

ADVERTIMENT. La consulta d'aquesta tesi queda condicionada a l'acceptació de les següents condicions d'ús: La difusió d'aquesta tesi per mitjà del servei TDX (www.tesisenxarxa.net) ha estat autoritzada pels titulars dels drets de propietat intel·lectual únicament per a usos privats emmarcats en activitats d'investigació i docència. No s'autoritza la seva reproducció amb finalitats de lucre ni la seva difusió i posada a disposició des d'un lloc aliè al servei TDX. No s'autoritza la presentació del seu contingut en una finestra o marc aliè a TDX (framing). Aquesta reserva de drets afecta tant al resum de presentació de la tesi com als seus continguts. En la utilització o cita de parts de la tesi és obligat indicar el nom de la persona autora.

ADVERTENCIA. La consulta de esta tesis queda condicionada a la aceptación de las siguientes condiciones de uso: La difusión de esta tesis por medio del servicio TDR (www.tesisenred.net) ha sido autorizada por los titulares de los derechos de propiedad intelectual únicamente para usos privados enmarcados en actividades de investigación y docencia. No se autoriza su reproducción con finalidades de lucro ni su difusión y puesta a disposición desde un sitio ajeno al servicio TDR. No se autoriza la presentación de su contenido en una ventana o marco ajeno a TDR (framing). Esta reserva de derechos afecta tanto al resumen de presentación de la tesis como a sus contenidos. En la utilización o cita de partes de la tesis es obligado indicar el nombre de la persona autora.

WARNING. On having consulted this thesis you're accepting the following use conditions: Spreading this thesis by the TDX (www.tesisenxarxa.net) service has been authorized by the titular of the intellectual property rights only for private uses placed in investigation and teaching activities. Reproduction with lucrative aims is not authorized neither its spreading and availability from a site foreign to the TDX service. Introducing its content in a window or frame foreign to the TDX service is not authorized (framing). This rights affect to the presentation summary of the thesis as well as to its contents. In the using or citation of parts of the thesis it's obliged to indicate the name of the author

AFFORDABLE KILO-INSTRUCTION PROCESSORS

MIQUEL PERICÀS GLEIM

OCTOBER 2008

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

DOCTOR OF PHILOSOPHY / DOCTOR PER LA UPC
Doctor Europeus Mention

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya

ACTA DE QUALIFICACIÓ DE LA TESI DOCTORAL

Reunit el tribunal integrat pels sota signants per jutjar la tesi doctoral:

Títol de la tesi:

Autor de la tesi:

Acorda atorgar la qualificació de:

No apte

Aprovat

Notable

Excel·lent

Excel·lent Cum Laude

Barcelona, de/d' de

El President

El Secretari

.....
(nom i cognoms)

.....
(nom i cognoms)

El vocal

El vocal

El vocal

.....
(nom i cognoms)

.....
(nom i cognoms)

.....
(nom i cognoms)

AFFORDABLE KILO-INSTRUCTION PROCESSORS

MIQUEL PERICÀS GLEIM

OCTOBER 2008

ADVISORS:

Mateo Valero Cortés

Universitat Politècnica de Catalunya

Barcelona Supercomputing Center

Adrian Cristal Kestelman

Barcelona Supercomputing Center

COLLABORATORS:

Francisco Cazorla Almeida

Barcelona Supercomputing Center

Daniel A. Jimenez

University of Texas at San Antonio

Alex Veidenbaum

University of California at Irvine

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

DOCTOR OF PHILOSOPHY / DOCTOR PER LA UPC

Doctor Europæus Mention

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya

*Als meus pares, Miquel Àngel i Anneliese,
a la meua germana Elisabet, i a la Núria.*

ACKNOWLEDGMENTS

Many people helped out during my work on this thesis. I would like to take the occasion to thank them all:

- First of all I would like to thank my advisors Mateo Valero and Adrian Cristal for their guidance during the development of my thesis. Many thanks also to Daniel A. Jimenez for multiple technical discussions and extensive help during the preparation of several manuscripts that were submitted during this thesis. I would also like to thank the other researchers that have collaborated in the development of this thesis for their insightful comments and discussions that have greatly improved the quality of this work: Francisco J. Cazorla, Alex Veidenbaum and Ruben González.
- Second, I would like to thank the many professors and researchers who have been involved in the procedures that have allowed this thesis become a reality: Per Stenstrom, Stefanos Kaxiras, Georgi N. Gaydadjiev, J.M. Llaberia, Agustin Fernandez, Alex Ramirez, Eduard Ayguadé, Theo Ungerer, Osman Unsal, Mario Nemirovsky, Rosa M. Badia and Gurindar H. Sohi.
- Third, I want to thank Trinidad Carneros for her help in guiding me through the procedures required for the completion of this thesis. Without her help I would surely have messed up with the procedures.
- Finally, my most sincere thanks to all the good friends at UPC and BSC that have suffered me since I started my phd in early 2003. I am deeply in debt for their kindness and for their help.

For the completion of this work I have used resources provided to me by my employer, the Barcelona Supercomputing Center (BSC-CNS), and by the Computer Architecture Department of the Universitat Politècnica de Catalunya, where I have pursued my PhD, and where I worked as *Visiting Professor* from September 2003 to February 2005.

Several institutions have provided additional funding for this project. The Spanish Ministry of Science and Technology has supported this work under contracts TIN-2004-07739-C02-01 and TIN-2007-60625. In addition, funding has been obtained from the Framework Programme 6 HiPEAC Network of Excellence (IST-004408). In particular, HiPEAC provided the funding for my three-month stay at the Technical University of Delft from April 2007 to June 2007.

Finally, I would like to thank Horst Zuse for allowing me to use one of his pictures of the Zuse Z1 reproduction currently housed in the Deutsches Technikmuseum Berlin as the cover art for the print edition of this thesis.

ABSTRACT

Evolution of computer architecture is in a moment of great turmoil. After the tremendous performance progress experienced during the 80s, 90s and early 21st century, this trend has stopped, and multiple walls make it uncertain which direction to follow.

The two main limiters are the memory wall and the power wall. The memory wall is the result of the increasing disparity between processor clock cycle time and memory access time. Since the early 80s, both have diverged at an approximate rate of 40-50% per year, with processor frequency improving much faster than memory latency. This represents a thousand-fold increase in about 20 years of evolution. For today's multi-GHz processors, every access to main memory translates to around 100-1000 processor cycles. Current microarchitectures are not well suited to handle these latencies. The influence of the memory wall is significant and reduces the advantages of new microarchitectural improvements. This is the result of Amdahl's law, as new improvements are only beneficial to parts of the program not affected by main memory access.

The main reason that the memory wall has grown this large is the impressive increase in processor frequencies experienced during the same period. As a way to increase performance, industry has concentrated on reducing the processor cycle time. The downside is that with higher frequency comes higher power consumption. The power consumption of a processor is roughly proportional to its frequency. Modern multi-GHz processors have power consumptions surpassing the 100 Watt barrier. The power dissipated per cm^2 has reached levels where operation is no longer safe, despite extensive use of cooling systems, including passive cooling (dissipators) and active cooling (such as air-cooling or water-cooling).

In addition to these two walls there is a third wall with profound consequences: the complexity wall. Modern microarchitectural techniques such as speculation and dynamic scheduling, along with billion transistor designs and intricate on-chip networks make verification of high-performance microprocessors a real nightmare. In addition, currently known algorithmic solutions for verification are running out of capacity. Chip verification time has become the largest part of the design process of modern microprocessors. Its importance cannot be underestimated.

All these walls have led industry to focus on multicores. Multicore processors place multiple processing cores on the same die along with communication hardware such as shared caches with coherence protocols. By using multiple smaller cores, power can be kept within bounds. However, the overall power cannot surpass the thermal design power (TDP) of the system, so care needs to be taken. By

having multiple cores operate in parallel, the problem of the memory wall is slightly diminished, as other cores can proceed while a core is stalled waiting for a cache miss to be serviced. Finally, since the cores that make up a multicore can be simpler than traditional single-core high performance microprocessors, complexity can be limited and better handled. However, things like cache-coherence protocols and more modern techniques such as transactional memory have their own complexities that cannot be neglected.

Overall, the introduction of multicores seems to be a good solution to the three aforementioned walls. However, multicores introduce a new wall with a large impact: the *programming wall*. The way to extract performance in a multicore is to exploit thread-level parallelism (TLP). However, reasoning about a program as a collection of threads is a complex task. To make it worse, current programmers are not well trained for this task. Parallel programming has found application in markets such as high-performance computing, where it is required to obtain performance. The question whether parallel programming will see generalized application in computing is yet to be answered.

Both single-core as well as multi-core processors share the single processor core as the basic computing unit. In this thesis we research how to design a core capable of overcoming the three traditional walls (memory, power and complexity). Our focus is on high performance and thus we first research how to overcome the memory wall, the single largest performance limiter of current microprocessors. Based on our findings we then propose implementation strategies that simultaneously target the power and complexity walls. By using a high performance ILP processor as baseline the programmer can focus on higher level program partitioning and obtain high performance for threaded applications –accelerating each thread– as well as for applications that disallow any further thread partitioning.

Our strategy to overcome the memory wall consists in increasing the instruction window to handle thousands of instructions in flight. Such processors are called KILO-Instruction Processors (KIPs). KIPs have been proposed by the High Performance Computing group (CAP) at the Computer Architecture Department (DAC) of the Technical University of Catalonia (UPC) using a partitioning of the instruction window into two parts: one tracked at instruction granularity and one tracked at checkpoint granularity [1, 2, 3, 4, 5]. In this thesis we propose a new KILO-Instruction processor by dividing the processor into two smaller processors. The first core focuses on processing instructions depending only on cache accesses while the second core processes instructions depending on cache misses. This code partitioning is based on our observation of *Execution Locality* (Chapter 4), which shows that, despite the big impact of memory misses on cycle count (over 50% in numerical benchmarks), less than 30% of the dynamic code actually depends on cache misses.

Based on this model we propose two implementations of KILO-Instruction Processors. The first implementation proposes to process the miss-dependent code with a

fully in-order machine (Chapter 5). Despite the simplicity of this model, sub-optimal performance motivates us to propose a more powerful and complexity-efficient version, which emulates out-of-order execution in the back-end using multiple iterative in-order pipelines (Chapter 6). This version proposes to build the back-end based on multiple sequential in-order machines –called *memory engines*– each one tracking a relatively small number of instructions. This allows to build a high performance back-end completely out of small structures. In addition, it allows to keep a low power profile. Our evaluation shows that this processor, despite featuring structures smaller than modern microprocessors, performs similar to an ideal out-of-order processor with an instruction window of around 1500 instructions. This multiplies by more than 10 the window size of current machines.

Despite the advantages of this second implementation, resource-wise it is quite aggressive. Each memory engine contains its own set of functional units which end up suffering from low utilization. To alleviate this problem the functional units can be shared among memory engines. Alternatively, utilization of memory engines can be increased by applying multithreading techniques to the processor (Chapter 7). This is effective since the performance of the processor depends on the number of allocated memory engines. The resulting processor is capable of adapting to running workloads. We therefore call it the *Flexible Heterogeneous MultiCore*.

The final contribution of this thesis is to provide a memory subsystem capable of supporting this architecture (Chapter 8). We keep the cache subsystem fixed and instead concentrate on the Load/Store Queues (LSQ). The LSQ is known to be very difficult to scale. This is because of its complex functionality: ensuring correct dynamic scheduling of loads and stores, correct ordering of the commit of stores, tracking store-loads forwarding and detecting load-store ordering violations. The solution we propose is based on *execution locality* coupled with a two-level approach. We show that this queue correctly emulates an ideal KILO-LSQ while being based only on small structures.

CONTENTS

1	HISTORICAL INTRODUCTION	7
1.1	A brief history of the transistor	8
1.2	A brief history of the microprocessor	8
1.3	Evolution of Processor Microarchitectures	10
1.4	Current Trends	14
2	MOTIVATION FOR THIS THESIS	17
3	EVALUATION INFRASTRUCTURE	23
3.1	Simulation Infrastructure	23
3.2	Benchmarks and Traces	24
3.3	Collecting and Reporting Results	24
4	THE MEMORY WALL AND EXECUTION LOCALITY	27
4.1	Overview	27
4.2	Effects of the Memory Wall	28
4.3	Execution Locality	29
4.3.1	EQUAKE Example	33
4.4	Summary and Conclusions	36
5	THE DECOUPLED KILO-INSTRUCTION PROCESSOR	39
5.1	Overview	39
5.2	Emulating the performance of KILO-Instruction Processors	39
5.3	A Decoupled KILO-Instruction Processor	40
5.3.1	Implementation of a D-KIP processor	40
5.3.2	Load/Store Queues	45
5.3.3	Pipeline	46
5.3.4	Execution Locality: A different perspective	46
5.4	Performance Evaluation	47
5.4.1	Simulation Infrastructure	47
5.4.2	Performance Comparison	47
5.4.3	Impact of Scheduler Policies and Queue Sizes	50
5.4.4	Impact of Cache Sizes	51
5.4.5	Storage Requirements	53
5.5	Design Issues	54
5.6	Related Work	56
5.7	Conclusions	58
6	THE FLEXIBLE HETEROGENEOUS MULTICORE PROCESSOR	59
6.1	Motivation	59
6.2	The Flexible Heterogeneous MultiCore Processor	61
6.2.1	Some problems with the initial D-KIP design	61
6.2.2	Approximating Dataflow in the Memory Processor	61
6.2.3	A resizable window based on sequential partitioning	63
6.3	Evaluation of the FMC processor	66
6.4	Related Work	73
6.5	Conclusions	75

6 CONTENTS

7	ADAPTIVE HETEROGENEOUS PLATFORM USING FMC	77
7.1	Motivation	77
7.2	Assigning Memory Engines	78
7.3	Multi-core Simulation Infrastructure	79
7.3.1	Evaluation of a 4-way Multi-Core Architecture	80
7.4	Related Work	82
7.5	Conclusions	83
8	EPOCH-BASED LOAD/STORE QUEUE	85
8.1	Motivation	85
8.2	Background	86
8.2.1	Memory Handling for Large Windows	86
8.2.2	Execution Locality Analysis	88
8.3	Epoch-based Load Store Queue	89
8.3.1	Generic Processor Model	89
8.3.2	Epoch-based Load/Store Queue	89
8.3.3	Restricted Disambiguation Models	91
8.3.4	Hardware Disambiguation Schemes	91
8.3.5	Non-Associative Load Queue with Load Re-Execution	95
8.3.6	Coherence and Consistency	95
8.4	Integration with Locality-based Processor	96
8.4.1	Exceptions and Recovery	97
8.5	Evaluation	97
8.5.1	Simulation environment	97
8.5.2	Tuning Epoch Size	98
8.5.3	Performance of Epoch-based LSQ	99
8.5.4	Performance of Global Disambiguation	100
8.5.5	Restricted Disambiguation Models	103
8.5.6	Large Window Load Re-Execution	104
8.6	Energy Considerations	106
8.7	Related Work	109
8.8	Conclusions	110
9	CONCLUSIONS	111
10	LIST OF PUBLICATIONS	115

HISTORICAL INTRODUCTION

This thesis describes research activities related to the design of microprocessors. To start, we initiate this thesis by giving some introductory information on microprocessor history and technology. I have made an effort to make this introduction understandable for readers with no previous knowledge in this field. Tech-savvy readers may want to skip this section and proceed directly with chapter 2.

The advent of the microprocessor has largely influenced computer technology and, indirectly, society. Microprocessors are now at the core of almost all modern computers. The key development that allowed microprocessors to be developed was Bell Labs' public introduction of the transistor in 1948. Previously, in 1928, the austrian-hungarian physicist Julius Edgar Lilienfeld registered three patents describing this device. However, he failed to produce any working demonstration and his work remained largely unknown. Before the introduction of the transistor, computers were built using vacuum tubes. As a result they generated large quantities of heat, were huge in size and unreliable. Only large corporations could afford these machines. The introduction of the transistor and the subsequent miniaturization race allowed computers to be considerably reduced in size, which finally led to their introduction in personal computers, laptops and devices as small as mobile phones.

Computers have been used in a variety of situations. Flexibility is one of the main features of computing systems. Computers are able to solve a multitude of problems by transforming inputs (with sources such as keyboard, mouse, files, etc.) into outputs (files, monitors, sound, etc). These inputs and outputs represent the interface through which computers interact with users and are capable to provide solutions to problems.

What makes a computer useful is that it can perform tasks with high precision, high speed, and deterministically. The tasks need to be well-defined and in a language that is understood by the computer. In computer nomenclature the set of instructions that a computer executes to complete a certain task is known as the *program*. Modern computers are multiprogrammed. They execute many programs simultaneously in an overlapped/interleaved way. Not all of these programs perform activities for the end user. Many programs just manage the hardware so that other programs can make use of it in an ordered and simple way. These programs are

known as *system software*. A typical example of system software is the operating system. Programs that directly interact with the user are known as *application software*. Applications are what motivates the existence of a market for computers.

1.1 A BRIEF HISTORY OF THE TRANSISTOR

Transistors are the basic blocks of integrated circuits. As aforementioned, the basic idea of the *field-effect transistor* (FET) had been described long before its first physical implementation by Bell Labs in 1947. A transistor is basically a three-gate device where the current flowing between two gates can be controlled by a small current or voltage applied to the third gate. The material used for this first-ever transistor was germanium. Germanium as a commercial material is limited due to its sensitivity to humidity and temperature. Silicon, with identical crystal structure to germanium, is a much better choice for commercial use. However, it was not until early 1954 that M. Tanenbaum at Bell Labs implemented a high performance silicon transistor using NPN junctions. [6] Silicon transistors have been used extensively ever since. Engineering has enabled their average size to be miniaturized with every new technology generation. This enables the complexity of chips to grow very fast. Intel co-founder Gordon Moore identified a pattern in this development which he explained in *Electronics Magazine* in 1965 [7]:

“The complexity for minimum component costs has increased at a rate of roughly a factor of two per year (see graph). Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least ten years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65 000.

I believe that such a large circuit can be built on a single wafer.”

This quote is the foundation for what is known today as *Moore’s Law*. Although Moore was actually talking about complexity of chips, Moore’s law actually states that the number of transistors that can be found on the latest generation of highest performance chips doubles every 24 months. This *law* assumes that chip complexity is proportional to the number of transistors.

1.2 A BRIEF HISTORY OF THE MICROPROCESSOR

The history of the microprocessor can be explained from several perspectives. For instance, there is a notable relationship between the development era and the width of the processor datapath. This small history will be organized based on processor bit-width.

It is generally believed that the first microprocessor ever built is the Intel 4004 processor. The 4004 had a 4-bit wide datapath. It employed a $10\mu\text{m}$ silicon-gate PMOS technology, implementing a total of 2300 transistors. The operation speed was around 740KHz. Two more processors claim the title of first-ever microprocessor. They are Texas Instruments' TMS-1000, and Garrett AiResearch's Central Air Data Computer. It is difficult to know which company actually built the first working microprocessor. A patent for the concept of the microprocessor was granted to Gary W. Boone from Texas Instruments [8]. It was filed on August 31, 1971. Gary W. Boone is also the author of the computer-on-a-chip patent [9], the so-called *microcomputer patent*.

Intel eventually evolved its 4004 chip to an 8-bit version called 8008 in 1972. Shortly after, many successful microprocessors appeared, like Intel's 8080 or Zilog's Z80. Motorola introduced the MC6809 in 1978. This chip was remarkable for its optimization which led to a completely hardwired logic design. For a chip of this complexity it was much more usual to use a microcoded design, in which a small microprogram controls the execution of instructions inside the processor. In a design with hardwired logic, instructions can be executed much faster. The downside is that the design complexity is considerably increased.

The first 16-bit microprocessor most likely was National Semiconductor's IMP-16, introduced in early 1973. It was a multi-chip implementation. The first single-chip 16-bit microprocessor probably was TI's TMS 9900. However, it was Intel's 8086 design which was most successful. Intel introduced the 8086 in 1978 as a cost effective way of porting software from the 8080, and succeeded in winning much business on that premise. The 8088, a version of the 8086 that used an external 8-bit data bus, was the microprocessor in the first IBM PC, the model 5150. Following up their 8086 and 8088, Intel released the 80186, 80286 and, in 1985, the 32-bit 80386, cementing their PC market dominance with the processor family's backwards compatibility. The 8086 was Intel's first processor to feature a memory management unit (MMU). The MMU gave the 8086 the capability of multiprogramming through the use of virtual memory system.

Shortly after the introduction of the 8086, the first chips with 32-bit datapaths started to appear. Motorola introduced the first processor of its 680Xo line, the 68000, in 1979. The 68K, as it was widely known, had 32-bit registers but used 16-bit internal data paths, and a 16-bit external data bus to reduce pin count, and supported only 24-bit addresses. The combination of high speed, large (16 MB) memory space and fairly low costs made it the most popular CPU design of its class.

Intel's first 32-bit chip was not an x86 chip. It was the iAPX-432, which it introduced in 1981. It had an advanced capability-based object-oriented architecture, but poor performance compared to other competing architectures such as the Motorola 68000. Probably for this reason it was never a commercial success and Intel decided to abandon the project.

It was during this era that the concepts of RISC (Reduced Instruction Set Computer) started to grow in importance. The RISC philosophy is based on the observation that complex instructions are only rarely used, and that most compilers don't know to generate them anyway, and thus proposed to design chips with small instruction sets, focusing on those instructions which are most used. MIPS was an important player in this field with the introduction of the R2000 (1984) and R3000 (1989) processors. They were used in high-end workstations and servers by Silicon Graphics, among others.

From 1985 to 2003, the 32-bit x86 architectures have become increasingly dominant in desktop, laptop, and server markets, and these microprocessors became faster and more capable. Intel had licensed early versions of the architecture to other companies, but declined to license the Pentium, so AMD and Cyrix built later versions of the architecture based on their own designs. During this timespan, these processors experience a 1000-fold increase in complexity and performance.

64-bit processors have been in several marketplaces since the 90's. The first 64-bit processor was the MIPS R4000, introduced in 1991. However, it was not until AMD's introduction of the first 64-bit IA-32 backwards-compatible architecture, AMD64, in September 2003, that these chips saw its introduction in the PC market. Intel followed shortly after by introducing its own x86-64 instruction set extension.

Increase of bit-width is of course not the only source of performance that microprocessors have exploited. In fact, increasing the bit-width does itself not provide speed-ups. The most important consequence of this is that programmers can make use of larger data-words efficiently and, more importantly, that they can efficiently index larger memory. This allows them to increase the capabilities of programs, a task which often involves increasing the memory footprint.

1.3 EVOLUTION OF PROCESSOR MICROARCHITECTURES

Technology miniaturization and frequency increase has been the main source of speed-up for microprocessors. We have already talked about Moore's Law and the transistor. The other source of speed-up have been improvements in the processor's microarchitecture. We will now provide a small historical perspective on this issue.

Early processors were extremely simple. Given an instruction they would fetch it, decode it, execute it and finalize it by either modifying register contents or main memory. All these operations were conducted in a single cycle of machine operation. This is not a very efficient way of processing as not all instructions need to perform the same number of operations. A first improvement was to split the execution of an instruction into different phases, called *stages*. In this way, fetch, decode, execution, etc.. all execute in different clock cycles. This allows to increase the clock frequency as the work performed in every cycle is much smaller. In addition, it allows to adapt instruction latency to the real work carried out.

A much larger benefit can be obtained when realizing that the different stages can be overlapped, as long as access to structures and instruction ordering is correctly maintained. This technique, called *instruction pipelining*, was first implemented in the IBM 7030 "Stretch". Only two machines of this model were ever delivered, the first of them to Los Alamos Scientific Laboratory in 1961.

Although the Stretch did not achieve the expected performance, it remained the fastest machine in the world until 1964, when the CDC 6600 was introduced. This processor introduced many new concepts, like the use of *scoreboarding* – which allows the processor to issue instructions based on dependencies rather than ordering – or being able to exploit multiple functional units. This resulted in a processor that was up to three times faster than the IBM Stretch. It remained the fastest computer until 1969, when it was replaced by its successor, the CDC 7600, which made extensive use of integrated circuit technology.

The introduction of the IBM 360/91 in 1967 marks another key point in the development of microarchitecture. This machine introduced many new concepts that are now extensively used in microprocessors, such as register renaming, branch prediction or dynamic scheduling via Tomasulo's algorithm. Tomasulo's algorithm combined dynamic execution with register renaming. This enables dynamic scheduling even in the presence of write-after-write hazards. The previous technique of scoreboarding was limited in this respect since it forced the processor to stall whenever an output dependence was about to be violated.

Branch prediction is one of the most important innovations to keep processor pipelines filled. Tomasulo already pointed out the advantages of speculation, but it was not until two decades later that heavy research was conducted on branch prediction. Jim Smith first described a per-branch prediction scheme using a 2-bit saturated counter in 1981. Ditzel and McLellan described how to predict the branch target address in the context of AT&T's CRISP processor (1986) using a structure now called the Branch Target Buffer (BTB). Finally, Patt et al. described multilevel predictors using a branch history for every branch. This allows every branch to be predicted based on the last N outcomes of the same branch. These schemes can be combined with global history information (i.e. the last M outcomes for all branches) for higher prediction accuracy.

Another important concept that allowed to increase processor performance was multiple-issue. Multiple-issue means that the processor can send multiple instructions to the functional units in the same cycle. Of course this can be done only if the instructions are independent. *Multiple-Issue* concepts were first introduced in an IBM project called ACS that was being developed in the 1960s. However, no ACS processor was ever built. John Cocke from IBM made a subsequent proposal for a multiple issue processor that dynamically makes issue decisions. He coined the term *superscalar* for this kind of processor. The processor he proposed was called *America* (1987) and its design principles can be found in the IBM POWER-1 processor (1989).

Another interesting proposal developed in the 1980s was the decoupled approach proposed by J. E. Smith at the University of Wisconsin. In his proposal, memory instructions and arithmetic instructions are executed in different units and ordering between both is achieved using a set of queues. In the decoupled approach, computation of one type of instructions can advance the other type, but ordering is always correctly maintained. Smith introduced these concepts into a processor called the Astronautics ZS-1 (1987). The POWER-2 processor also made use of similar concepts.

Hardware speculation is another key concept used in today's microprocessors. Speculation is the result of executing instructions without knowing for sure that they are on the correct path. This can, of course, only happen in processors using branch prediction. The problem is how to integrate it into the machine without violating program semantics. In other words, wrongly speculated instruction should not affect the correct execution of the program. In recent processors this is solved by adding an in-order *commit* buffer generally called the *reorder buffer*. Smith and Plezskun explored how to use buffering to maintain precise instructions and first described the reorder buffer in 1988. Later, in 1990, Guri Sohi added register renaming and dynamic scheduling, making this mechanism useful for speculation. Following a different path, in 1986, Patt's group extended Tomasulo's algorithm to support speculation by studying schemes that checkpoint the processor state and allow the processor to restart if speculation is shown to be wrong. Many concepts developed by these research activities are still found in today's out-of-order processors.

Besides instruction pipelines, another component that has been the focus of much research is the memory subsystem. The memory system is currently a bottleneck for many applications, particularly due to the long access times to main memory when compared microprocessor speeds. However, this has not always been the case. In the first generations of microprocessors, main memory ran at speeds similar to that of the microprocessor, if not faster. These systems were never starved due to lack of data from memory. But evolution of microprocessors has changed this picture dramatically. While processors have been increasing performance at a speed of around 50% every year, main memory has improved at most at a yearly rate of 10%. As a result, memory has moved farther and farther away from processors. With today's frequencies and memory access times, a main memory round-trip will take over a hundred cycles. How to overcome this disparity has been an issue for a long time. The fundamental technique to alleviate long memory latencies has been the introduction of memory caches, first proposed in 1964 by Maurice Wilkes. Caches allow repetitively-access data as well as closely grouped data to be accessed at microprocessor speeds. Although this is very beneficial for execution, the memory access time disparity has still a large impact on performance. This thesis deals with the memory problem from a different perspective. It proposes microprocessor techniques to overcome this problem for applications where caches are not an effective solution.

The schemes that have been presented so far have greatly improved the performance of microprocessors. They all share one characteristic: they are hardware schemes independent from the software. They also share one major disadvantage: they all increase the complexity of the processor. An alternative to these schemes is to use software schemes to increase performance by explicitly specifying the parallelism of the software. In such a scheme, the processor fetches an instruction word that consists of multiple independent operations. In general, each operation slot drives a different functional unit. In this sense it shares similarity to horizontal microcode. Architectures following these design principles are called *Very Long Instruction Word* (VLIW) architectures. Much of the early work in these topics was conducted by Josh Fisher and his students at Yale University, including coining the term VLIW and introducing the compilation technique of *trace scheduling*. VLIW architectures are among the first processors that were able to perform multiple-issue. They were able to advance their dynamically scheduled counterparts in these respects since they didn't need to deal with problems resulting from exception handling or virtual memory. One early VLIW proposal was Fisher's ELI-512 (ELI standing for *Enormous Longword Instruction*) in 1983. It aimed at an instruction word consisting of 512 bits. In the long term VLIW processors were not able to take over the market as they are not well suited to execute under variable memory latencies (such as current multi-level cache hierarchies). In addition, current compiler capabilities to extract high ILP and fill the VLIW slots is sub-optimal. Finally, VLIW processors have a very hard time dealing with code compatibility, almost an anti-natural concept in the VLIW world, but of utter importance in some markets. Concepts from VLIW architectures can be found in some recent processors, such as the Transmeta Crusoe/Efficeon, the Phillips Trimedia or Intel's line of Itanium processors.

Vector Processors are a different approach that deserves attention. A vector processor is a processor that includes instructions which, in addition to registers, can operate directly on vectors. In this context, a vector is a collection of data of the same type organized into an array. By operating on vectors we can express large quantities of operations in a concise way. This not only reduces fetch bandwidth, it also allows the processor to better schedule the operations, allowing it to considerably increase performance at low cost. Two early vector processors were the CDC STAR-100 and the TI ASC. Both were introduced in 1972. Their architecture was *memory-memory*, meaning that vectors were stored in memory, sent to the processor for computation and stored back in memory. Such a processor is not really too different from a scalar or superscalar processor. Only the ISA and some logic for the processing of these instructions needs to be added. It is even possible to reuse the functional units of the scalar part for the vector part.

One major breakthrough in vector processing was the Cray-1 (1976). It was the first vector processor by Seymour Cray, who earlier had worked on the CDC 6600 and CDC 7600. The Cray-1 introduced many new concepts. To start, it was the first *vector-register* architecture. Vectors of data were stored in large on-chip vector registers that

could be accessed very fast. This considerable reduced start-up overhead time for vector operations and was a big win for the architecture. Other notable innovations include being able to efficiently process vectors with non-unit strides (i.e., linear but non-sequential positions in memory) and the introduction of *chaining*. Chaining allows the individual results of the vector operations to be bypassed to a following vector operations that uses them, resulting in overlap of vector operations. Finally, Cray-1 also had strong scalar performance, which is important as focusing only on the parallel/vector part of the program would shift the application bottleneck to the scalar part. This is an interpretation of *Amdahl's Law*.

The 80s saw the appearance of a new concept of supercomputer. The minisupercomputer was a small scale vector processor-based computer that sold for about a tenth of the price of supercomputers. This allowed many new customers in this market. The most successful company in this segment was Convex which produced a uniprocessor vector processor (the C-1 in 1985) and a small multiprocessor (the C-2 in 1988). One of the reasons for the success of their systems was that Convex focused considerably on software performance. First, they were software-compatible with Cray. Second, the implementation of their vectorizing compiler was very efficient. And third, their implementation of the UNIX operating system was of high quality. This shows the growing importance of software, even in the supercomputer market which traditionally has focused only on hardware performance.

This completes the small historical introduction on microprocessor-related techniques. During this introduction we have left out references for technical papers covering the materials. If the reader is interested in a more detailed historical perspective from a more technical point of view, and including references, I suggest taking a look at the *Historical Perspective and References* sections in Hennessy and Pattersons famed computer architecture book [10].

1.4 CURRENT TRENDS

Since the invention of the 8-bit chips until now, microprocessor performance has increased considerably. As already explained, the foundation of this increase is twofold. First, technology miniaturization has allowed to build smaller and faster transistors. Frequency is, of course, one of the main components of processor performance. The other is microarchitecture. Inventions such as superscalar processors, out-of-order execution or branch prediction have allowed the performance of processors to gain an additional order of magnitude. However, both trends are slowing down. While Moore's law continues and transistor count increases, the frequency of these chips has not been growing a lot. High-frequency chips consume large amounts of energy and generate a lot of heat. Packaging technology is having trouble handling all the dissipated power. On the other hand, available parallelism within applications and

the disparity in memory access times makes new aggressive microarchitectural less attractive.

As a consequence, chip makers have changed their focus towards development of multi-core architectures. A multicore processor places multiple processing cores on the same chip. This increases the maximum performance, but requires a different programming model. One of the first multicore architectures was IBM's POWER4, introduced in 2001. Almost all major microprocessor manufacturers have followed this path and offer multicore chips today.

2

MOTIVATION FOR THIS THESIS

Improvement in processor design is motivated by bottlenecks. In the early days, and until the mid 90s, the main limitations in the evolution of processor design have been chip area and design complexity. This has somewhat shifted recently to include power consumption instead of chip area. In fact, this is the main reason why industry has chosen to pursue multicores instead of attempting to further improve single-thread performance.

The kind of bottlenecks that microprocessor industry has focused on in the early days were mainly related to instruction timing issues. The first generation of microprocessors were single-cycle architectures. These machines fetched, decoded and executed one instruction in every cycle. The cycle time of the processor was determined by the latency of the longest instruction. Execution proceeded as shown in Figure 2.1.

Such a processor suffers from imbalance due to long-latency instructions slowing down all other instructions. To make it worse, the most common instructions tend to be small and fast instructions, such as integer additions or comparisons. Slow operations such as multiplications or divisions appear quite infrequently in the instruction mix. Thus a great deal of time these processors are idle waiting for the next clock cycle. As a solution, multicyle processors were introduced. These

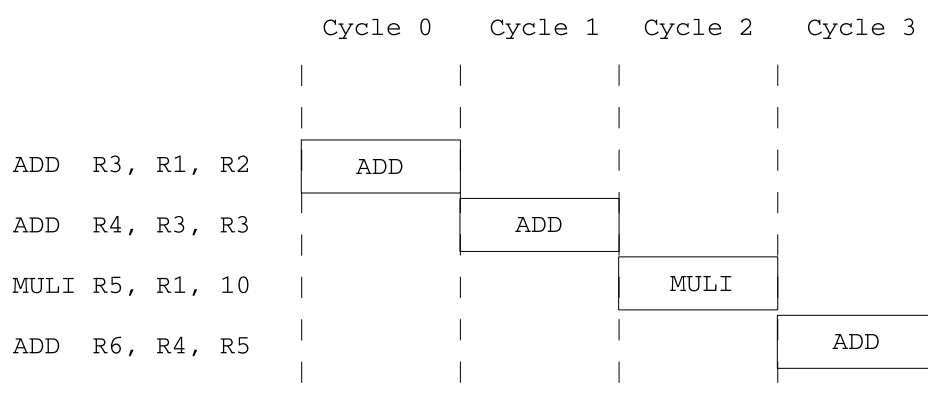


Figure 2.1: Execution of example code in a single-cycle single-issue processor

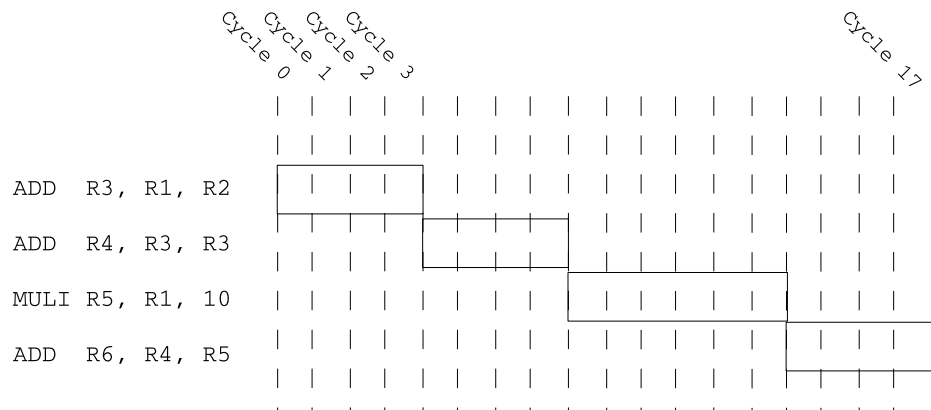


Figure 2.2: Execution of example code in a multicycle single-issue processor

processors partition the execution of an instruction into multiple stages (each one cycle). For example, an addition may consist of: fetch, decode, execute and writeback. A load could include fetch, decode, address calculation and some memory access stages. Finally, a multiplication could consist of fetch, decode, N execution stages and a writeback stage. In a multicycle implementation additions and loads will not need to wait for the N execution stages of the multiplication in order to complete (see Figure 2.2). As a result, performance can be increased. In addition, the cycle time of the processor is considerably reduced.

With the introduction of multicycle processors it becomes apparent that the functional units suffer from a relatively low utilization. For instance, in the previous example the fetch unit is accessed only once every four cycles. The utilization is even worse if long-latency instructions are present in the instruction stream. One solution is to introduce *pipelining*. With pipelining, the processor starts a new instruction every cycle. This means that instruction execution is partially overlapped with other instructions. Complexity is introduced because it is now necessary to track dependencies among different instructions. Enforcing dependencies may result in stalls in the pipeline. However, the amount of parallelism that this technique achieves is very high. In fact, pipelining is probably the single technique in processor design that provides most speed-up. Current processors cannot be understood without a comprehensive understanding of pipelining. Figure 2.3 shows a pipelined processor executing the example code segment.

Motivated by this observation, the focus of much research switched to further exploiting *Instruction Level Parallelism (ILP)*. One technique are multiple-issue processors. These architectures fetch and execute multiple instructions every cycle. The complexity is similar to pipelining, but adds additional pressure to the dependency tracking and resource scheduling hardware. Figure 2.4 shows the benefits obtained by this technique.

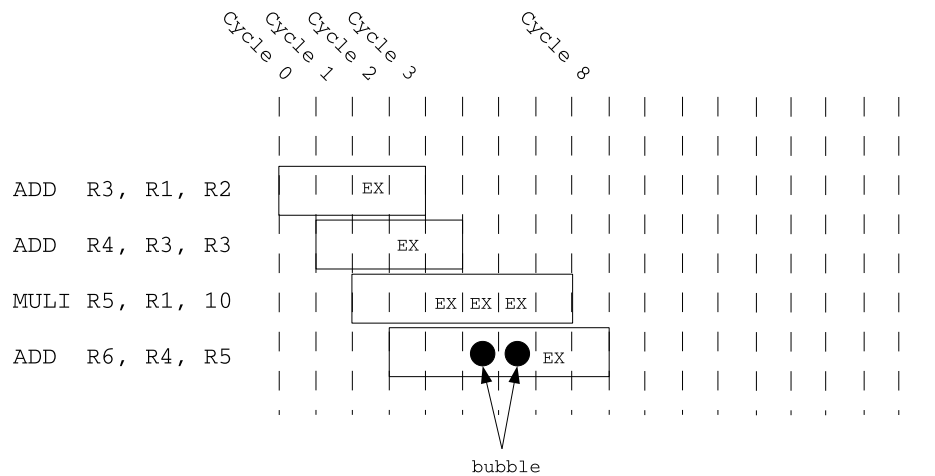


Figure 2.3: Execution of example code in a pipelined single-issue processor

ILP can be further exploited by introducing techniques such as dynamic scheduling and register renaming, collectively known as *out-of-order execution* (OoO). Out-of-order enables a great deal of flexibility in instruction processing. Thanks to dynamic scheduling, instruction execution no longer needs to follow program order. Instead the real dependencies specified by logical source and destination registers can be honored. Speed-ups are obtained because instructions need not wait on previous instructions on which they do not depend. For example, in a loop it is very common that the code controlling the loop boundaries is independent from the loop body. Since the loop body is in general slower (i.e., it is larger and often has worse locality), in an OoO processor the control code will be able to progress forward without having to wait on the loop body to complete. In general, out-of-order execution is very interesting due to its ability to hide the latencies arising from L1 misses (L2 hits) whenever independent instructions can be executed. Figure 2.5 tries to show the main characteristics of out-of-order execution.

Out-of-order execution enables exploitation of large amounts of parallelism but at the cost of much higher complexity. First, registers need to be renamed to eliminate false dependencies like Write-after-Write (*output dependencies*) and Write-after-Read (*antidependencies*). Second, select and wake-up logic needs to track instruction tags sent on the result busses to schedule when instructions are issued to functional units. Third, new hardware is necessary to enforce the ordering specified by the program. This is called *in-order commit* and is normally performed using a buffer called *ReOrder Buffer* (ROB). Finally, in the event of exceptions –most commonly misspeculations– the processor needs to be able to roll back to a previous state and restart. This is normally accomplished with the ROB structure or taking checkpoints at branches. In short, out-of-order allows a lot of performance improvement, but it

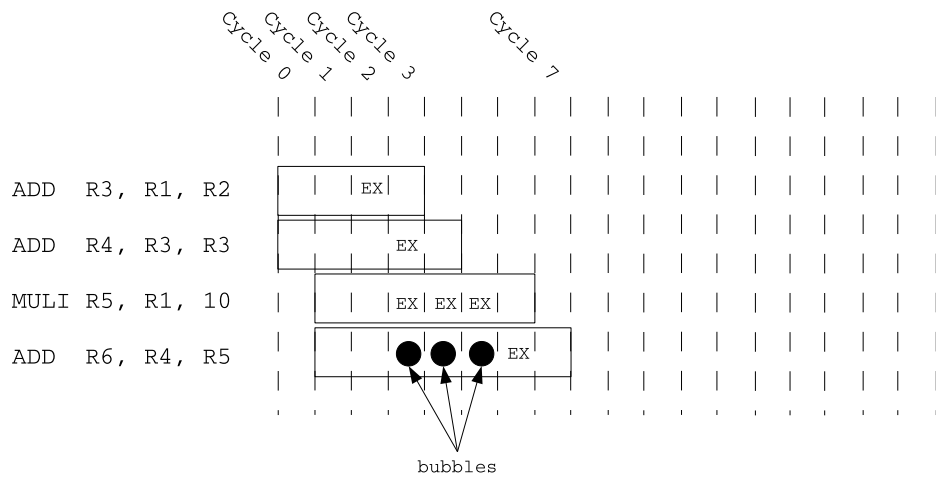


Figure 2.4: Execution of example code in a pipelined multiple-issue processor

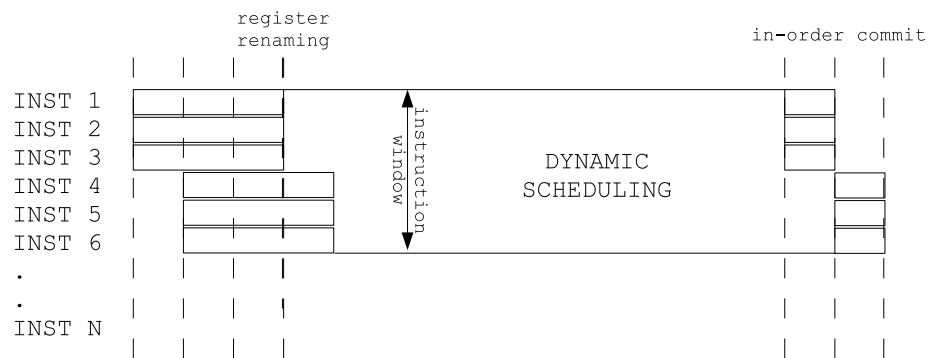


Figure 2.5: Conceptual Execution on a three-way out-of-order processor

comes at considerable hardware and energy cost. For moderate ROB sizes (~ 100 entries or smaller) this technique is acceptable.

Many current designs use the technique of OoO execution. However, the number of instructions that can be in-flight at the same time, and the number of instructions that can be fetched in a single-cycle are no longer increasing. The reason is that at window sizes of around one hundred instructions very large ROB increases are necessary to obtain significant speed-ups. Performance is now increasingly being limited by the *Memory Wall*. The memory wall refers to the disparity in speed between processors and main memory. It is now common for processors to wait hundreds of cycles until a memory miss completes. During this time a processor could fetch more than thousand instructions. But current ROB's can only keep around one-hundred instructions in the buffer. Thus the processor needs to stall for around 90% of the cache miss service time, notably reducing performance. In chapter 4 we provide a more detailed analysis of the effects of the memory wall.

As mentioned, slightly increasing the instruction window no longer adds much performance. However, adding a couple more instructions may considerably increase the pressure on the structures that support out-of-order execution. To make it worse, it increases the power consumption of the chip. This observation has motivated a change of focus in industry, which now focuses on integrating multiple cores on a single chip. Often each of these cores will be a speculative out-of-order processor with moderate structure sizes so that the cycle time and power consumption of the chip can be kept within limits.

But multicores have their own problems. The most notorious problem is that software developers are not well trained to exploit this kind of architecture. To fully exploit multicores it is necessary that the developer is able to come up with a parallel execution model for the application. This is not always trivial. In addition, it depends on the algorithms that have been chosen for the application. Some algorithms are inherently sequential and cannot be parallelized. Non-parallelized applications will only make use of a single core of the multi-core system, an undesired inefficiency that plagues current systems. Yet many of these algorithms contain important quantities of irregular parallelism as well as distant parallelism. Irregular parallelism can be difficult to exploit in the presence of the memory wall. On the other hand, with current ROB sizes it is impossible to exploit distant parallelism.

While the multicore trend will likely proceed into the future, ILP studies have confirmed that still a lot of unexploited parallelism exists in general codes. In order to extract this parallelism we need to operate at a larger granularity. L2 accesses are already well hidden by mechanisms such as out-of-order execution. Instead we need mechanisms specifically targeted at hiding L2 cache misses and exploiting distant parallelism.

In this thesis we will propose a design methodology to design processors capable of efficiently operating at multiple granularities. Such processors will need to efficiently exploit:

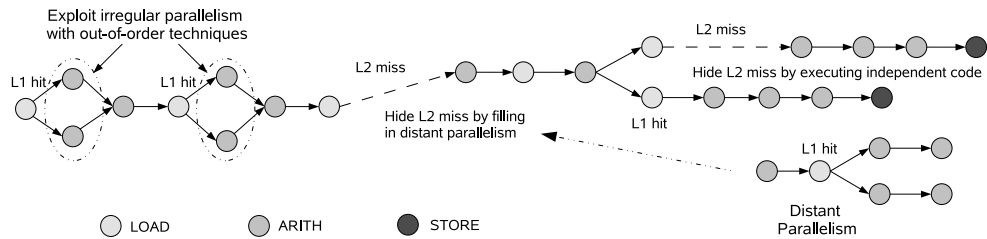


Figure 2.6: Obtaining performance in memory-wall limited processors

- Irregular ILP code with good locality. This code represents the vast majority of fetched instructions and is thus very important to execute it efficiently.
- Irregular ILP code with bad locality. Although miss-dependent instructions are a small fraction of the program, the impact of memory accesses on performance is large. It is thus necessary that instructions depending on these latencies can be reordered to recover lost performance.
- Distant Parallelism. When memory accesses cannot be reordered to exploit parallelism, the only chance is to recover performance by executing distant instructions belonging to an independent function or code segment.
- Early execution of cache-missing loads within short latency code results in very beneficial memory lookahead (accurate prefetching).

Figure 2.6 depicts a Data Dependence Graph with annotated cache behavior. The figure tries to show how an architecture exploiting these three kinds of parallelism can obtain considerable speed-ups.

EVALUATION INFRASTRUCTURE

Throughout this thesis an effort has been made to reuse the simulation infrastructure to evaluate all proposals. This has the advantage that, as long as microarchitectural parameters are invariant, results can be directly compared from chapter to chapter.

3.1 SIMULATION INFRASTRUCTURE

The simulator that has been used for this research is the result of several research iterations performed on top of the SimpleScalar 3.0 code [11] within the Computer Architecture Department at the Technical University of Catalonia (UPC). We worked on top of the kilo-instruction simulator used during the research of *Out-of-Order Commit Processors* [2].

Since the simulator is based on the original code of SimpleScalar, it is an *execution-driven* simulator. From SimpleScalar, the simulator inherits the functional execution front-end. The cycle-accurate back-end has been completely rewritten over the time. Out of the two main front-ends for SimpleScalar we choose to use the Alpha ISA front-end [12].

We will now describe the internal architecture for the generic out-of-order processor model. The architectures presented throughout the following chapters required specific modifications to be conducted on top of the basic simulator. These additional modifications are described in the corresponding chapters.

Internally, the simulator models a pipeline with five stages. Although the pipeline stages perform tasks similar to the original 5-stage MIPS R2000 pipeline, the simulated processor model is heavily out-of-order and speculative. The simulator focuses on simulating the out-of-order execution loop. To simulate deeper pipelines we increase the misprediction penalty, which models the additional cycles required to refill the execution window. The modeled stages are: `fetch`, `decode`, `issue`, `writeback` and `commit`. The `decode` stage handles decoding, renaming and IQ insertion. `Issue` and `writeback` form the out-of-order window while `commit` stalls until instructions complete. The simulator models a sixth stage named `storebufferrelease` which frees resources and sends completed stores from the store buffer to the cache.

3.2 BENCHMARKS AND TRACES

The simulator front-end can execute two kinds of formats:

- *Binaries*, statically compiled on Digital Unix (Tru64 UNIX).
- *Traces*, composed of an initial memory/register state plus a registry of system calls that the simulator will encounter during execution. The system calls include information on the modifications on memory that the system call produces and thus need not really be executed.

Traces have the advantage that they implicitly encode a fast forward which is thus not necessary during simulation. In addition, a trace does not require the user to have the simulated binaries and input files available. While this is of interest in some cases, in our case we will be using traces because of the first reason.

The traces that we have used throughout the evaluations correspond to SPEC CPU 2000 binaries. The binaries themselves originate from the `simplescalar` webpage and were compiled on Tru64 UNIX version 4.0 using Digital CC compiler version DEC C V5.9-008.

We reuse the trace collection that has been used in previous research from the Computer Architecture Department at the Technical University of Catalonia [2]. This trace collection consists in single execution points of 1000 million instructions for both SPECINT and SPECFP –one per benchmark– totalling 26 simulation points. Although the traces collect up to 1000 million instructions, only the second 100 million instructions belong to the simulation point. The first 100 million instructions are used for warming up caches and remaining processor structures. The simulations performed throughout this thesis correspond to the mentioned 100 million simulation points. The behavior shown by the benchmarks in these simulation points varies considerable. Very few have a single phase, while most have multiple recurring phases or just a couple of distinct phases. Only one of simulation points (`gcc`) shows completely irregular behavior over the execution length.

3.3 COLLECTING AND REPORTING RESULTS

When reporting results, data should be collected and reported for all benchmarks. However, the explosion of information resulting from reporting 26 benchmarks for every evaluation doesn't help in understanding the overall picture. Therefore some sort of averaging is often employed in reporting results.

The most intuitive mean for reporting results in single-thread evaluations is the arithmetic mean. Many researchers also use the harmonic mean. In some cases, the geometric mean has also been used. Which metric to use depends on what is being measured. One of the main metrics in which we are interested is the average Instructions per Cycle (IPC). IPC is a measure of the speed of the processor. Being

a velocity-like quantity, many researchers have opted for the harmonic mean to measure this average. This follows this reasoning based on car travel: Imagine you are travelling a distance of 200km by car. You like driving relaxed so the first 100km you drive at just 50 km/h. When you notice that you are running late, you react and the second 100km you drive at 200 km/h. Your total driving total time has been $(100km/50km/h) + (100km/200km/h) = 2h + 0.5h = 2.5hours$. So the average speed must have been $200km/2.5h = 80km/h$, which does not correspond to the arithmetic mean but to the harmonic mean. This reasoning applies well to cars. However, it doesn't really fit too well in computer evaluation. First, the length of the simulation point is not correlated with the length of the application. Second, a benchmark set does not measure the behavior of sequential runs of all the benchmarks on the target machine. If this were the case –and all benchmarks contained the same number of instructions– the harmonic mean would make sense, but sequentially running different benchmarks is *not* representative of computer usage.

The harmonic mean gives more weight to the slower parts. This intuitively makes sense for the previous example of a car running at 50 km/h and then at 200 km/h, as the final driving time is more affected by the slower run. But in the case of computer evaluation it doesn't make sense. For example, does the fact that a benchmark such as *mcf* progresses much slower than, e.g, *mgrid*, make it more important? This doesn't make much sense, as users that run *mcf* will likely not run *mgrid* afterwards. They are two completely different applications.

For this reasons we decided to use the arithmetic mean throughout this thesis. This mean gives every benchmark the same weight and, in addition, it is simpler to understand. We did not consider using the geometric mean as no arguments supporting it were found.

The discussion so far has focused on single-threaded workloads. When multiple threads are evaluated this picture changes somewhat. The problem for multi-threaded workloads is that the metric choice involves also the important topic of *fairness*. Average mean, or equivalently, *throughput* just measures the total number of instructions per cycle of all threads together. This is interesting, but it does not take into account that one thread's benefit may come at the cost of another thread's performance. In other words, quiet frequently the best way to improve the throughput is to focus only on the subset of benchmarks for which specific techniques/optimizations exist that considerably increase their speed. When implemented in a multithreading processor, these techniques may reduce the performance of other threads, but the global throughput may still be increasing. Such a situation is unfair, particularly if the threads belong to different users. For this case, using a technique based on the harmonic mean makes more sense. The common procedure is to measure the speed-ups experienced by all the threads individually and then compute the harmonic mean. This metric is generically referred to as the *harmonic mean* and it is specifically interesting for measuring fairness. In chapter 7 we will be measuring

multithreaded workloads. Both throughput and fairness results will be provided for the evaluated architectures.

4

THE MEMORY WALL AND EXECUTION LOCALITY

The goal of this thesis is to propose a processor microarchitecture that efficiently overcomes the memory wall for memory-limited programs. In particular we target codes with high memory-level parallelism (MLP) but which cannot currently be efficiently exploited because instruction windows of conventional processors can only keep a very limited number of in-flight instructions. A large instruction window can also overcome single cache misses by executing distant instructions independent of the cache miss. In other words, the new microarchitecture needs to exploit three kinds of parallelisms: a) unstructured parallelism in code with high data locality, b) unstructured parallelism in code with bad data locality and c) distant parallelism. Parallelism in code with good locality is efficiently exploited by consolidated techniques such as dynamic scheduling [13]. Unfortunately, to exploit parallelism in the shadow of cache misses we cannot rely on current microarchitectural design techniques. The window size of current processors is too small to hide memory accesses or reorder distant code. Novel concepts need to be developed to complete this goal.

This chapter introduces the *Execution Locality* principle, a new concept on which all later work in this thesis is based. Execution Locality is a conceptual way to understand program execution in the presence of long latency events and clustered execution. An early version of the concepts explained in this chapter was included in our HPCA-12 contribution [14]. It has since undergone several refinements. This chapter includes a more comprehensive overview of the concept.

4.1 OVERVIEW

The most important impediment to single-thread processor performance is the memory wall [15]. Thanks to improvements in process technology and microarchitecture, modern microprocessors are now clocked in the gigahertz range and reach peak performances of thousands of MIPS. Unfortunately, improvements in memory systems, particularly memory latency, have not kept pace with improvements in microprocessors. As the rate of improvement continues to diverge, we reach a point where some instructions can issue in just a few cycles while others

Config	L1 access time	L1 size	L2 access time	L2 size	memory access time
L1-1	1	∞	-	-	-
L2-10	1	32KB	10	∞	-
MEM-100	1	32KB	10	2MB	100
MEM-400	1	32KB	10	2MB	400
MEM-1000	1	32KB	10	2MB	1000

Table 4.1: Configurations used to quantify the effect of the memory wall

may take hundreds or even thousands of cycles, because they depend on uncached data. High-latency instructions caused by cache misses can slow a processor well below its peak potential.

The analysis we perform in this chapter allows establishing an important relationship between the memory hierarchy and the number of cycles instructions wait for issue. This relationship gives rise to the new concept of execution locality, that is used in subsequent chapters to build a decoupled processor architecture with multiple design advantages over a centralized architecture.

4.2 EFFECTS OF THE MEMORY WALL

Memory latency is by itself not necessarily a limitation to the ability to exploit instruction-level parallelism. Processor microarchitecture and program characteristics also play a crucial role in determining the effect of memory latency on performance.

We have quantified this effect by observing the impact of several memory subsystems on a range of out-of-order cores. Out-of-order execution is necessary to evaluate the ability of these cores to hide latencies of independent instructions. The resources of all out-of-order cores evaluated are sized such that stalls can only occur due to shortage of entries in the ROB. Thus, providing the number of ROB entries is sufficient to describe the processors. The fetch-and-decode bandwidth is 4 instructions per cycle and the models fully support speculative execution. Using SPEC2000 as the workload, six different memory subsystems are evaluated for IPC. Table 4.1 details the configurations. In the table, memory access latencies are given in processor clock cycles.

Figures 4.1 and 4.2 show the IPC resulting from these six memory subsystems. In these figures *Size of Instruction Window* is equal to the size of the ROB. An analysis of SPEC FP benchmarks shows that even for the slowest memory subsystems it is possible to recover most IPC simply by scaling the processor to support thousands of in-flight instructions. With a ROB of 4K entries almost all architectures perform close to the perfect L1 cache configuration. The reason for this speed-up is that in SPEC FP there is a vast amount of independent, correct-path instructions that can be executed while long latency events are in-flight.

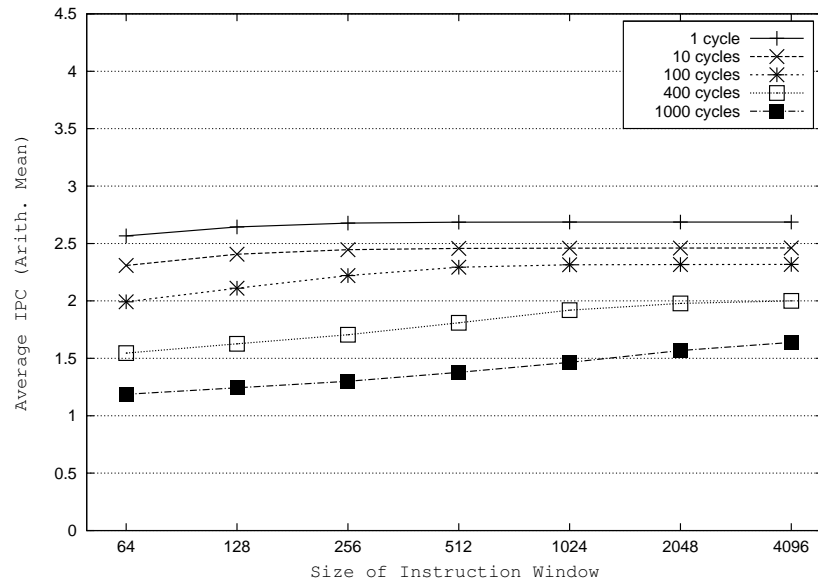


Figure 4.1: Effects of memory subsystem on SPECINT

For SPECINT benchmarks the analysis is quite different. These workloads often misbehave in three ways that can put a high latency load on the critical path even for large instruction windows: pointer chasing behavior, lack of independent instructions and branch mispredictions depending on uncached data. The latter will force a complete squash of the instruction window with a devastating effect on performance. Note that branch mispredictions depending on short latency events can be recovered from quickly (e.g., using a checkpoint-based scheme similar to that of the MIPS R10000 [16]) and thus have less impact on IPC. Figure 4.1 shows that, in the case of SPECINT, recovering the lost IPC by using large instruction windows is not an effective solution. Different techniques need to be researched for this case. In any case, large-instruction windows are not detrimental to integer codes. They simply are not as helpful as they are for floating point workloads.

4.3 EXECUTION LOCALITY

Clearly, one direction to build an architecture capable of overcoming the memory wall for numerical codes is to produce a chip with resources to handle thousands of in-flight instructions. This is analogous to the design methodology used for current out-of-order chips with respect to handling L1 cache miss latencies. These latencies are quite small. An L2 hit normally takes anywhere from 5 to 20 cycles. As new technologies have appeared and higher frequency chips have been produced, the cache distance has slowly increased. To hide the newer latencies, the resources on the chip (instruction queue, register file, load/store queue and reorder buffer) need

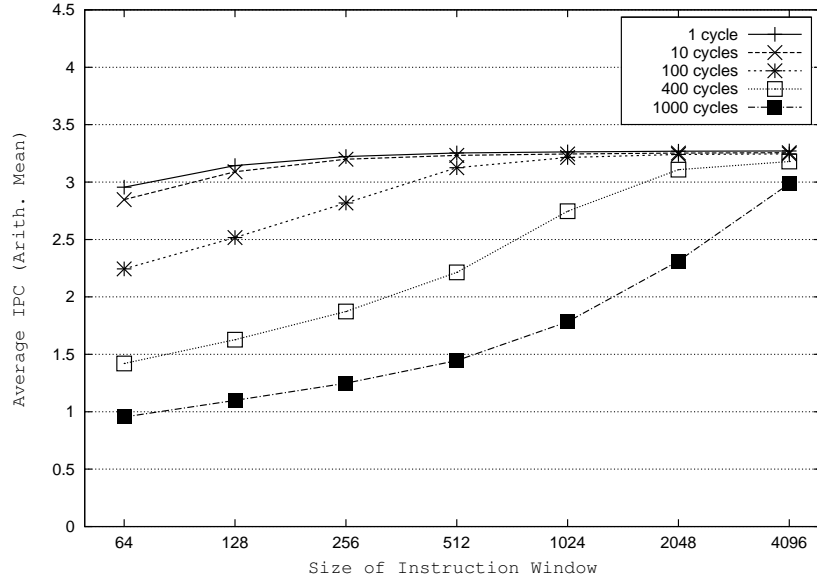


Figure 4.2: Effects of memory subsystem on SPECFP

to be commensurately increased to maintain the previous IPC rates. This approach is feasible for dealing with increasing L1 miss latencies. However, increasing the structures more than tenfold to overcome the memory wall is totally impractical due to power and timing issues [17]. Thus, most research on large-window processors has focused on replacing non-scalable resources with new structures that scale much better, using a variety of techniques like virtualization, hierarchical implementations, distributed approaches, etc.. Unfortunately, applying a specific design solution for every resource can also considerably increase the complexity of the design.

Instead, we would like to overcome the memory wall problem with a single integral design solution. This way we believe that low complexity designs can be accomplished. We start by collecting additional information on the execution of programs dominated by the memory wall. To this end, we perform the following instruction-centric analysis: using an out-of-order architecture with a memory latency of 400 cycles, a 2MB L2 cache and a 4K ROB we run SPECFP benchmarks and analyze the average number of cycles instructions wait in the instruction queue until they are issued for execution. Figure 4.3 shows that the *issue latency* of instructions follows defined patterns. This regularity is highly correlated with the behavior of the memory subsystem.

Several groups/peaks can be seen in the figure. Most instructions (about 73%) execute in fewer than 300 cycles while the remaining 27% of all instructions execute around 400 cycles or more. This distribution is highly related to the memory accesses of loads present in the instruction stream. The front-end normally advances at high speed, fetching up to 4 instructions per cycle. Thus, an instruction slice can be

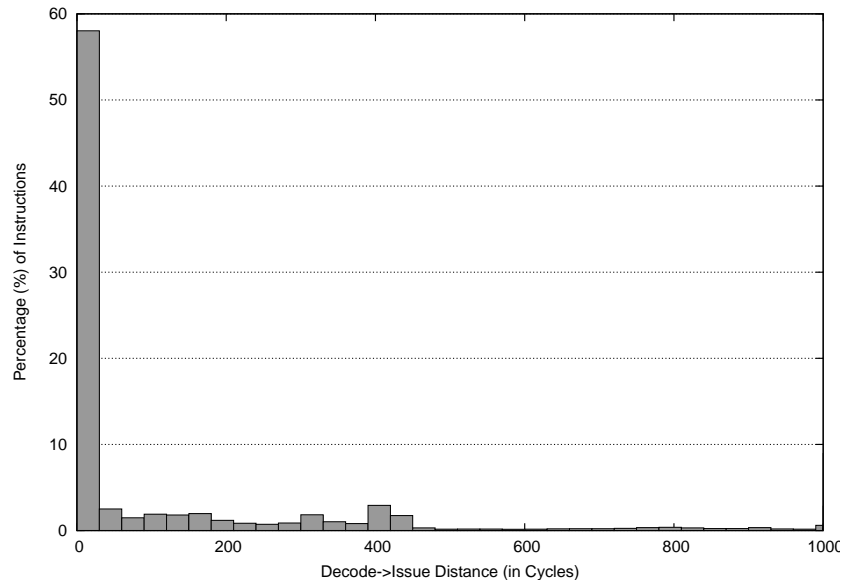


Figure 4.3: Average Distance between Decode and Issue/Writeback

fetched in a relatively small number of cycles. The 73% of instructions that execute in fewer than 300 cycles are instructions that read operands from the cache, from registers or from instruction bypass. The small peak around 400 cycles corresponds to instructions that depend on a single cache miss. Similarly, the small peak around 800 cycles corresponds to instructions that depend on two chained cache misses. For SPECINT applications we observed that almost all correct-path instructions are issued shortly after decode. The reason is that, in SPECINT, it is very likely that the instruction chain following a long latency event ends up on the wrong path. After recovery, long latency loads may have been turned into cache hits (prefetching effect) and thus most correct-path code ends up having short issue latency. In addition, SPECINT benchmarks also tend to have smaller working sets and generate fewer cache misses (*mcf* being the single exception in SPECINT2000).

This distribution can be interpreted in three ways. The first is in terms of register availability. An instruction within the instruction window can have its source registers in one of four states: (1) READY, i.e., with computed value; (2) NOT READY, waiting for cached data; (3) NOT READY, waiting for other instructions to writeback; and (4) NOT READY, waiting for a cache miss or for a cache-miss dependent instruction. In terms of execution, cases 2 and 3 have a very similar behavior. This allows to establish a new classification of instructions: Instructions that are linked in the data dependence graph (DDG) via at least one long-latency register (ie, in state 4) are classified as having mutually *low execution locality*. The remaining instructions which are issued within short intervals have *high execution locality*. Execution locality is a property that describes pairs of instructions as a

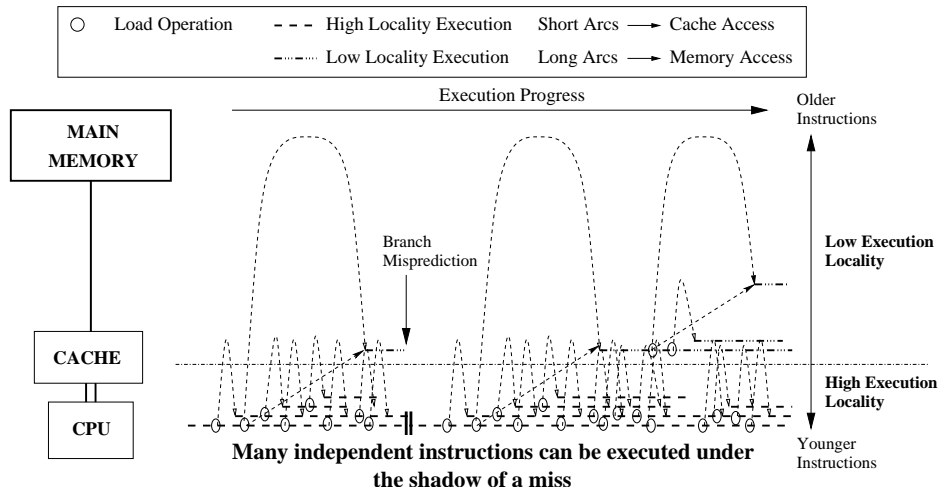


Figure 4.4: Execution Progress and Execution Locality

function of the number of cache misses that happen between both. The group of instructions that has high locality among themselves is referred to as an *instruction cluster*. We call this the *cluster-view* of *Execution Locality*.

These concepts are described by Figure 4.4 which depicts the likely execution of a sample program. This figure represents one of the key properties of programs which is Karkhanis' observation that *many independent instructions can be executed under the shadow of a load miss* [18]. Large window processors, such as KILO-Instruction processors [5], profit from this characteristic to hide the latencies of long-latency misses. Figure 4.4 also shows the detrimental effects of mispredictions depending on cache misses. Therefore, to establish a relationship between execution locality and performance it is necessary to take into account the criticality of loads. In general, loads that drive a low-confidence branch are very critical. These cache accesses will strongly determine performance [19].

Execution locality is a concept mainly derived from caches and data locality. However, execution locality can be given a second interpretation, which will be more interesting in later chapters. This second interpretation takes the point of view of the decode stage and observes locality as the issue distance from the decode stage. From this point of view, instructions have high locality if they execute *local* to the decode stage (*local* defined as execution within a short amount of cycles) and they have low locality if they execute *far* from the decode stage (i.e., many cycles after decode). This interpretation does not take into account the fact that instructions belonging to a low-locality instruction cluster have, among themselves high locality. This is not very important because low-locality instructions are very tolerant to additional latencies and the performance of low-locality clusters is not very important. What matters is, as will be seen, that the clusters can be reordered to hide memory access latencies. This second interpretation is the *decode-view* of *Execution Locality*.

To avoid confusion we will be using this second definition by default, i.e., we will explain the microarchitecture of our proposals from the point of view of the decode unit. When referring to the first definition we will explicitly refer to it as locality among groups of instructions. As mentioned, there is yet a third interpretation of execution locality. This third interpretation is a direct consequence of the *decode-view* when applied to the decoupled microarchitecture introduced in the next chapter. For consistency, we delay its introduction until the next chapter.

4.3.1 EARTHQUAKE Example

The previous definitions of high locality are based on observing the large-scale behavior of execution in a memory wall dominated environment. In this section we want to show a detailed example to verify the observation and gain better understanding of the behavior. To this end we will focus on the function `smvp()` belonging to SPEC2000's earthquake benchmark. The code corresponding to this function is shown next:

```
void smvp(int nodes, double ***A, int *Acol,
         int *Aindex, double **v, double **w) {
    int i;
    int Anext, Alast, col;
    double sum0, sum1, sum2;

    for (i = 0; i < nodes; i++) {
        Anext = Aindex[i];
        Alast = Aindex[i + 1];

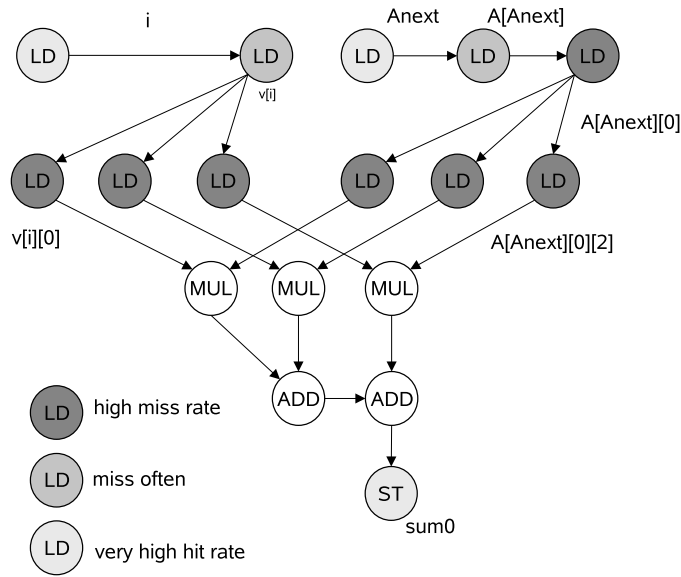
        sum0 = A[Anext][0][0]*v[i][0] + A[Anext][0][1]*v[i][1] + A[Anext][0][2]*v[i][2];
        sum1 = A[Anext][1][0]*v[i][0] + A[Anext][1][1]*v[i][1] + A[Anext][1][2]*v[i][2];
        sum2 = A[Anext][2][0]*v[i][0] + A[Anext][2][1]*v[i][1] + A[Anext][2][2]*v[i][2];

        Anext++;
        while (Anext < Alast) {
            col = Acol[Anext];

            sum0 += A[Anext][0][0]*v[col][0] + A[Anext][0][1]*v[col][1] + A[Anext][0][2]*v[col][2];
            sum1 += A[Anext][1][0]*v[col][0] + A[Anext][1][1]*v[col][1] + A[Anext][1][2]*v[col][2];
            sum2 += A[Anext][2][0]*v[col][0] + A[Anext][2][1]*v[col][1] + A[Anext][2][2]*v[col][2];

            w[col][0] += A[Anext][0][0]*v[i][0] + A[Anext][1][0]*v[i][1] + A[Anext][2][0]*v[i][2];
            w[col][1] += A[Anext][0][1]*v[i][0] + A[Anext][1][1]*v[i][1] + A[Anext][2][1]*v[i][2];
            w[col][2] += A[Anext][0][2]*v[i][0] + A[Anext][1][2]*v[i][1] + A[Anext][2][2]*v[i][2];
            Anext++;
        }
        w[i][0] += sum0;
        w[i][1] += sum1;
        w[i][2] += sum2;
    }
}
```

This function consists of a very parallel `for(;;)` loop. There are no loop-carried dependencies, except for memory dependencies on the "`w[][]`" matrix that need to be determined at runtime via memory disambiguation. The input file provided

Figure 4.5: DDG corresponding to representative `smvp()` statement

by SPEC2000 specifies a total of 30,169 nodes, which means that the outer loop performs over 30,000 iterations. Overall the `smvp()` function executes about 20 million instructions every time it is executed. This function is notable for the high amount of misses to the L2 data cache. Let us focus on the following statement which is representative of this code:

```
sum0 = A[Anext][0][0]*v[i][0] + A[Anext][0][1]*v[i][1] + A[Anext][0][2]*v[i][2];
```

Figure 4.5 shows the data-dependence graph of this statement with additional information on the cache behavior of the loads.

When the `smvp()` function is run on a large window processor an issue-latency histogram like the one shown in figure 4.6 is observed. The resulting distribution is fully in accordance with the DDG shown in the previous figure. As can be seen, most instructions have latencies either around 800 cycles (two chained misses) or 1200 cycles (three chained misses). Another important group of instructions are those that can issue shortly after being decoded. The shown bar corresponds to instructions with an issue-latency shorter than 30 cycles. In the case of the `smvp()` function these instructions correspond to the control path. In general, control path instructions tend to have high locality. This means that recoveries can normally be detected shortly after fetch. This is good for performance and also for power, as the number of dropped instructions will be small. Note, however, that many codes also feature some long latency loads driving branches. Even though the number may not be very high, the large latency of memory accesses combined with a hard-to-predict branch can make these branches quite problematic.

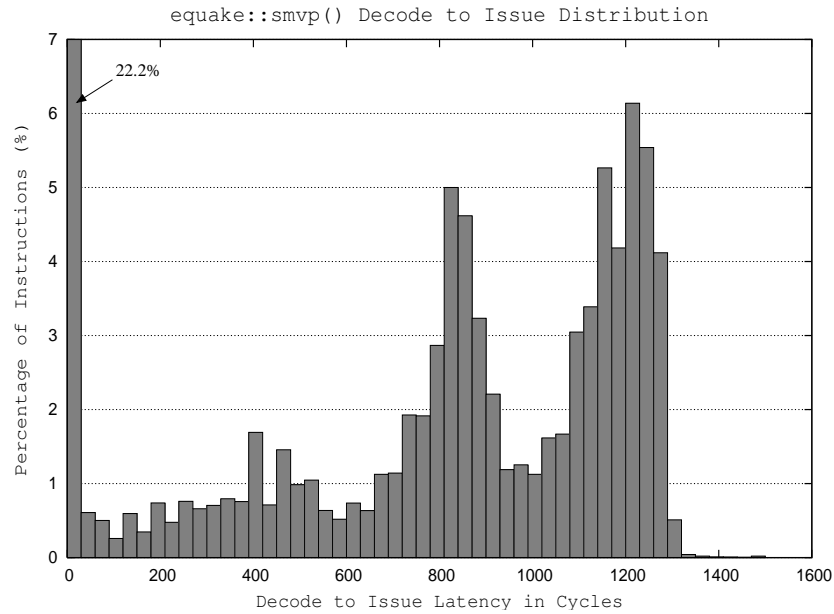


Figure 4.6: Instruction Decode \rightarrow Issue Distribution for `smvp()` function

The `smvp()` function is a clear example of instruction clustering. Each statement of the function has a series of input loads, some computation and an output store. The arithmetic core of each statement forms a cluster. The control path code also forms an instruction cluster, but with considerably better data locality. Figure 4.6 shows that some of the clusters have high locality and execute immediately (*control path clusters*), while others execute at a distance of around 800 cycles yet others at around 1200 cycles (*computation clusters*). A processor with a small window will have no chance of overlapping instructions among these different clusters and will therefore be severely limited by the instructions depending on either two or three chained cache misses. A processor that wants to avoid stalling should be able to handle around $1200 \times \text{FetchWidth}$ instructions, which for our 4-way architecture results in 4800 instructions. This number is the upper bound since the real fetch rate is smaller than the fetch width. Jumps in the control code will reduce the fetch rate since: First, the instructions after the jump are wrong path and, second, there is a limit to the number of branches that can be predicted every cycle. However, this is not really a big problem for a 4-way architecture.

Figures 4.7 and 4.8 show the performance that the two kinds of processors achieve on a fraction of the `equake` code performing several calls to the `smvp()` function. Both processors have 4-way fetch/decode units, yet the first one can only handle about 200 instructions in flight while the latter handles about 1500 instructions¹. Out of the 100 million instructions that are shown, the `smvp()` function corresponds to the

¹ this second model is actually that of a checkpointed architecture called FMC, which is explained in more detail in chapter 6. However, the overall behavior is similar to that of an out-of-order processor

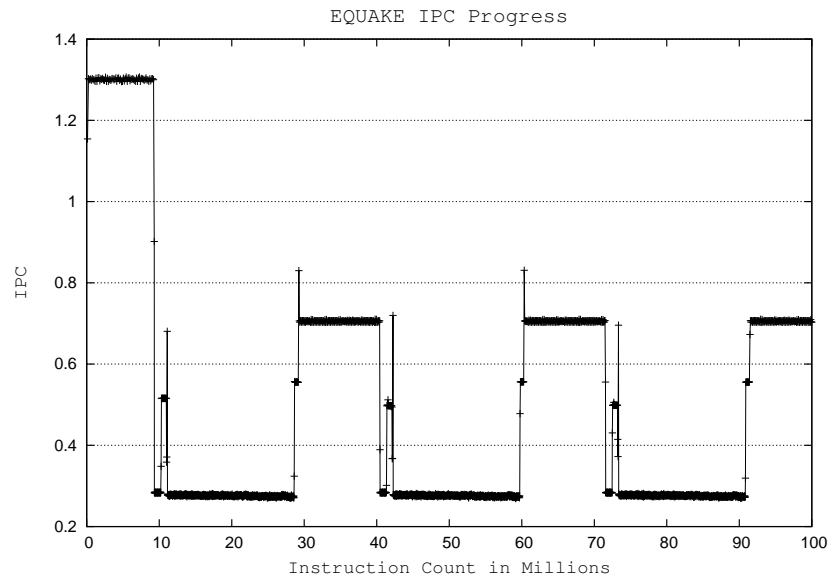


Figure 4.7: IPC progress for 200 instructions-in-flight processor

segments between 10-30 millions, 40-60 millions and 70-90 millions of instructions. The cluster reordering capabilities of the large-window processor are evident by the notable 7x speed-up that is obtained in this part of the code. Note that the remaining part of the code also achieves a considerable 5x speedup. There are many kinds of codes that can be sped up using large windows. This section has just shown a single example representative of the execution we are trying to speed up.

4.4 SUMMARY AND CONCLUSIONS

The main contribution of this chapter has been to introduce the concept of *Execution Locality*. We have started by making the observation that traditional out-of-order (OoO) execution is not a cost-effective way to handle code limited by long off-chip memory accesses. However, we made the point that this technique is effective to handle code dependent on cache hits. As will be seen in a later chapter (e.g., see Figures 5.6 and 5.7), it provides around 30% IPC improvement for this kind of code.

Studying program behavior, we observed that over 70% of all instructions in numerical codes are executed a short time after they are decoded. Making an analogy with memory subsystems we introduce the concept of *execution locality* and describe program execution using it. Instructions depending on short latency events are said to have high execution locality while instructions which depend on off-chip memory accesses are said to have low execution locality.

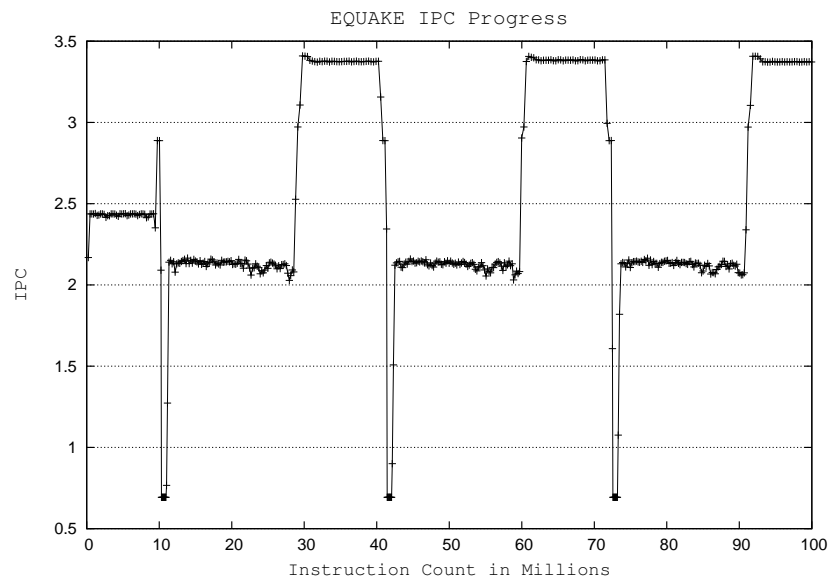


Figure 4.8: IPC progress for a KILO-Instruction Processor (~ 1500 in-flight instructions)

5

THE DECOUPLED KILO-INSTRUCTION PROCESSOR

5.1 OVERVIEW

In the previous chapter we have presented an analysis of program execution in the presence of long latency misses. This has led us to analyze issue locality among instructions and consider program execution as instruction clusters linked via memory misses. We refer to this representation as *Execution Locality*. In this chapter we propose a general processor design approach and a particular microarchitecture based on this definitions. This microarchitecture, called the Decoupled KILO-Instruction Processor, was introduced in our HPCA-12 contribution in 2006 [14].

5.2 EMULATING THE PERFORMANCE OF KILO-INSTRUCTION PROCESSORS

We start this chapter by taking a second look at Figure 4.4. As the processor fetches and executes instructions, the gap between the youngest and the oldest instruction tends to increase. Only some specific events, like mispredictions, exceptions or stores that end long-latency slices, may reduce this gap. Three features provide the most benefit to an architecture with a very large instruction window:

1. The fetch unit never stalls. Thus, high locality code continues to execute in the presence of several long latency loads.
2. Low locality instructions can be executed in parallel with recently fetched high locality instructions.
3. Many cache misses are issued from within high locality code, thus providing accurate memory lookahead (prefetching effect).

As Karkhanis *et al.* point out, most instructions that are fetched under the shadow of a miss are independent of it [18]. This means that *the amount of low locality code is small when compared to high locality code*. Most of the execution bandwidth is consumed by high locality code, as was shown in figure 4.3. Nevertheless, current architectures have to stall everytime they encounter a main memory access. Thus,

the small amount of low locality code present in the instruction stream considerably reduces performance.

To obtain a large window processor we will propose a microarchitecture that emulates the principles that have just been enumerated. Such a microarchitecture will need to follow these design guidelines:

- *Avoid Stalling Fetch*: It is important to continue fetching and executing instructions as a large part of the execution bandwidth is used for instructions with short issue latency. This allows executing load operations as soon as possible.
- *Relaxed Large Windows*: We will need to store many non-executable instructions during a cache miss. However, they can be executed in a relaxed fashion and they do not require high execution bandwidth. Therefore it is not necessary to rely on associative logic (CAM) for these low locality instructions.
- *Chained-Decoupled Execution*: Low execution locality code is highly decoupled from high execution locality code. There is no need to communicate values to high locality code as low locality code necessarily provide only values for low locality code. Only for speeding-up load execution it is interesting to provide back-communication for store-load forwardings.

5.3 A DECOUPLED KILO-INSTRUCTION PROCESSOR

Based on the guidelines outlined in the previous section we now introduce the main technique that this thesis proposes for the design of KILO-Instruction Processors: the *Decoupled KILO-Instruction Processor* (D-KIP). We will start by presenting a particular implementation for this processor. However, as will be seen, D-KIP is actually a design approach that allows for many implementations. The first D-KIP microarchitecture is the result of exercising the simplest implementation of the presented guidelines; it features 1) two different execution points, 2) unidirectional communication from high locality to low locality code (except for store-load forwardings) and 3) high latency tolerance within low locality code. The natural development of these ideas is to use one processor for each locality type linked via an unidirectional instruction buffer. This structure, shown in Figure 5.1, fully complies with the execution locality design guidelines. We will now describe the details of this microarchitecture before proceeding to analyze its performance and complexity characteristics.

5.3.1 Implementation of a D-KIP processor

In implementing the two-level D-KIP processor we will restrict ourselves to structures already implemented in conventional processors, allowing it to be built reusing

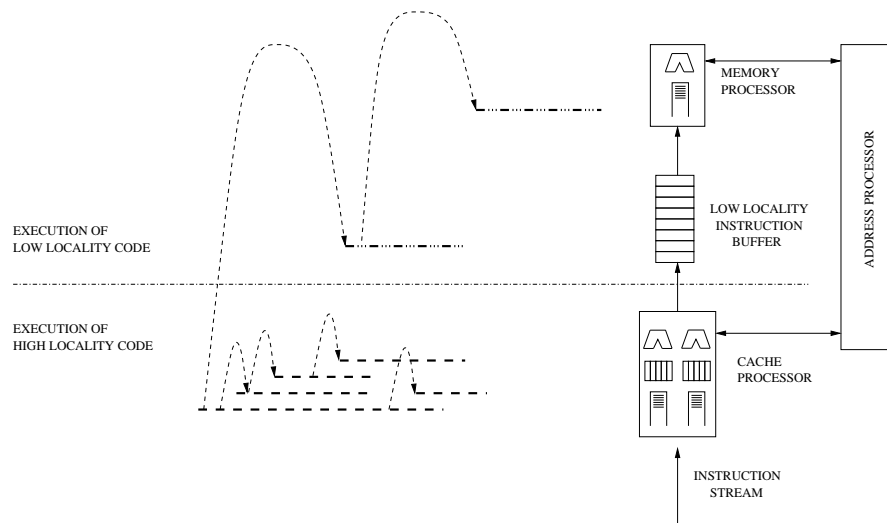


Figure 5.1: Two-Level Decoupled Processor

standard modules and focusing on the interfaces. Thus most of the work consists in adapting the structures to comply with the interfaces.

Figure 5.1 shows that the microarchitecture consists of two processors and a simple, non-issue-capable instruction buffer. As seen, there is also an address processor that handles memory operations. A small and fast *Cache Processor* is efficiently implemented using the MIPS R10000 as model [16]. It is useful to use a register-mapped architecture to provide fast branch recovery in the front-end. However, ROB-based recovery is also an option as long-latency instructions are precommitted from this processor and will not delay the recovery procedure. Because the Cache Processor does not handle miss-dependent instructions it can be smaller than current generation processors. On the other hand, the *Memory Processor* can be even simpler since it does not require high execution bandwidth nor dynamic scheduling. In this study we have modeled it using a simple Future File architecture [20] with modest execution capabilities (either in-order execution or OoO execution with a small window).

The execution model is as follows: Instructions are fetched by the Cache Processor (CP). They wait there until they are issued to the functional units or they are determined to be miss-dependent, meaning that they belong to a low locality instruction cluster. If so they are moved from the CP into the Low Locality Instruction Buffer (LLIB), a FIFO buffer where they wait until all long-latency events they depend on have finished. Note that there is one LLIB FIFO for floating point instructions and another for integer instructions.

When the long-latency load completes the value is kept temporarily in the address processor. When the depending instructions arrive at the head of the LLIB and the

load value is available, both the instructions and the value are inserted into the Memory Processor (MP). Execution then proceeds in the MP.

The LLIB Queues do not provide issue capability. In addition, they use a FIFO policy which greatly simplifies the register management. We will now discuss some of the modifications necessary to implement this microarchitecture.

AGING-ROB Like conventional processors, the Cache Processor contains a reorder buffer (ROB). This ROB, in addition to the conventional uses for exception and misprediction recovery handling, is also used to determine if instructions belong to a low execution locality cluster or not.

The initial D-KIP implementation proposes to use a scheme known as the *Aging-ROB*, which improves and supersedes the *pseudo-ROB* scheme introduced in [2]. The *Aging-ROB* is a FIFO buffer in which instructions progress at a constant pace. This allows checking whether instructions are short latency or not based on a timer. The size of the ROB should be the number of *aging cycles* multiplied by the commit width, which in our study is 4. The *Aging-ROB* is implemented as a circular FIFO with a head and a tail pointer that is moved forward at the same speed as decode after a fixed delay.

LLIB INSERTION AND WAKE-UP The *Aging-ROB* will check instructions after a certain number of cycles. This operation is called *Analyze* in the D-KIP pipeline and it determines if the instruction has low locality. There are two instruction types that require different analysis. Load handling is special and is described first. When a load arrives at the *Analyze* stage it must be determined if it already issued and, if so, if it resulted in a cache miss. To detect whether a load has missed in the L2 cache it is necessary to wait until the tag array has been accessed and the hit/miss information is available. This provides a criteria for the minimal size of the ROB. If the load missed, then this information is recorded in a bit vector that identifies long-latency (*low locality*) registers. This bit vector is called the Low Locality Bit Vector (LLBV). Otherwise the register is marked as short latency (*high locality*). The analysis of instructions does not stall unless the current situation of the load is still undecided. When an instruction other than a load arrives at the *Analyze* stage, it is first determined whether it has already executed or not. This information can be obtained from the ROB. If it has already executed the destination register is marked as short latency in the LLBV. Otherwise the sources are analyzed. If one of the sources is not yet available and is marked as long-latency in the LLBV then the instruction is classified as low-locality and is extracted from the CP and inserted into the LLIB. If none of the previous two situations applies, then the instruction status is not yet known. However, it will be resolved soon. In this case the architecture stalls in the *Analyze* stage until the instruction's status is known.

As instructions source registers marked as long-latency, the destination registers are also marked as long-latency in the LLBV. It is theoretically possible that, after a

while, all registers are marked as long-latency, an undesirable situation that would not improve performance as all instructions would be processed by the slower memory processor. However, it has been observed that in general this does not happen. There are various reasons for that:

- Recovery restores the full state to the Cache Processor. This operations clears all the LLBV completely. Note that for a multi-checkpointed architecture like the one presented in the next chapter, only a subset of LLBV bits are cleared.
- Short-latency operations, which represent around 70% of all executed code (see section 4.3), may overwrite registers that were marked as long-latency. After completion, the corresponding bit in the LLBV will have been cleared.

Long-latency loads are executed in the address processor, where the LSQ is located. Upon completion, the load value is stored in a FIFO buffer, one per LLIB. Each entry in this FIFO is associated to a long latency load. When the first depending instruction is about to enter the Memory Processor, and the load value is available, the value is first inserted from this buffer into the Future File of the Memory Processor from where the operation will then obtain the value.

REGISTERS Register management is a critical operation. We want to keep a minimum number of registers while having a simple and implementable management algorithm. The D-KIP architecture provides a solution for both goals using a distributed organization of registers allowing for distributed and independent register management.

The Cache Processor does not need any modification. The traditional algorithm of freeing registers once the renaming instruction is committed can be used. The Memory Processor proposed in this chapter uses a Future File architecture. It requires a logical register file in the front-end plus the associated space in the reservation stations. The low requirements of the MP enable it to have a very small number of reservation stations, so register management is very efficient in this part of the architecture.

The only structure that requires more attention is the LLIB since it may need to store many registers. However, the LLIB has some nice properties. First, this structure is in-order, so the order in which registers are inserted/extracted is known before-hand. The LLIB register storage (called LLRF, *Low Locality Register File*) works as follows: During `Analyze` it is determined if a long-latency instruction has `READY` operands. These operands are then inserted into the LLRF. In the Alpha ISA, which we are targeting, there will never be more than one `READY` operand per instruction (because otherwise the instruction would have been executed). Thus we need to allocate at most one register per instruction. The `Analyze` width of the cache processor we are modeling is 4. This will require an insertion rate of 4 registers per cycle in the worst case. The same applies to the instruction extraction rate. However,

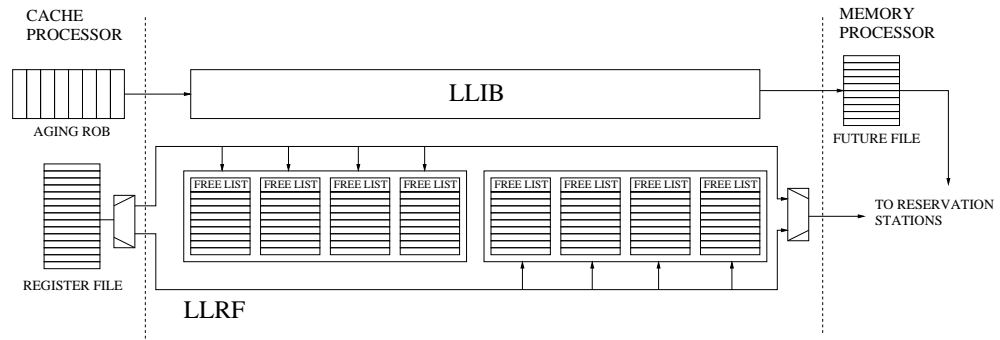


Figure 5.2: Schematic of the LLRF

the serial FIFO nature of the LLIB allows storing each register in a different bank. We model our LLRF as a banked register file with 8 banks. LLIB Insertion and LLIB Extraction always operates on a disjoint group of 4 banks. If a value to be read happens to be in a bank that is being written, instruction read is simply stalled for one cycle. This avoids conflicts in their access and allows to implement these banks as single-ported structures. The result is that each bank occupies minimal area and has minimal size. We calculated that the data array would be 6.6 times smaller than a centralized register file with 4 read ports and 4 write ports and the same number of entries based on Rixner's estimates for register files [21]. Each bank has a free list that works independently of the other banks. The instruction in the LLIB records the position of the READY operand during insertion. Actually, not all instructions have a READY register. There are many integer instructions that have a single operand and there are instructions with two long-latency sources. These will not require to allocate storage in the LLRF. We will analyze how to exploit this property to further reduce the size of the LLRF. A schematic showing the LLRF and the associated machinery is shown in Figure 5.2.

CHECKPOINTS The processor can recover mispredictions in the Cache Processor using the ROB or rename checkpoints. In the memory processor, these events, although less frequent, also occur. Recovery in the MP is supported by using checkpointing without a ROB [1]. Full checkpoints of available state are taken at specific instructions during the Analyze stage. In this stage, the instruction sees a register file composed of READY registers and some long-latency registers. Taking a checkpoint involves copying the ready values from the architectural register file (ARF) into the next free entry of the checkpoint stack. In addition, all operations that generate a long-latency register must be updated so that they writeback their destination values into this entry of the stack. This implementation is aided by a small RAM parallel to the LLBV. For each active bit in this vector, the RAM contains the position in the LLIB of the instruction that generates the value or a pointer to a

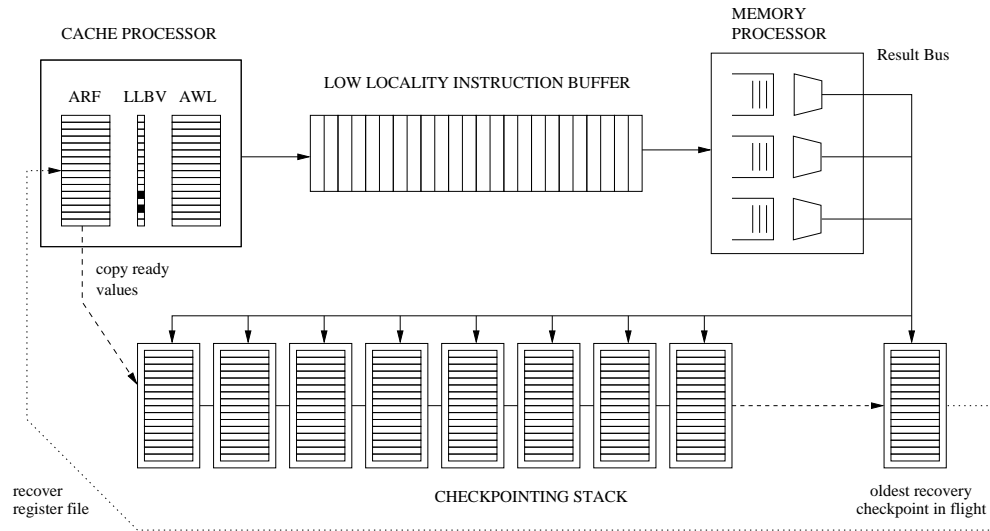


Figure 5.3: Schematic of checkpoints

previous checkpoint from which to copy the value. Keeping at least one checkpoint whenever long-latency instructions are in-flight is necessary so that no register value gets lost. This scheme is shown in Figure 5.3. The small RAM is referred to as the *Architectural Writers Log* (AWL). The number of ports of the checkpoint stack is not a problem as this structure is not frequently accessed.

In the LLIB subsection it was mentioned that the *Analyze* stage needs to wait for short latency instructions that have not written back their values. This simplifies checkpoint management as it makes sure that all short latency registers are available in the ARF when a checkpoint is taken.

5.3.2 Load/Store Queues

In this chapter we will not yet address a very important component of the microarchitecture: the Load/Store Queue (LSQ). A large window processor requires a scalable structure capable of supporting hundreds of in-flight loads and stores. In the D-KIP, the LSQ is decoupled from the remaining structures of the processor and it requires only small modifications to comply with the new interfaces. For now, we assume that the D-KIP features a centralized idealized Load/Store Queue. In chapter 8 we present a more efficient design of the Load/Store Queue that can be naturally integrated into decoupled KILO-Instruction designs.

The LSQ of this initial D-KIP design is decoupled from the execution cores in the same sense as the Decoupled Access-Execute Architecture [22]. In the D-KIP, the Address Processor needs to interface the Cache Processor and the Memory Processor. Load and Store ports can be asymmetrically partitioned – with more capacity for the

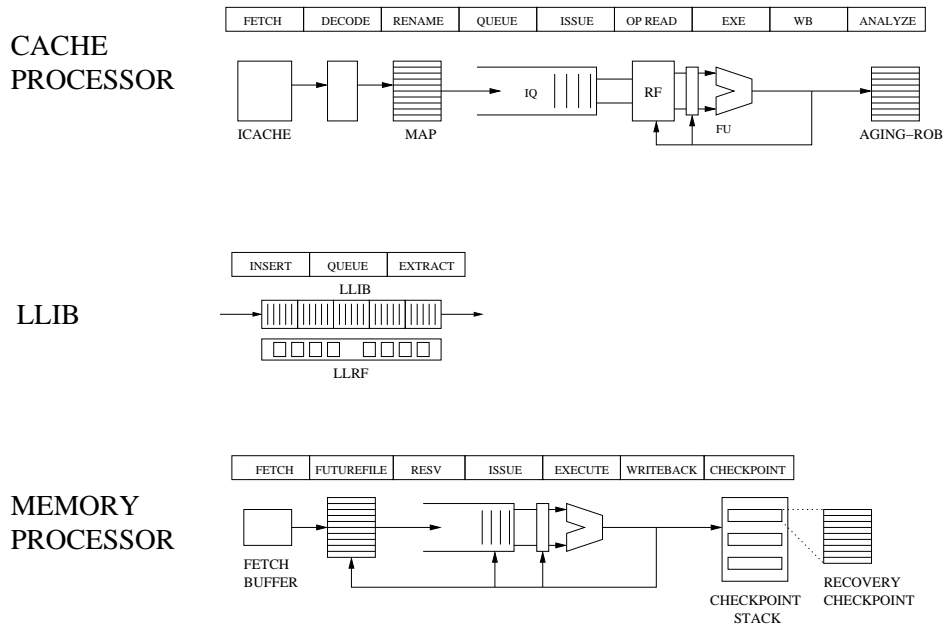


Figure 5.4: Full Pipeline of the D-KIP

CP – to support both cores. As was already mentioned, the address processor also needs to keep a FIFO buffer to store the results of long latency loads.

5.3.3 Pipeline

The full pipeline of the D-KIP architecture is shown in Figure 5.4. The three pipelines (Cache Processor, LLIB, Memory Processor) are cascaded. Most instructions only traverse the CP pipeline. Instructions will enter the LLIB when the *Analyze* stage determines that they belong to a low locality slice. Finally, insertion into the Memory Processor happens when the oldest instruction in the LLIB is either a load that has completed or a different instruction. Non-load instructions do not require additional checks as they must be high locality to instructions already executing in the Memory Processor.

5.3.4 Execution Locality: A different perspective

Register management in decoupled KILO-Instruction processors allows us to establish a third interpretation of Execution Locality. The basic mechanism by which the D-KIP determines if an instruction is high-locality or not is by analyzing register availability. If source registers are available in the Cache Processor, then the consuming instruction has high locality. If the register is not available, then the instruction

has low locality. In this case the register will be made available within the Memory Processor, where the consumer will finally execute. Thus, in the context of D-KIPs, execution locality can be understood as a consequence of register locality. In this sense, instructions are high locality if they source registers that are available *locally*. In this interpretation it is not reasonable to talk about degrees of locality as registers are either local or not. However, we keep the high/low locality terminology for simplicity since it is unambiguous. Since registers are local or not, instructions will also be either local or not. Instructions then have *high locality* when they execute in the same core as they currently are or *low locality* if they need to be migrated to a different core to be executed. This is the *register-view* of execution locality. It is probably the simplest to understand, yet it makes sense only in the context of D-KIP processors. It is similar to the *decode-view* of execution locality, with the difference that low locality also implies execution in a different processor core.

5.4 PERFORMANCE EVALUATION

The evaluation of the D-KIP is oriented toward verifying the introduced concept of execution locality and analyzing the efficiency of the processor.

5.4.1 *Simulation Infrastructure*

To model the D-KIP architecture we used the simulation infrastructure described in Chapter 3. The simulator is based on the infrastructure used in [2]. The back-end was extensively modified to model the LLIB as a FIFO structure, and the instruction queues were extended to handle in-order execution.

We will be evaluating several sizes for structures in the architecture. Table 5.1 summarizes architectural parameters that are invariant throughout this evaluation. Table 5.2 gives defaults for parameters that are going to be analyzed in this section.

5.4.2 *Performance Comparison*

First of all we have tested the performance of our architecture against other existing and experimental architectures. For this test we will use all the default values shown in Table 5.2. All LSQs are identical and have 512 entries. We will compare against three architectures:

R10-64 An out-of-order processor that models a MIPS R10000 processor. It has a ROB size of 64 entries and 40-entry issue queues. It is thus identical to the default Cache Processor.

Cache Processor	
Architecture	Merged Register File [16]
Fetch/Decode/Analyze Width	4
Branch Predictor	Perceptron [23]
ROB Timer	16 cycles
ROB Capacity	64 entries
ALU Units	4
Integer Multipliers	1
FP Adders	4
FP Multipliers/Divisors	1
LLIB	
Architecture	FIFO Queue
Number of Entries	2048 each
Insertion/Extraction Rate	4
Register Storage	8 banks, 256 regs each (max)
Integer Memory Processor	
Architecture	Future File [20]
Decode Width	4
ALU Units	4
Integer Multipliers	1
FP Memory Processor	
Architecture	Future File [20]
Decode Width	4
FP Adders	4
FP Multipliers/Divisors	1
Address Processor	
Architecture	Centralized
Load/Store Queue Size	512 entries
Number of Memory Ports	2 R/W ports (global)
L1 Cache Size	32 KB
L1 Cache Hit Latency	1 cycle
L2 Cache Hit Latency	10 cycles
Memory Access Latency	400 cycles

Table 5.1: Parameters of the architecture

L2 Cache Size	512 KB
CP Integer Queue Size	40
CP FP Queue Size	40
CP Scheduler	Out-of-Order
MP Integer Queue Size	20
MP FP Queue Size	20
MP Scheduler	In-Order

Table 5.2: Default values for variable parameters

R10-256 Another R10000-style processor, but with futuristic ROB and Queue sizes. The ROB here has 256 entries and the queues have 160 entries.

KILO-1024 This is an implementation of Out-of-Order Commit [2]. The pseudo-ROB has 64 entries and the *Slow Lane Instruction Queue* supports out-of-order extraction and can store up to 1024 instructions. The issue queues have 72 entries.

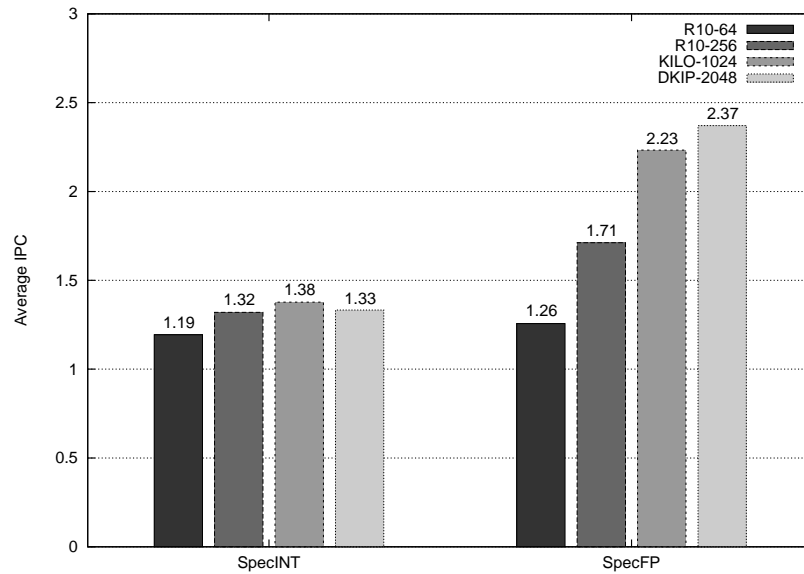


Figure 5.5: Performance of the D-KIP compared to baselines and a traditional KILO processor

D-KIP-2048 This is the microarchitecture described in this chapter. It features two LLIBs (one integer and one for FP) of 2048 entries each.

Figure 5.5 shows the IPC that these configurations yield. The figure shows large speed-ups when using the two large window processors. The floating point benchmarks in particular achieve a considerable performance benefit. The reason is simple: Branch prediction in these benchmarks is highly accurate. Thus, long latency instructions are almost never discarded and are simply processed in background after being reinserted from the long latency buffering system. Note that for integer benchmarks the performance of the D-KIP is less than that of the traditional KILO processor. The reason is that integer codes feature a lot of chasing pointers which will profit from an out-of-order instruction buffer such as the SLIQ [2]. However, the better performance is achieved at the cost of higher complexity, for instance, in the mechanism for register storage [3]. The performance advantage of the D-KIP in SPEC FP compared to the KILO comes from the fact that the D-KIP implementation implements two LLIBs and two memory processors, one integer and one for floating point, which allows it to exploit more parallelism and add limited out-of-order capabilities as the two pipelines operate independently.

Looking at Figure 4.2 we see that D-KIP-2048 achieves a SPEC FP performance similar to that of R10-768, with the difference that D-KIP-2048 has no associative structure larger than 40 entries. We will now analyze which parameters are most critical for these speed-ups.

5.4.3 Impact of Scheduler Policies and Queue Sizes

In this section we will evaluate the impact of the instruction queue sizes and impose a more severe restriction by forcing the queues to be in-order.

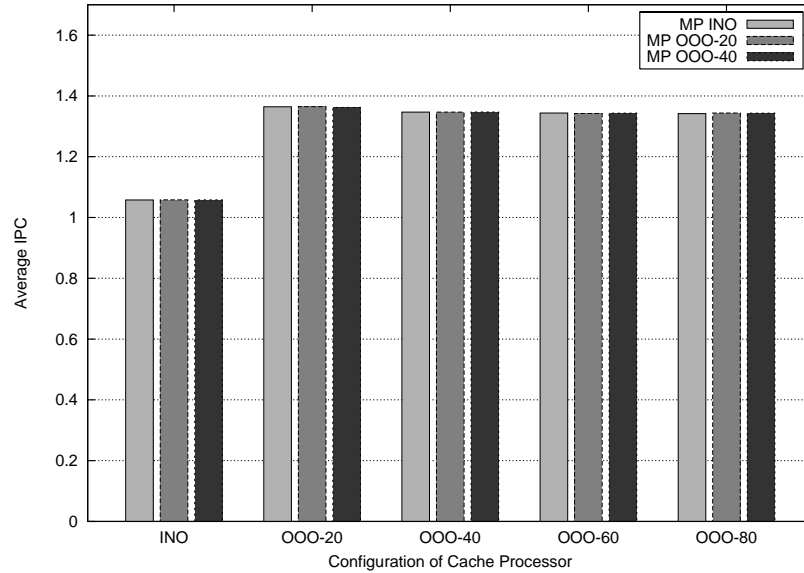


Figure 5.6: Impact of Scheduling Policy and Queue Sizes in SPECINT

We find that, for integer benchmarks (Figure 5.6), the D-KIP configuration is most sensitive to the scheduling policy in the Cache Processor. In this figure, *INO* means In-order, while *OOO-XX* means Out-of-order and *XX* refers to the size of the issue queue. Being out-of-order instead of in-order in this part of the pipeline increases the IPC by 29%. The D-KIP is insensitive to the configuration of the MP. This is reasonable as the MP processes only about 5% of all instructions for integer codes. The speed-ups are a sign that integer benchmarks profit from the D-KIP memory lookahead capabilities, but not from the additional processing capacity.

An analysis of SPECFP benchmarks shows that there is more potential for performance. Figure 5.7 shows the impact of the processor configurations on the execution speed for SPECFP.

First, the difference of in-order execution versus out-of-order execution in the Cache Processor again results in a speedup of 1.32. However, in this case there are still performance increases as we go to larger processors. Using an in-order Memory Processor, there is a 13% speed-up when going from an OoO Cache Processor with 20 in-flight instructions to an OoO Cache Processor with 80 instructions. In addition, as we go to larger CP processors, the configuration of the MP also has higher impact. Going from an in-order MemProc to an out-of-order MemProc with 40 entries in the queues increases performance in 1% when the Cache Processor is in-order.

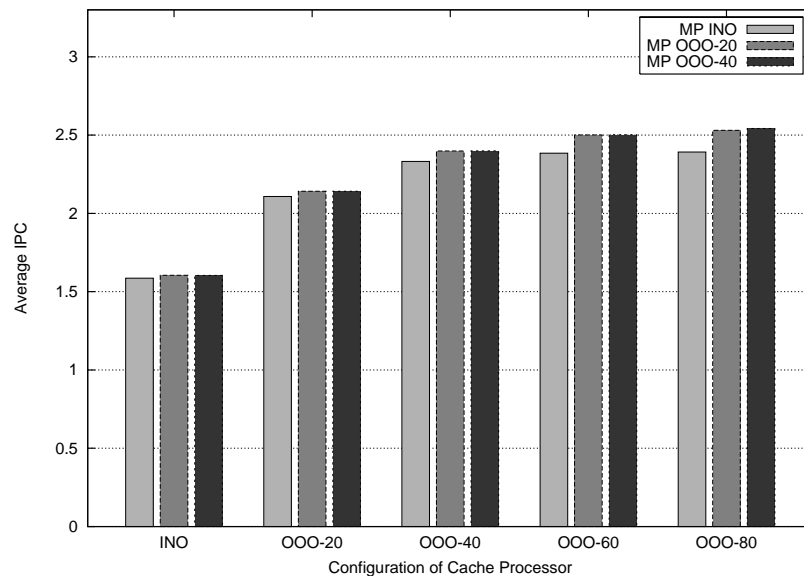


Figure 5.7: Impact of Scheduling Policy and Queue Sizes in SPECFP

The same variation improves performance in 6.3% when the Cache Processor is out-of-order with 80-entry queues. Figure 5.7 also shows that the OoO MP with 20 entries achieves almost the same IPC as the OoO MP with 40 entries. Thus, while OoO in the MP can be useful for aggressive configurations, the number of entries required can be kept small in general. The most aggressive configuration achieves an IPC of 2.54, up from the 2.37 achieved by our baseline D-KIP in Figure 5.5.

5.4.4 Impact of Cache Sizes

Our next analysis focuses on the memory subsystem. We want to see how the D-KIP behaves under a subset of different sized caches. Smaller caches result in higher miss rates which in the context of the D-KIP means that more instructions are going to be executed by the memory processor. If more instructions are executed in the MP, the scheduling policy there could be of higher importance.

Based on the previous section we select a subset of configurations labeled as *Config-CacheProc/Config-MemProc*. The configurations are: INO/INO, OOO-20/INO, OOO-80/INO and OOO-80/OOO-40. We will modify the size of the L2 Cache from 64KB to 4MB, maintaining all other parameters, and analyze the behavior of the architecture under different cache sizes. We also add the R10-256 configuration to show the differences with traditional OOO-based processors. The average IPC is shown in Figure 5.8 and Figure 5.9 for SPECINT and SPECFP.

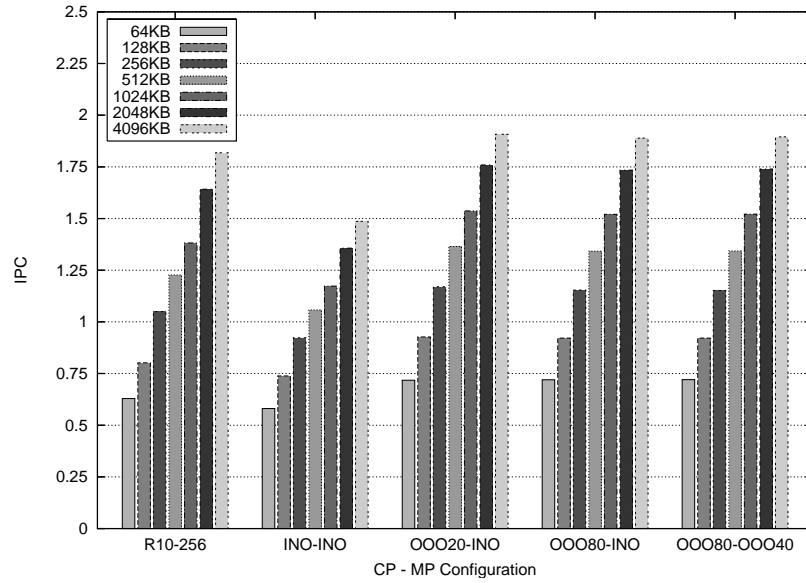


Figure 5.8: Impact of L2 Cache Size on SPECINT

The behavior of the D-KIP under cache variations in integer benchmarks is quite common. Each duplication of size in the L2 cache produces more or less a linear speed-up in the IPC. This is very similar to the conventional out-of-order processor. The interesting properties of the D-KIP appear in the SPECINT figure. IPC variations are much smaller here. The capacity of the D-KIP to process correct-path long-latency instructions without stalls allows it to be more cache insensitive. The difference between using a 64KB cache and a 2MB cache is less than 15%. It is only when a 4MB Cache is added that a considerable speed-up can be perceived. In any case, the maximum speed-up is still only about 24% compared to the INO-INO configuration.

From the figure it seems that the scheduling policy in the memory processor does not have that much influence on IPC variations. Thus we expect that even for the smallest cache of 64KB there is still enough execution locality so that the cache processor still processes most of the instruction stream. Our simulations confirm this hypothesis. Even for the 64KB cache, the 000-80/000-40 CP still processes 67% of all committed instructions in the cache processor (for SPECINT). When a 4MB cache is in use, the total number of high locality instructions increases to 77%. This difference is not so large considering that the cache size differs by about two orders of magnitude. It also explains why the D-KIP configuration is so tolerant to variations of the cache size. These properties get apparent when compared with the more conventional R10-256 configuration. For the range of caches observed the R10-256 configuration sees a total speed-up of 1.55 while the most aggressive D-KIP configuration sees

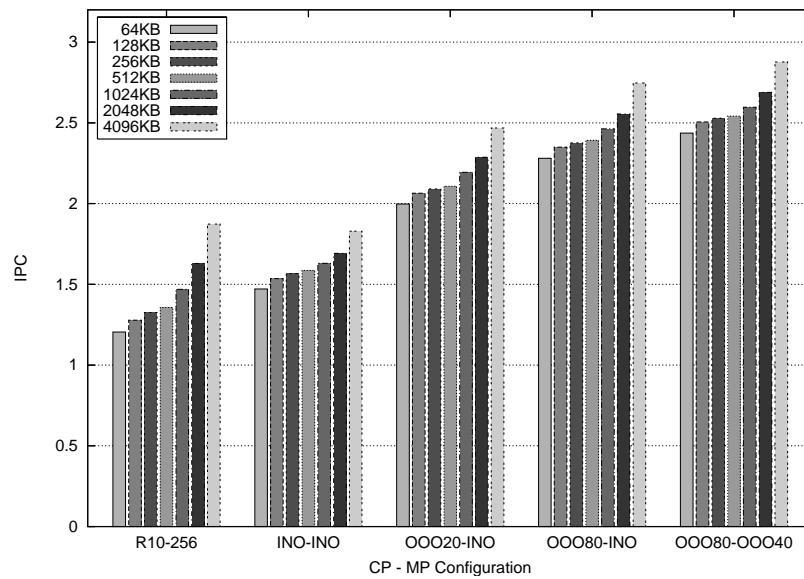


Figure 5.9: Impact of L2 Cache Size on SPECFP

only a speed-up of 1.18. This shows the tolerance of the decoupled architecture to different cache sizes when executing numerical codes.

5.4.5 Storage Requirements

The LLIB requires an associated register buffer that can be very large. However, not all instructions in the LLIB have an associated READY register. They can be single-source instructions or both sources may be long latency. If a large number of instructions do not require additional storage it may be possible to reduce the size of the register storage by not allocating a register.

Figures 5.10 and 5.11 show the maximum number of simultaneous instructions and registers in the LLIB during execution in our D-KIP architecture. Each LLIB can accommodate up to 2048 instructions and an equal number of registers values, which sums up to 16KB of storage divided into 8 banks, 2KB each.

The figures show that the real number of necessary registers can be much smaller. The worst case corresponds to integer data path instructions during SPECINT benchmarks. Many of these benchmarks contain long load chains with bad locality. This results in LLIB stalls due to fill-ups for four integer benchmarks. On the other side, none of the SPECFP benchmarks filled the LLIB. Note that in Figure 5.10 we are considering the integer LLIB while in Figure 5.11 only the floating point LLIB is being considered.

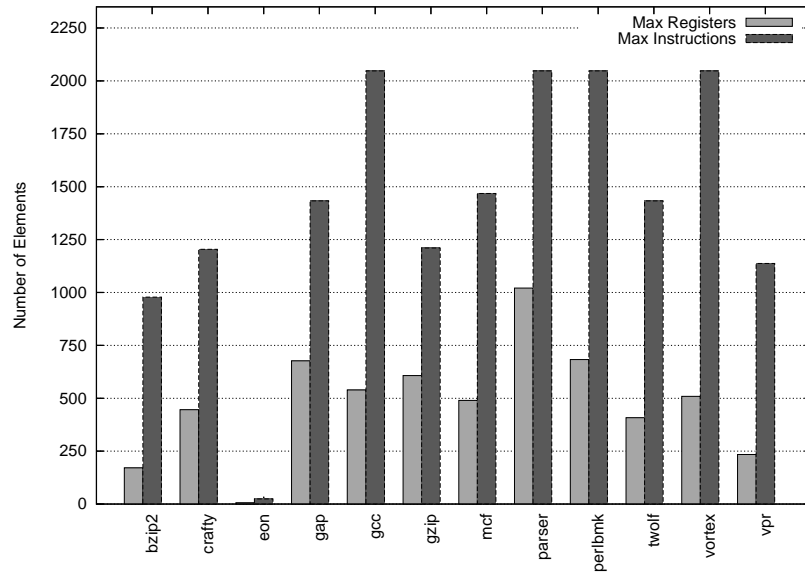


Figure 5.10: Maximum number of registers and instructions in the LLIB for SPECINT

These results suggest that the LLIB can probably be reduced considerably without significantly degrading IPC. For the code regions executed, an LLRF with only 1000 entries would have been enough. This number is large, however there is only a single benchmark that required more than 750 registers. The average number is much smaller, fewer than 500 registers. In any case, it must not be forgotten that the LLRF is a very regular structure consisting of 8 single-ported banks (the registers are distributed among the banks). Thus, the LLRF should not turn out to be a bottleneck, neither for area nor energy reasons.

5.5 DESIGN ISSUES

The D-KIP processor is an attempt to provide the benefits of KILO-Instruction processing at moderate cost. This section will focus on design issues and comment on the complexity of the approach.

The main technique used to reduce the complexity is decoupling [22]. While *decoupling* does not reduce the amount of hardware that has to be designed, it does limit the interaction between modules and keeps their individual sizes within bounds. The idea is to maintain only very narrow interfaces. Exploiting this property allows designer groups to work in isolation, with less effort to verify cross-module interaction correctness.

The following interfaces must be considered between the four structures:

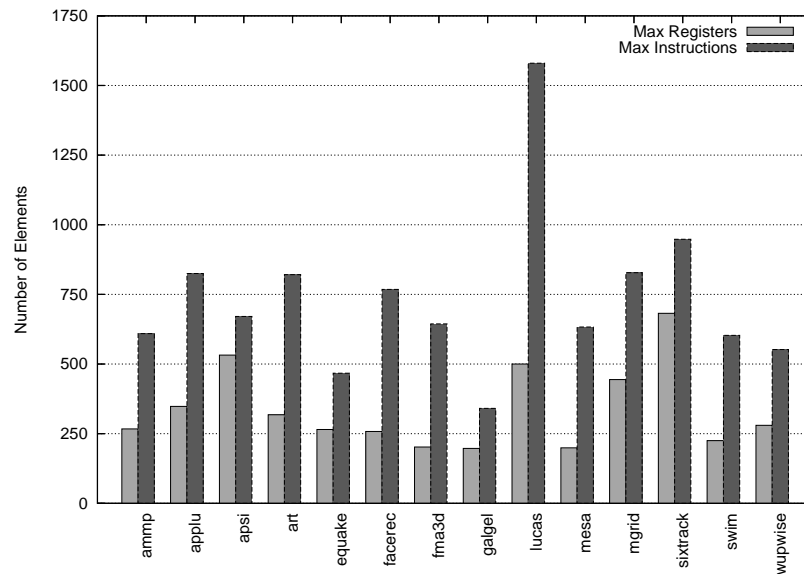


Figure 5.11: Maximum number of registers and instructions in the LLIB for SPECFP

CP→LLIB During the Analyze stage instructions may be sent to the LLIB in a procedure similar to instruction issue. The LLIB must synchronize with the CP and provide entries for the instruction and an associated register. Moreover, when a checkpoint is taken there needs to be a path into the LLIB to inform instructions that write checkpointed registers that they have to writeback into the checkpoint stack.

LLIB→MP When an instruction slice is ready it must be sent to the MP. This is simple considering that the LLIB is a FIFO. In addition, a register may need to be fetched from the LLRF.

LSQ→MP When long latency loads complete, their values are temporarily stored in a buffer whose entries are allocated in-order during Analyze. When the depending instruction arrives at the head of the LLIB it checks if the value is available at the head of the buffer. In that case, the value needs to be written into the Future File of the Memory Processor. Therefore, instead of keeping the destination register in the miss address file (MAF) the D-KIP keeps a reference to the entry in this buffer.

CHPT→CP When the architecture returns to a checkpoint (e.g., branch misprediction) the CP's register file has to be recovered and the LLBV cleared.

$CP \rightarrow CHPT$ Checkpointed registers must be copied from the ARF in the CP to the Checkpoint Stack when a checkpoint is taken.

$MP \rightarrow CHPT$ Checkpointed instructions must write their results into the Checkpointing Stack. Note how $MP \rightarrow CHPT \rightarrow CP$ is the only way back-communication can happen in the D-KIP.

5.6 RELATED WORK

Much research has been conducted to design microarchitectures able to overcome the memory wall. This research has concentrated on introducing techniques for the ROB, register files, instruction queues and load/store queues. The basic difference between these techniques and the proposal introduced here is that the D-KIP approaches the memory wall problem from an integral point of view based on the execution locality concept, while previous efforts have concentrated on overcoming the scalability problem of individual processor structures.

However, the memory problem is not new. The first approach to overcome the memory wall problem was the introduction of memory caches [24, 25]. Caches are small low-latency memory structures that use simple algorithms to capture the bulk of the locality in memory access. Data Prefetching [25, 26, 27] is a technique that tries to improve the efficiency of caches by fetching memory lines into the caches before the memory reference happens. However, in the scenario of increasing memory latencies, this technique is less efficient as the prefetch needs to happen more and more cycles before the memory access to be effective, and determining data access patterns early is difficult.

Several modern techniques try to improve the accuracy of prefetching by actually pre-executing the program but not committing the results. *Assisted threads* [28, 29, 30] rely on pre-executing future parts of the program, selected at compile time or generated dynamically at run time. *Runahead Execution* [31, 32] checkpoints the processor state and preceeds execution while an L2 cache miss is blocking the ROB.

Processor behavior in the event of L2 Cache misses has been studied in detail in [18]. Karkhanis *et al.* showed that many independent instructions can be fetched and executed in the shadow of a cache miss. This observation has fueled the development of microarchitectures to support thousands of in-flight instructions.

Many suggestions have been proposed for overcoming the ROB size and management problem. Cristal *et al.* proposed virtualizing the ROB by using a small sequential ROB combined with multicheckpointing [1, 2, 4]. Akkary *et al.* have also introduced a checkpointing approach [33] which consists in taking checkpoints on low-confidence branches. Cherry [34] uses a single checkpoint outside the ROB to divide the ROB into two regions: a speculative region and a non-speculative region.

Cherry is then able to early release physical registers and LSQ entries for instructions in the non-speculative ROB.

Instruction queues have also received much attention. The *Waiting Instruction Buffer* (WIB) [35] is a structure that holds all the instructions dependent on a cache miss until the data returns. The *Slow Lane Instruction Queue* (SLIQ) [2] is similar in concept to the WIB but is designed as an integral component of an overall KILO-instruction microarchitecture. Recently, Akkary *et al.* have proposed the *Continual Flow Pipelines* (CFP) architecture [36] which proposes an efficient implementation of a two-level instruction queue. It contains a Slice Data Buffer (SDB) which is similar in concept to the SLIQ. As with the SLIQ, the SDB is tightly integrated in a complete microarchitecture designed to overcome the memory wall.

Register Management has also been studied extensively. Several techniques have been developed in the context of out-of-order processors with centralized register storage. *Virtual Registers* [37] is a technique to delay the allocation of physical registers until the issue stage. On the other hand, *Early Release* [38] tries to release registers early by keeping track of the number of consumers. *Ephemeral Registers* [3] is an aggressive technique which consists in combining both approaches. The CFP architecture [36] stores long-lived registers along with the instructions in the Slice Data Buffer. Thus, each entry in the SDB is increased with the space to hold a register value. If the instruction has no READY registers, then the space is wasted. The D-KIP stores long-lived registers through an additional level of indirection and is able to save register storage by virtualizing the register space.

Finally, several techniques have been proposed to attack the scalability problem of the load/store queues. We delay the discussion of these approaches until chapter 8, where our LSQ approach is discussed.

Decoupling is an important technique that allows simplification of the design of loosely coupled microarchitectures while increasing its performance. Each partition results in a simpler core that is easier to verify. Overall, *decoupling* reduces complexity and opens the doors to new design approaches and research. Smith proposed decoupling the memory system from the processor in the Decoupled Access-Execute computer architecture [22]. In this approach two cores are implemented: The address processor handles all load/store operations and the arithmetic processor handles all the computation. The two cores are connected via a set of queues. This approach has also been used in array processors [39] and, more recently, in vector processors [40], where three cores are implemented: a vector processor, a scalar processor and an address processor. All three cores are connected via a set of queues. These decoupled architectures separate instructions based on their type. The approach presented here, the decoupled KILO-instruction processor, distributes the instructions depending on their issue latencies. The two cores are also connected using a queue (the LLIB).

5.7 CONCLUSIONS

In this chapter we have proposed to use *decoupling* as a way to design processors based on Execution Locality and presented an implementation based on this idea.

We showed that high locality instructions are best processed by an out-of-order *Cache Processor* while low locality instructions can be efficiently processed by a simple in-order *Memory Processor*. Our basic implementation of the architecture featuring out-of-order queues in the Cache Processor with 40 entries and an in-order Memory Processor obtains a speed-up for SPECFP of 40% compared to a large out-of-order processor with 256 entries in the issue queues and an 88% speed-up when compared to a smaller, Cache Processor-like, out-of-order processor. For SPECINT we found that the gains are limited by the irregular branch behavior and by the presence of load chains. In addition, we found that some out-of-order instruction insertion from the LLIB into the Memory Processor may provide speed-ups for some codes.

The nature of the decoupled design offers the promise for reduced design complexity. Both the Cache Processor and the Memory Processor are based on well known designs such as the R10000 [16] or the Future File [20]. In addition the Low Locality Instruction Buffer uses a FIFO architecture with an extremely simple register management algorithm.

If we compare the obtained results in Figure 5.9 with Figure 4.2 we observe that the performance of the proposed D-KIP implementation is still far from the performance of the ideal out-of-order core with a 4K window. This suggests that providing dynamic scheduling capabilities over the whole window can still provide important improvements. In the next chapter we present a new microarchitecture that exploits this feature while further simplifying additional interfaces of the microarchitecture.

THE FLEXIBLE HETEROGENEOUS MULTICORE PROCESSOR

Decoupling processor design based on the locality of the instruction stream has been shown to be an effective technique to design high performance processors able to overcome the memory wall problem. However, the initial design that has been introduced in the previous chapter is still suboptimal. It can be improved both in terms of performance and complexity.

In this chapter we will introduce a new processor design based both on decoupling and on a linear partitioning of the instruction window. This scheme will considerably simplify many interfaces. In addition it allows out-of-order execution of multiple instruction clusters which considerably improves performance for some applications. A fundamental property of this scheme is that the instruction window size of the architecture can be modified by adding or removing so-called *engines*. Based on this we developed a multithreading proposal introduced in the next chapter. This flexibility in modifying the instruction window size, along with its adaption capability, gives the proposal its name: the *Flexible Heterogeneous MultiCore Processor* or FMC, in short. The FMC was introduced in our PACT-2007 contribution [41] and has been kept as the base microarchitecture for later proposals.

6.1 MOTIVATION

Recent years have seen a new trend in the design of high performance microprocessors. Rather than continuing to improve performance through exploiting instruction-level parallelism (ILP), processors have begun to improve performance through thread-level parallelism (TLP). The shift in focus is driven by three factors limiting ILP-alone designs: the wide disparity between processor speeds and memory speeds (i.e. the aforementioned *memory wall* [15]), increasing power budgets and the design complexity of large monolithic designs. By contrast, *multi-core processors* take advantage of increasing transistor budget and can achieve high performance by running multiple threads simultaneously. For thread-parallel applications, the advantages of multi-core are obvious. However, by focusing on TLP, multi-core processors sacrifice performance for applications with a large sequential component. Despite the best efforts of the programming languages community, exploiting large

numbers of threads for high performance is still a complex resource that most programmers do not know how to handle correctly.

In this chapter and the next we will propose a microarchitecture capable of running a single or multiple threads with high performance and fairness. The architecture is based on a processor microarchitecture that allows the instruction window size to be changed at runtime. This is achieved by distributing the work among multiple small cores that can as well be reallocated to different threads. The microarchitecture exploits ILP by having an effective instruction window of thousands of instructions spread across the processing elements, largely overcoming the negative effects of long-latency memory operations. The microarchitecture also exploits TLP for parallel workloads by allowing multiple threads to automatically allocate the processing elements it needs to achieve the best performance, rather than giving each thread the same kind of core regardless of its needs. As an additional benefit, all of these advantages are obtained without changes to the ISA, compiler or operating system.

The variable window processor is based on the technique of decoupling, presented in the previous chapter. The instruction window of the previously introduced D-KIP can be large, but the issue windows are small since the cache and memory processors handle a relatively small amount of instructions at a time. However, this design has several shortcomings. For instance, the intermediate buffer, being in-order, serializes all miss-dependent instructions regardless of the memory writeback times. As will be seen, the penalty due to this serialization is not negligible, resulting in about a 20% performance loss for numerical codes. In addition, this design features only two execution modes, *small window* or *full window*, instead of offering a scalable performance range. This makes it undesirable as a building block for a flexible chip multiprocessor.

In this chapter, we add to the decoupled nature of the D-KIP to overcome its limitations and allow it to scale to many cores. The result is a processor with a variable window size using simple interfaces. This variable-size window processor uses multiple small cores, called *memory engines*, linked by a network, to compute memory dependent instructions. The network introduces latency, but this latency is well absorbed as low locality instructions are already waiting hundreds of cycles due to cache misses. The memory engine network can then be shared among threads to build a reconfigurable heterogeneous multi-core architecture. Although the memory engine sharing is a fundamental part of the flexible multicore (FMC) processor we delay its discussion to the next chapter, where we discuss the multithreading variant of the FMC architecture.

6.2 THE FLEXIBLE HETEROGENEOUS MULTICORE PROCESSOR

6.2.1 *Some problems with the initial D-KIP design*

As was pointed out in the previous chapter, the D-KIP does not handle well the case when the LLIB contains a code mix of different localities, i.e. if there are uncached loads in the LLIB driving non-local clusters. The LLIB, being in-order, prevents dataflow execution of these instruction clusters, limiting performance. In addition, the suppression of the ROB and implementation of a centralized checkpointing stack with distributed registers creates some complex conditions for operation.

In the D-KIP there is a checkpoint stack associated to the Memory Processor from which the processor can restart in case of an exception. Values generated in the Memory Processor should always be written into the last active checkpoint. Complexities arise because the distributed register scheme (LLIB and Memory Processor) is complex to handle with a centralized checkpoint stack and due to the presence of two different LLIBs (one integer and one FP). Since these two pipelines can advance at different speeds it can happen that at some point writes belonging to different checkpoints occur. To solve this, instructions need to be given a checkpoint ID which indicates the target checkpoint. In addition, global paths need to exist that allow writing to different checkpoints. Alternatively, a single LLIB may be implemented. This will lose some performance but would allow to implement a lower complexity D-KIP.

Further, when a checkpoint is taken during *Analyze*, the instruction that writes the value may still be in the LLIB. This means that all the checkpoints in the checkpointing stack (starting from the instruction) will reference this register. Thus, these values need to be copied from one checkpoint to another once they have been computed.

6.2.2 *Approximating Dataflow in the Memory Processor*

The problems of in-order execution in the Memory Processor can be solved by introducing a relaxed form of out-of-order execution that we call *multi-scan execution*, explained next. Full out-of-order capabilities are not necessary since the Memory Processor is very latency tolerant. Small scheduling delays are negligible compared to the latencies resulting from cache misses.

To implement *multi-scan execution* we profit from a special property of the Memory Processor that enables a different execution paradigm. In contrast to conventional processors, the memory processor does not perform fetch, decode nor branch prediction. Instead, instructions are provided externally by the Cache Processor and inserted into a buffer in the Memory Processor. When the long-latency load that triggered the MP finally obtains the value from memory, the MP may contain hun-

dreds of pre-decoded instructions awaiting execution. At this point, the instructions in the low-locality instruction buffer are long latency instructions that have not yet executed. The ordering of these instructions is program ordering. All higher locality instructions must necessarily have already executed. If we also insert executed loads and stores into the LLIB, the result is a compressed program image that does not include the higher locality instructions. Loads and stores cannot be precommitted due to memory consistency. The problem is then how to efficiently execute these instructions.

The method that we devise for efficient coarse-grain dataflow execution is based on the locality of the miss-dependent instructions. The idea is to take the LLIB and perform multiple runs through the buffer, attempting to execute those instructions no longer waiting for cache misses to complete. After a series of passes all instructions will have been executed. At this point, the whole group of instructions can be committed. This includes sending all stores to the data cache.

The new Memory Processor is a considerable departure from the D-KIP. However, the concepts on which it is built are more straightforward. In addition to the instruction buffer, which we now call *Completion Buffer*, the memory processor includes integer and floating point units, load and store buffers, source register buffers, and a pair of register files associated to the head and tail of the Completion Buffer. The input register file is used to keep the precise state corresponding to the first instruction of the Memory Processor. Furthermore, it keeps a small local register file to keep the partial register state necessary to process the instructions. As instructions are processed and the destination registers are computed, the source register buffers are updated in a procedure that is similar to reservation stations. Contrary to the D-KIP, this scheme does not require the existence of a checkpointing stack. A checkpoint is implicitly kept at the head of the Completion Buffer. This new processor will be called a *memory engine*. The basic structure with a sample filling of instructions and registers is shown in Figure 6.1. Memory Engines are composable, which allows multiple of these engines to be combined to create a large instruction window. This is explained next in the next section. *Multi-scan execution* vaguely resembles *Flea-Flicker Multipass Pipelining* [42]. However, while *flea-flicker* is a technique specifically targeted at overcoming L1 cache misses in an in-order processor, *multi-scan execution* is a way to achieve coarse grain dataflow execution among a set of miss-dependent instructions.

The output register file will initially be empty. After each scan new output registers will have been computed. The memory processor then proceeds to update these registers in the output RF. The input RF has always all registers, since it represents the register view when the first long-latency load is detected and the memory processor is initiated. At this point instructions start to be inserted into the instruction buffer. Until then the commit register file of the Cache Processor had a precise view of the register set. This precise view is then copied into the Memory Processor before the instruction window is allowed to start growing.

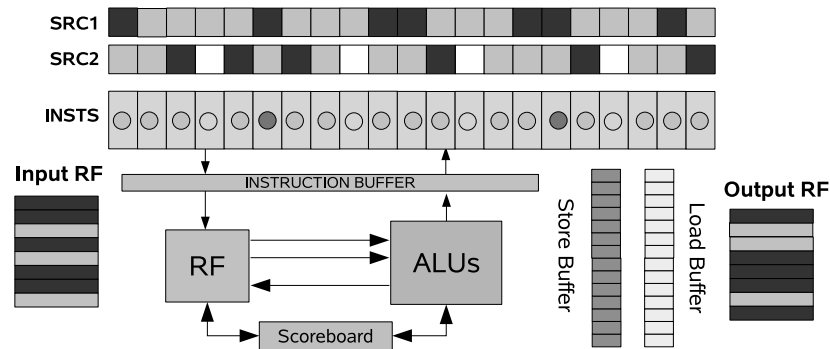


Figure 6.1: Architecture of a single Memory Engine

The main problem with this scheme is that completing a scan requires the memory processor to be filled completely. The memory processor cannot start processing the low locality instruction stream earlier because newer high-locality instructions may still be inserted. On the other hand, to allow effective lookahead this scheme will require a very large completion buffer. But, as memory access latencies are not uniform, the larger this buffer, the worse the dataflow approximation. An effective scheme should be able to handle low locality instructions earlier.

6.2.3 A resizable window based on sequential partitioning

The aforementioned problem can be solved by partitioning the Completion Buffer into multiple smaller buffers sequentially and providing each one with its own scheduler. Functional units can be local to the buffer or shared, with some additional complexity to the scheduler. The buffers are then allocated round-robin to the Cache Processor whenever new buffers are needed. In this scheme, instead of having a Memory Processor with a single memory engine, we have a Memory Processor consisting of multiple memory engines, as shown in Figure 6.2. Each memory engine handles only a subset of the compressed instruction window. Registers are passed between the engines whenever new registers are written into the output register file. Since the engines are sequentially allocated, the output register file corresponds to the input register file of the following engine. This means that there is no real need to keep an output register file. Only a way to track the set of live-outs is needed. This can be done by applying register renaming to the incoming instructions and checking who the last writer is. During each *scan* the engine checks for newly generated output registers (*live-outs*). These registers are then sent over a network to the input register file of the next logical memory engine. The input register files also provide state for the processor to perform precise recovery. If an exception occurs, e.g. a branch misprediction, only engines handling younger instructions are

squashed and the contents of the input RF of the engine where the exception was triggered are copied to the Cache Processor register file so that execution can resume. Uncomputed registers are marked as long-latency so that depending instructions will still travel to the Memory Processor. A diagram comparing the instruction windows of several architectures can be seen in Figure 6.3.

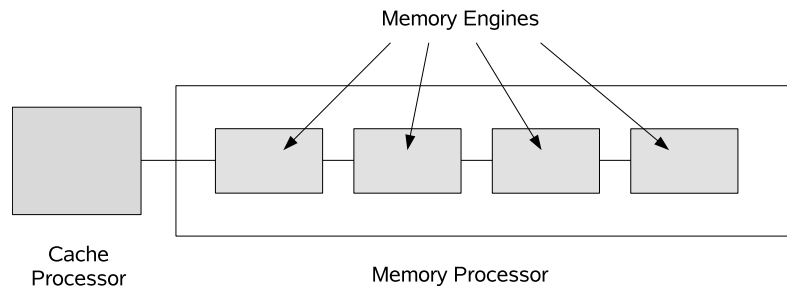


Figure 6.2: Overall architecture showing Cache Processor, Memory Processor and Memory Engines

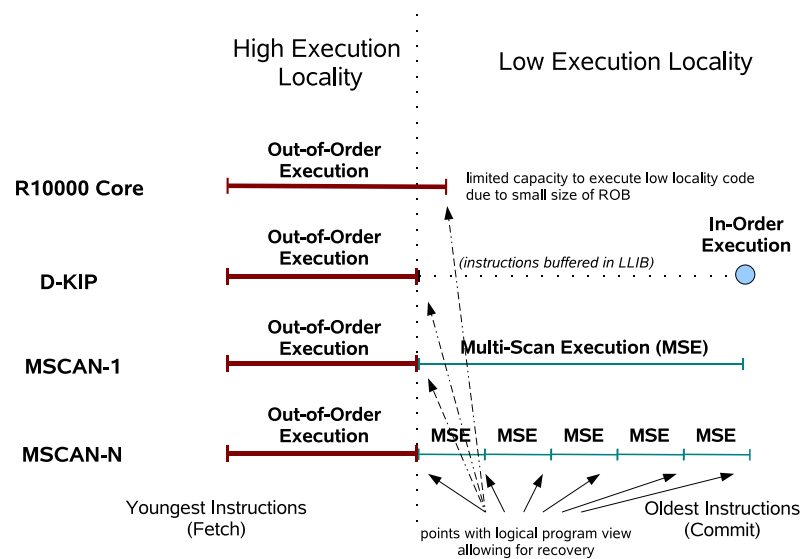


Figure 6.3: A comparison of Instruction Windows of the R10000, the D-KIP, and a single- and multiple-ME design with multi-scan execution

There are many parameters to tune in this microarchitecture. After a design space exploration we have settled on a memory engine design with in-order instruction queues scanning two instructions per cycle. Each memory engine can handle up to 128 execution-pending instructions and up to 128 loads and stores. Reducing the issue width to one reduced IPC in 2.3% while implementing out-of-order instruction

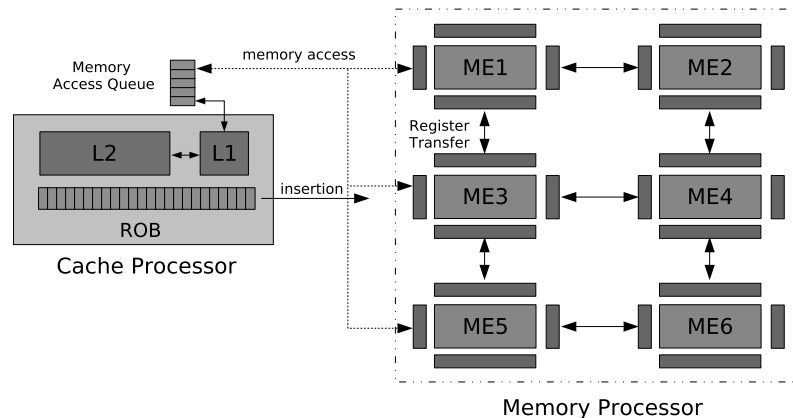


Figure 6.4: Generalized Processor with ME Network

queues increased performance in about 0.1%. In-order performs well since each scan only processes instruction clusters which additionally provide good latency tolerance.

The interconnection of the memory engines is another critical issue. Memory engines require a path to all other engines. Communication itself happens only between conceptually adjacent engines. However, the *previous* and *next* engine can be located anywhere on the chip. To provide this all-to-all communication some sort of network needs to be implemented. We will analyze the trade-offs of such a network later. A diagram of the complete architecture, showing the MEs connected with a mesh network, can be seen in Figure 6.4. The figure also highlights the different paths that are necessary for communication: memory access, register transfer and instruction insertion.

Memory Management

Memory management is a critical component of high-performance architectures. In this chapter we consider the LSQ to be centralized accessed in the Cache Processor. This scheme is shown in Figure 6.4. This makes the FMC scheme compatible with traditional memory consistency models for multiprocessors. We will be considering a centralized monolithic LSQ to evaluate the FMC, although such a LSQ is not a good candidate to be implemented in a real processor due to power and latency issues. Chapter 8 focuses on building an efficient LSQ for the FMC processor.

The architecture presented so far will be the basic building block for our goal of building a chip multiprocessor that can dynamically reconfigure itself to support various degrees of heterogeneity. This is explained in chapter 7. Before proceeding

with the description of the CMP architecture we evaluate the proposed architecture in single-threaded mode.

6.3 EVALUATION OF THE FMC PROCESSOR

The microarchitecture of the FMC has been evaluated using the same execution-driven infrastructure developed for the previous chapters. The cycle-time accurate back-end has been extended to model an array of memory engines using multiscan execution. In addition the Cache Processor's ROB has been slightly modified. It does not use the Aging ROB scheme any longer and instead checks only the locality of source registers. This scheme is simpler than the Aging ROB and achieves the same performance while removing the hardware used for the timers. Again we reused the SPEC CPU 2000 benchmark suite with selected simulation points of 100 million instructions.

In this evaluation we focus on determining the equivalent window size of the FMC processor and compare its performance with other proposals. We will also analyze the important topic of memory bandwidth which may become one of the major bottlenecks for future CMP processors.

The following lists the evaluated microarchitectures:

000-64: A 4-way R10k-like processor with out-of-order scheduling logic and a 64-entry ROB. The integer and FP instruction queues have 40 entries. Other resources are idealized. **000-64-PREF** is the same configuration extended with an aggressive stream prefetcher that can hold up to 256 streams of 256 bytes, totalling 64KB of prefetched data.

000-256: Like the previous, but with a 256-entry ROB. The instruction queues can hold up to 160 instructions each. This model is a lot more aggressive than current microarchitectures. **000-256-PREF** is the same configuration but including the aforementioned stream prefetcher.

RA-64, RA-256: Two runahead processors [32] with 64/256-entry ROB. **RA-64-PREF**, **RA-256-PREF** are the same models but including the stream prefetcher. These configurations include an unrestricted fully-associative runahead cache which allows them to take full advantage of data forwarding during runahead periods.

DKIP: This is the D-KIP model as presented in [14]. The size of each LLIB is 2048 entries. **DKIP-PREF** adds the prefetcher to the D-KIP model.

FMC: This is the proposed FMC microarchitecture. It includes 16 memory engines. All transfers between CP and MP suffer an additional delay of 4 cycles representing network latency. Among memory engines, every single hop is accounted as an additional cycle. The Cache Processor is equivalent to **000-64** in terms of structure sizes. **FMC-PREF** includes the stream prefetcher.

The Load/Store Queue has been idealized for all models. The memory model is modeled after an idealized pipelined memory that is capable of transferring 8 bytes every 4 processor cycles. Table 6.1 lists parameters that are equal for all configurations. Note that the FMC architecture is built completely out of medium-sized structures.

Fetch/Decode Bandwidth	4
Branch Predictor	Perceptron [23]
Store/Load ports	2 shared ports
L1 Cache Size, Associativity & Access Latency	32 KB / direct mapped / 1 cycle
L2 Cache Size, Associativity & Access Latency	2 MB / 4-way set assoc / 10 cycles
Memory Latency	400 cycles
Cache Processor & OoO: IQ/FPQ/ROB/RegFile	40/40/64/96
Cache Processor & OoO: Scheduler	Out-of-Order
Memory Processor: IQ/FPQ/RegFile	20/20/32
Memory Processor: Scheduler	In-Order

Table 6.1: Common Parameters for all Microarchitectures

Figures 6.5 and 6.6 show the IPC for selected microarchitectures side-by-side. The FMC performance gets close to the limit shown in Figure 4.2. Using 16 memory engines the IPC for SPEC FP reaches 2.97 using no prefetcher. There is a considerable speed-up of 12% compared to the D-KIP configuration, which reaches 2.66, and a 31% speed-up compared to RA-256. When prefetchers are in use, the speed-ups are 5% compared to DKIP-PREF and 18% compared to RA-256-PREF. The performance difference between runahead and FMC is due to the need of refetching instructions in runahead every time a runahead period ends (i.e., whenever off-chip memory accesses are in-flight). Because FMC does not need to refetch instructions it can look further into the future and achieve even farther memory lookahead.

On FP codes, the FMC architecture achieves speed-ups of 53% and 90% compared to the 0o0-256 and 0o0-64 (not shown) microarchitectures, respectively. These microarchitectures are severely limited by the sizes of their ROB's and cannot overcome memory stalls.

The speed-up achieved with integer codes is not as large: up to 13% for 0o0-64-PREF. FMC sees a speed-up of 9% compared to the more aggressive 0o0-256 model. There are no notable differences with runahead and the D-KIP models. All these techniques hit a wall due to the frequent recoveries caused by branch mispredictions and the lack of memory level parallelism in many integer applications. In addition, the large second level cache is large enough to capture the locality of most SPEC INT benchmarks. Thus, trying to overcome the memory wall with special techniques is unlikely to give major speed-ups unless the other limiters are attacked first.

In terms of branch misprediction, it is interesting to observe how the high-locality/low-locality decoupling applies to control code. One of the reasons the D-KIP is interesting is because control code does in general not depend on long-latency events. We have evaluated this property in the evaluated FMC model. We

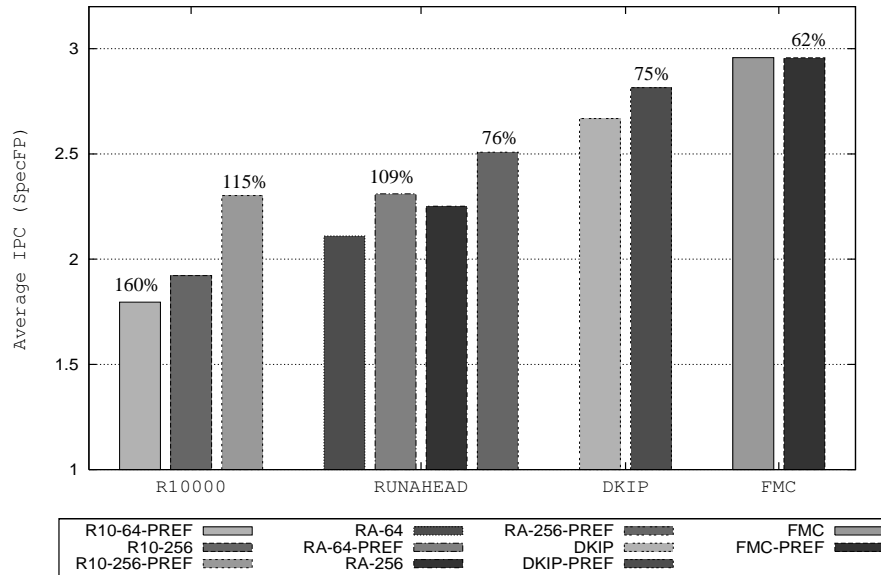


Figure 6.5: SPEC FP IPCs for selected MicroArchitectures

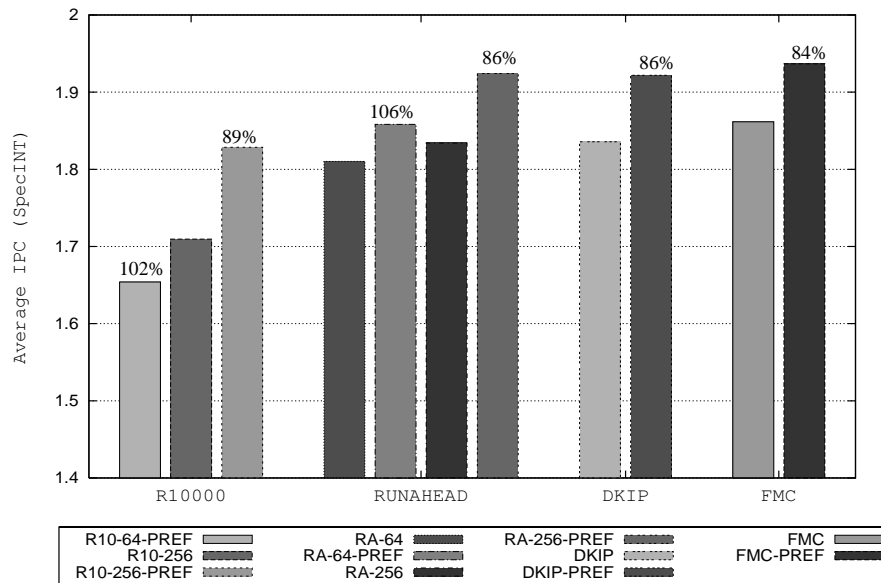


Figure 6.6: SPEC INT IPCs for selected MicroArchitectures

found that in SPEC FP, FMC generates on average 1 long-latency misprediction every 86000 instructions. For the case of SPEC INT, the average number of long-latency mispredictions is one every 3740 instructions. This confirms that long-latency mispredictions are quite infrequent. Particularly for SPEC FP, the number of long latency mispredictions can hardly have an effect on performance. The decoupled design exploit this feature to provide high performance. Even for SPEC INT the number is not very large. However, considering that FMC can hold thousands of instructions in-flight, generating a complete window squash every 3700 instructions could still have a small, though noticeable performance penalty. Future work will attempt to reduce this number by applying techniques such as control path independence.

Figures 6.5 and 6.6 also shows the percent increase in the off-chip memory traffic that the prefetcher is generating. Note that, although the prefetching approaches provide good speed-ups, they do this at the cost of considerable traffic increase. In the case of FMC, the addition of the prefetcher does not improve performance for FP codes. The reason is that the window size achieved by FMC is large enough to reorder memory accesses and make and make the processor insensitive to the memory wall when enough MLP is available. The fact that FMC does not require a prefetcher at all has important benefits. In addition to the reduction of memory traffic the FMC benefits from less area, complexity and power.

Allocation and Efficiency of the Memory Engines

The evaluation so far has been conducted using a FMC processor with 16 memory engines allowing us to emulate a core with a window of around 1500 instructions (see figure 4.2). This is enough for a 400-cycle latency in most benchmarks. To analyze the effective requirements we have evaluated the average performance of the FMC processor using different numbers of memory engines, ranging from 0 to 16. The resulting IPC curve for both SPEC INT and SPEC FP can be seen in Figure 6.7.

The figure shows the progression of IPC starting from zero memory engines, which is equivalent to the OoO-64 processor, up to 16 memory engines. The IPC value at this point differs in less than 1% from the value achieved with 30 MEs. Using 8 memory engines is still enough to achieve 95.7% of the maximum IPC for SPEC FP. The curve for SPEC INT saturates earlier, reaching 96.8% of the final IPC with only 4 memory engines. The architecture of the FMC is therefore well suited for power-performance trade-offs. If we want to reduce power consumption, we can deactivate memory engines and make the window smaller.

To characterize the behavior of the applications we measure two parameters:

- the average allocation of memory engines when run with 16 memory engines; and
- the minimum number of memory engines necessary to reach 95% of the performance of a FMC with 16 memory engines.

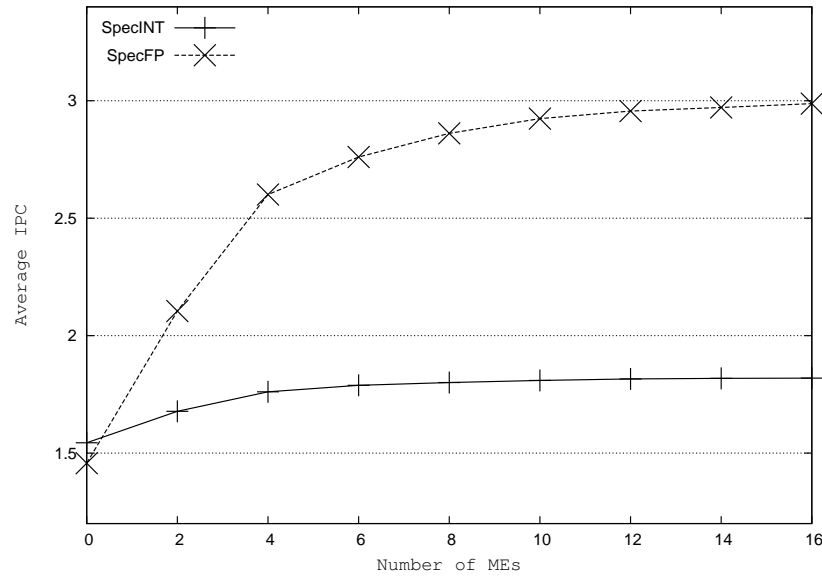


Figure 6.7: SPEC CPU 2000 IPC for varying number of Memory Engines. The case of zero engines is equivalent to OoO-64

Benchmark	Average Allocation	MEs for 95% IPC	Benchmark	Average Allocation	MEs for 95% IPC
bzip2	6.19	0	ammp	4.72	8
crafty	0.28	2	applu	9.83	6
eon	0.02	0	apsi	6.51	8
gap	2.08	4	art	8.77	6
gcc	4.32	2	equake	9.32	12
gzip	0.61	0	facerec	7.51	4
mcf	5.41	8	fma3d	14.16	10
parser	8.78	2	galgel	0.72	2
perlbnk	4.79	2	lucas	7.65	8
twolf	0.30	0	mesa	0.91	2
vortex	6.24	8	mgrid	3.47	4
vpr	3.0	4	sixtrack	1.94	6
			swim	3.29	4
			wupwise	3.13	6

Table 6.2: Behavior of SPEC INT (left) and SPEC FP (right) applications

The results for SPEC INT and SPEC FP are listed in Table 6.2. These numbers allow establishing a classification of applications depending on the speed-up they experience when they can use memory engines and the average number of engines that they allocate:

HIGH AVERAGE ALLOCATION, HIGH SPEED-UP (TYPE A): This type includes applications that experiment large speed-ups when additional MEs are given to them. The additional MEs allow these applications to extract more MLP and execute more distant parallelism. The benchmarks in this category are: *ammp*, *applu*, *apsi*, *art*, *equake*, *fma3d*, *lucas*, *mcf*, *sixtrack*, *vortex* and *wupwise*.

HIGH AVERAGE ALLOCATION, LOW SPEED-UP (TYPE B): This type includes applications that consume many MEs but do not noticeably improve IPC. The reason is that these applications have not enough MLP to exploit and instead perform sequential memory accesses. The benchmarks in this category are: *bzip2*, *facerec*, *gcc*, *parser* and *perlbmk*.

LOW AVERAGE ALLOCATION (TYPE C): This includes the remaining apps that do not allocate many MEs to reach their maximum speed-ups. The reason is that the working set of these benchmarks fits nicely within a 2MB L2 cache. The benchmarks in this category are: *crafty*, *eon*, *galgel*, *gap*, *gzip*, *mesa*, *mgrid*, *swim*, *twolf* and *vpr*.

Interaction with Memory Engine Network

Given that memory engines are connected by means of a network one cannot assume that operations requiring the use of the network will be able to complete with a zero-cycle delay. The impact of these delays needs to be evaluated in detail, particularly in the scenario of future highly integrated multi-GHz cores where a single signal may take many cycles to traverse the chip. There are three cases in which data needs to be passed through the ME network: insertion of instructions, register transfers between engines and execution of low locality loads.

Three types of networks have been implemented in the FMC simulator: a butterfly (fixed delay) network, a ring network (where delay is proportional to distance), and a mesh network in which the 16 memory engines are organized as a 4x4 mesh. There are two parameters that define this network: the distance in cycles between the cache processor and the memory processor and the number of memory engines that can be reached in a single cycle, i.e. the number of hops per cycle. Note that for the butterfly network there is only one network latency to consider for all types of transfers. The microarchitecture evaluated so far has assumed 4 cycles of CP→MP access latency and 1 hop per cycle using a mesh network where memory engines are organized as a 4x4 matrix. Our initial evaluations show that performance is not very sensitive to the kind of network that interconnects the memory engines. Instead, the parameter that has the highest influence is the distance between the cache processor and the memory processor. We have therefore opted to simulate a mesh network, which naturally fits in a tiled architecture like the memory engine network. We model CP→MP interconnect so that the four memory engines in the center of the ME mesh have direct access to the CP. This is shown in figure 6.8. When the Cache Processor interacts with a memory engine it will always enter the mesh through the

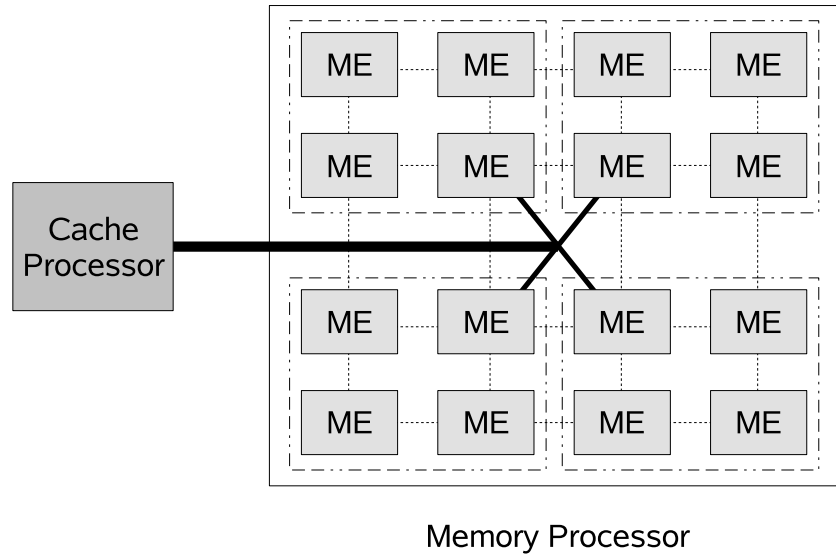


Figure 6.8: Interconnect between CP and MP with mesh network

same engine, as shows by the subsets of 4 memory engines also shown in the figure. Using networking terminology we call this memory engine the *gateway* memory engine.

Figure 6.9 shows the impact of these delays in the performance of the system both for SPEC INT and SPEC FP.

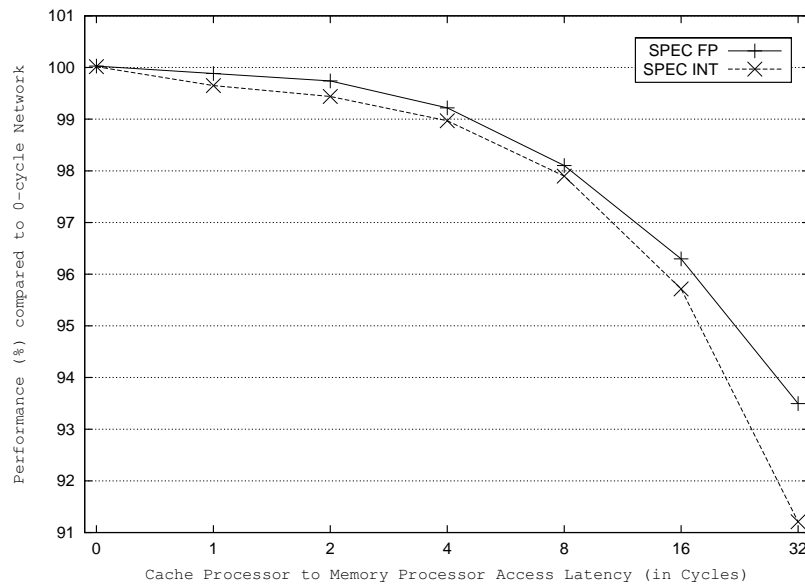


Figure 6.9: Effect of network delays on processor performance

Results show that even with a 4-cycle distance (minimum 8-cycle round trip), performance is still within 1% of the maximum for both SPEC FP and SPEC INT. The effect of the ME network on loads has the major impact on the final performance. Experiments in which only register transfers and instruction insertion delays were simulated resulted in no measurable performance penalty. The 4-cycle extra latency implies that loads issued in the Memory Processor observe a 8-cycle round-trip delay when accessing the cache resulting in total access latencies of at least 9 cycles for L1 and 18 cycles for L2. The internal latency of the ME network can add at most 4 cycles resulting from 4 hops (worst case round-trip). For an 8-cycle CP→MP latency, performance degrades around 2%. It is because low locality code is tolerant to additional latencies that these delays result in relatively small performance overheads.

6.4 RELATED WORK

The body of related work corresponding to this chapter builds on top of that of the previous chapter. This chapter has focused on presenting a large-window architecture capable of adapting its execution resources to the application requirements. To adapt to varying requirements, the architecture varies the size of the instruction window. This provides different degrees of capacity to overcome the memory wall problem.

An alternative way to overcome the memory wall is to modify memory management so to reduce the window requirements. One direction consists in fetching data before the requesting load appears. This is known as prefetching [43, 44, 45]. Both software and hardware prefetching schemes exist. Using prefetchers has good performance, but it comes at the expense of increasing bus traffic due to unnecessary prefetches. Memory bandwidth is a valuable resource and multicores already put a lot of pressure on the memory bus. The additional traffic coming from a prefetcher may be unacceptable for an already saturated bus. Vector processors do not directly attack the memory wall, but are an alternative way to increase performance by organizing data in vectors whose management enables a simple yet high-performance memory system. Vector processors are very good for the type of numerical applications that we are concentrating on, but, unfortunately, the vector logic is difficult to partition and thus this kind of processor is not useful for our goal of building an adaptive processor.

A different way to improve performance is to partition programs into tasks that can be executed in parallel. MultiScalar performs such a partitioning of the program and then executes the tasks speculatively on a set of processing units [46]. The FMC architecture shares some similarities to MultiScalar, but it performs the partitioning dynamically based on execution locality instead of being generated by the compiler. In MultiScalar, speed-ups are obtained by the parallel execution of tasks. Instead,

in FMC, speed-ups are obtained by also exploiting parallelism in codes with bad locality and by looking ahead in the program to execute loads early.

Another work of Sohi's group, the Expandable Split Window (ESW) paradigm [47], proposes a processor based on multiple small processors executing sequential *basic windows* to provide a large instruction window. This is the same principle as that of FMC, but the design philosophy is quite dissimilar. In ESW, the whole window is executed by small independent sequential processors. ESW does not distinguish high locality code from low locality code and thus does not profit from the fact that control path is mostly cache dependent. To overcome control path problems in the large window, the compiler determines the boundaries of the basic windows so that *if-then-else* statements are fully contained within a single basic window. This allows later basic windows to be executed without the need of squashing them in the event of mispredictions. In the case of FMC, we do not dedicate special attention to control path in the memory processor since mispredictions are very rare in the low locality window. Even in integer codes a misprediction in the memory processor is observed only every ~ 4000 instructions. An additional difference between FMC and ESW is that the basic windows executed by ESW are normally much smaller than the size of the memory engines. The goal of FMC's memory engines is to provide coarse grain out-of-order capabilities to overcome memory latencies. Together with multi-scan execution, this allows the size of the memory engine to be quite large.

A more current architecture that shares some similarities with FMC is the TRIPS project. TRIPS [48] is a complete architecture redesign aimed at extracting parallelism using a dataflow-oriented ISA and a distributed/tiled microarchitecture. Both FMC and TRIPS contain mechanisms to transmit registers between tiled structures, can reassign hardware between threads and are capable of supporting a large number of in-flight instructions. Note that other large window approaches [36, 2, 49] cannot reassign hardware between threads. The main differences between TRIPS and FMC are that FMC does not modify the ISA, that it makes use of a decoupled processor design that exploits Execution Locality and that it retains a certain degree of centralized control. The idea behind FMC is to build an architecture that can tolerate a certain degree of latency while the distributed TRIPS design tries to completely avoid long latency communications.

Integrating multiple cores on a die presents some important issues such as how to interconnect multiple cores. The implications of doing this when using shared busses as interconnect have been studied in [50]. The FMC itself uses a packet-routing network to reduce the problems of wire parasitics and to reduce complexity [51, 52]. Recent proposals have proposed on-chip packet routers capable of forwarding packets with a almost single-cycle latency [53, 54].

6.5 CONCLUSIONS

In this chapter we have presented a new decoupled KILO-Instruction architecture with the goal of overcoming several limitations of the previous D-KIP design. We observed that the D-KIP, due to the in-order nature of the LLIB, suffers a considerably performance penalty for benchmarks that keep multiple instruction clusters with different localities in-flight. To solve this, this chapter's proposal (the FMC) partitions the large instruction buffer of the D-KIP into smaller parts, and provides each one with execution resources. Each partition works like an in-order processor, but globally the resulting processor is out-of-order. Second, handling state in the D-KIP is too complex as it requires a centralized checkpointing scheme integrated in a distributed register scheme, with additional particularities like two LLIBs advancing at different speeds but sharing the checkpointing stack. In addition, the integration of the Address Processor in the overall microarchitecture is not a natural choice and increases the complexity of the checkpointing scheme. Instead, the FMC keeps all state locally and handles memory instructions locally, while providing centralized access to the LSQ. As a result, the processor yields higher performance and less complexity compared to the D-KIP. Another feature of the FMC, its capacity to adapt to the workloads, has been mentioned, but not yet exploited. The next chapter shows a technique based on this feature to increment the utilization of the memory engines and the overall processor throughput.

ADAPTIVE HETEROGENEOUS PLATFORM USING FMC

In the previous chapter we have introduced a new microarchitecture that allows to build affordable KILO-Instruction processors with coarse-grain out-of-order scheduling over the full instruction window. Compared to the D-KIP, the key idea of this processor is the partitioning of the Memory Processor into multiple sequential windows, each of them equipped with in-order scheduling logic and functional units. Although the functional units can be shared between multiple engines we have so far modelled a set of functional units for each memory engine. While Moore's law progresses, making it technically possible to implement, the low utilization that the units observe may be undesirable. In this chapter we provide a technique based on the multithreading paradigm to improve the utilization of the FUs. This technique was introduced together with FMC in our PACT-2007 contribution [41].

7.1 MOTIVATION

A fundamental property of the FMC is its ability to change the instruction window size at runtime. It can do so by dynamically adding or removing memory engines from the system. This property allows the processor to adapt to the requirements of the application and activate only those memory engines that are expected to lead to improved performance.

In this chapter we propose to use the memory engines to construct a dynamically adapting multi-core architecture that can provide high throughput to sets of threads with low requirements, mixes of applications with different resource requirements and mixes of identical high-performance programs, particularly in the quite common case that less software threads than available hardware contexts are running.

To this end we extend our proposed flexible multi-core architecture with multiple Cache Processors, each one with a static partition of memory engines, and keep a pool of memory engines that can be dynamically assigned to the different threads. Figure 7.1 shows a general view of this extended microarchitecture.

The microarchitecture has good potential to adapt to application mixes as threads that do not require Memory Engines can relinquish their engines and give them to threads that require more memory engines. Moreover, when there are fewer threads

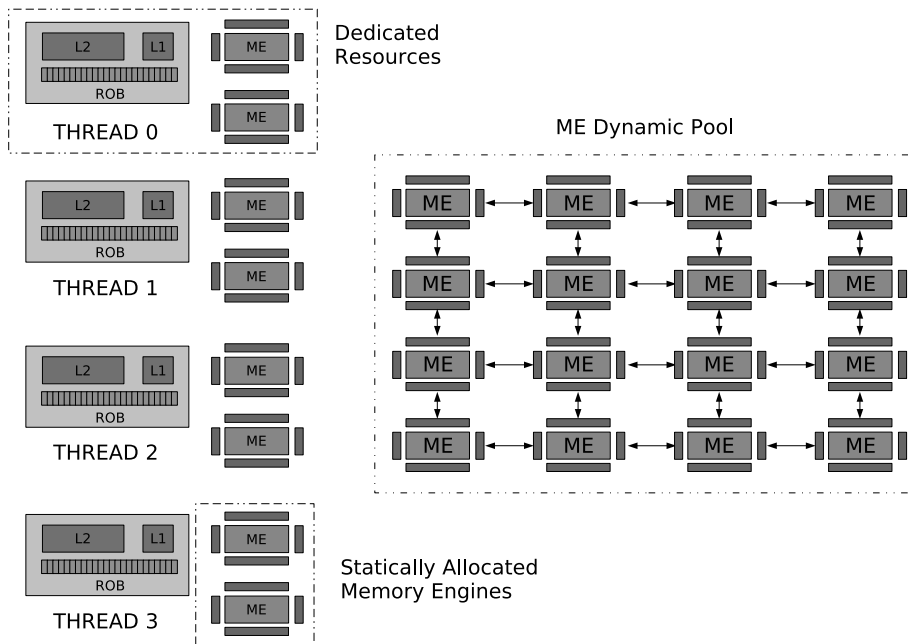


Figure 7.1: The microarchitecture of the flexible multi-core microarchitecture, including a set of Cache Processors, 2 statically assigned ME per thread, and a dynamic pool of memory engines

than Cache Processors, those threads that are running can access the dynamic pool of memory engines with less competition from other threads.

7.2 ASSIGNING MEMORY ENGINES

We have developed a memory engine assignment algorithm with the goals of simplicity and reasonable performance. The algorithm works as follows: Every fixed number of cycles (we chose 256 cycles) a piece of logic, called the *arbiter*, collects information from the Cache Processors regarding the number of dynamic memory engines that a thread has allocated but is not using. The arbiter collects all unused engines and reassigns the engines to all active cache processors, one at a time, using a round-robin policy starting with the thread that currently has the smallest number of MEs allocated. Engine are always allocated to some thread, although the engine might be in a power-saving mode. It is the responsibility of the CP to activate an engine when the application is going to use it. Powering-up an engine cannot be done in a single cycle, but since instruction insertion into engines proceeds more or less at a constant speed it is easy to predict if an engine will need to be used soon. The Cache Processor needs to power-up this unit within time, otherwise the system

may need to stall before new low-locality instructions can be inserted into a memory engine.

7.3 MULTI-CORE SIMULATION INFRASTRUCTURE

The multi-core implementation that we have proposed so far is highly decoupled. There are only three elements that are shared: the dynamic pool of MEs, the system bus and the main memory. Everything else is local to the thread. This includes the two levels of cache, TLBs and functional units. This partitioning has been implemented on some commercial processors such as the Intel Montecito, Intel PentiumD or AMD AthlonX2 processors. The sharing of the memory engines has been modeled by implementing a server process acting as the arbiter. The processor simulators are clients to this process. Every 256 cycles they synchronize with the server and send a packet containing the number of free engines. The server answers by sending a new engine allocation. Sharing of the system bus has been modeled by implementing a virtual memory system that statically partitions the bus bandwidth among the threads. In our model, each thread gets the same bandwidth. This assumption is pessimistic as a memory controller could perform a much better bus cycle assignment between the threads.

Evaluating multi-core/multi-threaded architectures requires the generation of workload mixes for the simulations. We evaluate a multi-core architecture with 4 Cache Processors. To generate the workload mixes we use the application classification provided in section 6.3 of the previous chapter. In constructing the workloads, we order the benchmarks alphabetically and select them using a round robin algorithm. Table 7.1 shows the generated workload mixes for the architecture with 4 Cache Processors. In Table 7.1, 'r' means *repeated*, i.e. the same application is run multiple times concurrently. This is a frequent scenario in scientific/engineering computations (e.g., evaluation of multiple inputs) and also usual in server workloads (e.g., database/web servers attending multiple petitions in parallel).

Class Mix	Benchmark Combinations
{A,A,A,A}	{ammp, applu, apsi, art} {equake, facerec, fma3d, lucas}
{A,A,B,B}	{mcf, vpr, bzip2, gcc} {ammp, applu, parser, perlbnk}
{A,A,C,C}	{apsi, art, crafty, eon} {equake, facerec, galgel, gap} {fma3d, lucas, gzip, swim} {mcf, vpr, mesa, mgrid}
{A,r,r,r}	{ammp, ammp, ammp, ammp} {applu, applu, applu, applu} {apsi, apsi, apsi, apsi} {art, art, art, art}
{A,r,-,-}	{ammp, ammp, -, -} {applu, applu, -, -} {apsi, apsi, -, -} {art, art, -, -}
{A,-,-,-}	{ammp, -, -, -} {applu, -, -, -} {apsi, -, -, -} {art, -, -, -}

Table 7.1: Workload mixes for 4-way Multi-Core

Running multi-threaded simulations has special requirements as we cannot simply run 100 million instructions and stop. Different benchmarks take different amounts of time to execute and stopping in the middle of a simpoint distorts the results. To avoid this situation we use the methodology proposed in [55]. The idea behind this methodology is to re-execute the benchmarks in a workload as many times as

needed until the measurements obtained (IPC in our case) are representative. For our evaluation we used a Maximum Allowable IPC Variance (MAIV) value of 5%. The number of memory engines has been fixed to a total of 20. This number includes statically allocated engines and dynamic engines.

7.3.1 Evaluation of a 4-way Multi-Core Architecture

The goal of this study is to check the effect of dynamically sharing the MEs. To this end, we compare a configuration in which each thread has 5 statically assigned MEs and no dynamic sharing (S5D0) versus a configuration where each thread has 2 statically assigned engines and there is a pool of 12 ME to share (S2D12). The S5D0 configuration models a symmetric CMP. Such a model focuses on throughput with a special emphasis on fairness. We are interested in analyzing if the dynamically reconfiguring S2D12 is capable of exceeding the homogeneous S5D0 both in throughput *and* fairness. Note that we do not evaluate an asymmetric CMP configuration. An analysis of asymmetric architectures can be found in [56].

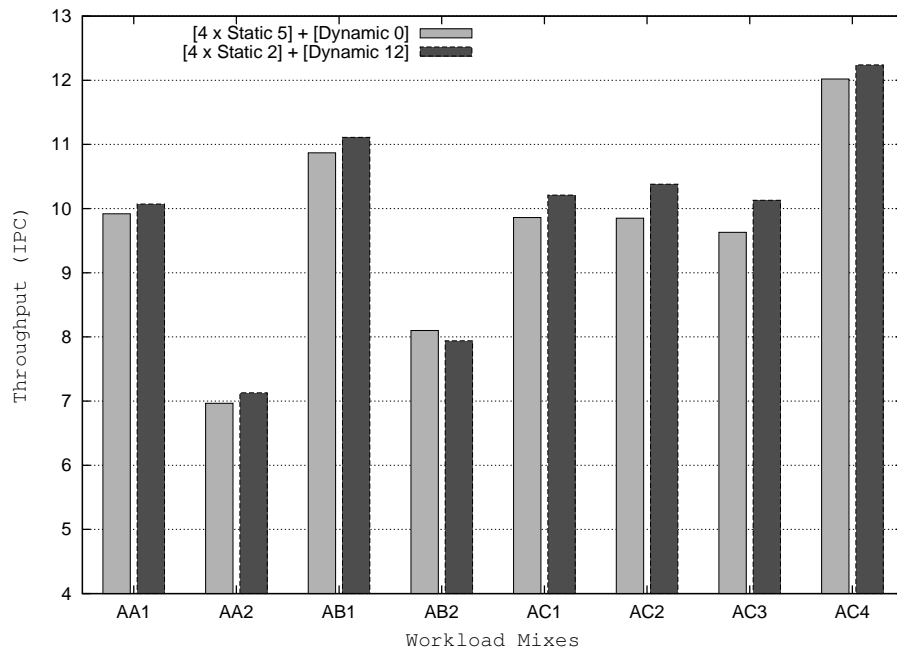


Figure 7.2: Throughput of Mixed Workloads

The throughput results for the 4-way Multi-Core are shown in Figures 7.2 and 7.3. The model of the FMC architecture that was used does not include a prefetcher. The workload identifiers have been abbreviated for spatial reasons. AA workloads refer to $\{A,A,A,A\}$, AB workloads refer to $\{A,A,B,B\}$ and AC workloads refer to $\{A,A,C,C\}$. Finally, workloads where a benchmark is run multiple times in parallel

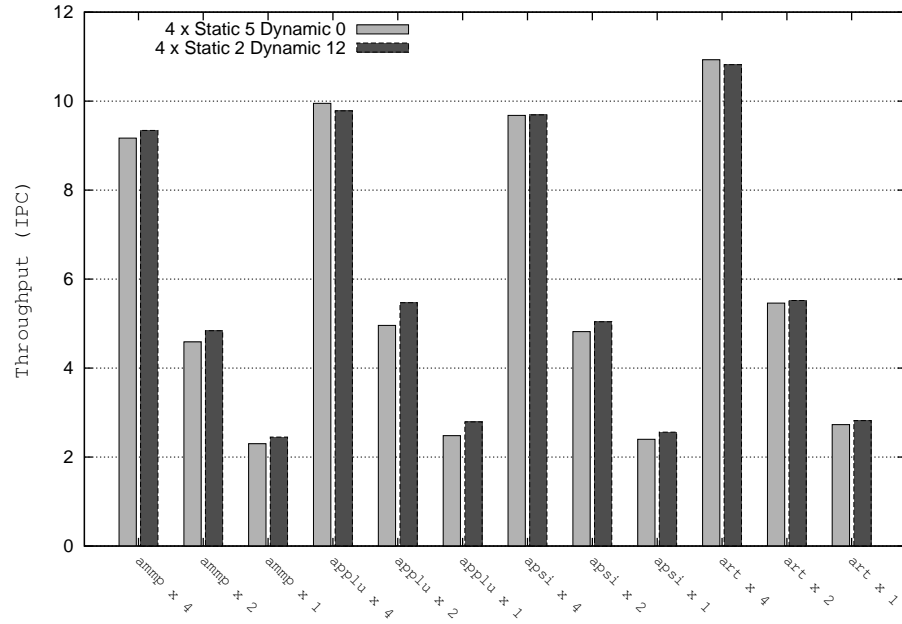


Figure 7.3: Throughput of Repetition Workloads

are identified as $benchmark \times y$. In this case y identifies the number of parallel occurrences of the benchmark. Because each benchmark advances at the same speed, these repetition workloads have been simulated with different fast forwards. The fast forwards differ in 10 million instructions and they average to the same fast forward used in the single thread evaluations in this thesis. While running applications with a difference of 10 million instructions may not allow testing the assignment algorithm for different program phases, it will allow to see how the algorithm behaves for multiple runs that have been started almost simultaneously.

While using more memory engines improves IPC considerably for class-A applications, there is always a point of saturation independent of the benchmark. Many applications will try to go past their saturation point and consume more memory engines without improving IPC. To handle this particular case of unfair behavior we have limited the maximum amount of dynamic memory engines that the arbiter will assign to a single thread to eight engines. This number represents two thirds of the size of the dynamic pool and is applied to all applications, irrespective of their type. Note that this limitation is only enforced when 4 threads are running. For the cases of two threads and one thread it is not meaningful.

The $\{A,A,A,A\}$, $\{A,A,B,B\}$ and $\{A,A,C,C\}$ mixes show promising results when run with a shared pool of memory engines. For this particular configuration $\{A,A,A,A\}$ workloads experienced a 1.9% speed-up in throughput, AB workloads experienced

a 0.4% speed-up and {A,A,C,C} saw a 3.9% improvement when running on the S2D12 configuration. We also measured the harmonic mean of workloads and found that the dynamic assignation algorithm improves its value between 2-4%. We used the *harmonic mean* defined as the mathematical harmonic mean of the relative IPCs compared to the case when the thread is running alone, i.e. when there is no competition from other threads [57]. In short, the technique does not only improve throughput, but it also provides fair execution for all threads in a workload.

When running repeated workloads the situation improves even more, particularly when fewer threads than the number of Cache Processors are running. For example, if only a single copy of *applu* is running, the throughput of this benchmark is 12.5% higher on the the S2D12 configuration compared to the S5Do configuration. This is because under these circumstances the arbiter can assign up to 14 memory engines to a single application without having to compete for resources with other threads. This is far better than what can be obtained using a homogeneous multi-core or even a heterogeneous multi-core, as none of these architectures are able to reassign all hardware resources, a limitation from which the FMC architecture does not suffer. Only the small subset of statically assigned engines is wasted in the FMC. In addition, the FMC architecture performs all these reconfigurations dynamically and can thus adapt to variations in program behavior.

7.4 RELATED WORK

Recently multi-core architectures have become very popular. All major microprocessor vendors have introduced multi-core versions of their high-performance cores, such as the Intel Core 2 [58], the AMD Barcelona [59] or the IBM POWER6 [60].

The popularity of multi-cores raises the problem of how to partition programs among cores. Traditional multicore approaches are homogeneous, i.e., all the cores look the same, while some proposals advocate using heterogeneous cores [61]. Both scenarios have difficulty accommodating many sorts of workloads [56]. Core Fusion [62] addresses this problem by joining multiple 2-wide processors into clustered processors of widths 4, 6 and 8. This proposal differs from ours in two basic ways: First, it widens the processor width instead of the instruction window—limiting its ability to exploit MLP— and second, reconfiguration is triggered by a system call to the operating system whereas FMC performs this transparently to the programmer. A more transparent solution are Composable Lightweight Processors [63] (CLP), an evolution of TRIPS [48]. CLP allows to dynamically aggregate simple, low-power cores to form larger, more powerful single-threaded processors without changing the application binary. To do so, however, it requires a different ISA that explicitly encodes parallelism.

Enhancing multi-cores with thread-level speculation (TLS) [64, 65] is an alternative strategy to improve single-thread performance in the presence of unused cores.

TLS speculatively spawns new threads running future blocks of the program in the hope that the performed work will be useful. As with FMC, TLS has the advantage that it does not require to extend or modify the ISA. While TLS is able to increase performance of single-threaded applications, the speculative nature of the thread spawning mechanism results in very wasteful resource handling.

The technique we have studied here dynamically adapts processor resources to application needs. Contrary to other proposals, it does not require to modify the instruction-set architecture (ISA) nor the application source code or operating system. The technique employs a set of small cores to provide fine-grain heterogeneity. Such a technique could be useful in the scenario of recent manycore architectures such as Larrabee [66], which consists of many in-order 2-wide cores that support traditional multithread programming techniques such as *pthread*s.

7.5 CONCLUSIONS

In this chapter we have evaluated an extension to the FMC processor that provides high performance in multithreaded environments. The processor is extended with multiple Cache Processors and the Memory Engines are collected into a dynamic pool that can then be shared between Cache Processors. Since the dynamic pool covers only the latency tolerant Memory Processor, reassignment latencies are easily hidden and performance is unaffected.

This chapter shows how a simple greedy algorithm is enough to provide good throughput and fairness in the context of a multicore FMC architecture. More sophisticated algorithms may however provide better performance. This is a topic of future research.

EPOCH-BASED LOAD/STORE QUEUE

In the previous chapters we have introduced a family of processors capable of hiding the memory wall by exploiting localities in the instruction stream. This has been achieved mainly by decoupling the abundant high-locality instructions from the less-frequent low-locality instructions which normally stall the processor when they appear. The nice properties in terms of performance and complexity that the FMC processor shows (Chapter 6) suggest that this microarchitecture is an interesting building block for future work. One of the more interesting features of the FMC architecture is that it is built out of small-sized structures, a necessary feature to enable high operating frequencies at low power. However, the Load/Store Queue has so far been modelled as a large centralized structure. Such a structure cannot be implemented in a real processor, at least not at reasonable speed and power. In this chapter we tackle this problem and complete the design goals of this thesis by presenting a new LSQ called the Epoch-based Load/Store Queue (ELSQ). The ELSQ was first introduced and evaluated in our ISCA-35 contribution in 2008 [67].

8.1 MOTIVATION

In this chapter we focus on the most critical component of a kilo-instruction processor and the only one not yet included in our FMC design: the Load/Store Queue (LSQ). Many previous designs have been proposed for the LSQ. However, in architectures that can handle thousands of in-flight instructions, most techniques fail to deliver performance. In Section 8.2 we analyze the reasons for this.

To overcome the bottlenecks we will apply the concept of *Execution Locality* to our LSQ design. Based on Execution Locality we design an LSQ with two-level disambiguation, dividing the non-completed instructions into two parts depending on whether they are miss-dependent or not. Low locality instructions are further partitioned into epochs, which are implemented in different banks. A two-level disambiguation scheme is implemented based on the epochs. On issue, loads and stores first search the local epoch for matches, then the global level. The implementation makes local searches much more power efficient than global searches and profits from store-load locality.

This chapter makes the following contributions:

- A Load/Store Queue based on *Execution Locality*, the ELSQ, is proposed. (Section 8.3.2)
- Exploiting locality, several restricted disambiguation schemes are proposed that can considerably reduce the implementation complexity. (Section 8.3.3)
- The LSQ further classifies low-locality memory instructions into *epochs* based on their age. Epochs are the building blocks for the proposed two-level disambiguation scheme. (Section 8.3.4)
- Several filtering schemes are proposed to reduce activity in global disambiguation. (Section 8.3.4)
- The energy-efficiency is analyzed. (Section 8.6)

8.2 BACKGROUND

8.2.1 Memory Handling for Large Windows

Building a scalable load store queue (LSQ) is challenging. LSQs are more difficult to implement compared with normal instruction queues due to their higher number of states and functionalities.

In a normal instruction queue there are only two states: *Ready* or *Not Ready*. The functionality of the LSQ is more complex. Issuing loads need to search the Store Queue (SQ), while stores need to search the Load Queue (LQ). The overall functionality that needs to be supported in an uniprocessor environment is as follows:

Store-Load Forwarding: When a load issues, in addition to accessing the data cache, it also needs to search the store queue for older stores matching the load address. If there is a match, the load should use the data from the store queue instead of the cache. Store-Load Forwarding involves two special cases. First, the matching store might still be waiting for data. In this case it is common to periodically reissue the load every few cycles until the data is available. Second, the load access might only partially match the store. In this case, special action should be taken to recover the correct value. Some implementations squash the load and do not issue it again until the store has committed its data to the cache [27].

Store-Load Ordering: When a store issues, it is necessary to check whether a younger load with a matching address has already executed, potentially violating program semantics. In general, store-load violations squash the instruction window starting from the violating load. Fortunately, these violations are rare.

Commit: At commit, stores update the memory in program order, maintaining program semantics. All loads and stores need to be buffered during their whole lifetime.

Due to the age-based operation of LSQs it is typical to implement age-indexed LSQs. In this scheme, when a memory instruction is decoded, an entry is allocated at the tail of the LQ or SQ. The size of the LSQ needs to be balanced so that it does not overly constrain the instruction window. We now introduce two relevant solutions that have been proposed for the LSQ.

HIERARCHICAL STORE QUEUES One solution that has been proposed to overcome the problem of the Store Queue is to use hierarchical store queues (HSQ) [33]. In this scheme, the SQ has two parts: A small and fast first-level store queue stores the X youngest stores in the window and a large and slower second-level SQ stores all older stores. This scheme optimizes loads that are ready soon after decode and forward from close stores, but penalizes loads that depend on chasing pointers or forward from distant stores. The hierarchical store queue also suffers higher complexity due to the way it manages checkpoints. Second-level stores are tracked by hashing into a set of counters. Checkpoint recovery consists of decrementing the counters, one by one, for every squashed store. This is costly and takes extra time.

LOAD RE-EXECUTION The largest group of techniques addressing the load queue are those related to load re-execution [68, 69] with the goal of making the LQ non-associative, thus solving the LQ scalability problem. In these schemes, when a store issues, it does not search the LQ for violations. Instead, when the load commits, the load re-executes and checks whether it obtained the correct value. Research has concentrated on reducing the number of loads that need to re-execute. Lipasti et al. [68] propose re-executing only loads that issue while there are older stores with unknown addresses in-flight. Store Vulnerability Windows (SVW) [69] is another way to decrease re-executions. SVW uses a Bloom filter to determine whether a re-execution might be necessary, substituting data access with filter access and possible re-execution.

Large instruction windows can increase the number of necessary re-executions, making this technique less applicable. For example, using SVW with a 10-bit SSBF, a conventional out-of-order processor with a 64-entry window observes an average of 1 re-execution every 715 instructions for SPEC FP. The same execution using a large window processor with about 1500 in-flight instruction results in 1 re-execution every 95 instructions.

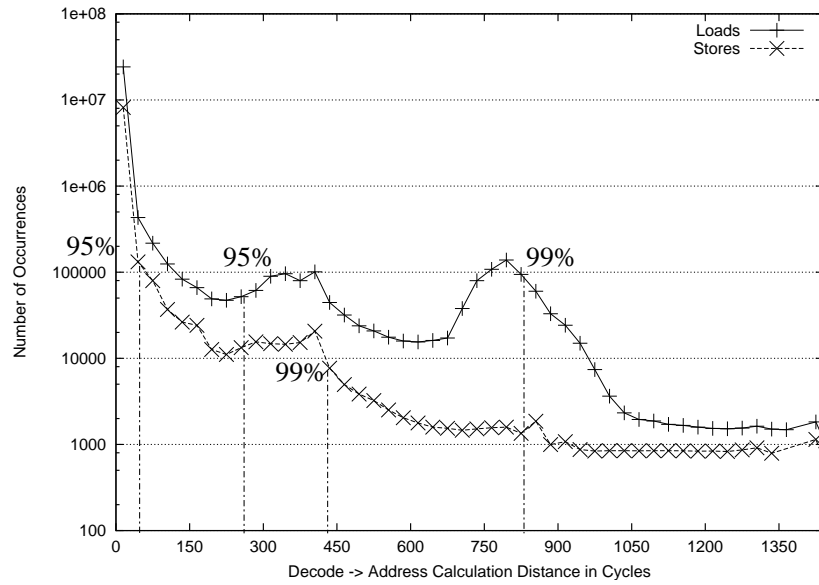


Figure 8.1: Floating Point Decode→Issue Distribution for 100 million instructions

8.2.2 Execution Locality Analysis

Figures 8.1 and 8.2 show how the concept of Execution Locality, explained in Chapter 4, applies to address calculations. These plots classify loads and stores depending on the latency between instruction decode and address calculation. Each data point represents the number of loads or stores that have a similar decode→issue latency measured in cycles. Similarity is grouped in blocks of 30 cycles. The test ran SPEC CPU 2000 on a 4-way out-of-order processor with a large window (up to 4096 in-flight instructions) and a memory subsystem with L1, L2 and main memory with distances of 1, 10 and 400 cycles, respectively. The numbers are averages for 100 million committed instructions over all benchmarks. The plots show the latencies within which 95% and 99% of all loads/stores are covered. For SPEC2000, around 91% of all loads and 93% of all stores calculate their addresses within 30 cycles after decode. The figures show that most address calculations do not depend on cache misses, explaining the prefetching effect achieved by large-window processors. For address calculations that depend on cache misses, loads are more frequent than stores. Few stores have address calculations depending on a cache miss, and almost none depending on multiple cache misses.

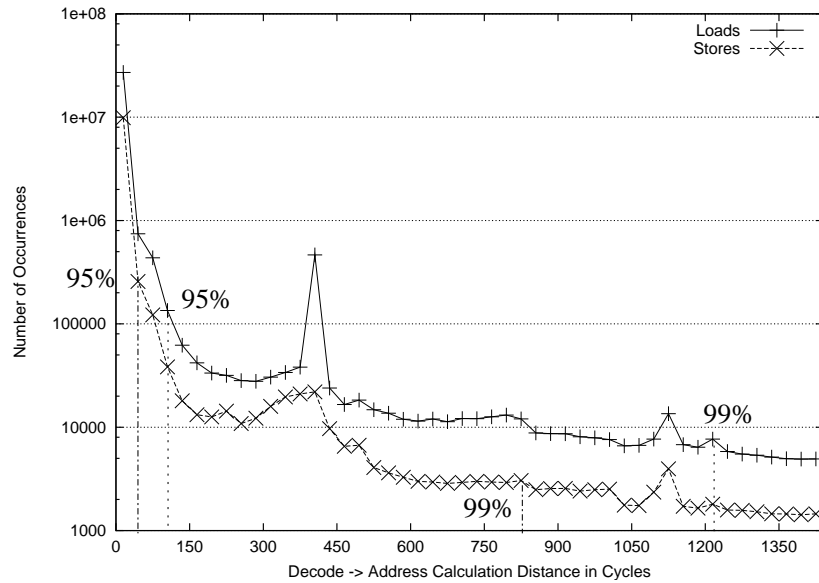


Figure 8.2: Integer Decode→Issue Distribution for 100 million committed instructions

8.3 EPOCH-BASED LOAD STORE QUEUE

8.3.1 *Generic Processor Model*

We first explain the LSQ model in the context of a traditional superscalar with a microarchitecture resembling that of a MIPS R10000 [16]. This processor features a reorder buffer and a centralized physical register file. Logical registers are renamed during decode and checkpoints are taken at branches. In Section 8.4 we will show how ELSQ can be integrated into a microarchitecture based on execution locality.

8.3.2 *Epoch-based Load/Store Queue*

We propose to partition the LSQ based on *Execution Locality*. For high-locality memory references we keep a first high-locality LSQ, while low-locality references occur in the low-locality LSQ. Low-locality address calculations are more latency tolerant. However, store-load forwardings from low-locality stores to high-locality loads are critical.

The partitioning that we propose enables fast access time and reduces power for high-locality memory instructions. In any given cycle, the number of these instructions is relatively small and moderate sized queues are sufficient to track them. Thus, the technique resembles schemes that partition the queue by using address interleaved LSQ banks. However, the conceptual differences imply completely

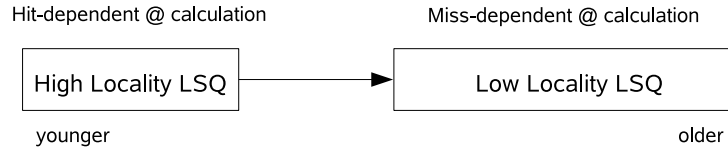


Figure 8.3: Basic Scheme of a two level LSQ based on Execution Locality

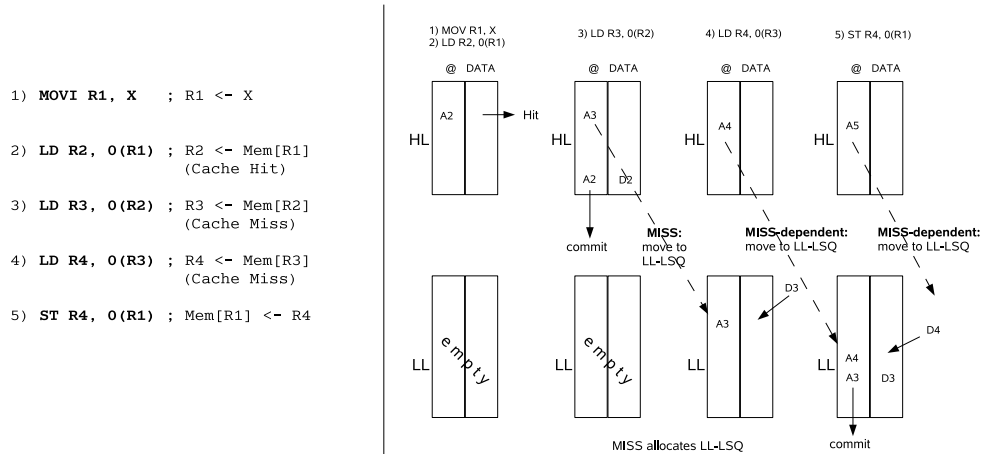


Figure 8.4: Execution (right) of example code segment (left) in a locality based Load/Store Queue

different logic designs. Address-interleaved LSQs require mechanisms to test the ordering between memory instructions that reside in different banks. In our model, memory instructions are physically ordered among the queues so that low-locality instructions are older than high-locality instructions. This idea is illustrated in Figure 8.3.

Loads and Stores are sequentially moved from the high-locality queue (HL-LSQ) to the low-locality queue (LL-LSQ) either when it is known that the address calculation is cache miss-dependent or whenever the low-locality queues are active. This is implicitly represented by the arrow in Figure 8.3. When the LL-LSQ is not active it can be kept in a low-power mode. This is beneficial to our design since the processor runs in *high-locality mode* for a large amount of time.

Consider the code segment annotated with cache behavior shown on the left of Figure 8.4. The right side of Figure 8.4 shows how execution proceeds. As long as address calculations do not depend on cache misses, address computation and issue proceed in the first queue (HL-LSQ). If the address calculation does depend on a cache miss, then the instruction migrates to the second queue (LL-LSQ) before address calculation and issue proceed. Loads that obtain their address in the HL-LSQ

but miss in the cache are also migrated to the LL-LSQ. Migration from HL-LSQ to LL-LSQ follows a scheme based on the Virtual ROB [2]. The goal of these techniques is to maintain instructions separated in two queues based on age and locality.

8.3.3 *Restricted Disambiguation Models*

The scheme so far presented allows loads and stores to disambiguate either in the HL-LSQ or in the LL-LSQ. However, disambiguation also needs to occur between locality levels. As we will see, some support logic is needed to make this work. This logic can be simplified if we restrict the disambiguation capabilities. We consider four disambiguation models:

- **Full Disambiguation:** In this model, loads and stores are allowed to disambiguate in both the HL-LSQ and the LL-LSQ. This model requires associative queues in both locality levels for loads and stores.
- **Restricted SAC:** Store Address Calculation (SAC) is restricted mainly to the HL-LSQ. If a store address depends on a long-latency register the store is allowed to migrate, but no later memory reference can be migrated until the store address calculation completes. This model simplifies disambiguation by removing LL-LSQ searches for store-load violations. The model benefits from the fact that store addresses are usually calculated in the HL-LSQ and rarely occur in the LL-LSQ (see Figures 8.1 and 8.2). Thus, stalls will be infrequent.
- **Restricted LAC:** In this model, Load Address Calculation (LAC) is restricted to the HL-LSQ. The benefits in terms of logic are less than those of the restricted SAC model as store-load forwardings will still require searching for stores in both HL-LSQ and LL-LSQ. Moreover, loads tend to have many more long-latency address calculations than stores so performance is likely to degrade more noticeably.
- **Restricted LAC/SAC:** In this model both loads and stores are restricted. Disambiguation resources are considerably simplified. However, the store window may be large so a solution for the LL-LSQ is still necessary.

For common parameters like a 10-cycle L2 cache and a 4-way processor, the HL-LSQs need not be larger than 24-32 entries, so a conventional-sized queue is enough.

8.3.4 *Hardware Disambiguation Schemes*

Since the LL-LSQ holds all low-locality loads and stores, it may need to buffer hundreds of memory references. We address the problem of the large LL-LSQ by banking the structures. To keep the sequence of memory instructions we bank

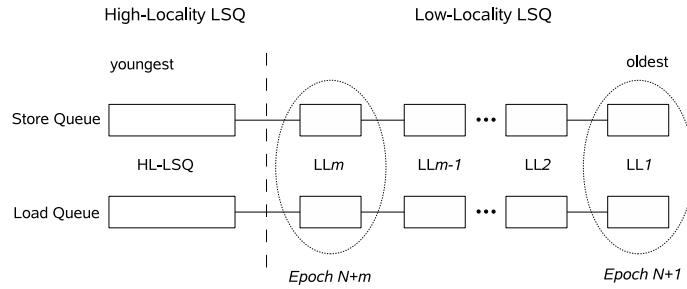


Figure 8.5: LSQ with banked LL-LSQ

based on age, not address. The number of banks is a design parameter. For the implementation we want to have as few banks as possible to minimize complexity, but enough banks for each to have smaller size. Figure 8.5 shows the partitioning.

Despite the multiple structures, this scheme still represents a sequential window of memory instructions. Since each partition of the LL-LSQ contains a sequential portion of loads and stores of the instruction window –which we call a *memory epoch*– we call our LSQ scheme the *Epoch-based Load Store Queue* or *ELSQ*, in short. Note that instructions never travel between epochs. We will use the banked scheme as the basis and on top of it implement a scalable disambiguation scheme for ELSQ.

Implementing an eager disambiguation scheme consists of implementing store-load forwarding at load issue and store-load violation detection at store issue. We now describe how this task is accomplished in the Epoch-based LSQ. The ELSQ uses a two-level disambiguation. The first level is *Local Disambiguation*. This disambiguation occurs within the epoch and involves no global searches. Loads search the local epoch’s store queue for matches while stores search the local epoch’s load queue for violations. If a load finds a matching store, the procedure stops as there is no need to perform a global search. In this case the power of the search is reduced to a single epoch. The scheme benefits from the fact that the majority of store-load forwardings happen among close store-load pairs. A similar procedure is applied to stores when they find a local violation.

If the local search does not hit a global search is conducted. *Global Disambiguation* provides the overall integration necessary for correctness. Its goals are the same as local disambiguation, i.e. having loads get the correct value from matching stores and having stores check loads for violations.

We propose two filtering schemes to avoid unnecessary searches. The goal of these schemes is to minimize the number of searches with a minimal hardware budget. One constraint that the filters need to satisfy is that the access time must be no longer than the time it takes to search the local store queue or the L1 cache access latency. If the filter cannot satisfy this condition load execution time will grow with

a noticeable performance penalty. For the ELSQ we study two filters: one based on L1 cache lines (*Line Filter*) and one based on Bloom filters (*Hash Filter*).

LINE-BASED FILTER In the first filter two bit-vectors (one for loads and one for stores) with as many entries as the total number of epochs are associated with every L1 cache line. The full collection of bit-vectors forms a table that we call the *Epoch Resolution Table* (ERT). There are two cases in which the ERT is updated: First, when a memory instruction with a known address is inserted into an LL-LSQ epoch, it sets the bit corresponding to its epoch and cache line in the ERT; and second, when an instruction obtains its address while in the LL-LSQ.

Global disambiguation proceeds as follows. In parallel with *local* store queue search during load issue, the cache line and the ERT store bit-vector are accessed. The value from the cache is used only when there is no active bit in the ERT store bit-vector. If an active bit is found it means that there is a possible match with a store and a remote search is conducted. A load pays an additional latency penalty while waiting for the search, even if it does not result in a match. The information from the bit-vector contains the epoch to which the likely matching store belongs. Using this information the load accesses the LL-SQs searching for the store. It searches the epochs that were active in the bit vector, one at a time, starting from the most recent one. This considerably reduces the energy required for the searches.

For this scheme to work it is necessary that the address-known memory instructions in the low-locality queues have an associated bit in the cache ERT. Thus, the system requires that all referenced lines be allocated in the L1 cache. Note that the data need not be available. When a new address appears in the LL-LSQ it is necessary to allocate the line and lock it in the cache. Locking is necessary because a replacement would break the disambiguation mechanism. If the new line cannot be allocated because all the lines in the set are already locked then special action needs to be taken. If the address belongs to an instruction that is being inserted from the HL-LSQ, then the insertion procedure is simply stalled. However, when the address is due to a memory reference that issued in the LL-LSQ the situation is more complex. The problem is that the loads that are locking the set may be younger than the load that issued. Stalling would result in a deadlock. As a solution, when this happens we proceed to squash the instruction window starting from the load that tried to lock the line and restart execution. This is supported by the recovery logic.

Locking cache lines does not involve any additional structures as the replacement algorithm can take care of everything. It will only replace lines for which there are no active bits in the ERT.

ADDRESS HASHING BASED FILTER To avoid the complexity resulting from modifying the algorithm to handle cache-line locking we also study a more conventional method based on Bloom filters [70]. In this method, the ERT is indexed using a hash consisting of a set of the lower-bits from the address. Thus, when a memory

instruction is inserted into the LL-LQs or computes its low-locality address it takes n bits of the address, it indexes the ERT and activates the bit corresponding to its epoch. This scheme is decoupled from the L1 cache. The access time to the ERT will depend on its size, so this is a parameter to take into account.

We have described the two global disambiguation schemes using the *Full Disambiguation* configuration. Using restricted schemes may simplify the hardware considerably. Restricted SAC, for instance, eliminates the need for the *Loads-ERT*. The *Stores-ERT*, however, is always necessary for operation.

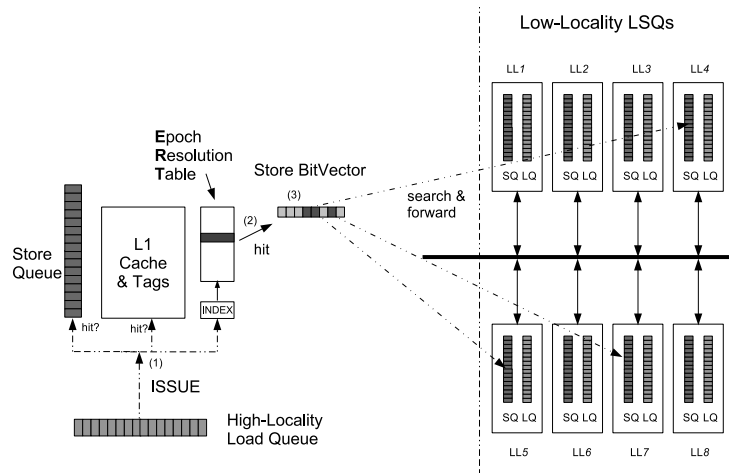


Figure 8.6: Store-load forwarding for High-Locality Loads hitting in the ERT

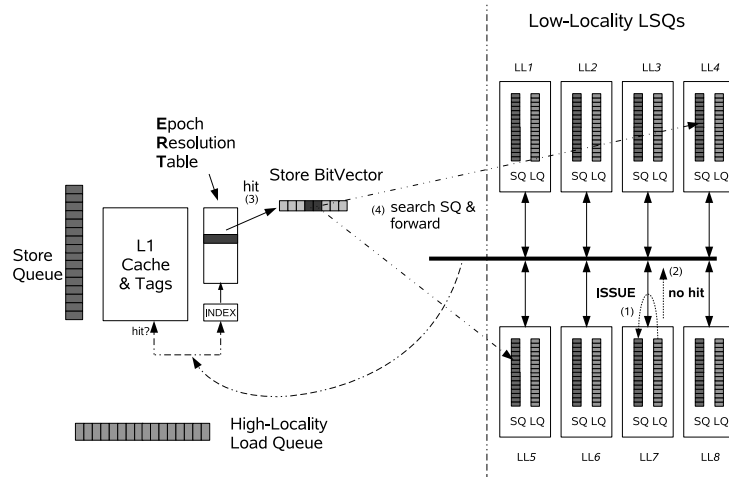


Figure 8.7: Store-load forwarding for Low-Locality Loads hitting in the ERT

Figures 8.6 and 8.7 shows the operation of store-load forwarding for both high-locality and low-locality loads forwarding from a low-locality store. The access to the ERT is guarded by a structure with the label INDEX. This structure reads the address and decides where in the ERT to index, depending on the filtering mechanism. Note

that the figure only shows the store-search part of the forwarding process. The data needs to be sent back to the load, following the same path backwards.

In both filtering schemes, when an epoch commits, the stores are sent to memory and the two columns that represent this epoch in the ERT are cleared. This way, lines in the line-based ERT get automatically unlocked by the bit handling mechanism. This method is notably simpler than the HSQ [33] method that requires counters to be decremented one-by-one for every store in the checkpoint.

8.3.5 *Non-Associative Load Queue with Load Re-Execution*

The methods we have introduced work well as a way to reduce search activity in the Load and Store Queues. A different approach is to attempt complete removal of parts of the LSQ. As we have mentioned before, much research has concentrated on removing the Load Queue and maintaining program semantics via *load re-execution* [68, 69]. Load Re-execution consists of executing an optimized load again during the commit stage to check for the validity of the optimization.

Only loads that may have incurred a store-load ordering violation should re-execute. It is important to take this into account, because cache access bandwidth is limited and expensive. Many techniques to reduce the number of re-executing loads have been researched. For instance, Roth proposed tracking whether the load is vulnerable to any recently committed store [69]. An alternative way to reduce the re-execution rate is to track whether there are stores in flight with unknown addresses and, in the case of store-load forwarding, see if they are younger than the store that is forwarding. If this is not the case then there is no need to re-execute. This is called the *no-unresolved-store-filter* [68].

These techniques can be added to ELSQ to make the Load Queue non-associative. However, care needs to be taken with the *no-unresolved-store-filter*. The filtering that guards searches in the LL-SQ does not track stores with unknown addresses. As a solution, it is possible to track which epochs contain address-unknown stores by adding a new ERT table, and adding counters in the epochs to track unresolved stores. The additional ERT table would need to be accessed by all executing loads (except locally forwarded loads). This is a trade-off that needs to be analyzed in relation with performance (see Section 8.5.6).

8.3.6 *Coherence and Consistency*

The epoch-based LSQ is designed with traditional memory semantics in mind. Externally the system sees a huge load-store queue. The cache subsystem features only one L1 data cache and there is only one L2 cache between L1 and main memory. ELSQ does not modify the problem of maintaining caches coherent, which can be solved using either *snooping* or *directory-based* schemes.

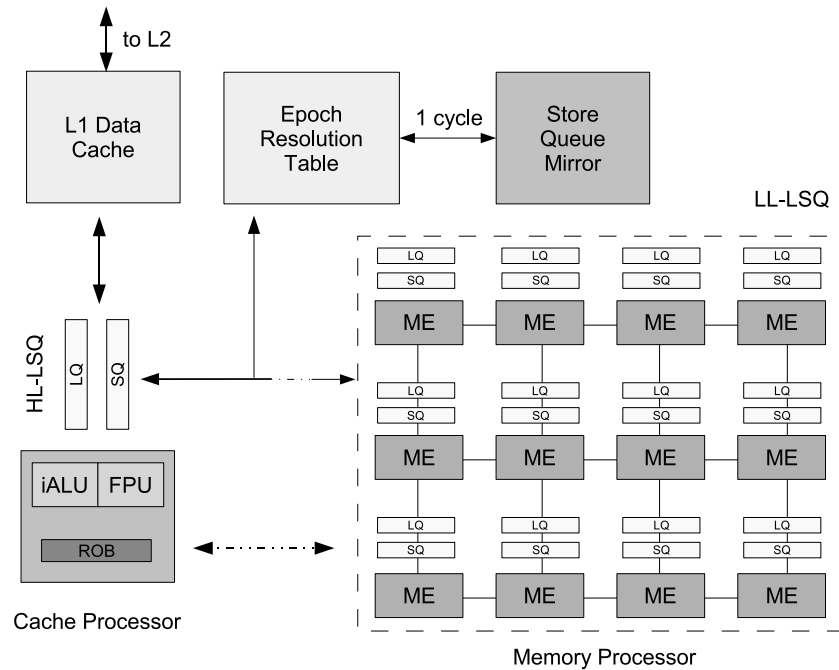


Figure 8.8: Integration of ELSQ on top of FMC

ELSQ is designed to support total store ordering. Most current processors operate under this model or use weaker models. Thus, ELSQ can be implemented on most architectures.

8.4 INTEGRATION WITH LOCALITY-BASED PROCESSOR

The processor model that have relied on so far is based on conventional technology. It is simple to understand and can conceptually work together with ELSQ, but in a real implementation it is not valid since it cannot scale to our goal of handling thousands of in-flight instructions. We now proceed to ELSQ on top of the FMC microarchitecture developed in Chapter 6. The natural way to complete the integration is to establish a one-to-one relationship between the memory engines and the ELSQ concept of epochs.

The overall organization and interconnect of this architecture is shown in Figure 8.8. FMC uses a mesh network to interconnect the different memory engines. By mapping ELSQ, every epoch is mapped to a memory engine. As a result the LL-LSQ is distributed along the memory engines. Access to the memory engine network is provided by a bus that interconnects the CP and the MP.

For the ELSQ, it is critical that store-load forwarding is as fast as possible. Often high-locality loads forward from low-locality stores. If this operation performs a round-trip every time (> 8 cycles) the penalty may be noticeable. To alleviate this problem we suggest a final addition to the ELSQ: implementing a *Store Queue Mirror* (SQM). The SQM is a replica of the LL-SQs located next to the ERT. It is updated when a store address appears in the Memory Processor. Accessing the SQM in the Cache processor costs only one additional cycle after ERT access. Figure 8.8 shows the location and interconnect of the SQM. When implemented, the SQM also acts as the buffer for stores before they commit. Thus, the SQM does not incur any additional power due to network trips.

8.4.1 Exceptions and Recovery

Maintaining correct state when exceptions occur is another important issue in the design of ELSQ. Being able to recover at any point of the LSQ is a complex issue even for smaller designs. In ELSQ we simplify this issue by relying on checkpoints. ELSQ considers checkpointing only for the low-locality LSQ, which holds many more instructions. The Cache Processor checkpoints branches so recovery may proceed like in a MIPS R10000 processor [16]. For LL-LSQ recovery, a checkpoint is associated with every epoch. When an exception occurs, the processor restarts execution starting from the instruction that initiates the epoch. To keep the state of the ELSQ consistent with program semantics, all loads and stores belonging to this epoch and to younger epochs –including the HL-LSQ– are squashed. This means that some correct path instructions get squashed. Nevertheless, low-locality recoveries are much less frequent than high-locality ones. Using checkpointing for the ELSQ is similar to the use of checkpointing for large window processors such as [36, 33, 14, 41].

8.5 EVALUATION

We now evaluate the performance of the ELSQ. First we analyze global performance issues, establishing the epoch size and comparing the overall performance (Sections 8.5.2 and 8.5.3). We then proceed by focusing on the store queue, analyzing the performance of the filtering schemes (Section 8.5.4). Finally, we analyze the load queue and evaluate the restricted disambiguation and re-execution schemes (Sections 8.5.5 and 8.5.6).

8.5.1 Simulation environment

The ELSQ models have been implemented top of an execution-driven simulator modelling the FMC microarchitecture. Conventional speculative out-of-order pro-

Parameter	Value
Fetch/Decode BW	4 insts per cycle
CP ReOrder Buffer Size	64
ME Max Instructions	128
ME Max Loads	64
ME Max Stores	32
CP Integer Issue Queue Entries	40
CP FP Issue Queue Entries	40
CP Scheduling Policy	Out-of-Order
CP INT/FP Registers	96/96
ME Issue Queue Entries	20
ME Scheduling Policy	In-Order MultiScan [41]
ME Issue Width	2-way
OoO-64 Integer IQ Entries	40
OoO-64 FP IQ Entries	40
OoO-64 Scheduling Policy	Out-of-Order
OoO-64 INT/FP Registers	96/96
Number of Cache Ports	2 read/write ports
L1 Cache Configuration	32KB 4-way, 32-byte lines
L1 Cache Latency	1 cycle
L2 Cache Configuration	2MB 4-way Assoc.
L2 Cache Latency	10 cycles
Main Memory Access Time	400 cycles

Table 8.1: Default Processor Parameters

processors are simulated by disabling the Memory Processor part of the simulator. For ELSQ, both Line-based and Hash-based global disambiguation may be simulated. An unlimited conventional LSQ is also modeled. We also implement a model of load re-execution using Store Vulnerability Windows [69] and the *no-unresolved-store-filter* [68]. Table 8.1 shows the default values that apply for the different microarchitectures unless explicitly stated. The simulation points and evaluation methodology is the same as presented in Chapter 3.

8.5.2 Tuning Epoch Size

First we establish the sizes of checkpoints and epochs. The number of epochs is the same as the number of checkpoints. We choose 16 epochs since this value has been shown to work well for the FMC architecture. We set the maximum number of low-locality instructions per epoch to be 128. This number includes integer ops, floating point ops, control ops and address calculations. Using this size for the checkpoint and a total of 16 epochs, we find that the maximal IPC for SPEC FP that can be reached lies at 2.99. We will now size the LSQ trying to stay within 1% of the unlimited LSQ performance. Setting the Load and Store Queue to 64 and 32 entries

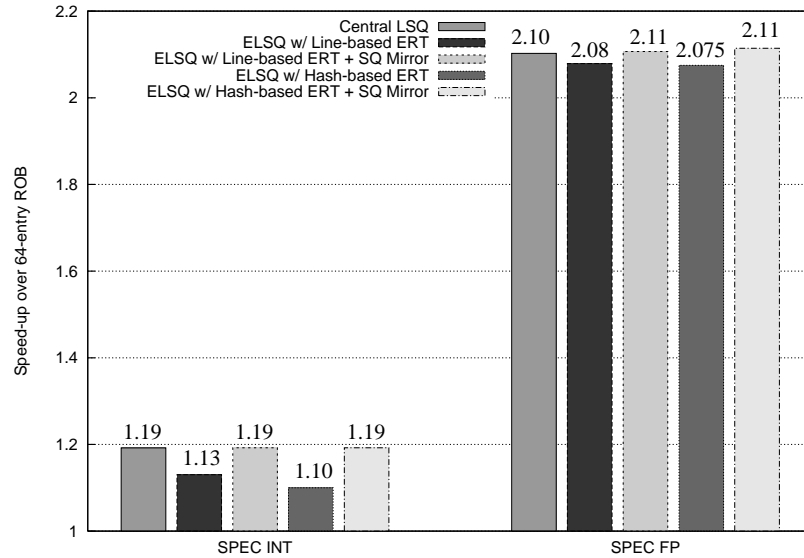


Figure 8.9: Speed-up of large window LSQ schemes over conventional 64-entry ROB

yields a average slow-down of 0.9% (7% worst-case). This seems a good trade-off from a power perspective. For this sizing procedure SPEC FP was used since at large window sizes it is more sensitive to variations than SPEC INT.

8.5.3 Performance of Epoch-based LSQ

We now evaluate the performance of the large window scheme. We evaluate both the cache-line-based and the hash-based ERT schemes. For a fair comparison we model the size of the hashing-based ERT to be the same as that of the cache line-based ERT. For the 32KB 4-way L1 that we use the size of the ERT amounts to 4KB of storage (2KB for the Load-ERT and 2KB for the Store-ERT). This translates to a 10-bit address hashing. For each ERT scheme we evaluate the impact of implementing the Store Queue Mirror. Finally, we also model a single-cycle unlimited-size centralized Load Store Queue. This LSQ is located in the Cache Processor to minimize store-load forwarding occurring in high-locality code. However, loads that execute in the Memory Processor suffer the corresponding round-trip penalty. The results are shown in Figure 8.9 as speed-ups over a conventional processor with a 64-entry ROB that yields IPCs of 1.55 and 1.42 for SPEC INT and SPEC FP, respectively. Note that this processor size is not representative of current technologies. It is chosen because it features the same sizes as the Cache Processor in the FMC architecture and emphasizes the impact of the Memory Processor.

The figure shows that for SPEC FP the performance is quite good for all schemes. The presence of the SQM improves performance by about 1% and provides a performance that is slightly larger than that of the centralized queue. This small performance gain comes from local forwardings in the LL-LSQ that require a full round-trip in the case of the ideal centralized queue. SPEC INT, on the other hand, is much more sensitive to the store-load forwardings from low-locality stores to high-locality loads. The presence of the SQM has thus a big impact on the performance, providing up to 8% more performance. Once the SQM is implemented, ELSQ performs at the same speed as the idealized queue.

We have measured the number of forwardings to better understand the behavior of the ELSQ. For SPEC FP we measured an average of 3.9 forwardings every 100 instructions. Out of these, 1.4 forwardings happen locally while the remaining 2.5 occur globally. This means that about one third of forwardings can be resolved locally. About one third of these (i.e., 1/9th of all forwardings) are local forwardings in the Memory Processor. This explains the performance advantage of the ELSQ compared to the centralized queue. For SPEC INT the number of forwardings is slightly larger. The average number of forwardings for integer applications was measured to be 8.5 forwardings every 100 instructions. In this case, local forwardings outnumber global forwardings (4.6 local forwardings compared to 3.9 global forwardings).

We also measured the number of store-load ordering violations. For the kind of speculative disambiguation scheme that we model, with stores issuing as soon as the address register is available, we found that only about 1300-1600 violations occur every 100 million instructions. This results in one violation every 60000 instructions on average. The impact of these violations is thus negligible. For this reason we have not implemented a memory dependence predictor scheme such as store sets [71].

8.5.4 *Performance of Global Disambiguation*

In this section we evaluate the performance of the filtering mechanisms for global memory disambiguation. The efficiency of the mechanism has in principle little impact on IPC. Different filtering schemes affect the number of searches that happen in the LL-LSQ, either for store-load forwarding or ordering violations (in the latter case it affects also the number of searches in the HL-LSQ). The schemes will have an impact on area, complexity and power, but the impact on performance is small. The Line-based scheme could be a little bit of an exception here, since it requires to lock cache-lines. However, for the 4-way 32KB L1 cache that we use, the performance penalty still lies only around 0.4% and can be safely ignored.

The main effect that needs to be evaluated is the number of false positives that are generated. A false positive happens when the ERT directs the load or store to search in an epoch where a matching address is actually not present. Such a search is useless and wastes power. Thus, goal of the ERT scheme should be to minimize

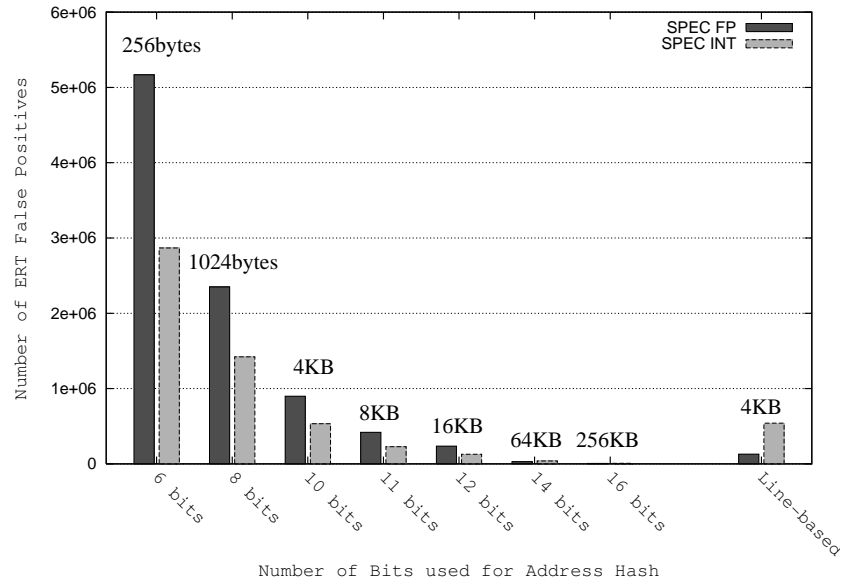


Figure 8.10: Performance of the filtering schemes varying ERT size

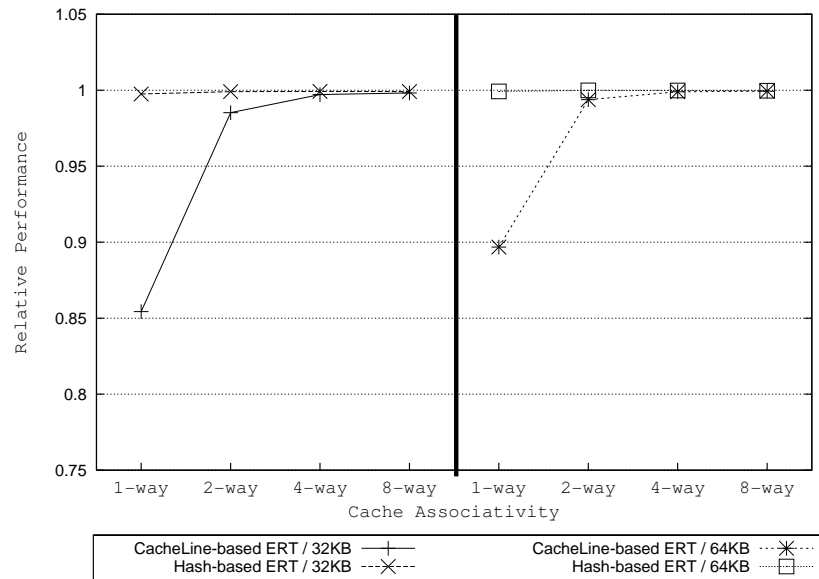


Figure 8.11: Performance of the filtering schemes varying L1 cache (SPEC FP)

these searches. For the address-hash based scheme this goal can be achieved by incrementing the number of bits. Doing so, however, increments the hardware budget of the scheme. Choosing the best scheme involves a trade-off. Figure 8.10 shows the average number of false positives for 100 million committed instructions in SPEC FP and SPEC INT together with the estimated hardware budget. The

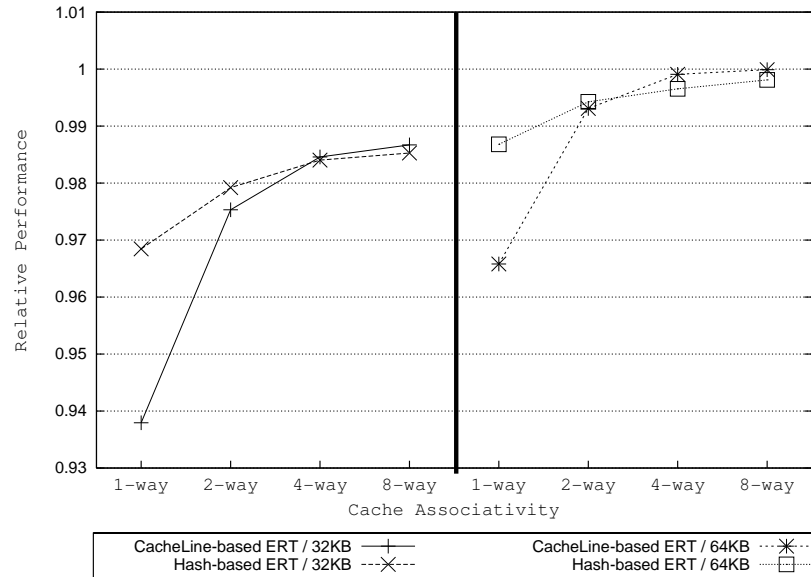


Figure 8.12: Performance of the filtering schemes varying L1 cache (SPEC INT)

hardware budget is estimated by taking into account that we need two ERT tables (one for loads and one for stores) and that every entry stores 16 bits. The figure shows that ERTs of at least 4KB are necessary to have less than 1 false search every 100 instructions. Note that 4KB here means 2KB for the Load-ERT and 2KB for the Store-ERT. The figure also shows that, using 32-byte lines, the line-based scheme requires about half the hardware budget for similar accuracy. Finally, it also shows that the filtering performance depends a lot on how well the filter maps to the application behavior. The line-based scheme performs much better on SPEC FP while the hashing scheme seems to have better performance on integer applications.

We also evaluate the impact of modifying the L1 cache on the Line-based ERT scheme. This scheme depends on the configuration of the cache since the ERT requires long-latency address calculations to have the corresponding cache lines *locked* in the cache. Intuitively this means that high-associativity caches may be necessary since line conflicts are handled via processor stalls or squashes. This section will evaluate how large the L1 cache need to be to minimize the losses. We evaluate a series of cache configurations of 32KB and 64KB, with associativity ranging from 1 to 8 ways. To compare we also add a hash-based ERT architecture. The interleaving is set so that hardware budgets for ERT are the same in both schemes. Thus, for the 32KB cache 10 bits are used and for the 64KB cache 11 bits are used. Figures 8.11 and 8.12 shows the results for this test relative to the highest scored performance. The figure shows that an associativity of 4 recovers the lost performance for both SPEC INT and SPEC FP. It also shows that the L1 cache size

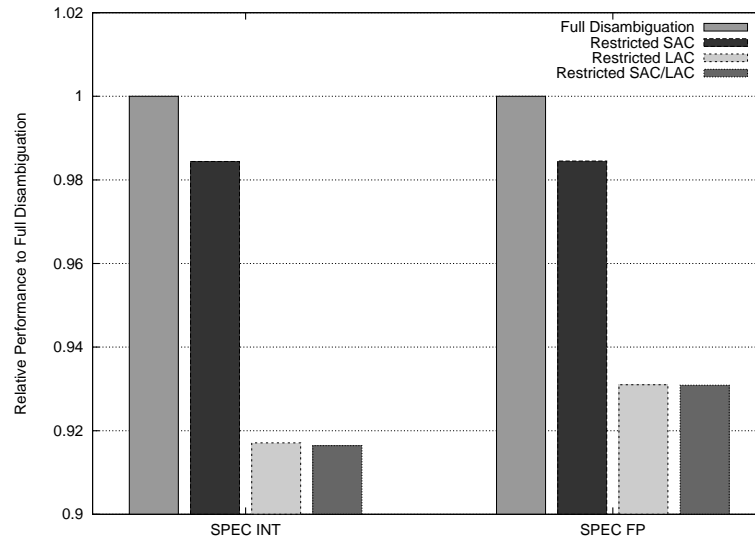


Figure 8.13: Relative performance of restricted disambiguation models

and associativity has a much higher effect on performance for SPEC INT than SPEC FP.

8.5.5 Restricted Disambiguation Models

We now analyze the performance of the restricted disambiguation schemes introduced in Section 8.3.3. Figure 8.13 shows the performance of the four disambiguation models. Full disambiguation has been chosen as the baseline against which comparison is being made. The figure shows that restricted LAC involves a higher penalty than restricted SAC. The reason for this has to be found, as shown in Figures 8.1 and 8.2, in the fact that many more loads with low locality address calculation exist than stores. Finally, when both stores and loads are restricted, performance is similar to just using restricted LAC. This is a result of low locality loads being much more frequent than stores. Thus the stalls arising from restricting their address calculation have a much higher penalty.

The performance of all four schemes is quite good. In particular, restricted SAC yields a slowdown that is below 2% for both SPEC FP and SPEC INT. Looking into the benchmarks further reveals that the slowdown is due to peculiarities of particular applications. For example, in the SPEC FP case, all the slowdown is attributable to *equake*, which suffers around a 30% performance loss. Much of the execution of the simulation point is covered by the `smvp()` function, which involves heavy multilevel pointer dereferencing, for both loads and stores.

Restricted SAC has a notable implementation advantage: it eliminates the need for a large associative Load Queue. Since stores may only compute their address in the Cache Processor, only loads in the small high-locality load queue (HL-LQ) may incur ordering violations. As a result, global disambiguation for load violation is no longer necessary, which eliminates the need for the Load-ERT. Note that, unfortunately, the converse does not hold. Restricted LAC does not allow to remove the store queue, so the global disambiguation scheme devised previously would still need to be implemented. Overall this means that restricted LAC is probably not a good idea.

8.5.6 Large Window Load Re-Execution

Finally, we analyze how Re-Execution performs in this context. We implement the technique of Store Vulnerability Windows [69] and remove the Load Queue. We do not modify any other structure. The implementation of SVW comes in two variants:

CheckStores: In this variant, when a load issues and forwards from the store queue, it checks if any stores with unknown address exist between the store-load pair. If so, the load re-executes during commit. This is the *no-unresolved-store-filter* [68]. As will be seen, doing so improves performance, but it adds complexity to the store-load forwarding machinery. Complexity is increased because it is necessary to implement a mechanism that tracks unresolved stores.

Blind: In this variant, we do not check whether stores with unknown addresses exist. Instead, we use only the SVW filtering mechanism to decide whether a load needs to be re-executed or not.

We evaluate the performance of the SVW scheme for both IPC and increase in cache activity. In the evaluation we take into account the fact that re-executing loads forces subsequent stores to commit after the cache access completes. In most cases this will be the next cycle (L1 access latency), but some loads re-execute from the L2 and, in some very rare cases, the load may re-execute from main memory. This behavior can affect performance when the re-execution rate is high.

Evaluating SVW in the context of large-window architectures is especially interesting as large windows are much more likely to create ordering violations compared to small windows. Figures 8.14 and 8.15 compare the performance of SVW on the FMC – emulating a window of around 1500 instructions – (right) and a smaller 64-entry ROB processor with a conventional out-of-order architecture (left). The small processor is provided to show how the number of re-executions increases. The figure shows three configurations for the Store Sequence Bloom Filter (SSBF), ranging from 8 to 12 bits.

From the results we see that using 12 bits has very good performance in all schemes. The resulting table might, however, be a little larger and more power-

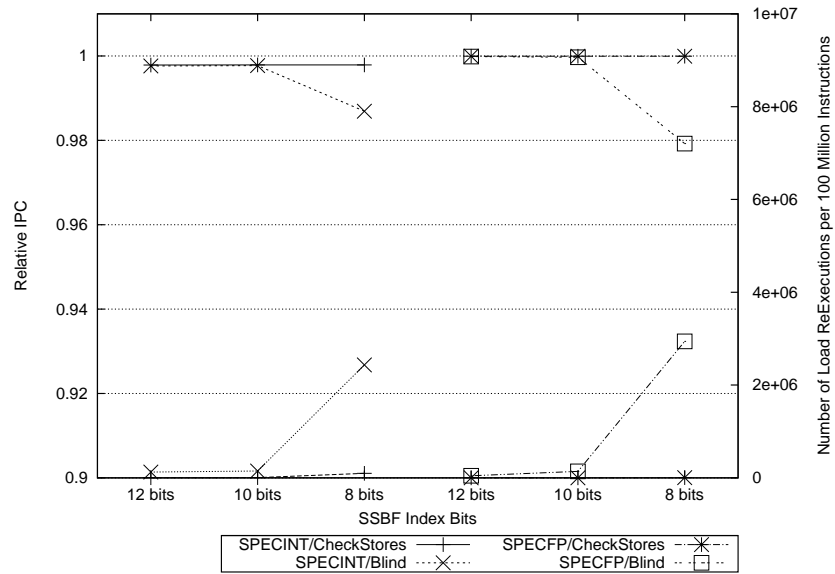


Figure 8.14: Performance of SVW on SPEC CPU 2000 relative to a model featuring Load Queue and a 64-entry ROB.

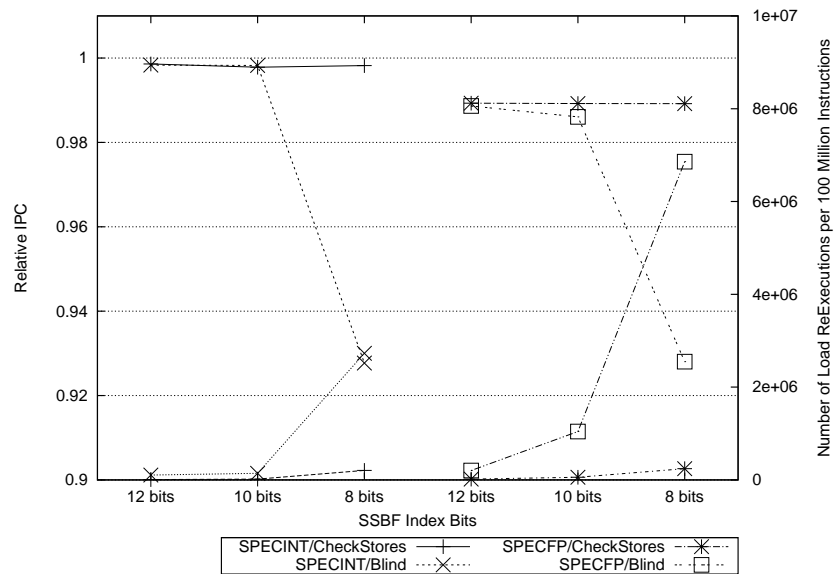


Figure 8.15: Performance of SVW on SPEC CPU 2000 relative to a model featuring Load Queue and a ~1500-entry ROB.

hungry than desired. Using 10 bits for the SSBF is still a good option. Performance is almost unaffected except for SPEC FP when the *no-unresolved-store-filter* is not used. The additional re-executions will increase the energy dissipated by the cache,

but it needs to be taken into account that the CheckStores mechanism implements additional structures that are accessed by all loads issue while the processor is in *low-locality mode*. This will add to the power consumption. Finally, even a SSBF with 8 bits works nicely if the filter is implemented. Otherwise, the performance for SPEC FP starts to degrade considerably ($\sim 7\%$ vs. 1%).

8.6 ENERGY CONSIDERATIONS

Our goal in introducing the ELSQ is to provide a large, high-performance LSQ that operates with little additional power and low complexity. We now analyze the power characteristics of the ELSQ. Increasing the size of a standard load-store queue would increase the energy consumption excessively, precluding its implementation. However, this is not true for the ELSQ. Although ELSQ keeps many queues, most of them are not active at any given moment and do not perform searches. In general, when the ERT returns a positive match, only one low-locality queue is searched. This happens because of the highly accurate filtering methods as was shown in Figure 8.10.

First, let us evaluate how much time the processor spends in *high-locality mode*. During this mode the Memory Processor does not need to track instructions because no cache misses have occurred recently (i.e. the Memory Processor is empty). Since the processor does not use low-locality resources, the LL-LSQ together with the ERT and the associated logic can be kept in a low power mode. Figure 8.16 shows the percentage of time in which the processor is in the high-locality mode as a function of the L2 data cache size.

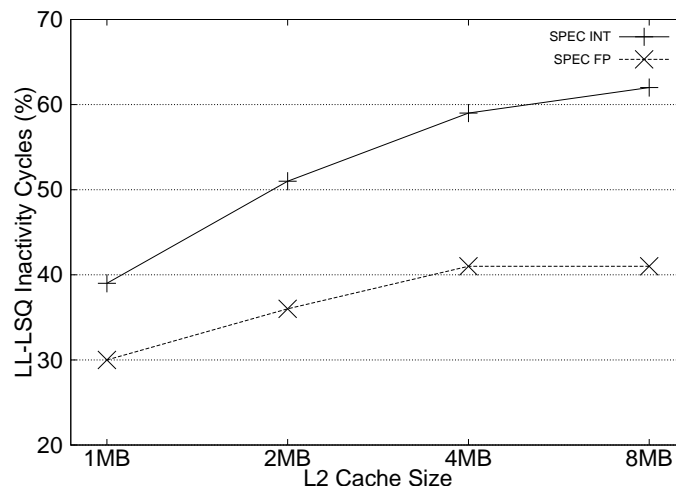


Figure 8.16: Percentage of time in which the FMC is in high-locality mode

The figure shows that even with a small 1MB cache, one third of the time the processor only uses the small HL-LSQ machinery that tracks only 32 loads and 24

	Configuration	HL-LQ	HL-SQ	LL-LQ	LL-SQ
SPEC FP	OoO-64	8.692	27.006	0	0
	OoO-64-SVW	0	27.006	0	0
	FMC-Line	8.761	25.929	0.119	8.902
	FMC-Hash	8.618	25.531	0.123	9.893
	FMC-Hash-SVW	0	26.010	0	9.795
	FMC-Hash-RSAC	8.732	25.815	0	9.378
SPEC INT	OoO-64	11.326	32.387	0	0
	OoO-64-SVW	0	32.387	0	0
	FMC-Line	13.356	37.703	0.115	10.348
	FMC-Hash	13.354	37.615	0.114	9.445
	FMC-Hash-SVW	0	37.602	0	9.606
	FMC-Hash-RSAC	12.867	36.294	0	8.056

	Configuration	ERT	SSBF	RoundTrips	Cache	Speed-Up
SPEC FP	OoO-64	0	0	0	33.375	1.0
	OoO-64-SVW	0	26.591	0	34.135	0.997
	FMC-Line	27.521	0	1.561	31.862	2.09
	FMC-Hash	27.281	0	1.701	31.662	2.10
	FMC-Hash-SVW	27.453	26.591	1.546	32.971	2.08
	FMC-Hash-RSAC	27.037	0	1.468	31.610	2.07
SPEC INT	OoO-64	0	0	0	37.328	1.0
	OoO-64-SVW	0	29.769	0	38.081	0.998
	FMC-Line	34.327	0	0.544	39.961	1.196
	FMC-Hash	34.250	0	0.541	39.887	1.195
	FMC-Hash-SVW	34.130	29.769	0.438	39.948	1.190
	FMC-Hash-RSAC	32.624	0	0.354	39.291	1.176

Table 8.2: Number of access to LSQ components (in millions)

stores. This is similar in size to what today’s processors use. As the data cache grows to 8MB the percentage of time during which the LL-LSQ runs in minimal-power mode averages 50%. Note that, when the Memory Processor is active, not necessarily all epochs queues are allocated. For the 2MB L2 cache an average of 5.73 epochs are allocated for SPEC FP while for SPEC INT this number drops to 4.77. Furthermore, if a designer wants to increase the efficiency of the queues, the Memory Processor can be shared between threads [41].

To estimate the dynamic power requirements of the implementation we track the utilization of the structures including accesses to the queues and the number of ERT lookups and network roundtrips of the searches (with or without data). Table 8.2 shows the average number of events for 100 million committed instructions for each of the SPEC FP and SPEC INT simulation points. Several large window and two small window configurations are evaluated. The structures of the 64-entry processor match the sizes of the cache processor allowing us to better understand the power behavior of the processor. The SVW implementation uses an SSBF with 10 bits and it does not implement the *no-unresolved-store-filter*.

There are several interesting observations to make from these tables. The structures that receive the most searches are the High-Locality Store Queue and the Epoch Resolution Table. The number of accessing instructions ranges from $25 * 10^6$ (FMC-Hash HL-SQ access in SPEC FP) to $37 * 10^6$ (FMC-Line HL-SQ access in SPEC INT). For the modeled 4-way fetch/decode architecture, the ERT and the HL-SQ will need to be dual ported, while the high locality load queues, which are accessed by around $8 * 10^6$ – $13 * 10^6$ instructions, may be designed as single-ported associative structures. The low-locality load/store queues see fewer instructions, between 0 and $10 * 10^6$, distributed to 5-6 subqueues (about $0.2 * 10^6$ each, on average). Thus a single port is also enough for these subqueues. Finally, note that network roundtrips can be implemented efficiently [53].

When control speculation works well, as in SPEC FP benchmarks, using ELSQ offers a good power–performance balance. For example, while OoO-64 performs 27 millions queue accesses to two-ported structures, FMC-Hash performs 25.5 million accesses to two-ported queues and 10 million accesses to single-ported queues (of similar size). This is an acceptable increase in power consumption. It is also necessary to account for the 27 million ERT accesses. The ERT is a 2KB SRAM with a similar access rate to a L1 cache, but its power consumption is much lower. Using CACTI-4.2 with a target technology of 70nm, the read energy for the ERT is 0.00195nJ while the read energy for the L1 cache amounts to 0.0958nJ. Thus, the read energy consumption of the ERT is only 2% that of the L1 Cache. With restricted disambiguation models additional gains can be achieved. RSAC reduces the number of accesses to the ERT, as stores do not access the ERT, and therefore it also reduces the number of round-trips. This is in addition to the benefit of removing the Load Queues from the Memory Processor.

Finally, unlike in SPEC FP, the number of LSQ access in SPEC INT grows with the aggressiveness of the processor. This is an effect of poor control path speculation. In integer programs, correctly speculating past multiple branches is difficult, so the processor instruction window grows, but many instructions are wrong–path. New control speculation mechanisms will be necessary to overcome this limitation.

Comparing the line-based filter and the address hash filter shows that both have similar behavior. FMC-Line reduces accesses to the LL-SQ and also reduces round-trips. On the other hand, stores in the MP need to lock the line. For SPEC FP, about 5.2 million stores access their cache line and lock it. Line locking and overflow squashes do not have a noticeable impact on performance. However, the higher implementation complexity makes this technique a less suitable candidate for implementation.

Finally FMC-Hash-SVW and FMC-Hash-RSAC are compared. This will tell us which of both methods is better for load queue simplification. Energy-wise, RSAC has some nicer properties than SVW. It reduces cache accesses (4% and 0%), round trips (5% and 19%), LL-SQ accesses (4% and 16%) and HL-SQ accesses (1% and 4%), for SPEC FP and SPEC INT, respectively. Other operations are not directly

comparable: the access frequency of the SSBF (1024-entry RAM) is three times that of the HL-LQ (32-entry CAM). On the other hand SVW has marginally better performance than RSAC (0.5% and 1.2%). Without taking HL-LQ and SSBF accesses into account, we conclude the performance-power behavior is better for RSAC than SVW. Another topic that needs to be taken into account is implementation complexity. Implementing RSAC is simple: stores that do not have computed address at the head of the HL-LSQ stall migration. SVW, on the other hand, makes the whole Load Queue non-associative but requires the implementation of an additional table (SSBF) and some logic to implement the vulnerability windows.

8.7 RELATED WORK

In addition to the schemes presented in section 8.2, several other important contributions have appeared in literature.

One proposal that shares similarities with our Epoch-based LSQ is the Address Resolution Buffer (ARB) [72], a work developed in the context of Multiscalar [46]. The main similarity is the use of local and global disambiguation levels, where the global level tracks groups of instructions and the lower level individual instructions. Despite this similarity, ELSQ controls global disambiguation via an Epoch Resolution Table, a concept inspired in directory-based cache coherence schemes [73, 74].

Several researchers have attempted to improve LSQ efficiency by introducing innovations into the traditional structure of the Load Store Queue. Sethumadhavan et al. [75] propose using hardware hashing to attack the issues of performance, power and latency. Single-bit hash tables are implemented via bloom filters with the goal of filtering unnecessary searches.

Park et. al [76] propose several optimizations to reduce search frequency on the LSQ. These include using a store-set predictor [71] to reduce the search requirements, implementing a load buffer for out-of-order loads that reduces the number of load queue searches and increasing the size of the LSQ by using segmentation.

Finally, Sethumadhavan et al. [77] propose a load-store queue architecture in which entries in the LSQ are not allocated at decode, but at issue. This technique reduces the size of the queues, but requires new methods to handle overflows.

Several works have developed high-performance LSQs on top of the basic technique of re-execution [68]. One group of optimizations is based on speculative memory bypassing (SMB) [78]. The goal of this optimization is to reduce the pressure on the Store Queue. However, an aggressive configuration that predicts all store-load forwardings [79, 80] yields a non-associative store queue.

8.8 CONCLUSIONS

In this chapter we have presented the design of a Load/Store Queue for both a conventional microarchitecture and the FMC microarchitecture. This LSQ design, called the Epoch-based Load/Store Queue, applies the concept of Execution Locality to address calculation and partitions the large low-locality window into multiple sequential *epochs*. On top of these epochs it implements a two-level disambiguation scheme that allows it to scale to thousands of loads and stores. The ELSQ design follows the design guideline of employing only small scale structures to emulate larger structures. Sequentiality is provided between epochs to simplify memory ordering and integration with FMC. The ELSQ achieves high performance by enabling high memory parallelism and fast load execution in the Cache Processor. In addition, it enables reasonable power consumption by concentrating accesses on small structures. With the proposal of the ELSQ and its integration on top of the FMC we consider that the goals of this thesis have been satisfactorily completed.

CONCLUSIONS

This thesis has dealt with the problem of single-thread performance in the context of the memory wall. While current industry trends are focusing on the construction of multicore chips, there are several reasons why single-thread performance is still going to continue to be important in the future:

1. Parallel programming is not a simple task. Correctly handling synchronization of multithreaded programs can be very complex, and so the programmer may decide to stick to a more conventional programming style. In addition, some codes do not have enough coarse-grain parallelism to be split into independent threads efficiently. Single-thread performance will be the key to accelerate these applications.
2. Even fully parallelized codes will continue to include a sequential component that we want to be able to execute at maximum performance. Thus this scenario will also profit from high-performance single-threaded cores.

Since parallel performance can be achieved by replicating cores and sharing the memory bus, we have concentrated on building a microarchitecture for single-thread performance to provide both single-thread and multi-thread performance. Our novel approach consists in separating the miss-dependent instruction stream from the cache-dependent instruction stream and distributing both streams into different processors, each one specifically design to provide the best power-performance relationship for the type of instructions it processes:

- The miss-dependent stream is processed by a latency tolerant *Memory Processor* featuring small issue widths and simple in-order instruction issue logic. This is the key feature that allows us to build a kilo-instruction window.
- The cache-dependent stream requires higher widths and more aggressive dynamic scheduling techniques like out-of-order execution. However, the window of this *Cache Processor* is small and thus profits from using only small structures.

This separation of streams is what we call *Execution Locality*. In this thesis we have proposed two architectures based on this principle. The Decoupled KILO-Instruction

Processor (D-KIP) has a single point of execution in the Memory Processor. It is a low power variant which, however, loses some performance due to serialization of independent loads in the Memory Processor. The Flexible Heterogeneous MultiCore (FMC) distributes the Memory Processor into multiple sequential Memory Engines which iteratively scan their instruction window. This proposal overcomes the load-serialization problems of the D-KIP and is well suited for the integration into a multicore, where the network of memory engines can easily be shared among different threads. On numerical workloads, the performance of the FMC is high enough that implementing a stream prefetcher provides no additional benefits. On a different note, FMC's checkpointing scheme is also a lot simpler than D-KIP's checkpointing scheme. This thesis has been completed with the proposal of a Load Store Queue based on the concepts of Execution Locality. Our LSQ proposal keeps traditional semantics while distributing the queues into a two-level approach coordinated by a directory. In doing so it profits from locality in store-load forwardings. Our evaluations show that this queue is energetically equivalent to the small queue of a conventional processor while providing kilo-instruction performance.

The main conclusion from this work is that providing a KILO-Instruction processor at reasonable cost can be done using the approach presented in this thesis. Our D-KIP/FMC processors are fully based on distributed structures similar in size to those found in a conventional microprocessor. The evaluations have also showed that for numerical applications a processor such as FMC performs at the same level of a conventional processor with an ideal L2 cache (compare Figures 6.5 and 4.2). Unfortunately for integer applications the performance impact of these techniques is much smaller. There are two main reasons why integer performance does not scale with window size:

- Control path speculation is not as efficient as in numerical codes. On the one hand the predictability of branches is much smaller in integer codes than numerical codes. On the other hand, the frequency of branches is much higher. This means that correct-path traces are much smaller for integer applications than numerical codes. As a consequence it is not possible to run far ahead. The processor is continuously recovering from mispredictions. Actually, much of the performance gains that we see in integer applications comes from wrong path prefetches that are later reused in the correct path.
- Chasing pointers are much more frequent in these kind of applications. While arrays are used everywhere in numerical applications, hashes and lists are most frequent in integer codes. Linked lists introduce serialization in the access of data structures which disallows dynamic parallelization by the processor. If, in addition, list traversal results in L2 misses, performance cripples.

Thus, future work should research novel techniques to overcome the problems of integer applications. Techniques such as multipath execution or control path independence have the potential to overcome the problem of control path misspeculation, but at the cost of additional hardware complexity. On the other hand, techniques such as value prediction can be used to speculate on the outcome of pointer dereferences and speculatively parallelize list processing. How to integrate these techniques into the FMC is the topic of open research.

LIST OF PUBLICATIONS

In addition to the work on KILO-Instruction Processors, on which this thesis is based, many other papers have been produced during my time as a PhD student. The following is a comprehensive list of all my publications, covering several topics such as Clustered VLIW architectures, Vector Organizations for Cryptography, Processor Design based on Value content, and KILO-Instruction Processors.

CONFERENCES

1. *Performance and Power Evaluation of Clustered VLIW Processors with Wide Functional Units*. Miquel Pericàs, Eduard Ayguadé, Javier Zalamea, Josep Llosa and Mateo Valero. Proceeding of SAMOS'03 Workshop. Samos, 2003.
2. *Power-Performance Trade-Offs in Wide and Clustered VLIW Cores for numerical Computations*. Miquel Pericàs, Eduard Ayguadé, Javier Zalamea, Josep Llosa and Mateo Valero. Intl' Symposium on High Performance Computing (ISHPC-V). Tokyo, 2003.
3. *Scalable Distributed Register File*. Ruben Gonzalez, Adrian Cristal, Miquel Pericàs, Alex Veidenbaum and Mateo Valero. Workshop on Complexity Effective Design (WCED). June 2004
4. *An Optimized Front-End Physical Register File with Banking and Writeback Filtering*. Miquel Pericàs, Ruben Gonzalez, Adrian Cristal, Alex Veidenbaum and Mateo Valero. Workshop on Power Aware Computer Systems (PACS). December 2004.
5. *An Asymmetric Clustered Processor based on Value Content*. Ruben Gonzalez, Adrian Cristal, Alex Veidenbaum, Miquel Pericas and Mateo Valero. International Conference on Supercomputing. June 2005
6. *Exploiting Execution Locality with a Decoupled Kilo-Instruction Processor*. Miquel Pericàs, Ruben Gonzalez, Adrian Cristal, Daniel A. Jimenez and Mateo Valero. International Symposium on High Performance Computing (ISHPC-VI) 2005

7. *The Decoupled State/Execute Architecture*. Miquel Pericàs, Ruben Gonzalez, Adrian Cristal, Alex Veidenbaum and Mateo Valero. International Symposium on High Performance Computing (ISHPC-VI) 2005
8. *Chained In-Order/Out-of-Order DoubleCore Architecture*. Miquel Pericàs, Ruben Gonzalez, Adrian Cristal, Daniel A. Jimenez and Mateo Valero. International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD). 2005
9. *A Decoupled KILO-Instruction Processor*. Miquel Pericàs, Ruben Gonzalez, Adrian Cristal, Daniel A. Jimenez and Mateo Valero. International Symposium on High Performance Computer Architecture (HPCA-12). 2006
10. *A Flexible Heterogeneous Multi-Core Architecture*. Miquel Pericàs, Ruben Gonzalez, Francisco J. Cazorla, Adrian Cristal, Daniel A. Jimenez and Mateo Valero. Parallel Architecture and Compiler Techniques (PACT-2007). 2007
11. *Vectorized AES Core for high-throughput secure environments*. Miquel Pericàs, Ricardo Chaves, Georgi N. Gaydadjiev, Mateo Valero and Stamatis Vassiliadis. The Future of Computing - essays in memory of Stamatis Vassiliadis. 2007
12. *Vectorized AES Core for high-throughput secure environments*. Miquel Pericàs, Ricardo Chaves, Georgi N. Gaydadjiev, Mateo Valero and Stamatis Vassiliadis. 8th International Meeting High Performance Computing for Computational Science (VECPAR'08). 2008
13. *A two-level Load/Store Queue based on Execution Locality*. Miquel Pericas, Adrian Cristal, Ruben Gonzalez, Alex Veidenbaum, Daniel A. Jimenez and Mateo Valero. The 35th International Symposium on Computer Architecture (ISCA-35), Beijing, 2008

JOURNALS

1. *Power-Performance Trade-Offs in Wide and Clustered VLIW Cores for numerical Computations*. Miquel Pericàs, Eduard Ayguadé, Javier Zalamea, Josep Llosa and Mateo Valero. Lecture Notes on Computer Science Volume 2858/2003. pp. 113-126
2. *High-Performance Low-Power VLIW Cores for Numerical Computations*. Miquel Pericàs, Eduard Ayguadé, Javier Zalamea, Josep Llosa and Mateo Valero. International Journal of High Performance Computing and Networking 2004 - Vol. 1, No.4 pp. 171 - 179
3. *Power-Efficient VLIW Design using Clustering and Widening*. Miquel Pericàs, Eduard Ayguadé, Javier Zalamea, Josep Llosa and Mateo Valero. International Journal of Embedded Systems. Volume 31204 (to appear).

4. *Performance and Power Evaluation of Clustered VLIW Processors with Wide Functional Units*. Miquel Pericàs, Eduard Ayguadé, Javier Zalamea, Josep Llosa and Mateo Valero. Lecture Notes in Computer Science. Volume 3133/2004, 2004. pp. 88-97.
5. *Kilo-Instruction Processors: Overcoming the Memory Wall*. Adrian Cristal, Oliverio J. Santana, Francisco J. Cazorla, Marco Galluzzi, Tanausú Ramirez, Miquel Pericàs and Mateo Valero. IEEE MICRO. Vol 25, Nr 3. pp 48-57. May/June 2005
6. *An Optimized Front-End Physical Register File with Banking and Writeback Filtering*. Miquel Pericàs, Ruben Gonzalez, Adrian Cristal, Alex Veidenbaum and Mateo Valero. Lecture Notes in Computer Science. Volume 3471 / 2005. Pages 1-14.
7. *Exploiting Execution Locality with a Decoupled Kilo-Instruction Processor*. Miquel Pericàs, Ruben Gonzalez, Adrian Cristal, Daniel A. Jimenez and Mateo Valero. Lecture Notes in Computer Science. Volume 4759/2008. pp. 56-67
8. *The Decoupled State/Execute Architecture*. Miquel Pericàs, Ruben Gonzalez, Adrian Cristal, Alex Veidenbaum and Mateo Valero. Lecture Notes in Computer Science. Volume 4759/2008. pp. 68-78

TECHNICAL REPORTS

1. *Banked Front-End Register File*. Miquel Pericàs, Ruben Gonzalez, Adrian Cristal, Alex Veidenbaum and Mateo Valero. Technical Report. Reference: UPC-DAC-2004-35. October 2004
2. *A Flexible Heterogeneous Multi-Core Architecture*. Miquel Pericàs, Adrian Cristal, Francisco J. Cazorla, Ruben González, Daniel A. Jimenez, Mateo Valero. Technical Report. Reference: UPC-DAC-RR-CAP-2006-15. August 2006
3. *A Reconfigurable Heterogeneous Multi-Core Architecture*. Miquel Pericas, Adrian Cristal, Francisco J. Cazorla, Ruben González, Daniel A. Jimenez, Mateo Valero. Technical Report. Reference: UPC-DAC-RR-CAP-2007-1. February 2007

BIBLIOGRAPHY

- [1] A. Cristal, M. Valero, A. Gonzalez, and J. Llosa, "Large virtual ROBs by processor checkpointing," Tech. Rep., 2002, technical Report number UPC-DAC-2002-39. [Online]. Available: <http://www.ac.upc.edu/recerca/reports/DAC/2002/index,en.html>
- [2] A. Cristal, D. Ortega, J. Llosa, and M. Valero, "Out-of-order commit processors," in *Proc. of the 10th Intl. Symp. on High-Performance Computer Architecture*, 2004.
- [3] A. Cristal, J. Martinez, J. Llosa, and M. Valero, "Ephemeral registers with multicheckpointing," Tech. Rep., 2003, technical Report number UPC-DAC-2003-51, Departament d'Arquitectura de Computadors, Universitat Politecnica de Catalunya.
- [4] A. Cristal, O. J. Santana, J. F. Martinez, and M. Valero, "Toward kilo-instruction processors," *ACM Transactions on Architecture and Code Optimization (TACO)*, pp. 389 – 417, December 2004.
- [5] A. Cristal, O. J. Santana, F. Cazorla, M. Galluzzi, T. Ramirez, M. Pericàs, and M. Valero, "Kilo-Instruction Processors: Overcoming the Memory Wall," *IEEE Micro*, vol. 25(3), pp. 48–57, May/June 2005.
- [6] M. Tanenbaum, L. B. Valdes, E. Buehler, and N. B. Hannay, "Silicon n-p-n Grown Junction Transistors," *Journal of Applied Physics*, vol. 6, no. 26, pp. 686–692, 1955.
- [7] G. E. Moore, "Cramming More Components onto Integrated Circuits," *Electronics*, pp. 114–117, April 1965.
- [8] G. W. Boone, "Computng Systems CPU," U.S. Patent 3,757,306, September 4, 1973.
- [9] G. W. Boone and M. J. Cochran, "Variable Function Programmed Calculator," U.S. Patent 4,074,351, February 14, 1978.
- [10] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed. Morgan Kaufmann, 2002.
- [11] T. Austin, E. Larson, and D. Ernst, "Simplescalar: an infrastructure for computer system modeling," *IEEE Computer*, 2002.
- [12] COMPAQ, *Alpha Architecture Handbook*. Compaq Computer Corporation, December 1998.
- [13] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of Research and Development*, pp. 25–33, January 1967.
- [14] M. Pericàs, A. Cristal, R. Gonzalez, D. A. Jimenez, and M. Valero, "A Decoupled KILO-Instruction Processor," in *Proc. of the 12th Intl. Symp. on High Performance Computer Architecture*, February 2006.
- [15] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *Computer Architecture News*, March 1995.
- [16] K. C. Yeager, "The MIPS R10000 superscalar microprocessor," *IEEE Micro*, vol. 16, pp. 28–41, Apr. 1996.

- [17] S. Palacharla, N. Jouppi, and J. Smith, "Complexity-effective superscalar processors," in *Proc. of the 24th Intl. Symp. on Computer Architecture*, 1997.
- [18] T. Karkhanis and J. E. Smith, "A day in the life of a data cache miss," in *Proc. of the Workshop on Memory Performance Issues*, 2002.
- [19] S. T. Srinivasan and A. R. Lebeck, "Load latency tolerance in dynamically scheduled processors," *Journal of Instruction Level Parallelism*, vol. 1, pp. 1–24, 1999.
- [20] J. E. Smith and A. R. Pleszkun, "Implementation of Precise Interrupts in Pipelined Processors," *Proc. of the 12th Intl. Symp. on Computer Architecture*, pp. 34–44, 1985.
- [21] S. Rixner, W. J. Dally, B. Khailany, P. R. Mattson, U. J. Kapasi, and J. D. Owens, "Register organization for media processing," in *Proc. of the 6th Intl. Symp. on High Performance Computer Architecture*, 2000, pp. 375–386.
- [22] J. E. Smith, "Decoupled access/execute computer architectures," in *Proc. of the 9th annual Intl. Symp. on Computer Architecture*, 1982.
- [23] D. A. Jimenez and C. Lin, "Dynamic branch prediction with perceptrons," in *Proc. of the 7th Intl. Symp. on High Performance Computer Architecture*, January 2001, pp. 197–206.
- [24] M. V. Wilkes, "Slave memories and dynamic storage allocation," *IEEE Transactions on Electronic Computers*, pp. 270–271, April 1965.
- [25] A. J. Smith, "Cache memories," *ACM Computing Surveys*, vol. 14(3), pp. 473–530, 1982.
- [26] D. G. Perez, G. Mouchard, and O. Temam, "MicroLib: A case for the quantitative comparison of micro-architecture mechanisms," in *Proc. of the 37th Ann. Intl. Symp. on Microarchitecture*, 2004, pp. 43–54.
- [27] J. Tendler, S. Dodson, S. Fields, and B. S. H. Le, "Power4 System Microarchitecture," *IBM Journal of Research and Development*, vol. 46, no. 1, 2002.
- [28] Y. H. Song and M. Dubois, "Assisted execution," Tech. Rep., 1998, technical Report #CENG 98-25, Department of EE-Systems, University of Southern California.
- [29] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt, "Simultaneous subordinate microthreading (SSMT)," in *Proc. of the 26th Intl. Symp. on Computer Architecture*, 1999.
- [30] A. Roth and G. Sohi, "Speculative data-driven multithreading," in *Proc. of the 7th Intl. Symp. on High-Performance Computer Architecture*, 2001.
- [31] J. Dundas and T. Mudge, "Improving data cache performance by pre-executing instructions under a cache miss," in *Proc. of the 11th Intl. Conf. on Supercomputing*, 1997, pp. 68–75.
- [32] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *Proc. of the 9th Intl. Symp. on High Performance Computer Architecture*, 2003, pp. 129–140.
- [33] H. Akkary, R. Rajwar, and S. T. Srinivasan, "Checkpoint processing and recovery: Towards scalable large instruction window processors," in *Proc. of the 36th Intl. Symp. on Microarchitecture*, 2003.
- [34] J. Martinez, J. Renau, M. Huang, M. Prvulovic, and J. Torrellas, "Cherry: Checkpointed early Resource recycling in out-of-order Microprocessors," in *Proc. of the 35th Intl. Symp. on Microarchitecture*, 2002, pp. 3–14.

- [35] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg, "A large, fast instruction window for tolerating cache misses," in *Proc. of the 29th Intl. Symp. on Computer Architecture*, 2002.
- [36] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton, "Continual flow pipelines," in *Proc. of the 11th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2004.
- [37] A. Gonzalez, M. Valero, J. Gonzalez, and T. Monreal, "Virtual registers," in *Proc. of the 4th Intl. Conf. on High-Performance Computing*, 1997.
- [38] M. Moudgill, K. Pingali, and S. Vassiliadis, "Register Renaming and Dynamic Speculation: an alternative approach," in *Proc. of the 26th. Intl. Symp. on Microarchitecture*, 1993, pp. 202–213.
- [39] E. U. Cohler and J. E. Storer, "Functionally parallel architecture for array processors," *IEEE Computer*, vol. 14(9), pp. 22–36, September 1981.
- [40] R. Espasa and M. Valero, "Decoupled vector architectures," in *Proc. of the 2nd Intl. Symp. on High Performance Computer Architecture*, February 1996, pp. 281–290.
- [41] M. Pericàs, R. Gonzalez, F. J. Cazorla, A. Cristal, D. A. Jimenez, and M. Valero, "A Flexible Heterogeneous Multi-Core Architecture," in *Proc. of the 16th Intl. Conf on Parallel Architectures and Compiler Techniques*, September 2007, pp. 13–24.
- [42] R. D. Barnes, S. Ryoo, and W. mei W. Hwu, "Flea-Flicker Multipass Pipelining: An Alternative to the High-Power Out-of-Order Offense," in *Proc. of the 38th. Annual Intl. Symp. on Microarchitecture*, December 2005.
- [43] J.-L. Baer and T.-F. Chen, "An effective on-chip preloading scheme to reduce data access penalty," in *Proc. of the 1991 ACM/IEEE Conference on Supercomputing*, 1991, pp. 176–186.
- [44] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proc. of the 17th annual Intl. Symp. on Computer Architecture*, 1990, pp. 364–373.
- [45] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," in *Proc. of the fifth Intl. Conf. on Architectural support for Programming Languages and Operating Systems*, 1992, pp. 62–73.
- [46] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar processors," in *Proc. of the 22nd. Intl. Symp. on Computer Architecture*, June 1995.
- [47] M. Franklin and G. S. Sohi, "The expandable split window paradigm for exploiting fine-grain parallelsim," in *Proc. of the 19th Intl. Symp. on Computer Architecture*, 1992, pp. 58–67.
- [48] K. Sankaralingam, R. Nagarajan, R. McDonald, R. Desikan, S. Drolia, M. Govindan, P. Gratz, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, S. W. Keckler, and D. Burger, "Distributed Microarchitectural protocols in the TRIPS Prototype Processor," in *Proc. of the 39th Annual Intl. Symp. on Microarchitecture*, December 2006.
- [49] H. Zhou, "Dual-core execution: Building a highly scalable single-thread instruction window," in *Proc. of the 14th Intl. Conf. on Parallel Architectures and Compilation Techniques*, September 2005.

- [50] R. Kumar, V. Zyuban, and D. M. Tullsen, "Interconnection in multicore architectures: Understanding mechanisms, overheads, and scaling," in *Proc. of the 32nd Intl. Symp. on Computer Architecture*, June 2005.
- [51] W. J. Dally and B. Towles, "Route packets, not wires: On-chip interconnection networks," in *Proc. of the 38th Design Automation Conference*, June 2001.
- [52] J. Kim, C. Nicopoulos, D. Park, V. Narayanan, M. S. Yousif, and C. R. Das, "A gracefully degrading and energy-efficient modular router architecture for on-chip networks," in *Proc. of the 33rd Intl. Symp. on Computer Architecture*, June 2006.
- [53] A. Kumar, L.-S. Peh, P. Kundu, and N. K. Jha, "Express Virtual Channels: Towards the Ideal Interconnection Fabric," in *Proc. of the 34th Intl. Symp. on Computer Architecture*, June 2007.
- [54] A. Kumar, P. Kundu, A. Singh, L.-S. Peh, and N. Jha, "A 4.6Tbits/s 3.6GHz Single-cycle NoC Router with a Novel Switch Allocator in 65nm CMOS," in *International Conference on Computer Design (ICCD)*, October 2007.
- [55] J. Vera, F. J. Cazorla, A. Pajuelo, O. J. Santana, E. Fernandez, and M. Valero, "FAME: FAirly MEasuring Multithreaded Architectures," in *Proc. of the 16th Intl. Conf. on Parallel Architecture and Compilation Techniques*, September 2007.
- [56] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai, "The impact of performance asymmetry in emerging multicore architectures," in *Proc. of the 32nd. Intl. Symp. on Computer Architecture*, June 2005, pp. 506–517.
- [57] K. Luo, J. Gummaraju, and M. Franklin, "Balancing throughput and fairness in SMT processors," in *Proc. of the 2001 IEEE Intl Symp on Performance Analysis of Systems and Software*, November 2001.
- [58] Intel, "Introducing the 45nm Next-Generation Intel Core Microarchitecture," Tech. Rep., 2007, intel White Paper. [Online]. Available: <http://www.intel.com/technology/architecture-silicon/intel64/45nm-core2-whitepaper.pdf>
- [59] B. Sander, "Barcelona: AMD's Next-Generation Quad-Core Microprocessor," Tech. Rep., 2007, aMD White Paper. [Online]. Available: <http://developer.amd.com/assets/CART2007-Barcelona.pdf>
- [60] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden, "IBM POWER6 microarchitecture," *IBM Journal of Research and Development*, vol. 51 (6), November 2007.
- [61] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas, "Single-isa heterogeneous multicore architecture for multithreaded workload performance," in *Proc. of the 31st Intl. Symp. on Computer Architecture*, June 2004.
- [62] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez, "Accommodating workload diversity in chip multiprocessors via adaptive core fusion," in *Workshop on Complexity-Effective Design*, June 2006, pp. 10–21.
- [63] C. Kim, S. Sethumadhavan, M. S. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler, "Composable Lightweight Processors," in *Proc. of the 40th Annual Intl. Symp on Microarchitecture*, December 2007, pp. 381–394.

- [64] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukoton, "The Stanford Hydra CMP," *IEEE MICRO*, vol. 20(2), pp. 71–84, March/April 2000.
- [65] V. Krishnan and J. Torrellas, "A Chip-Multiprocessor Architecture with Speculative Multithreading," *IEEE Transaction on Computers*, vol. 48(9), pp. 866–880, September 1999.
- [66] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerma, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: A Many-Core x86 Architecture for Visual Computing," 2008.
- [67] M. Pericàs, A. Cristal, F. J. Cazorla, R. González, A. Veidenbaum, D. A. Jiménez, and M. Valero, "A Two-Level Load/Store Queue based on Execution Locality," in *Proc. of the 35th Intl. Symp. on Computer Architecture (ISCA-35)*, June 2008.
- [68] H. W. Cain and M. H. Lipasti, "Memory ordering: A value-based approach," in *Proc. of the 31st Intl. Symp. on Computer Architecture*, June 2004.
- [69] A. Roth, "Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization," in *Proc. of the 32nd Intl. Symp. on Computer Architecture*, June 2005.
- [70] B. H. Bloom, "Space/Time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13(7), pp. 422–426, July 1970.
- [71] G. Z. Chrysos and J. S. Emer, "Memory Dependence prediction using store sets," in *Proc. of the 25th Intl. Symp. on Computer Architecture*, June 1998, pp. 142–153.
- [72] M. Franklin and G. S. Sohi, "ARB: A Hardware Mechanism for Dynamic Reordering of Memory References," *IEEE Transactions on Computers*, vol. 45, pp. 552–571, May 1996.
- [73] C. K. Tang, "Cache Design in the Tightly Coupled Multiprocessor System," in *AFIPS Conference Proceedings, National Computer Conference*, June 1976, pp. 749–753.
- [74] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, "An Evaluation of Directory Schemes for Cache Coherence," in *Proc. of the 15th Annual Intl. Symp. on Computer Architecture*, May-June 1988, pp. 280–289.
- [75] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S. W. Keckler, "Scalable hardware memory disambiguation for high ILP processors," in *Proc. of the 36th Intl. Symp. on Microarchitecture*, 2003.
- [76] I. Park, C. L. Ooi, and T. N. Vijaykumar, "Reducing design complexity of the load/store queue," in *Proc. of the 36th Intl. Symp. on Microarchitecture*, 2003.
- [77] S. Sethumadhavan, F. Roesner, J. S. Emer, D. Burger, and S. W. Keckler, "Late-Binding: Enabling Unordered Load-Store Queues," in *Proc. of the 34th Intl. Symp. on Computer Architecture*, June 2007.
- [78] A. Moshovos and G. S. Sohi, "Streamlining Inter-operation Memory Communication via Data Dependence Prediction," in *Proc. of the 30th Intl. Symp. on Microarchitecture*, December 1997.
- [79] S. Subramaniam and G. H. Loh, "Fire-and-Forget: Load/Store Scheduling with No Store Queue at All," in *Proc. of the 39th Annual Intl. Symp. on Microarchitecture*, December 2006.
- [80] T. Sha, M. Martin, and A. Roth, "NoSQ: Store-Load Communication without a Store Queue," in *Proc of the 39th Annual Intl. Symp. on Microarchitecture*, December 2006, pp. 285–296.