highly banked to allow multiple reads at the same time as long as they are in different banks. In any case the maximum throughput is gained when running all four threads, despite the fact that the same might be reached with fewer threads. The abundance of resources in each of the 8 POWER7 cores, combined with the instruction fetch balancing between different threads contribute to maximizing throughput when running 2 or 4 threads in a core.

5.2 Thread Prioritization Performance Characterization

5.2.1 Thread Prioritization of 2 Single-Thread Micro-Benchmarks

As we have seen in section 3.5, when applying priorities on 2 different threads, one thread receives more fetch/decode slots at the expense of the other. As a result, one thread is expected to benefit and the other is presumed to suffer from the application of priorities. In this section we investigate the effect of thread prioritization to both running threads. We execute `cpu_int` with five representative micro-benchmarks under different priority settings in the same cluster with the other two core contexts disabled (SMT2 mode). Subsequently, we repeat the same for `ldint_l2` against the five micro-benchmarks. Figure 5.6(a) and Figure 5.7(a) shows the effect of priorities on `cpu_int` and `ldint_l2` respectively, while Figure 5.6(b) and Figure 5.7(b) show the effect of priorities on the five of their co-runners.

When running with priorities 1/1 both threads have very low performance. As explained in Section 3.5 this is due to the fact that this priority combination puts the POWER7 core in low power mode, so it is only recommended for cases when all threads are executing low priority activities. We do not consider this thread priority setting in the rest of our study.

For all the other priority configurations we confirm that what really matters is not the priority of each thread, but the priority difference between threads. The higher the priority of a thread over the other thread the more the fetch/decode slots it receives and hence the better it performs. We observe that high-ipc cpu-bound benchmarks like `cpu_int`, are more sensitive to priorities than low-ipc benchmarks like `ldint_l2`. As cpu-bound high-ipc benchmarks constantly fetch and decode new instructions they are sensitive to the amount of fetch/decode bandwidth they receive. This is not the case for low-IPC benchmarks, that are stalled due to the latency of some execution unit or the cache hierarchy. Furthermore, we have observed that the IFU balances fetch rates between all threads in a core, causing the performance of a cpu-bound micro-benchmark to be remarkably lowered when co-executing with a cache-bound micro-benchmark. In this case the slowdown suffered by the high-ipc thread can be reversed with the use of priorities, without a significant cost. When executing `cpu_int` with `ldint_l2` and we increase the priority difference to +5 in SMT2 mode, the primary thread's performance improves by 14x, with only a 2x slowdown for the secondary thread cache-bound thread.

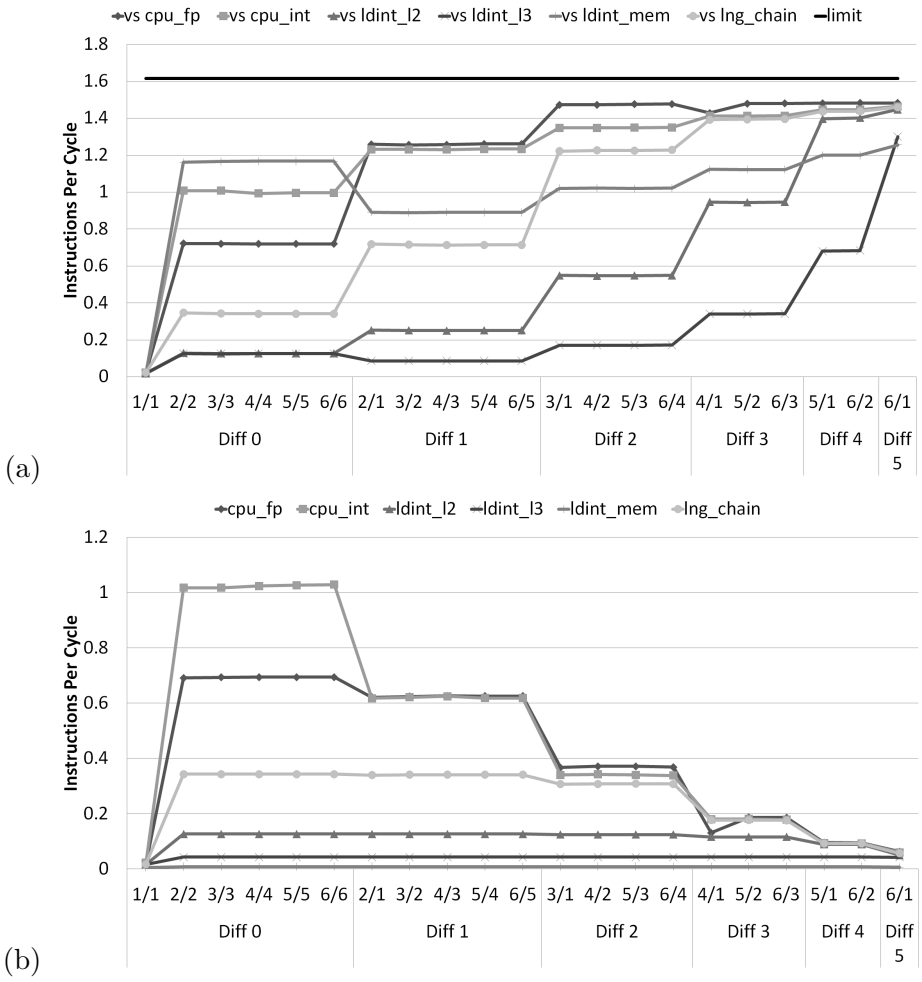


Figure 5.6: Effect of priorities on the IPC of (a) `cpu_int` and (b) each of its co-runners

As a general rule, we notice that by increasing a thread's priority over its co-runner, its performance is increased, since it is granted more fetch/decode slots. For instance, the performance of `cpu_int` against another micro-benchmark is higher with 6/4 than with 4/4 priorities.

However, exceptions to this rule are primarily `ldint_mem` and in a smaller extent `ldint_l3`. When any micro-benchmark is co-executing with `ldint_mem` or with `ldint_l3`, their performance is decreased as soon as we increase priority difference from 0 to 1.

We believe that the reason for this behavior is that POWER7's dynamic hardware resource control mechanisms. By monitoring key resources, applications that clog resources, such as `ldint_mem`, are detected and the system automatically reduces their priority difference. When the user changes the default priority difference, this automatic mechanism is overridden, causing the observed performance decrease. Nevertheless, by further increas-

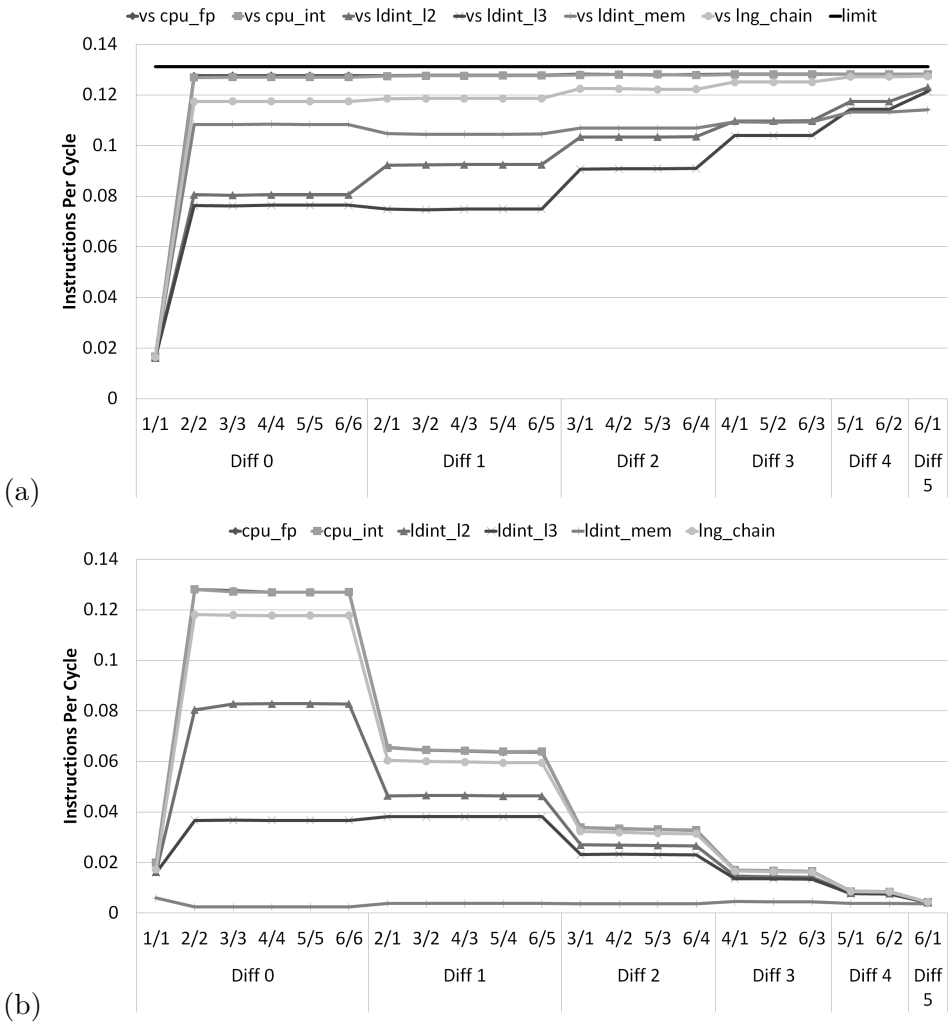
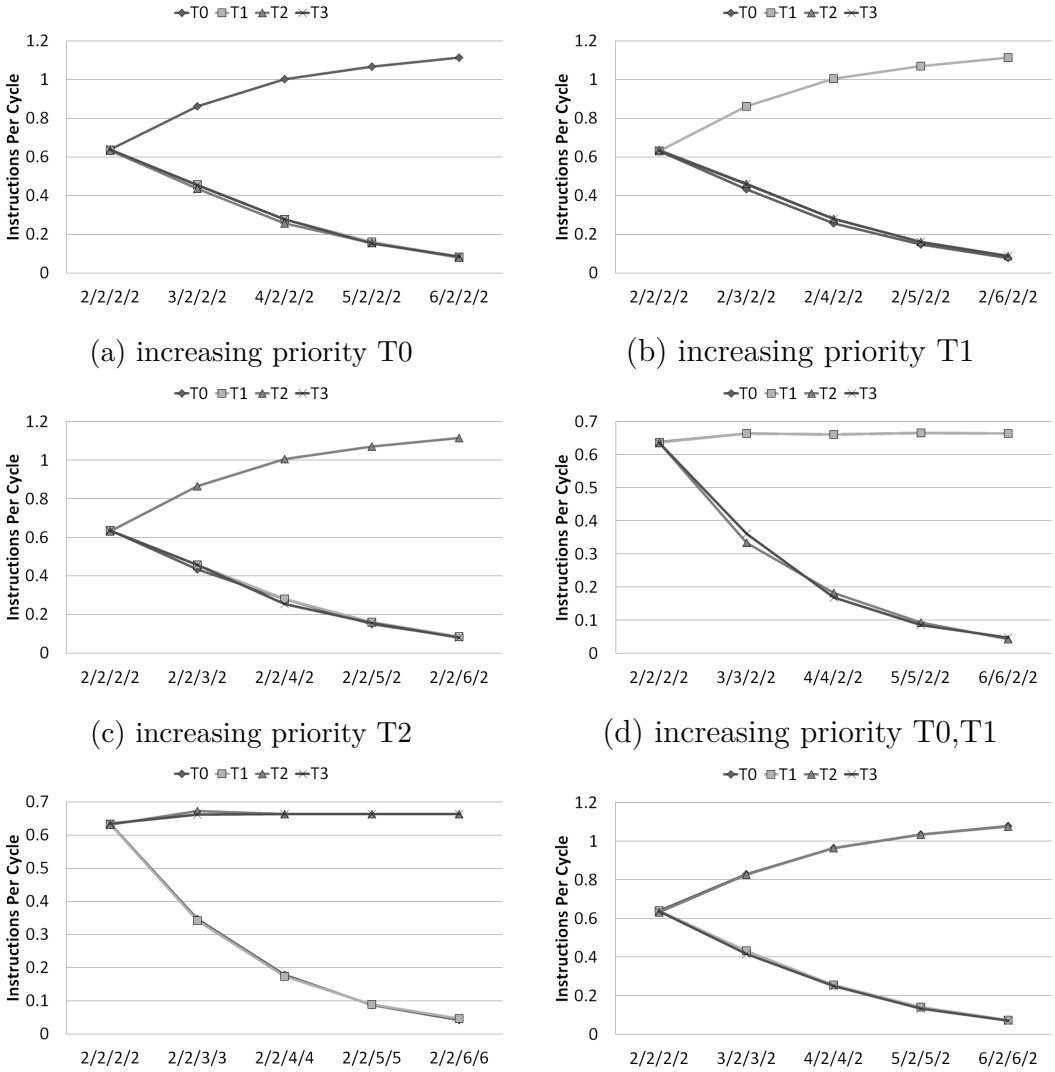


Figure 5.7: Effect of priorities on the IPC (a) of `ldint_l2` and (b) each of its co-runners

ing the priority difference we can see a relative increase in performance that eventually overcomes the performance with default priorities. For example, for `cpu_int` with `ldint_mem` a priority difference of 4 is needed to match the performance with default priorities, and for `cpu_int` with `ldint_l3` the default priorities performance is overcome with a priority difference of 2. If we consider that `ldint_mem` misses in the last level cache, thus causing long latency out-of-chip accesses, we can conclude that the automatic prioritization mechanism decreases the priority of a thread depending on the degree to which it clogs resources.



(a) increasing priority T0

(b) increasing priority T1

(c) increasing priority T2

(d) increasing priority T0,T1

(e) increasing priority T2,T3

(f) increasing priority T0,T2

Figure 5.8: Intra- and Inter-cluster effect of priorities on the four-thread `cpu_int` benchmark

5.2.2 Thread prioritization of a Four-Thread Micro-Benchmark

One of the main difference in the design of the prioritization mechanism of POWER7 and its predecessors, POWER5 and POWER6, is that POWER7 features 4 threads per core that are divided into two clusters. This impacts the way prioritization affects running threads. Figure 5.8 shows the result of four running copies of the `cpu_int` benchmark. In this section we refer to a thread running in Context 0 as T0, a thread in Context 1 as T1 and so on. In the default configuration we run all copies with the same priority of

2. In Figures 5.8(a)-(c) we increase the priority of T0(a), T1(b) and T2(c) over all other threads. We do not show the case of increasing the priority of T3 over other threads, as it is identical to all three cases. Figures 5.8(d)-(e) show the performance of the four threads when increasing the priorities of two threads in the same cluster over the threads in the other cluster and Figure 5.8(f) when increasing the priority of two threads in different clusters (T0 and T2) over the remaining two. Likewise, when increasing the priority of T1 and T3 over T0 and T2 the results obtained are the same as in Figure 5.8(f).

In Figure 5.8(a) we observe that the increase in priority difference of T0 affects all other threads equally, regardless of whether they are on the same or a different cluster than T0. We further observe that the behavior is exactly the same if we increase the priority of T0, T1, T2 or T3 over the rest of the threads in the core. In general, if we increase the priority difference of a thread we improve its performance (the degree of improvement depends on the particular benchmark as shown in Figure 5.6), but we affect the performance of all other 3 threads in the core and not only of the thread running in the same cluster. This fact makes it more difficult to achieve core throughput improvement or at least maintain it when changing priorities.

If we change the priority difference of both threads in the same cluster (T0-T1 or T2-T3), we observe in Figure 5.8(d)-(e) that their performance does not significantly increase while the performance of threads in the other cluster decreases. This situation is the worst possible and indicates that increasing the priority difference of threads in the same cluster provides the worst results.

Instead if we increase the priority difference of two threads in different clusters (Figure 5.8(f)) we obtain the desired behavior, as we manage to improve the performance of the prioritized threads.

Overall, prioritizing a thread in each cluster seems to be the best option in order to avoid affecting overall throughput significantly. In SMT4 mode threads in the same cluster share much more resources than threads in different clusters. When prioritizing one thread in a cluster, it can enjoy increased usage of resources in the cluster. However, when prioritizing both threads in a cluster, their increased instruction fetch rates result in higher intra-cluster resource conflicts.

We conclude that prioritization on POWER7 is applied on two separate levels, an inter-cluster and an intra-cluster level and that to obtain the highest throughput with priorities on four running threads, we should prioritize threads in different clusters.

5.3 Thread Placement & Prioritization Performance Characterization

5.3.1 Thread Placement & Prioritization of 2 Single-Thread Micro-Benchmarks

In this case study, we compare the different priority settings (differences of 0, 2 and 4) under the three placement configurations ([AB][XX], [AX][BX] and [AX][XB]) that we have examined thus far. Figure 5.9(a) presents `cpu_int` against five micro-benchmarks that are representative for the whole METbench suite: `cpu_fp`, `cpu_int`, `ldint_l2`, `ldint_mem` and `lng_chain` . `Cpu_int` is very sensitive to priority changes so it is the best candidate to use against all other micro-benchmarks.

A first conclusion is that threads in the same cluster are affected more by priorities than threads in different clusters. For example, with priority difference 0 `cpu_int` performs 1.4x better in [AX][BX] setup w.r.t. [AB][XX] when co-executing with `cpu_fp`. However, for the same pair when priority difference is increased to 2, [AB][XX] shows better performance by 1.3x compared to [AX][BX]. In [AB][XX] thread placement, the core is executing in SMT2 mode where practically all resources are dynamically shared between the two threads. On the other hand, in [AX][BX] and [AX][XB] thread placements the core executes in SMT4 mode, where the pipelines are clustered and resources are statically allocated to the two threads. When increasing a thread's priority, it is given more fetch/decode slots and so the more resources it has access to the more it can benefit. For that reason priority increase in SMT2 mode results in higher improvements in comparison to SMT4.

We have seen in the previous section, that the two SMT4 placements ([AX][BX] and [AX][XB]) lead to very close results. We notice that the two placements provide similar results for all priority differences.

5.3.2 Thread Placement & Prioritization of 2 Two-Thread Micro-Benchmarks

The same case study is repeated for 4 threads, i.e. 2 two-thread micro-benchmarks. Figure 5.9(b) shows the performance of `cpu_int` against the five representative micro-benchmarks we have used in previous experiments. We use priority difference steps of 2 and assign to threads of the same micro-benchmark the same priority. The placements under examination are [AA][BB], [AB][AB] and [AB][BA].

First of all, we confirm that prioritizing a thread in each cluster results in signifi-

cantly better performance than prioritizing both threads in a cluster, as we explained in section 5.2.2. For example `cpu_int`'s performance is improved by 1.8x against `cpu_fp` when running with a priority difference of 2 in [AB][AB] placement, compared to the same priority difference in [AA][BB] placement.

We have seen in section 5.1.3 that the optimal configuration when running 2 two-thread micro-benchmarks is [AB][BA] since it leads to the least amount of resource conflicts. Additionally we observed that the IFU balances threads fetch rates, so a high-ipc micro-benchmark's performance is lowered when co-executing with a low-ipc cache-bound micro-benchmark. However, by increasing the priority difference of the cpu-bound thread we increase its fetch and decode rate, thus increasing its performance impressively, while the cache-bound application has no visible degradation. `Cpu_int`'s IPC improves by 11x against `ldint_l2` in [AB][BA] placement when the priority difference is increased from 0 (6/6/6/6 priorities) to 4 (6/2/2/6), while the slowdown for `ldint_l2` remains small. In general, cache-bound micro-benchmarks depend on the latency of the relative level of cache and are rather insensitive to priority decrease.

Also, the two configurations where threads of the same kind are assigned to different clusters ([AB][AB] and [AB][BA]) lead to very similar results for all priority differences. This is expected as long as we have seen that METbench micro-benchmarks do not suffer from notable resource conflicts in the inter-cluster resource sharing level.

5.4 Micro-benchmarks Conclusions

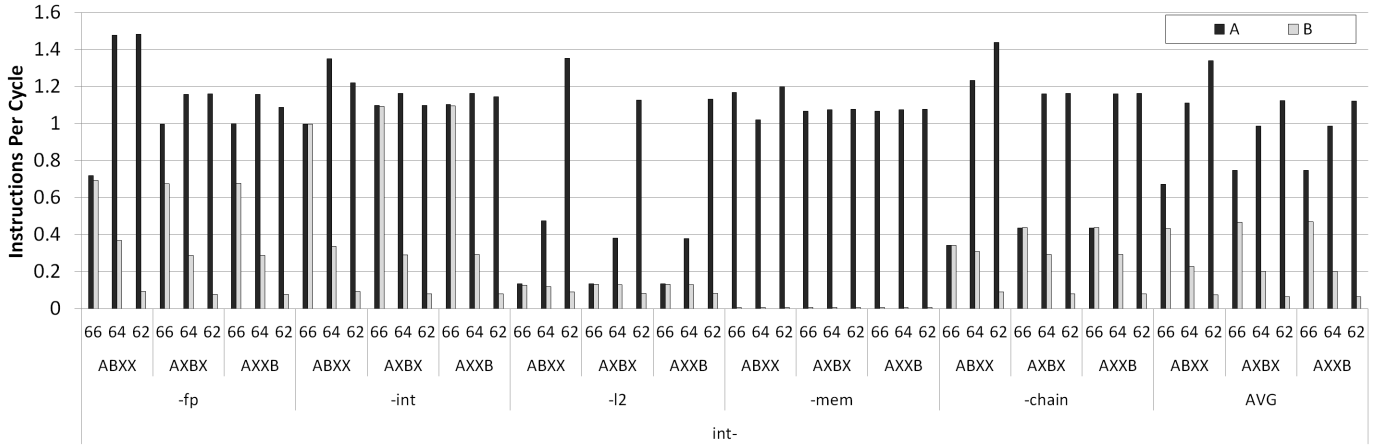
Throughout this section we drew conclusions on the resource sharing mechanism of the IBM POWER7 and how this knowledge can be applied to thread placement and thread prioritization. Following are the main conclusions we drew from micro-benchmarks.

- When running a single thread, the user should assign it to Context 0 as a general rule, thus putting the core in ST mode and allocating to it all available core resources. The only exception to this rule is when the task has a long chain of data dependencies, in which case it should be bounded to Context 2 or 3 in SMT4 mode.

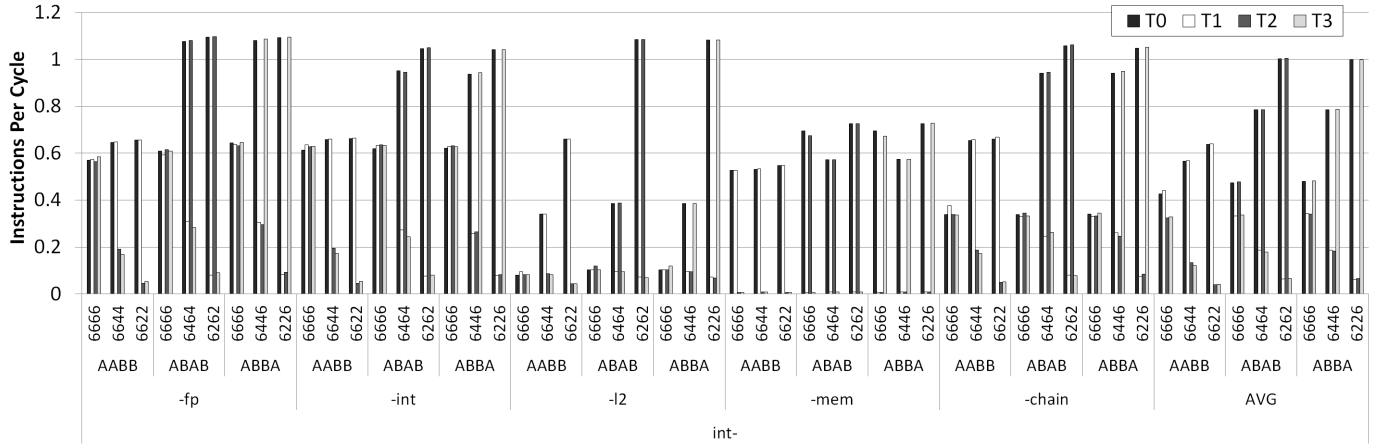
- When executing two threads, in case they belong to different applications the choice regarding thread placement comes down to either [AB][XX] (SMT2) or [AX][XB] (SMT4). In general, SMT2 performs better than SMT4 as long as none of the co-runners has a high percentage of inter-dependent instructions or low-latency operations. When co-running a high-IPC application with a low-IPC cache-bound application, the user can increase the high-IPC thread's performance by increasing

its priority difference, with almost no performance cost for the cache-bound thread. At the same time, it should be noted that a critical workload should rather be prioritized in SMT2 mode in contrast to SMT4 mode, as the latter limits the amount of maximum resources it can potentially access.

- When executing four threads, in case they all belong to different applications, the user should place the highest-IPC thread in the same cluster as a memory-bound thread if available and to ensure memory and cache-bound threads are assigned to different clusters. In case the four threads belong to two 2-thread applications, the optimal choice is to place threads of the same application in different clusters and specifically in [AB][BA] placement, as this configuration leads to the least amount of resource conflicts. Also under this thread placement, the use of priorities in favor of one of the two 2-thread applications leads to the best speedups, compared to other thread placements. In general, with four threads in a core, prioritizing one thread affects all other executing threads, while prioritizing two threads in different clusters leads to significantly better results than prioritizing two threads in the same cluster.



(a)



(b)

Figure 5.9: Effect of thread placement and priorities on (a) 2 single-thread or (b) 2 two-thread micro-benchmarks, where A is always `cpu_int` and B is one of five other micro-benchmarks

Chapter 6

Case Study

In this chapter, we perform a couple of case studies using PARSEC benchmarks on the POWER7 processor. We make use of the knowledge we acquired with METbench micro-benchmarks and build on it to draw the optimal performance in real-world applications, such as the PARSEC benchmarks.

Recall that we use a single POWER7 processor, with 8 cores, each supporting 4-way SMT. Every core has its private 32-KB data and instruction L1 caches, a 256-KB private L2 cache and a 4-MB local L3 region that is shared between all cores, forming a 32-MB global L3 cache.

6.1 Thread Placement of a Single Parallel Application

Firstly, we measure the scalability of each PARSEC benchmark with the number of threads. We start by running a single-thread version of each benchmark and, in subsequent experiments, we run parallelized versions of each benchmark with 2, 4, 8, 16 and 32 threads. Figure 6.1 shows the speedups obtained over the single-thread version of each benchmark.

When using 2 and 4 threads, we bind each thread to the first context of a different core, thus switching them to ST mode. In the case of 8 threads, we evaluate all four possible placements with one thread per core: [AX][XX], [XA][XX], [XX][AX] and [XX][XA]), with each core operating in ST, SMT2, SMT4 and SMT4 mode, respectively. We observe that the fastest execution time for all benchmarks is achieved in ST mode, followed closely by the SMT2 mode with a degradation of 3.1% in execution time. The worst performance is obtained in SMT4 mode, with an execution time degradation of 20.2%. These results are consistent with our findings with micro-benchmarks in the previous section.

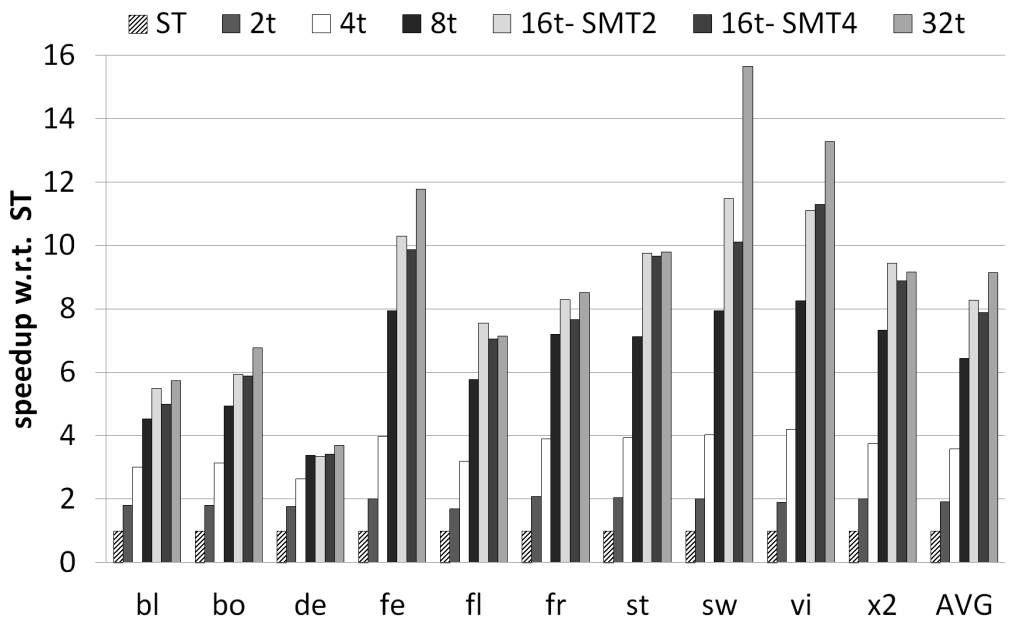


Figure 6.1: PARSEC speedup w.r.t. Single-thread execution of different thread-count and placement setups.

Under the 16-thread count setup, we bind two threads per core. We try three configurations: running both threads in the first cluster in SMT2 mode ([AA][XX]), or in different clusters in SMT4 mode ([AX][AX] and [AX][XA]). We notice that the two SMT4 configurations are very similar in performance, with [AX][XA] being slightly better on average. Consequently, we omit the [AX][AX] case from Figure 6.1. Between the two core modes, SMT2 performs better than SMT4 for all benchmarks (4.9% better on average), with the exception of `vips` and `dedup` that perform slightly better in SMT4 mode. We see that in contrast to several micro-benchmarks, few real-world PARSEC applications benefit from the extra rename registers in SMT4 mode, and therefore, generally perform better in SMT2 mode.

Under the 32-thread setup, we bind 4 threads per core. Given that all threads running in the same core execute similar code, thread placement does not significantly change the final performance of the application. We observe that the majority of applications scale up to 32 threads. Only two benchmarks perform better with 16 threads (`fluidanimate` and `x264`), but only by a small margin. We saw in Section 5.1.5 that with micro-benchmarks the maximum core throughput was reached with either 2 or 4 threads, although throughput with 4 threads was never lower than with 2 threads. Likewise, for the majority of PARSEC applications 32-thread setups execute in equal or shorter time than 16-thread setups.

Overall, to optimize the performance of a given parallel application, we have to take

into account the thread number and thread placement. We confirm the conclusions we drew with micro-benchmarks: When only one thread runs per core, the optimal configuration is ST mode. When 2 threads of a single application are executing in a core, the best configuration is SMT2 mode. Finally, we notice that the optimal throughput is achieved when running 2 or 4 threads per core, for a total of 16 and 32 threads respectively, with the latter leading to significantly higher speedup on average.

6.2 Thread Placement of 2 Parallel Applications

The increase in execution time when moving from 32 to 16 threads for PARSEC benchmarks is only 7.2% on average. All benchmarks except `swaptions` show very small differences between these two setups. This reduced performance degradation motivates us to execute two 16-thread PARSEC applications simultaneously. Several authors advocate for not allocating parallel applications to the same core [16][33], since threads from different applications fight for shared intra-core resources and do not benefit from having a shared L1 data cache. However, proper thread placement can help minimizing resource conflicts in the core. Moreover the considerably large size of the inner levels of the cache hierarchy in the POWER7, contribute to reduced cache contention between different applications. Given applications A and B, we evaluate the following configurations:

1. *Sequential*: We execute applications A and B sequentially. We start by executing A in isolation and once it finishes, we execute B in isolation. Each application is executed under its optimal thread number and placement, i.e., `x264` and `fluidanimate` run in 16 threads in SMT2 mode, while the rest run in 32 threads in SMT4 mode.

2. *Split cores*: We execute applications A and B together, with the first 4 cores running 16 threads of A, and the last 4 cores running 16 threads of B. Here each core is shared only by threads of the same application.

3. *Shared cores*: We execute applications A and B together, with all 8 cores executing 2 threads of each application simultaneously, for a total of 16 threads per application. After trying all three possible placements, we confirm our micro-benchmarks conclusion that [AB][BA] is the best configuration on average when running 2 pairs of threads in a core.

For cases 2 and 3, we compare the time required for both applications to finish executing, to the sum of execution times of A and B in case 1. Our goal is to find out whether we can benefit from running two applications simultaneously or not.

Figure 6.2 shows the execution time improvement of the parallel execution of applications, while sharing (case 3) or not sharing the core (case 2), over the sequential execution of applications case 1). We run each PARSEC application against all other 9 applications. For a given benchmark we compute the execution time improvement of cases 2 and 3 over case 1 for all 9 benchmarks. Figure 6.2 reports the average improvement per benchmark, where configuration 'AAAA-BBBB' stands for case 2 and 'ABBA' for case 3.

Since threads of the same application have more similar hardware resource requirements than threads of different applications, assigning threads of different applications to the same cluster leads to reduced resource conflicts. According to Zhang et al [39], PARSEC benchmarks do not share high amounts of data on the L1 and L2 caches and so co-running two of these applications in a core should not lead to significant cache contention.

Only `dedup`, `freqmine` and `vips` experience slowdowns due to a high contention for common core resources when running 4 threads per core (case 2). These benchmarks execute close to 50% of their instructions in the FXU, while the first two do not include any floating-point instructions. As a result, running 4 threads of these applications in the same core leads to many conflicts in the FXU and thus a decrease in performance. However, when the same benchmarks co-execute with other applications in a core (case 3), we can see clear improvements that stem from reduced contention for shared core resources.

Other benchmarks such as `blackscholes`, `bodytrack`, `ferret`, `fluidanimate` and `streamcluster` perform well when their threads share a core both with threads of the same application (case 2) and with threads of a different one (case 3). Especially `bodytrack` and `ferret` benefit significantly from co-execution despite the fact that they run noticeably slower with 16 than with 32 threads. Nevertheless, it is `streamcluster` -a memory-bound clustering application, that is the best co-runner in both cases. It is by far the application with the highest amount of memory loads amongst the ones we tried, though it does not fetch high amounts of data from memory that could cause bandwidth issues. We have seen that memory-bound micro-benchmarks are good co-runners in a core. Additionally, `streamcluster` has a balanced instruction mix: 30% load and store, 30% fixed-point, and 30% floating-point instructions. This balanced utilization of resources helps when sharing the core amongst its threads, or with threads of another application. The largest improvements are obtained for `streamcluster` with `fluidanimate` in case 3 (45.5%) and with `blackscholes` in case 2 (54%).

In general, few pairs demonstrate worse performance when sharing the core than when running sequentially (only 12.2% of the total pairs increase their execution time by 5%

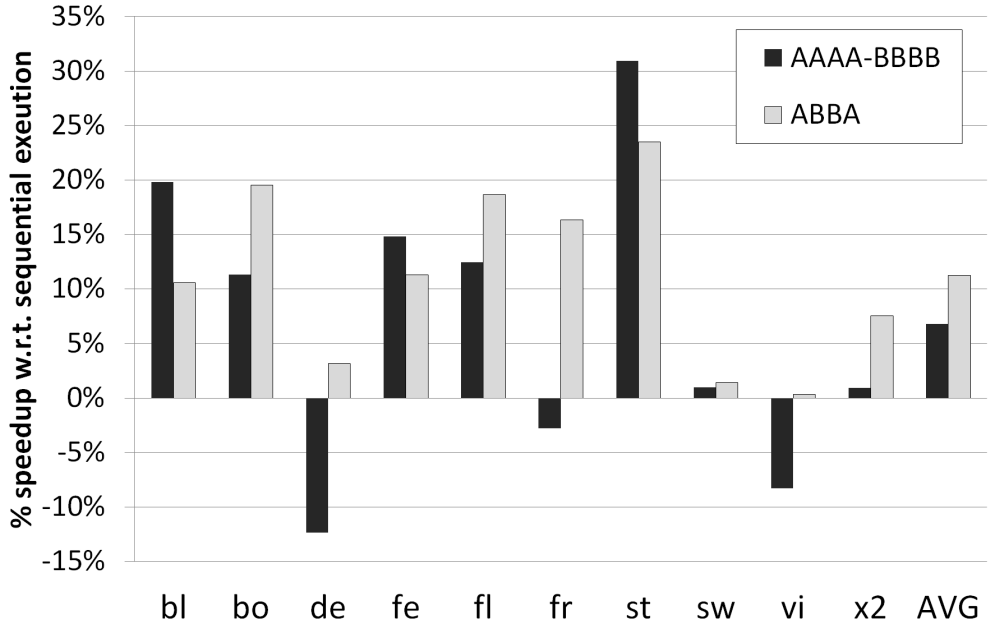


Figure 6.2: Average execution time improvement for each PARSEC application against all other in cases 2 and 3.

or more). These pairs are mainly formed by applications that suffer from running with a suboptimal thread number, like `swaptions` or `vips`. However, the number of pairs that suffer a slowdown in case 2 with respect to case 1 is significant (37.8% of the total pairs increase their execution time by 5% or more). This difference could be caused by increased intra-core resource conflicts between threads of the same application. In any case, the majority of applications benefit from co-executing simultaneously, primarily when sharing the core with another application (case 3), with an average execution time improvement of 11.2% for all pairs.

Figure 6.3 shows a selection of pairs with (a) the best and (b) the worst cases of core-sharing in [AB][BA] (case 3). When two applications run in parallel, one finishes before the other. The time the two applications run together is denoted *co-execution phase 1* in Figure 6.3. The extra time that the slowest application needs to run alone until it finishes is denoted *co-execution phase 2*. In contrast, the second bar per pair of benchmarks stacks the execution times of the two applications while running sequentially in isolation. The vertical axis shows execution time, so the shorter the bar the faster the execution.

Figure 6.3 complements Figure 6.2, as it shows some cases of co-executing pairs of parallel applications. It also provides a visualization of our run methodology and the different phases of co-execution, while depicting how both applications' execution times are affected when running together. We observe that, while for pairs in Figure 6.3 (a) the

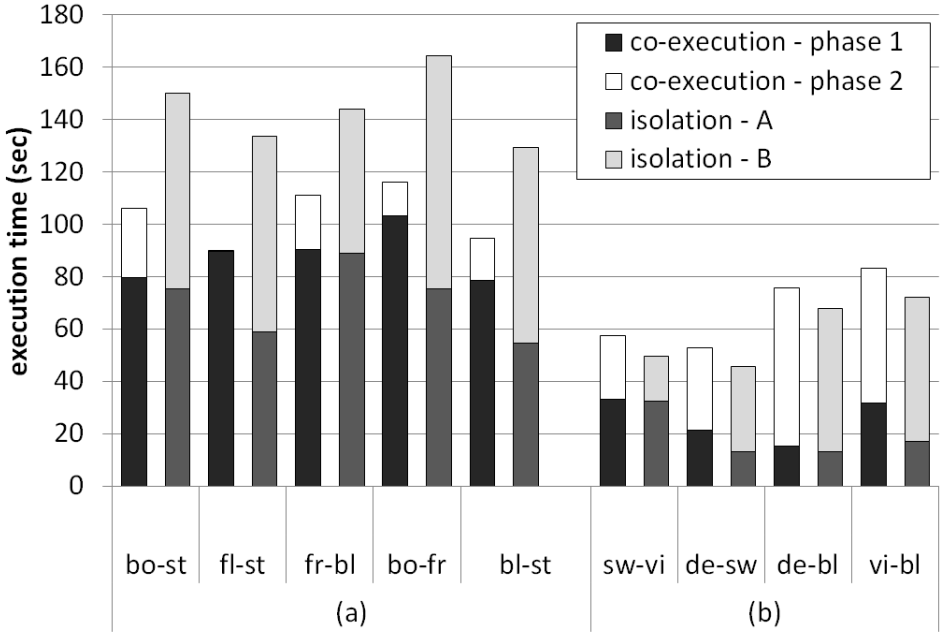


Figure 6.3: PARSEC applications pairs that (a) benefit and (b) suffer from sharing the core. Co-execution phase 1: when A and B are running together. Co-execution phase 2: when the slowest of A,B is running alone.

duration of co-execution phase 2 is small, for pairs in Figure 6.3 (b) it is relatively large. For the latter pairs, this suggests that a significant portion of the slowest application's workload could not be carried out during phase 1 of co-execution due to increased resource conflicts. Note that the pairs in (a) include four of the best co-runners we saw in Figure 6.2, while the pairs in (b) include the three worst co-runners.

Overall, we conclude that co-executing two parallel applications leads to clear improvements in performance, either by sharing the core with another applications (preferably in [AB][BA] configuration), or by occupying half of the cores each. Also, we have to make sure that execution with less threads does not imply a significant slowdown with respect to the optimal case.

6.3 Thread Prioritization of 2 Parallel Applications

In the previous section we discussed the need to co-execute parallel applications and drew conclusions on the optimal thread placements. In this section we build on the drawn conclusions and perform a case study with the software-controlled hardware thread

priorities provided in the POWER7. When co-running two parallel applications, priorities can be used for two purposes:

- First in case one of the two applications is more crucial, it can be prioritized at the cost of the secondary application. For example we might want to execute a real-time streaming application, like `streamcluster` while an ongoing 3D graphics rendering is taking place in the background (e.g. `fluidanimate`). In this scenario we want to make sure the primary application receives increased priority in order to maintain a high quality streaming service, even though the secondary application might suffer.

- Secondly, in order to improve the overall execution time of the two applications together without preference to any of the two applications. In case we have two applications with one finishing faster than the other we can improve total execution time by increasing the priority of the slower application.

Similarly to section 5.3.2, we run two applications with 16 threads each. The thread placement used is the one found to provide the best overall results, i.e. sharing all cores in [AB][BA] placement. Figure 6.4 shows the execution times of some selected pairs of PARSEC applications, including the pairs we used in Figure 6.4. Some extra pairs that benefit from sharing the core are also provided. We set the same priority for all threads of a specific application and use priority differences of plus/minus 3. For all priority differences, we can see the stacked values of both the co-execution phase (2 running applications) and the isolation phase (only the slowest application after the faster one has finished running). Additionally for each pair we present the stacked bars of the two applications execution times while running in isolation. Note that the graph shows execution time, so the shorter the bar the faster the execution.

It is clear how we can use priorities to improve the performance of an important workload, such as `streamcluster`. We can see that its execution time against `blackscholes` or `fluidanimate` improves significantly, almost reaching its optimal execution time in isolation when its priority is increased by 3. However, even though the secondary thread is clearly affected, overall execution time for these pairs remains lower than the sum of execution times of the two applications in isolation. The same is true for all pairs which benefit from co-executing rather than running in isolation. On the other hand, we notice that other benchmarks reach their peak performance with default priorities and cannot further benefit from the use of priorities. Such an example is `dedup` that reaches a threshold close to its isolation execution time when co-running with other PARSEC applications and cannot benefit additionally from an increase in its instructions fetch rate. Nevertheless it does not significantly suffer either from a reduction of priority.

We also observe that in many cases, when running a faster application we can gain by slowing it down in favor of the slowest of the two. Total execution time improves in several cases. Improvement w.r.t. default priorities reaches 12.7% for the pair `fluidanimate-blackscholes` with priorities 4/5. However the use of priorities can results in a slowdown of up to 24.4% w.r.t. default priorities for the pair `streamcluster-fluidanimate` with priorities 5/2. Of course priorities affect different applications in a different degree. As we have seen with micro-benchmarks, high-ipc applications, such as `blackscholes` and `freqmine` are affected more by priorities than low-ipc applications, like `dedup` and `vips`. Additionally, when the difference in execution time is so extended that the isolation phase is longer than the co-execution phase with default priorities, then we notice that priorities only slightly affect execution time. This could be explained by the comparison between co-execution time and isolation time, if we notice that the room for improvement in co-execution is relatively small in these cases.

In general, when co-executing parallel applications overall execution time improves when increasing the priority of the slower application and reducing the priority of the faster one. On average, `blackscholes` gains 2.57% execution time improvement, `bodytrack` 2%, `freqmine` 1% and `swaptions` 0.94% against all other benchmarks with priority difference +1. Also, `ferret` gains 2% and `streamcluster` 1% execution time improvement with priority difference -1. Finally, `x264` gains the maximum average improvement (1.24%) with priority difference -2.

In any case, a safe way to decide whether priorities would improve the performance of an application would be to compare the execution time of the benchmark in isolation and its run-time when co-executing with another application. If the two values are close then there is little room for improvement, however if there is significant slowdown when running in parallel then priorities would improve its performance.

All in all we can see that priorities can be used when co-executing real world applications to effectively reduce the execution time of a critical workload and also to reduce the overall execution time. In this direction, hardware priorities can be a useful tool in the hands of users or system administrators.

6.4 Thread Disabling

When there are no runnable threads in the task run-queue of a specific hardware thread, the OS assigns the idle thread to it. This is the lowest priority thread that performs an infinite loop, is always runnable and has very low IPC. Since the placement of a thread sets the core mode in the POWER7 processor, the existence of one or more idle threads can

reduce the amount of available hardware resources to running applications, thus potentially impacting their performance. Modern versions of the Linux kernel automatically disable hardware threads that are idle for more than a certain time (the default value is 20 ms): first the idle thread reduces its hardware thread priority to 1 (low-power mode) and then, after 20 ms, it disables the hardware thread. External events, such as interrupts raised by the decrementer, an I/O device or another hardware thread, re-activate the disabled hardware thread, which handles the interrupt and invokes the scheduler to check whether any runnable thread has been assigned to its run-queue. If no other thread is assigned to the context, the idle loop restarts. In order to further improve performance by preventing these alternating phases from dynamically switching the core mode, we need to manually turn off unused hardware threads.

In any case, the existence of this automatic mechanism guarantees an improvement in performance in comparison to the case without this feature. This is especially important in situations where manually disabling the threads is not an option, such as parallel applications with load imbalance. In such applications, some threads finish executing faster than others and idle threads take their place. Consequently the core mode remains set in SMT4 mode, even if only one thread from the application is running. As we saw, this results in an important performance decrease. When running PARSEC applications with 32 threads, we experienced a difference in performance between kernels with or without the automatic thread-disabling feature of 22% on average, and up to 143% for `blackscholes`.

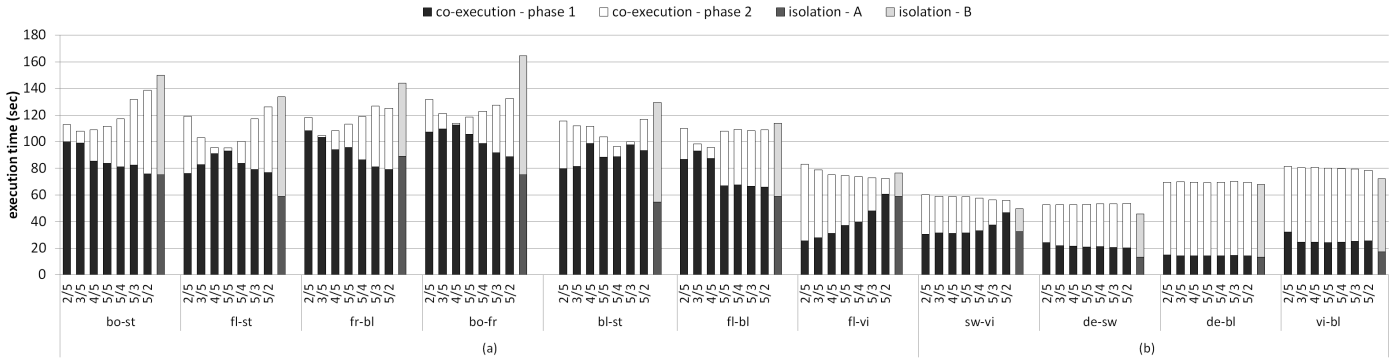


Figure 6.4: PARSEC applications pairs that (a) benefit and (b) suffer from sharing the core. Co-execution phase 1: when A and B are running together. Co-execution phase 2: when the slowest of A,B is running alone.

Chapter 7

Conclusions

7.1 Conclusions

Current CMP+SMT architectures provide complex multi-level resource sharing systems that are hard to analyze. In order to evaluate the performance of such systems, it is necessary to characterize the hardware resource sharing. Thread placement plays an important role in performance since it alters the allocation of resources and can help reduce conflicts between threads. Also, the IBM POWER family offers a thread prioritization system that changes the distribution of resources inside the core by modifying the instruction fetch rate of active threads.

In this study we have used the IBM POWER7 as an example of the current trend of TLP processors. It has 8 cores with each one supporting 4-way SMT, while also being the first processor to feature 4 different levels of resource-sharing. Our methodology consisted of getting insight on the complex resource sharing system of the POWER7 processor with the use of custom micro-benchmarks (METbench) and subsequently applying this knowledge to real-world parallel applications (PARSEC). We conducted this analysis with regard to both issues of thread placement and thread prioritization.

We investigated thread placement for 1, 2 and 4 threads in a core and thread priorities when running 2 and 4 threads simultaneously per core. We drew various useful conclusions about the resource sharing of the processor, as well as the optimal thread placement and prioritization under several configurations. In continuation, we applied this knowledge to the different case studies we executed using parallel applications.

The first case study we performed showed that the optimal number of threads for parallel applications is usually when using all available contexts (32 in our case) and only in a few cases the optimal number was half of the available contexts (16). However, we

noticed that running applications with 16 threads led to a small performance decrease, which motivated us to co-execute 2 parallel applications with suboptimal thread numbers in our second case study. In this case, we managed to extract an average of 11.2% performance increase over execution in isolation with proper thread placement. The latter was counter-intuitive to previous studies, as it was suggested that threads of different applications should rather not be co-scheduled in the same core due to issues of cache contention and resource conflicts. Finally, we used the optimal thread placement and applied thread priorities to show how we can prioritize a critical application or improve total execution time when co-running parallel applications. In this case we achieved 12.7% of additional improvement for some pairs. We also noticed the importance of disabling hardware threads. Without using a linux feature that automatically disables unused contexts, we observed a 22% performance degradation on average.

7.2 Discussion - Future Work

This characterization can have effects on three levels: the Software level, the Operating System level and the Hardware level.

On the software level, software developers or application users can manually bind threads to the desired contexts or apply priorities in order to execute a critical workload optimally. Developers should ideally provide specific application resource requirements, in order to facilitate thread placement decisions made by the OS.

On the OS level, job schedulers need to become more and more advanced in order to deal with the increasing problem complexity of thread placement, especially when considering the heterogenous design of CMP+SMT architectures. The OS can initially schedule threads on the CMP level, while using the extra SMT contexts for higher thread numbers. Preferably hardware contexts in the same core should be shared by threads of different applications, as threads of the same application have more similar resource sharing profiles. Additionally, thread re-assigning after the beginning of execution is important when co-executing two or more applications, or when running applications with load imbalance. After some threads finish running, the scheduler should bind active threads to their optimal placement, based on the conclusions of our study, so that they do not execute in a suboptimal configuration.

Finally, on the hardware level, processor manufacturers should provide hardware mechanisms that allow the OS to control the level of resource sharing between threads. Multithreaded processors should also provide better performance counters to identify the sensitivity of each hardware thread to different resources, simplifying the work of the OS to

better adapt to applications' varying resource requirements.

Future work involves the development of a dynamic job scheduling mechanism. The dynamic mechanism should take our conclusions into consideration and automatically decide on the thread number and thread placement in order to improve system performance. Additionally it could use the thread prioritization mechanism to improve total execution time for two or more simultaneously executing parallel applications.

Bibliography

[1] J. Abeles, L. Brochard, L. Capps, D. DeSota, J. Edwards, B. Elkin, J. Lewars, E. Michel, R. Panda, R. Ravindran, J. Robichaux, S. Kandadai, and S. Vemuganti. Performance guide for hpc applications on ibm power 755 system. Technical report.

[2] A. Agarwal, B. H. Lim, D. Kranz, and J. Kubiatowicz. APRIL: A processor architecture for multiprocessing. Technical Report MIT/LCS/TM-450, 1991.

[3] B. E. Alex Mericas and V. R. Indukuru. Comprehensive pmu event reference power7. Technical report.

[4] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[5] C. Boneti, F. J. Cazorla, R. Gioiosa, A. Buyuktosunoglu, C.-Y. Cher, and M. Valero. Software-controlled priority characterization of power5 processor. In *ISCA*, pages 415–426, 2008.

[6] D. R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[7] V. Cakarevic, P. Radojkovic, J. Verdú, A. Pajuelo, F. J. Cazorla, M. Nemirovsky, and M. Valero. Characterizing the resource-sharing levels in the ultrasparc t2 processor. In *MICRO*, pages 481–492, 2009.

[8] F. J. Cazorla, A. Ramírez, M. Valero, and E. Fernández. Dynamically controlled resource allocation in smt processors. In *MICRO*, pages 171–182, 2004.

[9] B. Chapman, G. Jost, and R. v. d. Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[10] M. DeVuyst, R. Kumar, and D. M. Tullsen. Exploiting unbalanced thread scheduling for energy and performance on a cmp of smt processors. In *IPDPS*, pages 140–140, 2006.

[11] B. Gibbs, B. Atyam, F. Berres, B. Blanchard, L. Castillo, P. Coelho, N. Guerin, L. Liu, C. D. Maciel, and C. Thirumalai. *Advanced Power Virtualization on IBM Eserver P5 Servers*. IBM redbooks.

[12] R. Halstead and T. Fujita. MASA: A multithreaded processor architecture for parallel symbolic computing. In *ISCA-15*, 1988.

[13] W. D. Hillis and G. L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, Dec. 1986.

[14] IBM. *PowerPC Architecture book: Book III: PowerPC Operating Environment Architecture*. 2005.

[15] V. Jiménez, F. J. Cazorla, R. Gioiosa, M. Valero, C. Boneti, E. Kursun, C.-Y. Cher, C. Isci, A. Buyuktosunoglu, and P. Bose. Power and thermal characterization of power6 system. In *PACT*, pages 7–18, 2010.

[16] C. Jung, D. Lim, J. Lee, and S. Han. Adaptive execution techniques for smt multiprocessor architectures. In *PPoPP*, pages 236–246, 2005.

[17] R. Kalla, B. Sinharoy, and J. M. Tendler. Ibm power5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24:40–47, 2004.

[18] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, Mar. 2005.

[19] O. Lempel. 2nd generation intel core processor family: Intel core i7, i5 and i3. *Hot Chips*, August 2011.

[20] D. Levinthal. Performance analysis guide for intel• core• i7 processor and intel• xeon• 5500 processors. Technical report.

[21] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, A. J. Miller, and M. Upton. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*, 6(1), Feb. 2002.

[22] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[23] A. Morari, C. Boneti, F. J. Cazorla, R. Gioiosa, C.-Y. Cher, P. Bose, and M. Valero. SMT Malleability in IBM POWER5 and POWER6 Processors. *IEEE Trans. Computers*, 99(Preprint), 2012.

[24] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, ASPLOS-VII, pages 2–11, New York, NY, USA, 1996. ACM.

[25] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly Media, 1st ed. edition, July 2007.

[26] V. Salapura, K. Ganesan, A. Gara, M. Gschwind, J. C. Sexton, and R. E. Walkup. Next-generation performance counters: Towards monitoring over thousand concurrent events. *Performance Analysis of Systems and Software, IEEE International Symmposium on*, 0:139–146, 2008.

[27] B. Sinharoy, R. Kalla, W. J. Starke, H. Q. Le, R. Cargnoni, J. A. Van Norstrand, B. J. Ronchetti, J. Stuecheli, J. Leenstra, G. L. Guthrie, D. Q. Nguyen, B. Blaner, C. F. Marino, E. Retter, and P. Williams. IBM POWER7 multicore server processor. *IBM J. Res. Dev.*, 55:191–219, May 2011.

[28] B. Smith. Architecture and applications of the HEP multiprocessor computer system. *Fourth Symposium on Real Time Signal Processing*, 1981.

[29] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. *SIGARCH Comput. Archit. News*, 28(5):234–244, Nov. 2000.

[30] B. Sprunt. The basics of performance-monitoring hardware. *IEEE Micro*, 22:64–71, 2002.

[31] S. Storino, A. Aipperspach, J. B. R. Eickemeyer, S. Kunkel, S. Levenstein, and G. Uhlmann. A commercial multithreaded RISC processor. In *45th International Solid-State Circuits Conference*, 1998.

[32] Sun. *OpenSPARC T2 Core Microarchitecture Specification*, a edition, Dec. 2007.

[33] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. The impact of memory subsystem resource sharing on datacenter applications. In *ISCA*, pages 283–294, 2011.

[34] J. M. Tendler, J. S. Dodson, J. S. F. Jr., H. Le, and B. Sinharoy. Power4 system microarchitecture. pages 5–26, 2002.

[35] N. Tuck and D. M. Tullsen. Initial observations of the simultaneous multithreading pentium 4 processor. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, PACT '03, pages 26–, Washington, DC, USA, 2003. IEEE Computer Society.

[36] D. M. Tullsen and J. A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *MICRO*, pages 318–327, Washington, DC, USA, 2001. IEEE Computer Society.

[37] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. 1995.

[38] J. Vera, F. J. Cazorla, A. Pajuelo, O. J. Santana, E. Fernández, and M. Valero. Fame: Fairly measuring multithreaded architectures. In *PACT*, pages 305–316, 2007.

[39] E. Z. Zhang, Y. Jiang, and X. Shen. The significance of cmp cache sharing on contemporary multithreaded applications. *IEEE Transactions on Parallel and Distributed Systems*, 23:367–374, 2012.