General-Purpose Architectures for Media Processing and Database Applications

Parthasarathy Ranganathan

RICE

Thesis: Doctor of Philosophy

Electrical and Computer Engineering

Rice University, Houston, Texas (August 2000)

RICE UNIVERSITY

General-Purpose Architectures for Media Processing and Database Workloads

by

Parthasarathy Ranganathan

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

Houston, Texas

August, 2000

General-Purpose Architectures for Media Processing and Database Workloads

Parthasarathy Ranganathan

Abstract

Workloads on general-purpose computing systems have changed dramatically over the past few years, with greater emphasis on emerging compute-intensive applications such as media processing and databases. However, until recently, most high performance computing studies have primarily focused on scientific and engineering workloads, potentially leading to designs not suitable for these emerging workloads. This dissertation addresses this limitation. Our key contributions include (i) the first detailed quantitative simulation-based studies of the performance of media processing and database workloads on systems using state-of-the-art processors, and (ii) cost-effective architectural solutions targeted at achieving the higher performance requirements of future systems running these workloads.

The first part of the dissertation focuses on media processing workloads. We study the effectiveness of state-of-the-art features (techniques to extract instruction-level parallelism, media instruction-set extensions, software prefetching, and large caches). Our results identify two key trends: (i) media workloads on current general-purpose systems are primarily compute-bound and (ii) current trends towards devoting a large fraction of on-chip transistors (up to 80%) for caches can often be ineffective for media workloads. In response to these trends, we propose and evaluate a new cache organization, called *reconfigurable caches*. Reconfigurable caches allow the on-chip cache transistors to be dynamically divided into partitions that can be used for other activities (e.g., instruction memoization, application-controlled memory, and prefetching buffers), including optimizations that address the compute bottleneck. Our design of the reconfigurable cache requires relatively few modifications to existing cache structures and has small impact on cache access times.

The second part of the dissertation evaluates the performance of database workloads like online transaction processing and decision support system on shared-memory multiprocessor servers with state-of-the-art processors. Our main results show that the key performance-limiting characteristics of online transaction processing workloads are (i) large instruction footprints (leading to instruction cache misses) and (ii) frequent data communication (leading to cache-to-cache misses). We show that both these inefficiencies can be addressed with simple cost-effective optimizations. Additionally, our analysis of optimized memory consistency models with state-of-the-art processors suggest that the choice of the hardware consistency model of the system may not be a dominant factor for database workloads.

Acknowledgments

If I have seen farther than others, it is because I was standing on the shoulders of giants.

Albert Einstein, 1879-1955

This dissertation would not have been possible without the guidance, help, and support of several people.

- Sarita Adve, my primary advisor at Rice, has been a tremendous source of support – technical, professional, and personal – throughout my graduate school years . Her clarity of thought and focus, patience, enthusiasm, and concern are qualities I hope to emulate in the rest of my professional and personal life.
- Norm Jouppi, my advisor at Compaq Western Research Labs, has been a great source of inspiration since my early graduate days. His perspective and technical depth and the great balance between his personal and professional lives are qualities I am still trying to aspire to.
- I would also like to thank the other members of my thesis committee, Keith Copper, Willy Zwaenepoel, and Joe Cavallaro for all their encouragement and feedback. I am particularly grateful to Joe for his support during my first year of graduate school. Special thanks to Keith and Willy for serving on my Master's committee as well.
- I am also thankful to Luiz Barroso and Kourosh Gharachorloo for our productive collaborations at Digital research; the database portion of this dissertation was a result of these collaborations. I learnt a lot about database workloads (and a little bit about shooting hoops) during my interactions with them.
- I am also particularly thankful to my office-mate, collaborator, and friend for the last six years, Vijay Pai. I will cherish our fruitful interactions – the long hours of simulator development, our numerous discussions, and our inside jokes (on MG/1 queues and everything else under the sun).
- My graduate research has also been enriched by interactions with several other faculty members and researchers. In particular, I am thankful to

Behnaam Aazhang, Hazim Abdel-Shafi, Vikram Adve, Mohit Aron, Rich Baraniuk, Murthy Durbhakula, Keith Farkas, Jon Hall, Chris Hughes, Bob Jump, Praful Kaul, Jack Lo, Raghu Madhyastha, Vivek Pai, Sridhar Rajagopal, and Steve Wallach for all the fruitful interactions over the last several years. Thanks are also due to all the other people, far too many to name, in the Rice ECE/CS community and the WRL community, for making my graduate experience memorable.

- To all my friends at Rice, I owe a tremendous debt of gratitude for making my stay at Rice so enjoyable. Special thanks to Karthick (my roommate of the last six years), Suman, Mahesh, Srikrishna, Krishna Kiran, and Ram Rajamony (my other roommates over the years), Dinesh (my badminton partner), Priya, Ranjani, Kishore, and Adria for all the support over the years.
- I am thankful to my brother Krishnan, sister-in-law Deepa, my uncle Prof. Raghunathan, and all my other relatives for all their love, support, and encouragement over the years. Finally, to my dear parents, Prof. Ranganathan and Vijayalakshmi Ranganathan, this dissertation is dedicated to you. Without your immeasurable love, encouragement, and pride in me, I would not be who I am today. Thank you.

Parts of the work presented in this dissertation were published in the Eighth International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 1998 [99], the 26th International Symposium on Computer Architecture (ISCA), 1999 [97], and the 27th International Symposium on Computer Architecture (ISCA), 2000 [98].

The research presented in this dissertation was supported by IBM Corporation, Intel Corporation, the National Science Foundation under Grant No.CCR-9410457, CCR-9502500, CDA-9502791, CDA-9617383, CCR-0096126, the Texas Advanced Technology Program under Grants No. 003604-016 and 003604-025, and the Alfred Sloan Research Foundation. Parts of this work were also supported by a Rice fellowship and a Lodieska Stockbridge Vaughan Fellowship. The database portion of this work was supported partly by an internship at Digital Western Research Labs.

Contents

Illustrations

Tables

Chapter 1

Introduction

1.1 Motivation

1.1.1 Key Trends

The digitization of traditionally analog representations of information and the need to process large amounts of digital information has led to an increasing use of computing systems in day-to-day activities. Microprocessor-based systems are now being used in a wider range of rapidly-growing markets than ever before – including high-end servers, mainstream PCs, low-cost PCs, mobile PCs, communication systems, and embedded information appliances. Two key developments in the past few years are likely to further accelerate these trends. First, the availability of greater computing power at lower costs and the emerging convergence of computing, communication, and consumer products have held out the promise of *universal computing*. Second, recent advances in communication technologies and the growth of the internet have made the prospect of *universal connectivity* realizable in the near-future.

In response to these advances in computing and communication, computing workloads have changed dramatically over the past several years. Several new aggressive applications have recently emerged affecting areas as diverse as entertainment, lifestyles, human-computer interfaces, business, and personal productivity. Some of these emerging workloads that are likely to dominate future computing cycles are listed below.

- *Entertainment and lifestyles.* These include applications such as collaborative tele-conferencing, tele-medicine, distance learning, interactive games, high-resolution digital television (HDTV), and advanced content-generation methods (e.g., photo-realistic rendering, real-time video authoring, and 3-dimensional visualization).

- *Advanced human-computer interfaces.* These include applications such as spontaneous speech recognition, speech synthesis, image recognition, handwriting and document recognition, and immersive and interactive virtual environments (e.g., immersadesks and virtual kiosks).
- *Business and personal productivity.* These include applications such as e-commerce, data mining, digital libraries, desktop publishing, security and encryption, and advanced communication technologies such as 3-G wireless.

Overall, these emerging trends indicate a growing shift from conventional application scenarios based on *scientific and engineering workloads* to emerging application scenarios based on *media processing and database workloads*. Consistent with this shift in computing workloads, it is important to ensure that the underlying computer architectures are also suitably defined to perform well for emerging workloads. In the rest of this thesis, we will focus on the implications of emerging media processing and database workloads on future computer architectures.

1.1.2 Emerging Media Processing and Database Workloads

Media processing applications deal with the computing associated with the creation, encoding/decoding, processing, display, and communication of digital multimedia information such as images, audio, video, and graphics. Such applications have been conjectured to dominate up to 90% of the total future computing cycles [104]. While current media processing systems use general-purpose processors accelerated with DSP or application-specific processors, the cost, performance, and flexibility advantages offered by general-purpose processors argue for their increasing use with these workloads [31, 38, 69]. A recent step in this direction has been the recently-announced media instruction-set architecture (ISA) extensions for most commodity general-purpose processors (e.g., 3DNow! [87], AltiVec [95], MAX [68], MDMX and MIPSV [58], MMX [93], MVI [19], VIS [123]).

At the high-end server market, *database and information processing applications* such as online transaction processing (OLTP) [121] and decision support systems (DSS) [122] have emerged as the largest and fastest growing market segment for shared-memory servers [115]. While database workloads were once exclusively the domain of main frame systems, the cost advantages of using commodity components with shared-memory servers have led to a greater use of these general-purpose systems for database workloads.

Several of these media processing and database workloads require significantly higher computing performance than what is available in current systems. However, until recently, most high-performance computing studies have primarily focused on scientific and engineering workloads, leading to design decisions optimized for these workloads. For example, commodity processors are primarily optimized to perform well on the SPEC benchmark suite [113], and system designs are focused on scientific and engineering benchmarks such as STREAMS [76] and SPLASH-2 [128]. It is not clear if these design decisions are suitable even with emerging media processing and database workloads. Given the growing importance of these emerging workloads, it becomes particularly important to re-evaluate key system design decisions in the context of these applications.

Unfortunately, in spite of this growing awareness of the need to refine architectures in the context of the growing importance of media processing and database workloads, there has been very little quantitative understanding of the behavior of these workloads prior to our work. The next section discusses the reasons for this mismatch between emerging workloads likely to be used in future systems and the workloads used in characterization studies.

1.2 Key Challenges

Key challenges that have limited the detailed study of these emerging media processing and database workloads have included (i) the lack of availability and/or restrictions in studying large and proprietary representative benchmarks, and (ii) the complexity and cost in setting up, studying, and simulating these systems.

Lack of Benchmarks. With media processing, a major challenge is the large number of application classes in this domain (e.g., image, video, audio, speech, communication, graphics, etc.) and the absence of any standardized representative benchmark suites and input data sets.[1] Additionally, the performance of media processing applications is often measured by qualitative perception metrics which makes it hard to consistently compare vendor-published results for different commercial architectures. Similarly, for database workloads, the proprietary nature of the software makes them hard to study. For example, software licenses for commercial databases typically prevent the publication of performance information and do not provide access to the source code. Additionally, both with media processing and database workloads, the competitive nature of the market leads to software changes at a very rapid pace, often obsoleting benchmarks even before they become accepted.

Cost and Complexity of Studies. One of the other major problems in studying emerging workloads is the complexity and cost involved in setting up and studying these systems. For example, setting up a audit-quality hardware system for database workloads can typically take hundreds of thousands of dollars. Scaling these workloads to more easy-to-study configurations requires making simplifications based on a deep understanding of the workloads and their interactions with the system. Additionally, commercial media processing and database workloads can be inherently complex with complex operating system and I/O interactions (with database workloads), hand-tuned assembly routines and the need to

[1] In his keynote speech at the Texas Instruments DSPS Fest, 1998, Dr. Will Strauss, President of Forward Concepts, compared the definition of multimedia processing to that for "true love" – everyone knows what it is, but nobody can define it exactly!

run multiple tightly synchronized applications at the same time (often the case with media processing workloads). Simulating such systems can often be very hard.

Given all these challenges in studying emerging applications, most uniprocessor studies in the architecture community tend to focus on the SPEC technical benchmark suite [113], while most multiprocessor studies focus mainly on scientific and engineering benchmarks such as STREAMS [76] and SPLASH-2 [128]. These trends have led to

- A lack of detailed quantitative understanding of the behavior of emerging workloads on state-of-the-art systems.
- A lack of clear consensus on the key performance challenges and best architectural solutions for future systems.

Lack of understanding of current systems. The lack of detailed quantitative understanding of database and media processing workloads has resulted in a number of fundamental unanswered questions for these workloads. What are the important performance bottlenecks for these applications? Are system features designed to perform well for scientific and engineering workloads equally relevant in the context of these emerging applications? For example, optimizing systems for the SPEC/SPLASH workloads has resulted in the emergence of aggressive out-of-order processors with features such as 32- and 64-bit data types, complex interlocking and speculation hardware to exploit instruction-level parallelism (ILP). Given the qualitative differences between database and media processing workloads and the SPEC/SPLASH workloads [31, 67, 75], how effective are these complex ILP-centric architectures for the emerging workloads? Are other current architectural techniques such as ISA extensions, large system caches, and software prefetching effective for these workloads?

Designing high-performance future systems. Similarly, there are a number of open research questions on designing new architectures for these workloads. Specifically, what are the key future performance bottlenecks for emerging workloads? How do we design architectures that can achieve the high future computing requirements of emerging workloads?

How can we benefit from emerging technologies that allow more transistors and greater on-chip system integration (e.g., reconfigurable logic on-chip)?

1.3 Contributions

This dissertation presents the first work to use detailed simulation to characterize and improve the performance of several emerging media processing and database workloads on state-of-the-art general-purpose systems.

We study a representative set of several realistic commercial media processing and database workloads. Our benchmarks were identified through collaboration and consultation with application developers in these areas and form key components of several of the application scenarios discussed earlier. The benchmarks we study include media processing workloads such as image processing, video processing, speech coding, audio coding, speech recognition and synthesis, as well as database workloads such as online transaction processing and decision-support systems.

We study these benchmarks on a wide variety of general-purpose architectural configurations using detailed simulation. For our experiments, we mainly use the detailed RSIM simulator [90] that was developed at Rice as part of this research. In addition, wherever needed, we complement our simulation experiments with the use of performance monitoring (with performance counters) and application profiling on currently-available real systems for tuning and scaling our workloads and for verifying the results of our simulations.

Our key high-level contributions include:

(i) The first detailed quantitative simulation-based studies of the performance of media processing and database workloads on systems using state-of-the-art processors.

(ii) Cost-effective architectural solutions targeted at achieving the higher performance requirements of future systems running media processing and database workloads.

Below, we present a brief summary of our specific results.

1.3.1 Media Processing Workloads

The first part of the dissertation focuses on media processing workloads. We use detailed simulation to study several representative image, video, speech, and audio coding applications, and speech recognition and synthesis applications.

We first focus on an important easy-to-understand sub-class of media processing, namely, image and video processing, and provide a detailed quantitative characterization of the effectiveness of state-of-the-art system features. Specifically, we study techniques to extract instruction-level parallelism (ILP), media instruction-set extensions, software prefetching, and large caches. Our results indicate that, contrary to previous qualitative perceptions of these workloads [31], complex ILP-centric architectures can be effective in improving performance even with media workloads (by factors of 2.3 to 4.2). Our results also quantify the performance benefits from instruction set extensions for media processing and identify key limitations that can potentially be addressed to achieve greater performance. The strong compute-centric performance benefits from ILP features and VIS extensions shift the bottleneck to the memory sub-system for several of these benchmarks. Software prefetching, however, is effective in addressing this memory stall time, improving performance by factors of 1.4 to 2.5 in the benchmarks where memory is a significant problem.

Focusing on future designs, we build on two key trends from our analysis: (i) media workloads on current general-purpose systems are primarily compute-bound and (ii) current trends towards devoting a large fraction of on-chip transistors (up to 80%) for caches can often be ineffective for media workloads because of their streaming data access patterns and large working sets. In response to these results, we propose and evaluate a new cache organization, called *reconfigurable caches*, that allows the on-chip cache transistors to be dynamically divided into partitions that can be used for other processor activities (e.g., instruction memoization, application-controlled memory, and prefetching buffers). Our design of the reconfigurable cache requires very few modifications to existing caches and analysis using a modification of the CACTI analytical model [102, 127] shows only small impact on cache access times. Finally, as one representative use of reconfigurable caches, we study instruction reuse for media processing workloads on a state-of-the-art general-purpose configuration. Our results show that this use can achieve good performance improvements by reusing otherwise under-utilized memory resources to address the computation bottleneck.

1.3.2 Database Workloads

The next part of the dissertation examines the performance of database workloads on shared-memory servers with state-of-the-art processors. We use detailed simulation to study both online-transaction processing (OLTP) and decision-support systems (DSS) running on the commercial Oracle database engine (version 7.3.2).

Our results show that current ILP-centric architectures are also very effective for database workloads. The combination of out-of-order execution and multiple instruction issue is effective in improving the performance of all our database workloads, providing factors of 1.5 to 2.6 performance improvements over an in-order single issue processor for OLTP and DSS. Additionally, our analysis of optimized memory consistency models enabled by speculative ILP techniques suggest that the choice of the hardware consistency model of the system may not be a dominant factor for database workloads.

The more challenging online transaction processing workloads exhibit radically different behavior compared to scientific/engineering applications. The key performance-limiting characteristics of these workloads are (i) large instruction footprints (leading to instruction cache misses) and (ii) frequent data communication (leading to cache-to-cache misses in the shared-memory system). We show that both these inefficiencies could be addressed with simple cost-effective optimizations. Stream buffers help alleviate instruction misses (approaching a perfect instruction cache within 15%). A combination of prefetching and producer-initiated communication directives address the communication misses. To the best of our knowledge, we are not aware of any database server at the time of our work that included any of these optimizations other than software prefetching.

1.4 Thesis Organization

The rest of the dissertation is organized as follows. Chapter 2 focuses on media processing benchmarks and characterizes the effectiveness of current processor and memory system features for these workloads. Chapter 3 builds on the insights from our analysis to propose and evaluate reconfigurable cache architectures to improve the performance of media processing workloads. Chapter 4 focuses on database workloads running on shared-memory servers. We present results characterizing the performance of database workloads and discuss cost-effective architectural techniques to address the main performance limitations. Chapter 5 summarizes this dissertation and Chapter 6 discusses future directions of research.

Chapter 2

Performance Evaluation of Media Processing

This chapter performs a detailed quantitative evaluation of the performance of media processing applications to understand their behavior on state-of-the-art general-purpose systems. We focus on an easy-to-understand, yet important class of media processing workloads, namely image and video processing, and attempt to cover the spectrum of the key tasks in this class. We use detailed simulation to study a variety of general-purpose architectural configurations to analyze the effectiveness of (i) state-of-the-art *processor features* including techniques to extract instruction-level parallelism (ILP) and instruction-set extensions for media processing, and (ii) current *memory system features* including software prefetching and support for large caches.

Section 2.1 describes the workloads that we study. Sections 2.2 and 2.3 describe the simulated architectures and the simulation methodology. Sections 2.4 to 2.6 discuss the performance benefits of current aggressive ILP techniques and media ISA extensions for image and video processing workloads. Sections 2.7 and 2.8 consider the impact of larger cache sizes and software prefetching on the performance of image and video applications. Section 2.9 discusses related work relevant to the research presented in this section. Section 2.10 summarizes the chapter.

2.1 Applications Studied

The benchmarks we study cover the spectrum of key tasks in image and video processing workloads and were identified through consultation and collaboration with application developers in these areas. These benchmarks form significant components of many current and future real-world workloads such as collaborative teleconferencing, scene-visualization, distance learning, streaming video across the internet, digital broadcasting, real-time flight imaging and radar sensing, content-based storage and retrieval, online video cataloging, and medical tomography [49]. Emerging standards such as JPEG2000 and MPEG-4 are likely to build on a number of components of our benchmark suite.

Table 2.1 summarizes the 12 benchmarks that we use in this chapter, and is divided into image processing (Section 2.1), image source coding (Section 2.1), and video source coding (Section 2.1). These benchmarks are similar to some of the benchmarks used in the image and video parts of the Intel Media Benchmark (described at the Intel web site) and the UCLA MediaBench [65]. (We did not use the Intel Media Benchmark or the UCLA MediaBench directly because the former does not provide source code and the latter does not include image processing applications.)

All the image benchmarks are run with 1024x640 pixel 3-band (i.e., channel) input images obtained from the Intel Media Benchmark. The video benchmarks are run with the *mei16v2* test bit stream from the MPEG Software Simulation Group that operates on 352x240 sized 3-band images. We did not study larger (full-screen) sizes because they were not readily available and would have required impractical simulation time.

Image Processing

Our image processing benchmarks are taken from the Sun VIS Software Development Kit (VSDK) [79], which includes 14 image processing kernels. These kernels include common image processing tasks such as one-band and three-band (i.e., channel) alpha blending (used in image compositing), single-limit and double-limit thresholding (used in chroma-keying, image masking, and blue screening), and functions such as general and separable convolution, copying, inversion, addition, dot product, and scaling (used in the core of many image processing codes like blurring, sharpening, edge detection, embossing, etc.). We study all 14 of the VSDK kernels, but due to space constraints, we report results for only 6 representative benchmarks (*addition, blend, conv, dotprod, scaling, and thresh*).

Image processing

Addition	Addition of two images (*sf16.ppm*, *rose16.ppm*) using mean of two pixel values
Blend	Alpha blending of two images (*sf16.ppm*, *rose16.ppm*) with another alpha image (*winter16.ppm*); the operation performed is $dst = alpha \times src1 + (255 - alpha) \times src2$.
Conv	General 3x3 image convolution of an image (*sf16.ppm*). The operation performed includes a saturation summation of 9 product terms. Each term corresponds to multiplying the pixel values in a moving 3x3 window across the image dimensions with the values of a 3x3 kernel matrix.
Dotprod	16x16 dot product of a randomly-initialized 1048576-element linear array
Scaling	Linear image scaling of an image (*sf16.ppm*)
Thresh	Double-limit thresholding of an image (*sf16.ppm*). If the pixel band value falls within the low and high values for that band, the destination is set to the map value for that band; otherwise, the destination is set to be the same as the source pixel value.

Image source coding

Cjpeg	JPEG progressive encoding (*rose16.ppm*)
Djpeg	JPEG progressive decoding (*rose16.jpg*)
Cjpeg-np	JPEG non-progressive encoding (*rose16.ppm*)
Djpeg-np	JPEG non-progressive decoding (*rose16.jpg*)

Video source coding

Mpeg-enc	MPEG2 encoding of 4 frames (I-B-B-P frames) of the *mei16v2rec* bit stream. Properties of the bit stream include frame rate of 30fps, bit rate of 5Mbps at the Main profile@Main level configuration. The image is 352x240 pixels in the 4:2:0 YUV chroma format, and is scaled to a 704x480 display. The quantization tables and the motion estimation search parameters are set to the default parameters specified by the MPEG group.
Mpeg-dec	MPEG2 decoding of the *mei16v2rec* video bit stream into separate YUV components.

Table 2.1 Image and video processing benchmarks used for performance characterization.

Image Source Coding

We focus on the Joint Photography Experts Group (JPEG) standard [50] and study the performance of the Release 6(a) codec (encoder/decoder) from the Independent JPEG Group. We study two different commonly used codecs specified in the standard, a progressive JPEG codec (*cjpeg* encoder and *djpeg* decoder), and a non-progressive JPEG codec (*cjpeg-np* encoder and *djpeg-np* decoder).

The JPEG encoding process consists of a number of phases many of which exploit properties of the human visual system to reduce the number of bits required to specify the image. First, the *color conversion* and *chroma-decimation* phases convert the source image from a 24-bit RGB representation domain to a 12 bit 4:2:0 YUV representation. Next, a linear *DCT image transform* phase converts the image into the frequency domain. The *quantization* phase then scales the frequency domain values by a quantization value (either constant or variable). The *zig-zag scanning* and *variable-length (Huffman) coding* phases then reorder the resulting data into streams of bits and encode them as a stream of variable-length symbols based on statistical analysis of the frequency of symbols.

Progressive image compression uses a compression algorithm that performs multiple Huffman coding passes on the image to encode it as multiple scans of increasing picture quality (leading to the perception of gradual focusing of images seen on many web pages).

The decoding process performs the inverse of the operations for the encoding process in the reverse order to obtain the original image from the compressed image.

Video Source Coding

We focus on the Motion Picture Experts Group-2 (MPEG2) video coding standard [40], and study the performance of the version 1.1 codec from the MPEG Software Simulation Group.

The first part of the video compression process consists of spatial compression similar to that described for JPEG and includes the color conversion, chroma decimation, frequency transformation, quantization, zig-zag coding, and run-length coding phases.

Additionally, MPEG2 has an inter-frame predictive-compression *motion-estimation* phase that uses difference vectors to encode temporal redundancy between macroblocks in a frame and macroblocks in the following and preceding frames. Motion estimation is the most compute-intensive part of *mpeg-encode*.

The video decompression process performs the inverse of the various encode operations in reverse order to get the decoded bit stream from the input compressed video. The *mei16v2* bit stream is already in the YUV format, and consequently, our MPEG simulations do not go through the color conversion phase discussed earlier.

2.2 Simulated Architectures

2.2.1 Processor and Memory System

We study two processor models – an in-order processor model (similar to the Compaq Alpha 21164, Intel Pentium, and Sun UltraSPARC-II processors) and an out-of-order processor model (similar to the Compaq Alpha 21264, HP PA8000, IBM PowerPC, Intel Pentium Pro, and MIPS R10000 processors). Both the processor models support non-blocking loads and stores. For the experiments with software prefetching, the processor models provide support for software-controlled non-binding prefetches into the first-level cache.

The base system uses a 64KB two-way associative first-level (L1) write-back cache and a 128KB 4-way associative second-level (L2) write-back cache. Section 2.7 discusses the impact of varying the cache sizes. All the caches are non-blocking and allow support for multiple outstanding misses. At each cache, 12 miss status holding registers (MSHRs) reserve space for outstanding cache misses and combine a maximum of 8 multiple requests to the same cache line.

Tables 2.2 and 2.3 summarizes the parameters used for the processor and memory subsystems. When studying the performance of a 1-way issue processor, we scale the number of functional units to one of each type. The functional unit latencies are chosen based on

Processor speed	1 GHz
Issue width	4-way
Instruction window size	64
Memory queue size	32
Branch prediction	
Bimodal agree predictor size	2K
Return-address stack size	32
Taken branches per cycle	1
Simultaneous speculated branches	16
Functional unit counts	
Integer arithmetic units	2
Floating-point units	2
Address generation units	2
VIS multipliers	1
VIS adders	1
Functional unit latencies (cycles)	
Default integer/address generation	1/1
Integer multiply/divide	7/12
Default floating point	4
FP moves/converts/divides	4/4/12
Default VIS	1
VIS 8-bit loads/multiply/pdist	1/3/3

Table 2.2 Default processor parameters for media processing workload characterization.

the Alpha 21264 processor. All functional units are fully pipelined except the floating-point divide (non-pipelined).

2.2.2 VIS Media ISA Extensions

The VIS media ISA extensions to the SPARC V9 architecture are a set of instructions targeted at accelerating media processing [60, 123]. Both our in-order and out-of-order processor models include support for VIS.

The VIS extensions define the packed byte, packed word and packed double data types which allow concurrent operations on eight bytes, four words (16-bits each) or two double

Cache line size	64 bytes
L1 data cache size (on-chip)	64 KB
L1 data cache associativity	2-way
L1 data cache request ports	2
L1 data cache hit time	2 ns
Number of L1 MSHRs	12
L2 cache size (off-chip)	128K
L2 cache associativity	4-way
L2 request ports	1
L2 hit time (pipelined)	20 ns
Number of L2 MSHRs	12
Max. outstanding misses per MSHR	8
Total memory latency for L2 misses	100 ns
Memory interleaving	4-way

Table 2.3 Default system parameters for media processing workload characterization.

words of fixed-point data in a 64-bit register. These data types allow VIS instructions to exploit single-instruction-multiple-data (SIMD) parallelism at the subword level. Most of the VIS instructions operate on packed words or packed doubles; loads, stores, and `pdist` instructions operate on packed bytes. Many of the VIS instructions make implicit assumptions about rounding and the number of significant bits in the fixed-point data. Hence, while using VIS instructions, it is important to ensure that their use does not lead to incorrect outputs.

In the rest of this section, we provide a short overview of the VIS instructions (summarized in Table 2.4).

Packed arithmetic and logical operations. The packed arithmetic VIS instructions allow SIMD-style parallelism to be exploited for add, subtract, and multiply instructions. To minimize implementation complexity, VIS uses a pipelined series of two 8x16 multiplies and one add instruction to emulate packed 16x16-bit multiplication. The VIS logical instructions allow logical operations on the floating-point data path.

Packed arithmetic and logical operations
Packed addition
Packed subtraction
Packed multiplication
Logical operations
Subword rearrangement and realignment
Data packing and expansion
Data merging
Data alignment
Partitioned compares and edge operations
Partitioned compares
Mask generation for edge effects
Memory-related operations
Partial stores
Short loads and stores
Blocked loads and stores
Special-purpose operations
Pixel distance computation
Array address conversion for data reuse
Access to the graphics status register

Table 2.4 Classification of VIS instructions.

Subword rearrangement and alignment. To facilitate conversion between different data types, VIS supports subword rearrangement and alignment using pack, expand, merge (interleave), and align instructions. The subword rearrangement instructions also include support for implicitly handling saturation arithmetic (limiting data values to the minimum or maximum instead of the default wrap-around).

Partitioned compares and edge operations. For branches, VIS supports a partitioned compare that performs four 16-bit or two 32-bit compares in parallel to produce a mask that can be used in subsequent instructions. VIS also supports the edge instruction to generate masks for partial stores that can eliminate special branch code to handle boundary conditions in media processing applications.

Memory-related operations. For memory instructions, VIS supports partial stores that selectively write to parts of the 64-bit output based on an input mask. Short loads and stores transfer 1 or 2 bytes of memory to the register file. Blocked loads and stores transfer 64 bytes of data between memory and a group of eight consecutive VIS registers without causing allocations in the cache.

Special-purpose operations. The pixel distance computation (`pdist`) instruction is primarily targeted at motion estimation and computes the sum of the absolute differences between corresponding 8-bit components in two packed bytes. The *array* instruction is mainly targeted at 3D graphics rendering applications and converts 3D fixed-point coordinates into a blocked byte address that allows for greater cache reuse. VIS also defines instructions to manipulate the graphics status register, a special-purpose register that stores additional data for various media instructions.

Overall, the functionality discussed above for VIS is similar to that of fixed-point media ISA extensions in other general purpose processors (e.g., MAX [68], MMX [93], MVI [19], MDMX [58], AltiVec [95]). The various ISA extensions mainly differ in the number, types, and latencies of the individual instructions (e.g., MMX implements direct support for 16x16 multiply), whether they are implemented in the integer or floating-point data path, and in the width of the data path. The most different ISA extension, the proposed PowerPC AltiVec ISA, adds support for a separate 128-bit vector multimedia unit in the processor. Though we do not consider floating-point media ISA extensions (such as Intel SSE [48], 3DNow! [30], MIPS V [58, 24], and AltiVec [95]) in this work, they exploit similar support for packed data types, subword rearrangement, and special-purpose operations to achieve their performance benefits.

Our VIS implementation is closely modeled after the UltraSPARC-II implementation and operates on the floating-point register file with latencies comparable to the UltraSPARC-II [123] (Table 2.2). Note that our VIS multiplier has a lower latency compared to the integer (64-bit) multiplier because it operates on 16-bit data. The increase in chip area associated with the VIS instructions was estimated to be less than 3% for the UltraSPARC-II [60].

2.3 Simulation Methodology

2.3.1 Simulation Environment

We use the RSIM simulator [90] to simulate the in-order and out-of-order processors described in Section 2.2.1. RSIM is a user-level execution-driven simulator that models the processor pipeline and memory hierarchy in detail including contention for all resources.

The processor microarchitecture modeled by RSIM is closest to the MIPS R10000 [80] and is illustrated in Figure 2.1. In particular, RSIM models the R10000's *active list* (which holds the currently active instructions, corresponding to the *reorder buffer* or *instruction window* of other processors), *register map table* (which holds the mapping from logical to physical registers), and *shadow mappers* (which store register map table information on branch prediction to allow single-cycle state recovery on mispredictions). The pipeline parallels the *Fetch, Decode, Issue, Execute,* and *Complete* stages of the dynamically scheduled R10000 pipeline. Instructions are *retired* (i.e. *graduated, committed,* or *removed from the active list*) after passing through this pipeline. In the base out-of-order processor model, instructions are fetched, decoded, and retired in program order; however, instructions can issue, execute, and complete out-of-order. The in-order model used in our experiments is based on the above out-of-order model with the aggressive features for out-of-order execution turned off.

Since RSIM is a user-level simulator, we do not model system effects like operating system interactions and I/O latencies. To assess the impact of not modeling system level code, we profiled the benchmarks on an UltraSPARC-II-based Sun Enterprise server. We found that the time spent on operating system kernel calls is less than 2% on all the benchmarks. The time spent on I/O is less than 15% on all the benchmarks except *mpeg-dec*. This benchmark experiences an inflated I/O component (45%) because of its high frequency of

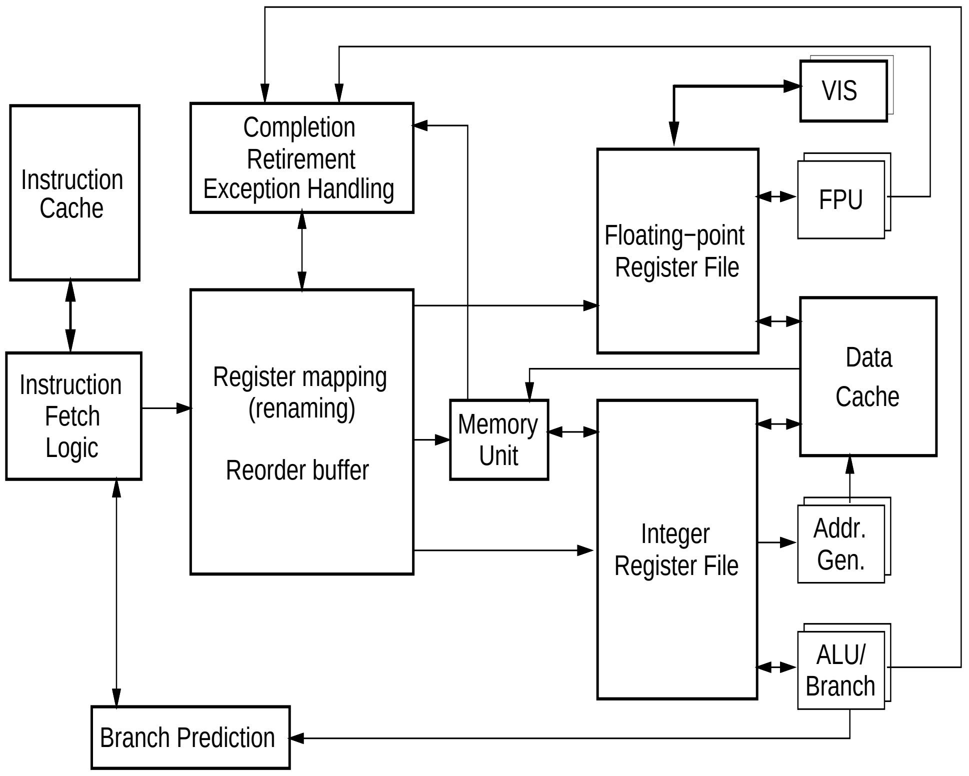


Figure 2.1 Default organization of out-of-order processor model.

file writes. However, in a typical system, these writes would be handled by a graphics accelerator, significantly reducing this component. Since our applications have small instruction footprints, our simulations assume all instructions hit in the instruction cache.

All the applications[2] were compiled with the SPARC SC4.2 compiler with the *-xO4 -xtarget=ultra1/170 -xarch=v8plusa -dalign* options to produce optimized code for the in-order UltraSPARC processor.

[2]We changed the 14 image processing kernels from the Sun VSDK to skew the starting addresses of concurrent array accesses and unroll small innermost loops. This reduced cache conflicts and branch mispredictions leading to 1.2X to 6.7X performance benefits. To facilitate modifying the applications for VIS, we replaced some of the key routines in the JPEG and MPEG applications with equivalent routines from the Sun *MediaLib* library.

2.3.2 VIS Usage Methodology

We are not aware of any compiler that automatically modifies media processing applications to use media ISA extensions. For our experiments studying the impact of VIS, we manually modified our benchmarks to use VIS instructions based on the methodology detailed below.

We profiled the applications to identify key procedures and manually examined these procedures for loops that satisfied the following three conditions: (1) The loop body should have no loop-carried dependences or control dependences that cannot be converted to data dependences (other than the loop branch). (2) The key computation in the loop body must be replaceable with a set of equivalent fixed-point VIS instructions. The loss in accuracy in this stage, if any, should be visually imperceptible. (3) The potential benefit from VIS should be more than the overhead of adding VIS; VIS overhead can result from subword rearrangement instructions to convert between packed data types or from alignment-related instructions.

For loops that satisfied the above criteria, we strip-mined or unrolled the loop to isolate multiple iterations of the loop body that we then replaced with equivalent VIS instructions. We used the inline assembly-code macros from the Sun VSDK for the VIS instructions. This minimizes code perturbation and allows the use of regular compiler optimizations. As before, the applications were compiled with the SPARC SC4.2 compiler with the *-xO4 -xtarget=ultra1/170 -xarch=v8plusa -dalign* options to produce optimized code for the in-order UltraSPARC processor. Wherever possible, we tried to use procedures available from the Sun VSDK Kit and the SUN MediaLib library routines that were already optimized for VIS.

Our benchmarks use all the VIS instructions except for the array and blocked load/store instructions. Array instructions are targeted at 3D array accesses in graphics loops, and are not applicable for our applications. Blocked loads and stores are primarily targeted at transfers of large blocks of data between buffers without affecting the cache (e.g., in operating system buffer management, networking, memory-mapped I/O). We did not use these instructions since the Sun VSDK does not provide inline assembly-code macros to support them. The alternative of hand-coded assembly could result in lower performance since it is hard to emulate the compiler optimizations associated with modern superscalar processors by hand [103]. Note that both the array and blocked load/store instructions are unique to VIS and are not supported by other general-purpose ISA extensions.

2.3.3 Software Prefetching Algorithm

We studied the applicability of software prefetching for the benchmarks where the cache miss stall time is a significant component of the total execution time ($>$20%). We identified the memory accesses that dominate the cache miss stall time, and inserted prefetches by hand (since we did not have access to a C compiler that implements software prefetching) for these accesses.

We followed the well known software prefetching compiler algorithm developed by Mowry et al. [83]. The prefetch algorithm handles both affine and indirect addresses to identify and schedule prefetches for possible cache misses. The algorithm is loop-based, and consists of an analysis phase and a scheduling phase. The *analysis phase* identifies the accesses that do not exhibit locality for the given cache parameters, and for which prefetches need to be inserted. The *scheduling phase* follows the analysis phase and uses loop peeling, unrolling, and strip-mining to insert prefetches only for the accesses that are expected to cause cache misses. Accesses to the same cache line are grouped into equivalence classes, and an exclusive prefetch is used to obtain ownership along with the data for the line if at least one member of the equivalence class is a write. The inner-most loop corresponding to an access is software pipelined to schedule a prefetch a certain number of iterations ahead of the demand access. The number of iterations is computed by the formula $\lceil \frac{L}{W} \rceil$, where L is the expected miss latency in cycles and W is an estimate of the shortest possible path through an iteration in cycles. This number is referred to as the *prefetch distance*, and is expected to represent the number of iterations needed to completely overlap the latency of a prefetch. For our results in Section 2.8, we performed multiple experiments varying the prefetch distance for each application and reported the results for the prefetch distance that gave the maximum benefit (unless stated otherwise).

2.4 Impact of Conventional ILP Features

For each benchmark, Figure 2.2 presents execution times for three variations of our base architecture, each without VIS (the first set of three bars) and with VIS (the second set of three bars). The three architecture variations are (i) in-order and single issue, (ii) in-order and 4-way issue, and (iii) out-of-order and 4-way issue. On the VIS-enhanced architecture, we use the VIS-enhanced version of the application as mentioned in Section 2.3. The execution times are normalized to the time with the in-order single-issue processor.

For all the benchmarks, the execution time is further divided into the busy component, the functional unit stall (FU stall) component, and the memory component. With out-of-order processors, it is difficult to assign stall time to specific instructions since each instruction's execution may be overlapped with both preceding and following instructions. We therefore use the following convention to account for stall cycles. At every cycle, we calculate the ratio of the instructions retired that cycle to the maximum retire rate and attribute this fraction of the cycle to the busy time. The remaining fraction is attributed as stall time to the first instruction that could not be retired that cycle. The memory component is shown further divided into the L1 miss and L1 hit components.

This section focuses on the system without the VIS media ISA extensions (the left three bars for each benchmark in Figure 2.2).

Overall results. Both multiple issue and out-of-order issue provide substantial reductions in execution time for most of our benchmarks. Compared to a single-issue in-order processor, on the average, multiple issue improves performance by a factor of 1.2X (range of

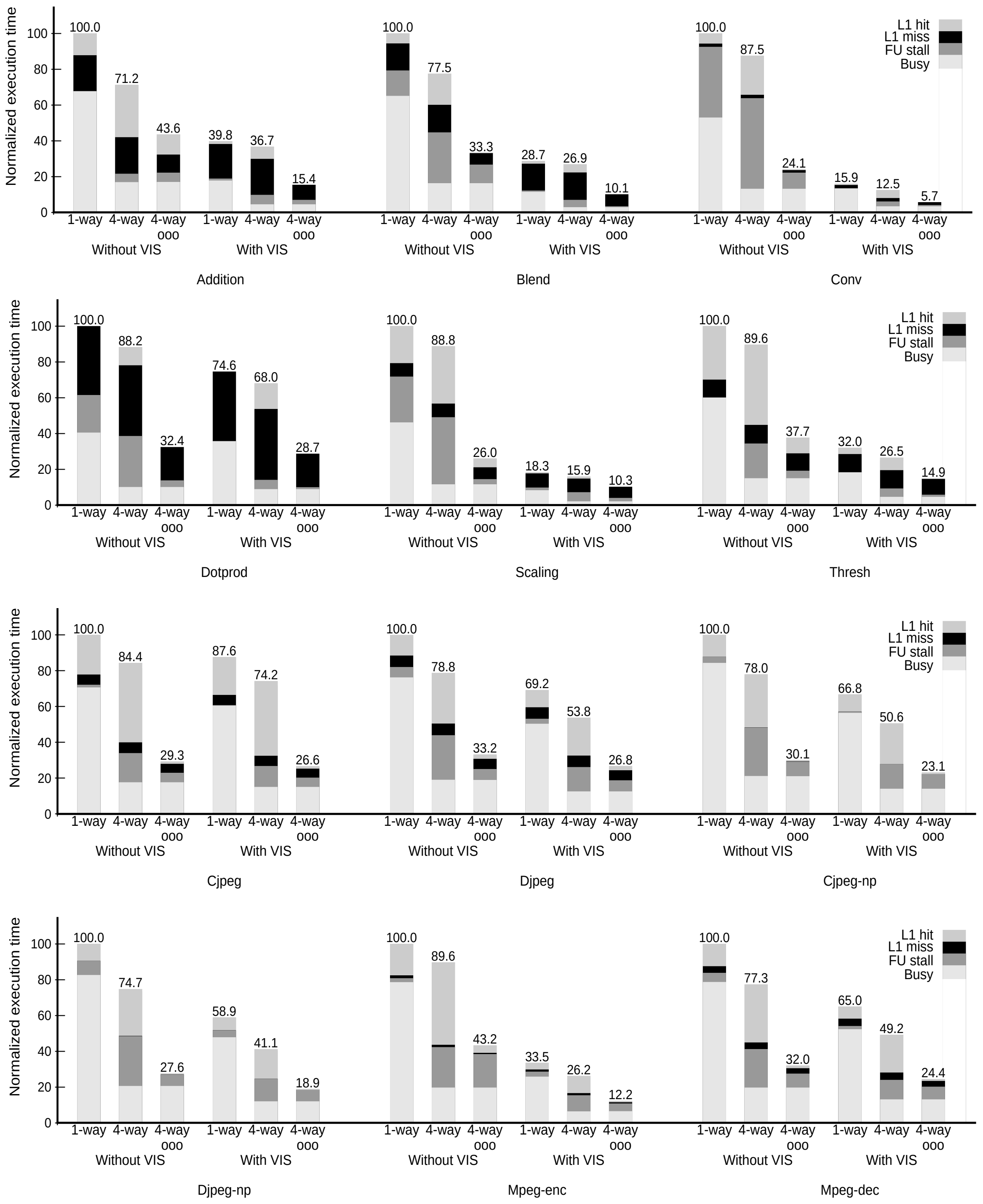


Figure 2.2 Performance of image and video benchmarks.

1.1X to 1.4X), while the combination of multiple issue and out-of-order issue improves performance by a factor of 3.1X (range of 2.3X-4.2X).

Analysis. Compared to the single issue processor, we find that multiple issue achieves most of its benefits by reducing the busy CPU component of execution time. Data, control, and structural dependences prevent the CPU component from attaining an ideal speedup of 4 from a 4-way issue processor, reflected in the increased functional unit and L1 hit memory stall time.

Some of the benchmarks see additional memory latencies when the number of outstanding misses to one cache line (MSHR) increases beyond the maximum allowed limit of 8. This is caused by the higher use of small data types associated with media applications which leads to a high frequency of accesses for each cache line (e.g., 64 pixel writes in a 64-byte line). Since the processors do not stall on writes, in benchmarks with small loop bodies (e.g., *addition*, *cjpeg*, *djpeg*), this leads to a backup of multiple writes. This backup leads to contention for the MSHR that eventually prevents other accesses from being serviced at the cache.

Out-of-order issue, on the other hand, improves performance by reducing both functional unit stall time and memory stall time. A large fraction of the stall times due to data, control, and structural dependences, as well as MSHR contention, is now overlapped with other useful work. This is seen in the reduction in the FU stall and L1 hit components of execution time. Additionally, out-of-order issue can better exploit the non-blocking loads feature of the system by allowing the latency of multiple long-latency load misses to be overlapped with one another. Our results examining the MSHR occupancies at the cache indicate that while there is increased load miss overlap in all the 12 benchmarks, only 2 to 3 misses are overlapped in most cases. The total capacity of 12 MSHRs is never fully utilized for load misses in all our benchmarks.

Overall, the impact of the various ILP features is qualitatively consistent with that described in previous studies for scientific and database workloads. Quantitatively, these ILP features are substantially more effective for the image and video benchmarks than for on-line transaction processing (OLTP) workloads [99], and comparable in benefit to previously reported scientific and decision support system (DSS) workloads [10, 89, 99].

It must be noted that the performance of the in-order issue processor is dependent on the quality of the compiler used to schedule the code. Our experiments use the commercial SPARC SC4.2 compiler with maximum optimizations turned on for the in-order UltraSPARC processor. To try to isolate compiler scheduling effects, we studied two other processor configurations with single-cycle functional unit latencies and functional unit latencies comparable to the UltraSPARC processor. In both these configurations, our results continued to be qualitatively similar; the out-of-order processor continues to significantly outperform the in-order processor. The impact of future, more advanced, compiler optimizations, however, is still an open question.

Interestingly, an earlier position paper [31] on the impact of multimedia workloads on general-purpose processors conjectures that complex out-of-order issue techniques developed for scientific and engineering workloads (e.g., SPEC) may not be needed for multimedia workloads. Our results show that, on the contrary, out-of-order issue can provide significant performance benefits for image and video workloads.

2.5 Impact of VIS Media ISA Extensions

This section discusses the performance impact of the VIS ISA extensions (comparing the left-hand three bars and right-hand three bars for each benchmark in Figure 2.2).

2.5.1 Overall Results

The VIS media ISA extensions provide significant performance improvements for all the benchmarks (factors of 1.1X to 4.0X for the out-of-order system, 1.1X to 7X across all configurations). On average, the addition of VIS improves the performance of the single-issue in-order system by a factor of 2.0X, the performance of the 4-way-issue in-order system by a factor of 2.1X, and the performance of the 4-way issue out-of-order system by a factor of 1.8X.

Multiple issue and out-of-order issue are beneficial even with use of the VIS media ISA extensions. On average, with VIS, compared to a single-issue in-order processor, multiple issue achieves a factor of 1.2X performance improvement, while the combination of multiple issue and out-of-order issue achieves a factor of 2.7X performance improvement. The reasons for these performance benefits from ILP features are the same as for the systems without VIS.

2.5.2 Benefits from VIS

Figure 2.3 presents some additional data showing the distribution of the dynamic (retired) instructions for the 4-way out-of-order processor without and with VIS, normalized to the former. Each bar divides the instructions into the Functional unit (FU, combines ALU and FPU), Branch, Memory, and VIS categories. The use of VIS instructions provides a significant reduction in the dynamic instruction count for all the benchmarks. The reductions in the dynamic instruction count correlate well with the performance benefits from VIS. We next discuss the sources for the reductions in the dynamic instructions.

Reductions in FU instructions. The VIS packed arithmetic and logical instructions allow multiple (typically four) arithmetic instructions to be replaced with one VIS instruction. Consequently, all the benchmarks see significant reductions in the FU instructions with corresponding, smaller, increases in the VIS instruction count. Additionally, the SIMD VIS instructions replace multiple iterations in the original loop with one equivalent VIS iteration. This reduces iteration-specific loop-overhead instructions that increment index and address values and compute branch conditions, further reducing the FU instruction count.

Reductions in branch instructions. All the benchmarks use the edge masking and partial store instructions to eliminate testing for edge boundaries and selective writes. They also

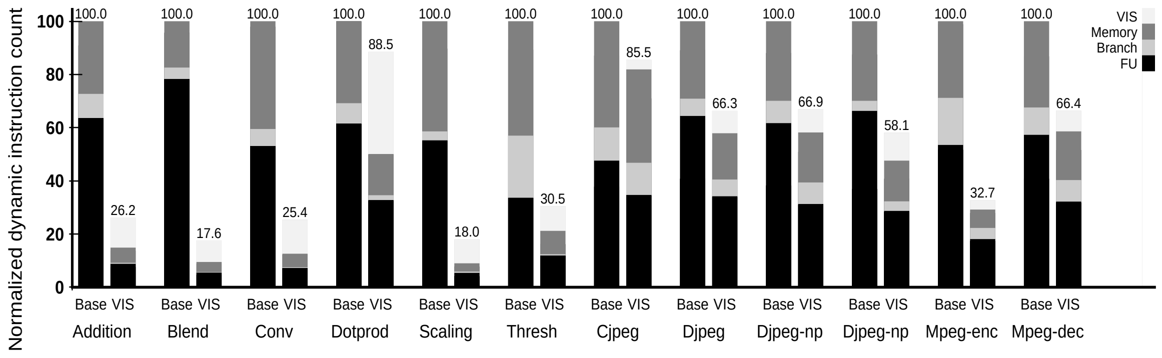


Figure 2.3 Impact of VIS on dynamic (retired) instruction count.

use loop unrolling when replacing multiple iterations with one equivalent VIS iteration. These lead to a reduction in the branch instruction count for all the benchmarks. For some applications, branch instruction counts are also reduced because of the elimination of the code to explicitly perform saturation arithmetic (mainly in *conv*[3] and the JPEG applications), and the use of partitioned SIMD compares (mainly in *thresh*).

Many of the branches eliminated are hard-to-predict branches (e.g., saturation, thresholding, selective writes), leading to significant improvements in the hardware branch misprediction rates for some of our benchmarks (branch misprediction rate decreases from 10% to 0% for *conv* and from 6% to 0% for *thresh1*).

Reductions in memory instructions. With VIS, memory accesses operate on packed data as opposed to individual media data. Consequently, most of the benchmarks see significant reductions in the number of memory instructions (and associated cache accesses). This reduces the MSHR contention discussed in Section 2.4.

Most of the memory instructions eliminated are cache hits, without a proportional decrease in the number of misses, causing higher L1 cache miss rates with VIS. The higher miss rate and the lower instruction count allows additional load misses to appear together

[3]The original source code from the Sun VSDK checks for saturation only in the *conv* code. The *add, blend*, and *dotprod* kernels are written in the non-saturation mode. These could, however, be rewritten to check for saturation in which case they would also see similar benefits.

within the instruction window and be overlapped with each other. However, the system still rarely sees more than 3 load misses overlapped concurrently.

Pixel distance computation for the *mpeg-enc* benchmark. *mpeg-enc* achieves additional benefits from the special-purpose pixel distance computation (`pdist`) instruction in the motion estimation phase. The `pdist` instruction allows a sequence of 48 instructions to be reduced to one instruction [123], significantly reducing the FU, branch, and memory instruction counts. The elimination of hard-to-predict branches to perform comparisons and saturation improves the branch misprediction rate from 27% to 10%.

2.5.3 Limitations of VIS

Examining the variation of performance benefits across the benchmarks, we observe that the JPEG applications, *mpeg-dec*, and *dotprod* exhibit relatively lower performance benefits (factors of 1.1X to 1.5X) compared to the remaining benchmarks (factors of 2.8X to 4.2X). We next discuss factors limiting the benefits from VIS.

Inapplicability of VIS. The VIS media ISA extensions are not applicable to a number of key procedures in the JPEG applications and *mpeg-dec*. For example, the JPEG applications (especially the progressive versions) spend a large fraction of their time in the variable-length Huffman coding phase. This phase is inherently sequential and operates on variable-length data types, and consequently cannot be optimized using VIS instructions.[4] Other examples of code segments in the JPEG and MPEG applications where VIS could not be applied include bit-level input/output stream manipulation, scatter-gather addressing, quantization, and saturation arithmetic operations not embedded in a loop.

VIS overhead. All our benchmarks use subword rearrangement and alignment instructions to get the data in a form that the VIS instructions can operate (e.g., packing/unpacking between packed-byte pixels and packed-word operands). This results in extra overhead

[4] It is instructive to note that even many media processors (e.g., the Mitsubishi VLIW Media Processor and Samsung Media Signal Processor) use a special-purpose hardware unit to handle the variable-length coding.

that limits the performance benefits from VIS (on the average, for our benchmarks, 41% of the VIS instructions are for subword rearrangement and alignment). Overhead is also increased when multiple VIS instructions are used to emulate one operation (e.g., 16x16 multiply in *dotprod*) or when the data need to be reordered to exploit SIMD (e.g., byte reordering in the color conversion phase in JPEG).

Limited parallelism and scheduling constraints. Most of the packed arithmetic instructions operate only on packed words or packed double words. This ensures enough bits to maintain the precision of intermediate values. However, this limits the maximum parallelism to 4 (on 16-bit data types), even in cases when the operations are known to be performed on smaller data types (e.g., 8-bit pixels). The limit on the SIMD parallel path,[5] in combination with contention for VIS functional units, limits the benefits from VIS on some of our benchmarks (most significantly in *mpeg-enc*).

Cache miss stall time. As discussed earlier, the reductions in memory instructions mainly occur for cache hits. The VIS instructions do not directly target cache misses though there are indirect benefits associated with increased load miss overlap due to instruction count reduction (discussed in Section 2.5.2).

2.6 Combination of ILP and VIS

The combination of conventional ILP features and VIS extensions achieves an average factor of 5.5X performance improvement (range of 3.5X to 18X) over the base single-issue in-order processor. The benefits from VIS are achieved with a much smaller increase in the die area compared to the ILP features.

On the base single-issue in-order processor, all the benchmarks are primarily compute-bound. With ILP features and VIS extensions, *cjpeg*, *djpeg*, and *mpeg-enc* continue to spend most of their execution time in the processor sub-system (87% to 97%). Five of the

[5]The MIPS MDMX provides support for a larger size accumulator that allows greater parallelism without losing the precision of the intermediate result [58]. The PowerPC AltiVec supports a larger 128-bit data path to increase the parallelism[95].

image processing kernels, however, now spend 55% to 66% of their total time in memory stalls. The strong compute-centric performance benefits from ILP features and VIS extensions shift the bottleneck to the memory sub-system for these benchmarks. The remaining 4 applications (*conv*, *cjpeg*, *djpeg*, and *mpeg-dec*) spend between 20% to 30% of their total time on memory stalls.

The next two sections (Sections 2.7 and 2.8) discuss the effectiveness of commonly-available memory system features such as support for large on-chip caches and software prefetching.

2.7 Impact of Caches

The first technique that we study to address the memory system bottleneck is support for large on-chip caches. The principle of caching is based on the key intuition that most programs typically tend to operate on a small portion of the data at a time exhibiting *locality* in memory accesses. Locality can either be (i) *spatial locality* where successive data items are accessed one after the other by the program or (ii) *temporal locality* where the program accesses the same data multiple times. Consequently, most processors include support for a smaller but faster memory that attempts to capture and store this reused data closer to the processor to help improve memory system performance. Caches have been demonstrated to achieve significant performance benefits for traditional engineering and scientific workloads. Consequently, the use of large caches is a common trend across general-purpose systems, sometimes consuming up to 80% of the total transistor budget and up to 50% of the die area [46].

Below, we first discuss the impact of varying the size of the second-level cache (Section 2.7.1) and then discuss the impact of varying the size of the first-level cache (Section 2.7.2). Section 2.7.3 summarizes our results.

2.7.1 Impact of Varying Second-Level Cache Sizes

Overall results. We varied the second-level (L2) cache size from 128K to 2M, keeping the L1 cache fixed at 64K. Our results (not shown here due to lack of space) showed that increasing the size of the L2 cache has no impact on the performance of the 6 image processing kernels and the *cjpeg-np* and *djpeg-np* applications. The remaining four applications, *cjpeg*, *djpeg*, *mpeg-enc*, and *mpeg-dec*, reuse data; but the cache size needed to exploit the reuse depends on the size of the display. For our input image sizes, L2 cache sizes of 2M capture the entire working sets for all the four benchmarks, and provide 1.1X to 1.2X performance improvement over the default 128K L2 cache. With the 2M cache sizes, memory stall time is between 7-9% on all the applications and is dominated by L1 hit time (due to cache stalls on contention for MSHRs) and L2 hit time.

Analysis. The image processing kernels have streaming data accesses to a large image buffer with no reuse and low computation per cache miss. Consequently, they exhibit high memory stall times unaffected by larger caches. *cjpeg-np* and *djpeg-np* do not see any variation in performance with larger caches because of their negligible memory components. These applications implement a blocked pipeline algorithm that performs all the computation phases on 8x8-sized blocks at a time, reducing the bandwidth requirements and increasing the computation per miss. The *cjpeg* and *djpeg* applications differ from their non-progressive counterparts in the progressive coding phase where they perform a multi-pass traversal of the buffer storing the DCT coefficients. The low computation per miss in this phase combined with the reuse of the image-sized (1024x640 pixels) buffer results in a 1.2X performance benefit from increasing the cache size to 2M. Larger images would increase the working set requiring larger caches. For example, a 1024x1024 image would require a 4M cache size. *mpeg-enc* and *mpeg-dec* perform inter-block distance vector operations and therefore need to operate on multiple image-sized buffers (as opposed to blocks-sized buffers in *cjpeg-np* and *djpeg-np*). The reuse of these 352x240 buffers across the frames in the video leads to 1.1X performance benefits with 512K (*mpeg-dec*) and 1M (*mpeg-enc*) cache sizes. Larger image sizes would require larger caches; for example a 1024x1024 image would require almost a factor of 12X increase in the cache size.

2.7.2 Impact of Varying First-Level Cache Sizes

Overall results. We also performed experiments varying the size of the first-level (L1) cache from 1K to 64K while keeping the L2 cache fixed at 128K. Our results showed that the L1 cache size had no impact on five of the image processing kernels. On the remaining benchmarks, a 64K L1 configuration outperforms the 1K L1 configuration by factors of 1.1X to 1.3X; 4K-16K L1 caches achieve within 3% of the performance of the 64K L1 cache configuration.

Analysis. Small data structures other than the main data, such as tables for convolution, quantization, color conversion, and saturation clipping, are responsible for these small first-level working sets. At 64K L1 caches, memory stall time is mainly due to L1 hits or to L2 misses. Again, the L1 hits are mainly associated to longer latency first-level cache accesses because of stalled cache cycles on contention for MSHR resources (similar to that discussed in Section 2.4). This stall time may be addressed with a more aggressive cache design that allows cache hits to proceed irrespective of contention for outstanding miss resources.

2.7.3 Overall Results

The memory behavior of these workloads is characterized by large working sets and streaming data accesses. Increasing the cache size has no impact on the image processing kernels and the non-progressive JPEG applications. The remaining benchmarks require relatively large cache sizes (dependent on the display sizes) to exploit data reuse, but derive less than 1.2X performance benefits with the larger caches.

These results are particularly interesting considering current trends towards large on-chip and off-chip caches and the associated large on-chip transistor investment into caches. Chapter 3 discusses a novel cache organization that addresses this trend.

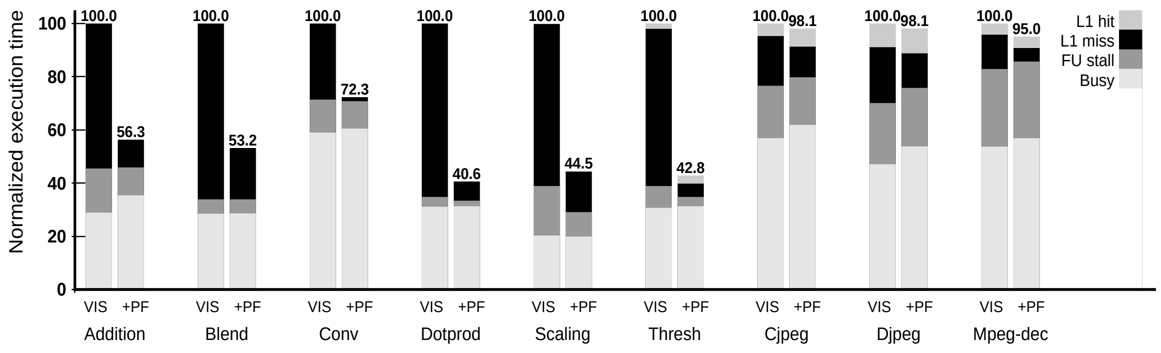


Figure 2.4 Effect of software-inserted prefetching on image and video processing workloads.

2.8 Impact of Software Prefetching

The second technique that we study to address the memory system bottleneck is software prefetching. To reduce memory stall time, all current commodity processors support software-controlled non-binding prefetching. With this technique, the compiler or programmer schedules an explicit *prefetch* instruction for a location that will be accessed by the processor at a later time, with the goal of bringing the location into the processor's cache before it issues a demand memory access [17]. Previous studies have shown that software-controlled non-binding prefetching can eliminate a large fraction of memory stall time in shared-memory multiprocessors [84, 124, 100]. However, these studies have been mainly limited to conventional scientific and engineering workloads. In this section, we study the effectiveness of software prefetching for our media processing workloads. As discussed in Section 2.3.3, we follow the well known software prefetching compiler algorithm developed by Mowry et al. [83] to insert prefetches by hand for our benchmarks.

Overall results. Figure 2.4 summarizes the execution time reductions from software prefetching relative to the base system with VIS (with 64K L1 and 128K L2 caches). We do not report results for *cjpeg-np*, *djpeg-np* and *mpeg-enc* since these benchmarks spend less than 6% of their total time on L1 cache misses. Our results show that software prefetching achieves high performance improvements for the six image processing benchmarks (an average of 1.9X and a range of 1.4X to 2.5X. The *cjpeg*, *djpeg*, and *mpeg-dec* benchmarks exhibit relatively small performance improvements. Overall, after applying software prefetching, all our benchmarks revert to being compute bound.

Analysis. For the image processing kernels, a significant fraction of the prefetches are useful in completely or partially hiding the latency of the cache miss with computation or with other misses. The addition of software prefetching also increases the utilization of cache MSHRs; in many of the image processing kernels, more than 5 MSHRs are used for a large fraction of the time. The remaining memory stall time is mainly due to late prefetches and resource contention. Late prefetches (prefetches that arrive after the demand access) arise mainly because of inadequate computation in the loop bodies to overlap the miss latencies. Contention for resources occurs when multiple prefetches are outstanding at a time. These effects are similar to those discussed in previous studies with scientific applications for ILP-based processors [100].

The other benchmarks (*cjpeg*, *djpeg*, and *mpeg-dec*) see lower performance benefits primarily because the fraction of memory stall time is relatively low and includes an L1 hit component (mainly due to MSHR contention). Software prefetches do not address the L1 component. Second, in *cjpeg* and *djpeg*, the prefetches are to memory locations that are indirectly addressed (of the form A[B[i]]). Consequently, the prefetching algorithm is unable to distinguish between hits and misses and is constrained to issue prefetches for all accesses. The resulting overhead due to address calculation and cache contention limits performance (seen as increased Busy and FU stall components[6]). Finally, as before, late prefetches and resource contention also contribute to the lower benefits from prefetching.

2.9 Related Work

This section discusses other related work relevant to the research presented in this chapter.

[6]Some of the image processing kernels see a reduction in the CPU component because of the reduction in instructions and better scheduling when loops are unrolled for the prefetching algorithm [83].

Most of the papers discussing instruction-set extensions for multimedia processing have focused on detailed descriptions of the additional instructions and examples of their use [19, 58, 68, 87, 93, 95, 123]. The performance characterization in these papers is usually limited to a few sample code segments and/or a brief mention of the benefits anticipated on larger applications. Eyre studies the applicability of general-purpose processors for DSP applications; however, the study only reports high-level metrics such as MIPS, power efficiency, and cost [35].

Daniel Rice presents a detailed description of VIS and 8 image processing applications without and with VIS [103]. The study reports speedups of 2.7X to 10.5X on an actual UltraSPARC-based system, but does not analyze the cause of performance benefits, the remaining bottlenecks, or the impact of alternative architectures.

Yang et al. look at the benefits of packed floating-point formats and instructions for graphics but assume a perfect memory system [129]. Bharghava et al. study some MMX-enhanced applications based on Pentium-based systems [11]; Nguyen and John study the same applications enhanced with AltiVec [86]. Again, in both these studies, no detailed characterization of performance bottlenecks or the impact of other architectures is done.

Following our work, a recent study by Sebot [108] has studied the performance improvements of several multimedia kernels with the AltiVec streaming SIMD extensions. Even though this work evaluates a different family of media extensions using a different simulation infrastructure, their results are qualitatively similar to our results. This corroborates our intuition that the media extensions studied in this chapter are representative of other media extensions as well. Specifically, Sebot's work, in addition to showing performance improvements from SIMD extensions, also highlights the transition from compute-bound to memory-bound with the use of media extensions and the benefits from software prefetching. Quintana et al. [96] also study a superscalar processor enhanced with a vector unit and show that this architecture can achieve good performance benefits.

Zucker et al. study MPEG video decode applications and show the benefits from I/O prefetching, software restructuring to use SIMD without hardware support, and profile-driven software prefetching [132, 133]. However, the studies assume a simplistic processor model with blocking loads and do not study the effect of media ISA extensions. Bilas et al. develop two parallel versions of the MPEG decoder and present results for multiprocessor speedup, memory requirements, load balance, synchronization, and locality [12]. Similar to our results, they also find that the miss rates for 352x240 images on "realistic" cache sizes are negligible. Iwata et al. also study parallel implementations of MPEG-2. Their results show that the coarse-grained parallelism in MPEG-2 is orthogonal to the fine-grained parallelism which can be exploited by instruction-set extensions like MIPS MDMX [52].

Lee and Stoodley discuss the performance of a simple in-order long vector media processor [66]. For the benchmarks they study (chroma-decimation, color-space conversion, image blending and convolution, and decryption), their proposed system outperforms a conventional MMX-enhanced superscalar processor by a factor of 1.6. Similar to our work, their results also show that most of the performance benefits come from reductions in the instruction count and the use of SIMD parallelism. However, in contrast to this work, our research focuses on general-purpose processor applicable to both conventional and emerging media processing workloads. Our results also show that many of the media benchmarks exhibit control-, data-, and resource-dependences which can be alleviated with the use of out-of-order execution.

Corbal et al. discuss a new matrix oriented set of extensions to the instruction-set architecture targeted towards media processing [23]. Their results identify several of the limitations that we discuss in Section 2.5.3 and use the matrix-oriented ISA extensions to improve performance in some of these cases.

Zhang et al. also study image processing applications (volume rendering, image warping, and image filtering), but they primarily focus on improving the memory system performance using the dynamic reordering of memory accesses using the Impulse memory controller [130].

2.10 Summary

This chapter focuses on image and video processing, an important class of media processing, and aims to provide a quantitative understanding of the performance of these workloads on general-purpose processors. We use detailed simulation to study 12 representative benchmarks on a variety of architectural configurations, both with and without the use of Sun's visual instruction set (VIS) media ISA extensions.

Our results show that conventional techniques in current processors to enhance ILP (multiple issue and out-of-order issue) provide a factor of 2.3X to 4.2X performance improvement for the image and video benchmarks. The Sun VIS media ISA extensions provide an additional 1.1X to 4.2X performance improvement. The benefits from VIS are achieved with a much smaller increase in the die area compared to the ILP features.

Our detailed analysis indicates the sources and limitations of the performance benefits due to VIS. VIS is very effective in exploiting SIMD parallelism using packed data types, and can eliminate a number of hard-to-predict branches using instructions targeted towards saturation arithmetic, boundary detection, and partial stores. Special-purpose instructions such as *pdist* achieve high benefits on the targeted application, but are too specialized to use in other cases. Routines that are sequential and operate on variable data types, VIS instruction overhead, cache miss stall times, and the fixed parallelism in the packed arithmetic instructions limit the benefits on the benchmarks.

On our base single-issue in-order processor, all the benchmarks are primarily compute-bound. Conventional ILP features and the VIS instructions together significantly reduce the CPU component of execution time, making five of our image processing benchmarks memory-bound. The memory behavior of these workloads is characterized by large working sets and streaming data accesses. Increasing the cache size has no impact on the image processing kernels and the non-progressive JPEG applications. This is particularly interesting considering current trends towards large on-chip and off-chip caches. The remaining benchmarks require relatively large cache sizes (dependent on the display sizes) to exploit data reuse, but derive less than 1.2X performance benefits with the larger caches.

Software-inserted prefetching achieves 1.4X to 2.5X performance benefits on the image processing kernels where memory stall time is significant. With the addition of software prefetching, all our benchmarks revert to being compute-bound.

The next chapter builds on this work to discuss a new architectural design that improves the performance of media processing workloads.

Chapter 3

Reconfigurable Caches

The previous chapter presented our results where we quantitatively characterized the performance of media processing workloads on general-purpose systems. Our results identified several characteristics of media processing that were different from the characteristics exhibited by conventional workloads (e.g., sub-word data types, SIMD parallelism, ineffectiveness of caches). As current high-performance general-purpose processors get used for an increasing variety of application domains, including scientific, engineering, database, and media processing, it becomes important to ensure that these processors perform well enough across all these different workloads. Consequently, a "one-size-fits-all" design philosophy will be inadequate.

An alternative design philosophy is to build some flexibility in the system so that features that use a large number of resources can be used in different ways by different applications. In this chapter, we apply this philosophy to the design of caches. We focus on caches because the use of large caches is a common trend across current general-purpose systems, typically consuming the largest fraction on the on-chip transistor resources (up to 80% of the total transistor budget and up to 50% of the die area [46]). Additionally, as discussed in Chapter 2, while large caches are effective for a variety of conventional workloads, they are often ineffective for media processing applications because of the streaming nature of data accesses, the large working sets, and the compute-bound nature of these applications.

In response to these observations, we propose a new cache organization called *reconfigurable caches*. Reconfigurable caches allow the on-chip SRAM to be dynamically divided into different partitions that can be assigned to different system activities other than conventional caching. For example, the partitions could be used as hardware look-up tables for techniques such as instruction reuse and hardware prefetching, or as storage area for prefetched information, or as compiler-controlled memory. Thus, the cache SRAM storage can benefit applications that would not otherwise exploit large conventional caches. Figure 3.1 illustrates the idea behind reconfigurable caches.

The idea of reconfigurable architectures itself is not new; however, conventional reconfigurable architectures typically involve large changes in both hardware and software. In comparison, our reconfigurable cache design continues to use a conventional cache design with minor hardware and software changes. Also, in our model, the reconfigurability is implemented in custom hardware at design time. We support a limited number of possible configurations (e.g., two or three) which are incorporated into the circuit design and verified as part of the original microprocessor design. This is in contrast to an approach that might use hardware similar to field-programmable gate-arrays (FPGAs). While useful in many applications, the ability to dramatically change the programming of FPGAs in the field results in FPGA clock cycle times that are typically 4 to 6 times slower than that obtained in full-custom microprocessors.

The rest of the chapter is organized as follows. Section 3.1 first discusses several applications of reconfigurable caches. Section 3.2 discusses the key challenges with reconfigurable cache organizations and our proposed solutions. Section 3.3 discusses the design issues involved with a detailed implementation of reconfigurable caches and uses the CACTI analytical timing model to show that our modifications do not significantly impact the cache access time of the system. Finally, Section 3.4 focuses on one representative application of reconfigurable caches – instruction reuse for media processing – and presents simulation results to quantify the benefits from reconfigurable caches. Section 3.5 discusses related work and Section 3.6 summarizes this chapter.

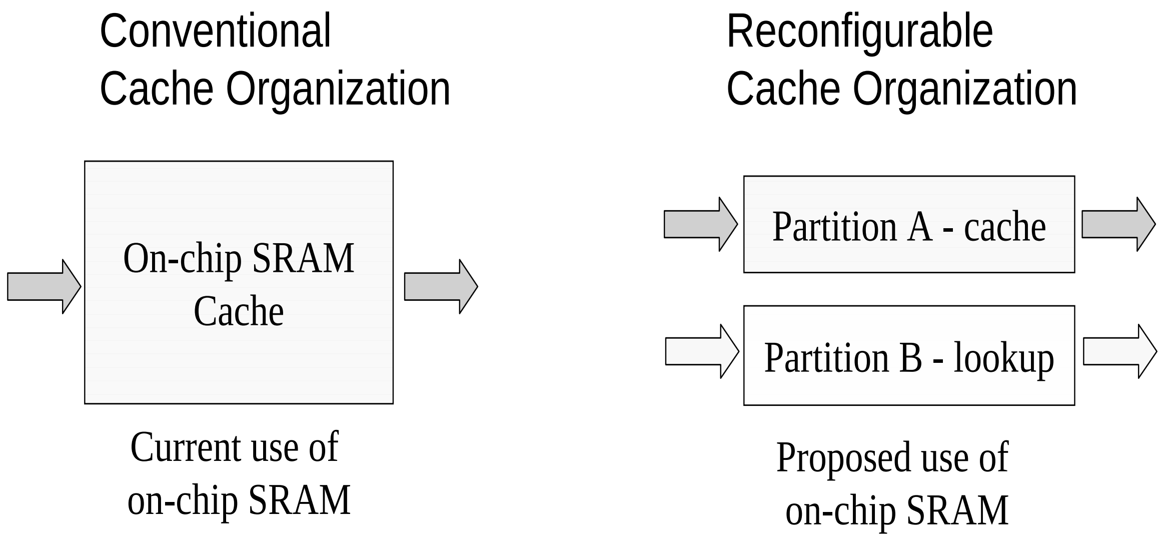

Conventional Cache Organization
On-chip SRAM Cache
Current use of on-chip SRAM
Reconfigurable Cache Organization
Partition A - cache
Partition B - lookup
Proposed use of on-chip SRAM

Figure 3.1 Reconfigurable caches: Key idea.

3.1 Potential Applications of Reconfigurable Caches

The reconfigurable cache organization provides the ability to dynamically divide the on-chip SRAM area into different partitions that can be used for different activities other than conventional caching. Some of the possible applications for reconfigurable caches are discussed below. We specifically discuss how these applications are relevant to the domain of media processing; however, codes from other domains can benefit from these as well.

Hardware optimizations using lookup tables or buffers.

Several hardware optimizations have been proposed that require maintaining lookup tables or buffers, where the effectiveness of the optimization improves significantly with larger table sizes. For example, value prediction, memoization, and instruction reuse have recently been studied to exploit redundancy in computation in the SPEC benchmarks [39, 70, 71, 107, 110, 111]. Other optimizations that require large lookup tables or buffers include coherence prediction, memory disambiguation prediction, compression-based branch prediction, hardware prefetching (where lookup tables are used to store information for address prediction), and dynamic optimizations triggered by performance information collected and stored in tables at runtime. Several of these techniques have been reported to have the capacity to perform better with larger lookup table spaces [107, 110]. The lookup tables and buffers for these optimizations could be implemented in a partition of a reconfigurable cache instead of using other valuable chip area. Section 3.4 studies one such technique, instruction reuse, with reconfigurable caches to address the computation bottleneck in media processing workloads.

Software and hardware prefetched data.

Software and hardware prefetching are widely used techniques to hide memory latency. However, if the prefetched data is fetched too far in advance, it can pollute the cache replacing other useful data or be replaced before use by a demand access, eliminating any performance benefits. On the other hand, prefetches that occur too late do not fully hide the latency. Therefore, prefetching techniques need to strike a careful balance when scheduling the prefetches, but are often unsuccessful in doing so. With reconfigurable caches, a separate partition can be used to prefetch data early while avoiding the problem of cache pollution or replacement of prefetched data. Such an application of reconfigurable caches could be particularly useful with media processing benchmarks which often have streaming behavior (Section 2.7).

Compiler or application controlled memory.

A partition of a reconfigurable cache could be configured as compiler or application controlled memory. As discussed in [22], the compiler could use such memory as a scratch area for spill code to achieve performance benefits. Alternatively, this area can be used by system code or device drivers as a separately addressable buffer area. Such a use may also be beneficial in ensuring real-time requirements of media applications with general-purpose processors. Many DSP processors hardwire their on-chip SRAM to be used as memory (as opposed to caches) to ensure predictability of memory latencies [36]. Cache line locking (as in the Cyrix MediaGX processor [73]) or controlled cache line replacement (as with *malleable caches* [21]) can provide the same functionality.

3.2 Cache Organization

Reconfigurable caches require a cache organization that allows the on-chip SRAM cache storage to be dynamically partitioned and reused for other processor activities requiring storage. There are several aspects to such an organization with multiple design options and tradeoffs. These are discussed in the rest of this section.

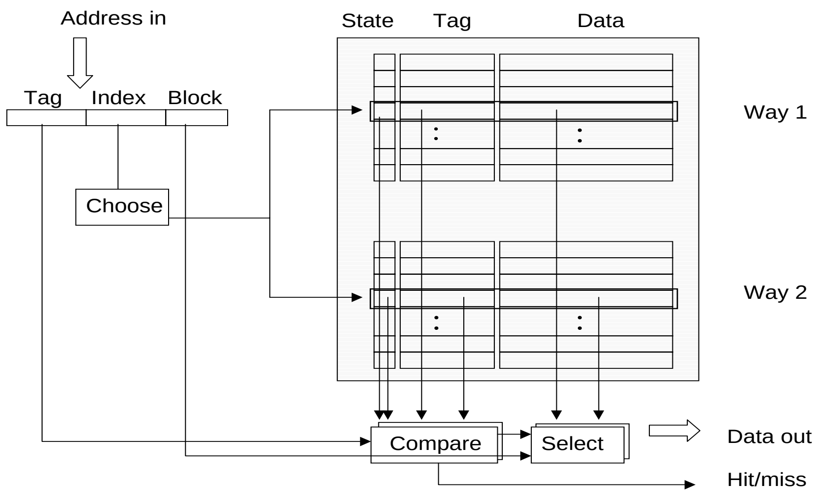
3.2.1 Partitioning and Addressing

The key challenge in designing a reconfigurable cache is to devise a mechanism to divide the SRAM storage into different (possibly variable-sized) partitions, and to efficiently be able to address these partitions. In particular, the addressing scheme must efficiently adapt to dynamic resizing of the partition sizes. Below, we first discuss a scheme based on cache associativity that is used in the rest of this chapter. We then discuss an alternative that does not rely on cache associativity.

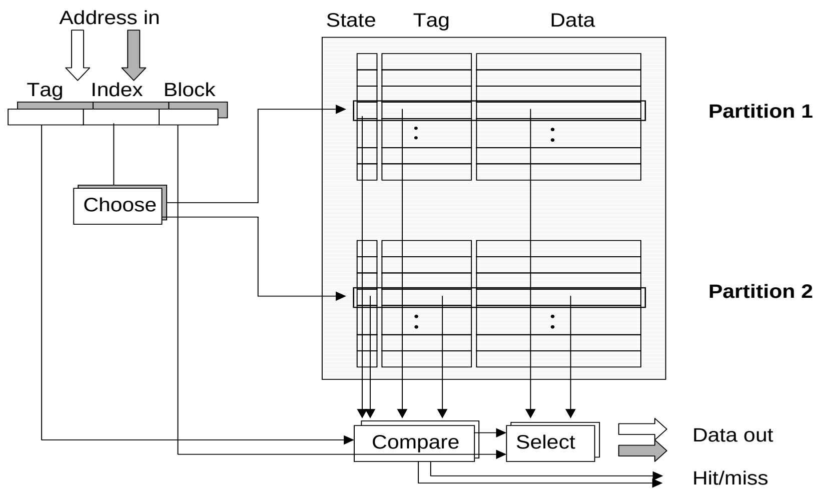
Associativity-based partitioning

Our primary design exploits set-associativity in current cache organizations. To understand the design, we briefly revisit a conventional set-associative cache organization. Figure 3.2(a) depicts the block diagram for the (conceptual) organization of a single-ported 2-way set-associative conventional cache. An N-way set associative cache is divided into N data and tag arrays, and each of these N pairs is referred to as a *way*. The index part of the input address is used to index all the ways of the data and tag array. The tag part of the input address is sent to the comparators of all the ways to determine if there is a match with any of the tags read from the tag array. If there is a match, a hit is signaled on the valid output line and data from the corresponding way of the data array is sent onto the output data lines.

The reconfigurable cache builds on the above organization in a natural way, as depicted in Figure 3.2(b). We divide the reconfigurable cache into partitions at the granu-



(a) Conventional cache organization



(b) Reconfigurable cache organization

Figure 3.2 Associativity-based partitioning organization for reconfigurable caches.

larity of the *ways* of the conventional cache, exploiting the conceptual division into ways already present in a conventional cache. For example, at the finest granularity, a 4-way set-associative 1MB cache can be dynamically reconfigured into 4 partitions of 256KB each, with each way in a separate partition. At coarser granularities, a partition of such a cache may contain two ways (for a total of 512KB organized as a two-way set-associative partition) or three ways (for a total of 768KB organized as a three-way set associative partition). In each case, the same bits of the address field would be used as tag, index, and block offset bits. In the 1MB cache example above, assuming 64 byte cache lines, the last 6 bits of the address are used for the block offset, the next 12 to index each way, and the remaining for the tag. The only changes to the conventional cache organization are as follows:

- *Multiple input and output paths.* A reconfigurable cache with up to N partitions must accept N input addresses and generate N output data elements with N hit/miss signals (one for each partition). Section 3.3 discusses a design that can achieve this without increasing the number of cache ports.

- *Cache status register.* The current partitioning of the cache controls which of the N input addresses is used to index a specific way and to perform the tag match at the comparator of that way. Similarly, for the ways that produce data hits, the current partitioning determines which of the N output data paths should get this data and which of the N hit/miss lines should be signaled as a hit. A special hardware register called the *cache status register* is therefore maintained to track the number and sizes of the partitions and control the routing of the above signals. The cache status register is part of the processor state and needs to be preserved across context switches similar to any other control register.

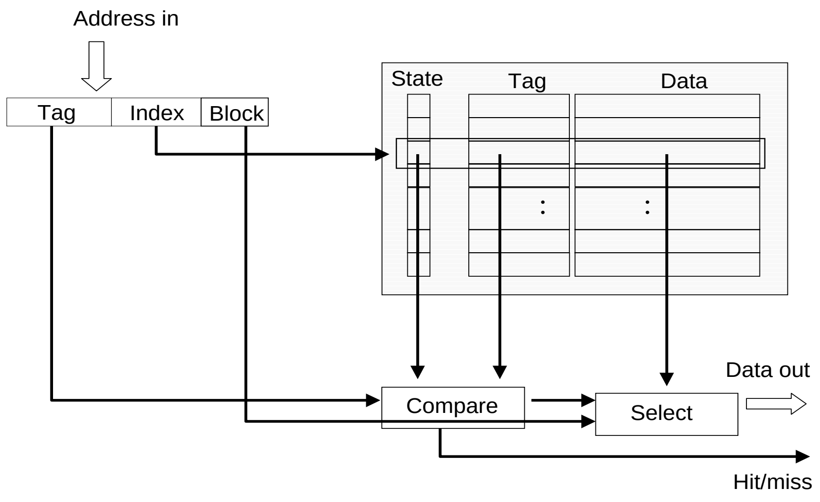
From the processor's viewpoint, the various partitions can be addressed either implicitly for internal hardware activities (e.g., value prediction lookup tables) or explicitly for software-controlled use (e.g., compiler-controlled memory). The latter would require some

ISA support to indicate the correct partition that the memory accesses need to be routed to (e.g., an approach similar to the address space identifiers could be used).

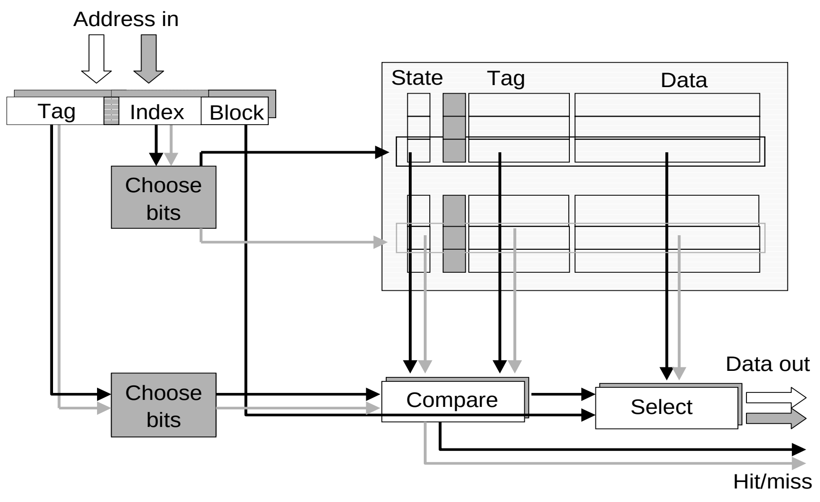
The above set-associativity based partitioning approach has at least three advantages. First, it requires only small changes to the current well-understood set-associative cache organization. Second, the mechanism for addressing the cache arrays scales well with the dynamic repartitioning of the cache. Third, this organization keeps requests for the different partitions isolated from each other, and so does not introduce any additional contention for the SRAM ways. However, the key drawback with this approach is that the number and granularity of the partitions are limited by the associativity of the cache. Larger or smaller granularity partitioning needs a higher-order associativity of the base cache which could increase cache access time and tag storage space. An alternative organization that is slightly more complex but does not suffer from the above drawback is discussed next.

Overlapped wide-tag partitioning

The overlapped wide-tag partitioning approach does not require a set-associative cache organization. The dark-shaded regions in Figure 3.3(b) indicate the additional changes to be made to the conventional direct-mapped cache organization in Figure 3.3(a). Since the number of bits in the index and tag fields of the address vary based on the size of the partition, the size of the tag array in the cache SRAM also needs to change dynamically with the size of the partitions. The wide-tag organization extends the current SRAM tag array to account for the maximum variation in the tag size with different partition sizes. For example, for a direct-mapped 1MB cache which supports multiple partitions each of which is at least 256KB large, the tag array would be extended by an additional two bits to support the the maximum possible tag length required by the 256KB partitions. Though the cache partitions can be of any size, we limit them to be powers of two to enable simpler decoding. When the data is read from the cache, the additional logic shown in Figure 3.3(b)



(a) Conventional cache organization



(b) Reconfigurable cache organization

Figure 3.3 Overlapped wide-tag partitioning organization for reconfigurable caches.

ensures that only the bits of the index and tag corresponding to the size of the particular partition are used for the cache access. This logic can be relatively simple.

3.2.2 Maintaining Data Consistency

Reconfigurable caches need a mechanism to ensure that after reconfiguration, the data belonging to a particular processor activity resides only in the partition associated with that particular activity. We discuss two approaches to ensure this below.

Cache scrubbing

This approach uses the partitioning information in the cache status register to ensure that at the time of reconfiguration, appropriate data gets moved between partitions or the data is moved from the cache to lower levels of memory. Such a *cache-scrubbing* approach requires examining all the locations of the cache to check for their validity and performing suitable actions on valid data at reconfiguration. For example, for a partition transformed from a cache to another activity, valid data needs to be sent to another partition used as cache or to lower levels of memory. Some applications such as compiler-controlled memory may require intervention at the operating system level to ensure that data is kept consistent. For partitions used as lookup tables for speculation, data consistency is not as critical an issue; not initializing the data to the correct values would just result in a longer "cold-start" period. Cache scrubbing can, in some cases, incur a high reconfiguration overhead, but can be acceptable in cases of infrequent reconfiguration such as at the start of the application. For example, for the 128KB L1 cache with 64 byte cache lines in Section 3.4, assuming a fully pipelined 20-cycle L2 cache and support for 12 outstanding writebacks, the time taken to write all the 2048 cache lines to the L2 cache is less than 3500 cycles. In practice only a smaller fraction of these 2048 cache lines will have to be written back to the cache.

Lazy transitioning

In some cases, more frequent reconfiguration may be desirable (due to frequent context switches between applications or aggressive adaptive reconfiguration for the same application). For such cases, an alternate scheme is possible where data is lazily moved into its correct partition only when it is accessed. For such a scheme, the state information associated with a cache line needs to be augmented with information on the processor activity associated with that line. For example, for a 128KB L1 cache with four partitions, if different activities are associated with each of the partitions, this would require about 512 bytes of extra state storage. In addition to the normal tag lookup, this technique requires the processor activity state to also be validated before the data is considered a hit. On a miss in the appropriate partition, other partitions need to be checked to determine if the data is still in those partitions. This approach can reduce the high overhead associated with moving large amounts of data at reconfiguration time and increase the data hit rates in the partitions; however, these benefits come at the expense of increased state storage, a more complex implementation, and possible increased contention for the SRAM partitions.

3.2.3 Reconfiguration Policy and Detection

The third issue that needs to be addressed is the policy and detection mechanism to determine when to reconfigure. Repartitioning may occur infrequently (e.g., just once at the start of the application) or frequently (e.g., at the beginning of certain loops), depending on the characteristics of the application and the processor activities for which the reconfigurable cache is used. The mechanism to detect when to reconfigure can be software or hardware controlled. A software-controlled approach can expose the cache status register to the code-generator (user or compiler) which can use information about the program behavior to invoke appropriate reconfiguration at the appropriate points in the program. Alternately, a hardware-controlled approach could use hardware performance monitoring support (e.g., DCPI [6]) to automatically decide when and how to change the partitions.

Design Issue	**Choices** *(the option used in this work is underlined)*
Partitioning mechanism	Overlapped wide-tag vs. *Associativity-based*
Address generation for non-cache partitions	*Hardware* vs. Software generated
Data consistency	*Cache scrubbing* vs. Lazy transitioning
Repartitioning policy	Frequent vs. *Infrequent*
Detection mechanism for reconfiguration	*Software* vs. Hardware control
Reconfigurable cache level	*L1*, L2, or lower levels

Table 3.1 Reconfigurable cache organization choices.

3.2.4 Reconfigurable Cache Level

The final issue with the reconfigurable cache organization is the level that is reconfigurable in a multi-level cache hierarchy. The cache organization described above does not preclude its application to any level. Tradeoffs in terms of the size, granularity, access time, and usage of the partitions will determine the level to partition.

3.2.5 Options Used in This Work

Table 3.1 summarizes the various aspects of the reconfigurable cache organization and gives the configuration we study in this chapter. Specifically, we study a hardware-addressed associativity-based partitioning approach with software control for infrequent reconfiguration and cache scrubbing to ensure consistency of data. We apply this configuration to the L1 cache in Section 3.4. This configuration represents a simple organization that is likely to achieve most of the performance benefits for the applications discussed in Section 3.1.

3.3 Design and Implementation

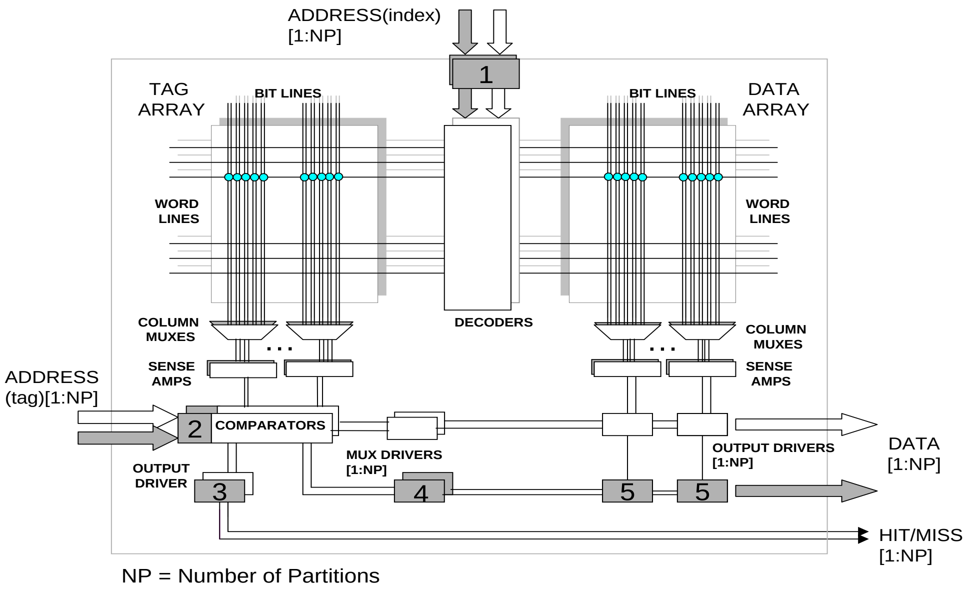
Sections 3.3.1–3.3.3 first discuss a more detailed implementation of the reconfigurable cache organization discussed in Section 3.2.5. Section 3.3.4 then uses a modification of the CACTI model for a detailed timing analysis of a reconfigurable cache. Section 3.3.5 discusses the transistor-count and energy tradeoffs with reconfigurable caches. Section 3.3.6 summarizes our results.

3.3.1 Conventional Cache Implementations

Figure 3.4 shows a typical implementation of the internal structure of a conventional SRAM cache. The numbered shaded blocks refer to the changes needed for reconfigurable caches and are discussed in subsequent subsections.

The key components are the data and tag arrays. The arrays consist of the storage cells, the horizontal wordlines that enable a single row of cells, and the vertical bitlines that transfer data from the selected cell of a column. A straightforward implementation of the data (or tag) array would have *S* rows and *8BA* columns of storage cells, where *S* is the number of sets in the cache, *B* is the line size in bytes, and *A* is the associativity. This implementation, however, would incur large cache access times because of the long and unbalanced wordline and bitline delays. Instead, existing implementations divide both the data and tag arrays into multiple subarrays, as shown in Figure 3.4. In this work, we use the CACTI timing model [102, 127] to identify the optimal values for the dimensions of the subarrays. Note that a row of a subarray can have cells from multiple ways of the same set.

Other components of the cache and the various delay components that comprise the operation of a cache (as modeled by CACTI) are as follows. A decoder for each SRAM subarray first decodes the address (incurring a *decoder delay*) and selects the appropriate subarray row by driving one wordline in the array (*wordline delay*); only one wordline in each subarray can be high at a time. Each memory cell along the selected row is associated with a pair of bitlines that is initially precharged high. When a wordline goes high, the memory cell in that row pulls down one of its bitlines which determines the value stored in the cell (*bitline delay*). If a bitline is shared between multiple columns or subarrays, a column multiplexor is used to choose the relevant bit lines for the particular SRAM access



(The numbered shaded blocks correspond to the modifications required over a conventional cache design.)

Figure 3.4 Implementation of reconfigurable caches.

(*column multiplexor delay*). A sense amplifier is used to detect which line goes low and determine the value in the memory cell (*sense amplifier delay*). For the tag array, the information read from the various ways is compared to the tag bits of the address (*comparator delay*). The results of these comparisons are used to drive a valid output signal that indicates whether there is a hit or a miss (*valid output driver delay*). If the cache access was a hit, in a set associative cache, the results of the comparisons also choose the correct data from the data array (*select* and *data output delays*). Depending on the cache configuration, the critical path could be either through the tag array or through the data array. The cache access time is the sum of the delays on the critical path (obtained by adding the appropriate delay components discussed above).

3.3.2 SRAM Partitioning for Reconfigurability

As discussed above, current cache implementations already partition the cache array into multiple subarrays. The partitioning required for reconfigurable caches can therefore naturally exploit this structure. A key difference between a conventional and reconfigurable cache, however, is that different partitions in the latter need to be indexed by different addresses. Therefore, the cache ways corresponding to different partitions must be implemented in physically different subarrays of the cache; i.e., there must be at least as many subarrays as the maximum number of partitions. In comparison, in a conventional set-associative cache, for a given set, all or some of the ways of the set can potentially be implemented within a single subarray. Thus, it may not be possible to implement a reconfigurable cache with the number and dimensions of subarrays that would have been optimal for a conventional cache.

In general, a reconfigurable cache will require an equal or greater number of subarrays than a non-reconfigurable cache. An increase in the number of subarrays can have the following effects: (i) if the number of subarrays is larger, the wire area and delay required to connect them will be larger, and (ii) a larger number of subarrays results in a reduction in the number of bits in each subarray, which means the individual subarray access times are reduced. The increase in wiring delay with increasing numbers of subarrays tends to balance the decreasing delay with decreasing numbers of bits per subarray. This yields a delay curve vs. the number of subarrays that has an optimum value. For balanced designs, the curve tends to have a fairly shallow minimum, in that configurations with similar numbers of subarrays tend to be close to the optimum. This means that a modest increase in the number of subarrays to support reconfiguration over a design point that is optimum for a non-reconfigurable design usually results in only a modest increase in delay.

One alternative to using a different organization of subarrays would have been to increase the number of ports on each subarray. However, additional ports can be very expensive: for example going from a single ported memory to a true dual-ported memory can almost double the required cache area and significantly increase its access time and power dissipation. Thus we do not further consider the provision of reconfigurability through additional cache ports in this work.

3.3.3 Additional Logic for Reconfigurability

Multiplexors at the address decoders. Each address decoder needs to be preceded by a multiplexor (marked 1 in Figure 3.4) that selects the correct address to forward to a subarray, based on the current partitioning specified in the cache status register. The additional multiplexor delay increases the total decoder delay time. Additionally, the latency of the previous stage is increased because of the extra capacitive load imposed by the multiplexor drain capacitances. The decoder driver also needs to be duplicated at the output of each multiplexor.

Multiplexors at the tag comparators. Analogous to the multiplexers before the address decoders, multiplexors need to be added before the inputs of the tag comparators to route the tag bits from the correct input address (marked 2 in Figure 3.4). However, this typically does not affect the critical path of the cache access as the delay for the tag bits from the input addresses to reach the comparators is less than the delay for the tag information to arrive from the tag subarrays.

Additional multiplexor drivers and output drivers. The reconfigurable cache needs to generate multiple hit/miss signals and send back multiple data elements to the processor (one for each currently invoked partition). Therefore, the outputs of the comparators now need to drive multiple multiplexor drivers and output drivers (marked 3, 4, and 5 in Figure 3.4) corresponding to the various partitions.

Additional wiring. Additional wiring needs to be routed from the cache to the processor for the data paths for the various partitions. This is similar to the data wires for the original cache. It will add to the complexity of routing, and modestly increase the area, delay, and power of the cache.

3.3.4 Impact on Cache Access Time

We use version 2.0 of the CACTI (Cache access and cycle time) model [102][7] to study the impact of the reconfigurable cache organization on the cache access time of the system. The CACTI model analytically models the cache access and cycle times for cache organizations, given the cache configuration and technology parameters, and has been validated with comparisons to SPICE simulations. To study reconfigurable caches, we modify the CACTI timing model to reflect the changes discussed in Sections 3.3.2 and 3.3.3.

The CACTI release used in this thesis integrates several improvements to the timing model used in the previous release including (i) tuned transistor widths to improve the access time of the model, (ii) support for multiple ports and process technology scaling, (iii) split comparators in the tag path, and (iv) variable-location output drivers to balance the data and tag path delays. Consequently, the results in this thesis supersede the results presented in our earlier work using a preliminary release of the CACTI simulator [98].

Table 3.2 summarizes our results for cache sizes of 128KB and 1MB for a 0.13μm process technology (our results for other process technologies were qualitatively similar). For each cache size, three columns are shown for associativity 2, 4, and 8 respectively. For each set-associative configuration, the number of partitions is varied from 2 to the maximum number of partitions allowed (equal to the associativity of the cache), in powers of two.

Our results show that for the various cache sizes, associativities, number of partitions, and process technologies studied, changing existing cache organizations to a reconfigurable cache organization can increase the cache access time by anywhere between 2% to 8%. For small numbers of partitions (2-way), reconfigurable caches usually increase the access time of the non-configurable baseline cache by less than 2-6% (4-6% for a 128KB cache and 2-

[7]The CACTI model assumes a physically-addressed cache with a two-region circuit model approximation; a more aggressive model could use a virtually addressed cache with TLB support and more complex timing models such as a four-region model [77]. However, for the first-level cache configuration we study in this work, we do not anticipate these changes to make a qualitative difference to our results.

	128KB caches, 64B lines					
	8-way		4-way		2-way	
	ns	Δ	ns	Δ	ns	Δ
base	1.94	-	1.55	-	1.43	-
2-part	2.01	4%	1.63	5%	1.51	6%
4-part	2.05	6%	1.64	6%	-	-
8-part	2.10	8%	-	-	-	-

	1MB caches, 64B lines					
	8-way		4-way		2-way	
	ns	Δ	ns	Δ	ns	Δ
base	4.72	-	3.95	-	3.54	-
2-part	4.80	2%	4.01	2%	3.64	3%
4-part	4.80	2%	4.03	2%	-	-
8-part	5.07	7%	-	-	-	-

The *base* configuration represents the conventional non-configurable system; configurations *2-part*, *4-part*, and *8-part* represent systems with 2, 4, and 8 partitions respectively. Δ gives the access time increase over a conventional non-reconfigurable cache organization.

Table 3.2 Reconfigurable cache access times for 0.13μm technology.

3% for a 1MB cache). For larger numbers of partitions (4-way and 8-way), reconfigurable caches suffer a relatively larger cache access time penalty, particularly for smaller cache sizes (6-8% for the 128KB cache and 2-7% for the 1MB cache). Our results also show that most of the impact on the cache access time stems from the additional logic for reconfigurability (Section 3.3.3). The impact of the added constraint on the number of sub-arrays (discussed in Section 3.3.2) is felt only for reconfigurable cache organizations with larger number of partitions.

Sensitivity to cache size. The increase in cache access time is higher for the smaller cache sizes. As the cache size is increased, the access time increases, and consequently, the impact of reconfigurable caches is a smaller fraction of the total access time. Also, due to the large number of subarrays usually required in larger caches, the change required to support reconfiguration is reduced.

Sensitivity to number of partitions. Across all our configurations, the cache access time increases with the number of partitions because of increased delays in driving additional loads at the multiplexors. However, with small number of partitions, the cache access time for reconfigurable caches is not significantly greater than that of non-reconfigurable designs.

3.3.5 Transistor-count and Energy Tradeoffs

Transistor count. As discussed in Section 3.3.3, the changes for reconfigurable caches require adding additional logic for reconfigurability, leading to an increase in the on-chip SRAM transistors. Specifically, this increase consists of three components.

(i) The transistors required for the address-bit multiplexors and the added decoder drivers. This is given by the function:
$((N_{dwl}N_{dbl} - 1) \times DecoderDriverTransistors \times AddressBits)$
$+((N_{twl}N_{tbl} - 1) \times DecoderDriverTransistors \times AddressBits)$
$+(AddressBits \times 2 \times NumberPartitions)$.

(ii) The transistors required for the tag bit multiplexor at the tag comparator.
$A \times TagBits \times NumberPartitions.$

(iii) The transistors required for the additional mux-drivers and output-drivers.
$(NumberPartitions - 1) \times \frac{8BA}{b_0} \times MuxDriverTransistors$
$+(NumberPartitions - 1) \times 8BA$
$+(NumberPartitions - 1) \times InverterTransistors.$

N_{dwl}, N_{dbl}, N_{twl}, and N_{tbl} represent the cache sub-array division parameters used in the CACTI model [127]. B,A, and b_0 represent the cache line size, the associativity, and the bits of data read into the cache respectively. Substituting values for a representative cache structure (128KB 4-way associative cache with 2 partitions), these equations yield an increase of 28,553 transistors to implement the reconfigurable cache organization. Given that a 128KB cache structure requires at least 6,291,456 transistors just to implement the on-chip memory structure (assuming 6-transistor SRAM cells), even a conservative estimate of the transistor increase for reconfigurable caches is less than 0.5% of the total transistors currently devoted to caches.

Energy. Reconfigurable cache organizations allow the on-chip SRAM area to be divided into multiple partitions that can be *concurrently* used for different applications other than caching. Consequently, a larger fraction of the on-chip SRAM is likely to be active with reconfigurable caches compared to conventional cache organizations, requiring greater energy dissipation. Results from studying the energy requirements of a reconfigurable cache organization using a preliminary energy model available with CACTI validate this observation. The increased energy dissipation per access is almost directly proportional to the number of partitions. In addition to this metric, the actual power consumed by the system is also dependent on the frequency of use of the various partitions. This frequency metric is a function of the workload's cache behavior and the particular application for which we use reconfigurable cache partitions. Focusing on the average energy required for a default cache access, our results indicate that this is not changed significantly relative to conventional cache organizations in most cases. Again, as with our results characterizing the impact on the access time, most of the increase in energy consumption comes from the additional logic required for reconfigurable caches.

3.3.6 Summary

Our results show that a reconfigurable cache organization can be implemented within the framework of a conventional SRAM cache design with a few minor changes. Our results using an analytical model of the cache access time indicate that for small number of partitions, the reconfigurable cache organization does not incur significant additional delay over traditional cache structures. Larger number of partitions have the potential to increase the cache access time, but this increase is still less than 8% with larger caches. Such an increase may or may not affect the cycle time of the machine or the number of cycles required for a cache access, depending on whether a conventional cache would limit the cycle time of the microprocessor. If the access time of a conventional cache is already somewhat below the latency of the cache (in cycles) times the cycle time of the whole microprocessor, reconfigurability could be added without negatively impacting either the machine cycle time or the number of cycles required for a cache access. Our analysis also shows that the extra logic to implement reconfigurability increases the cache transistor budget by less than 0.5%. However, the greater functionality is achieved at the expense of proportionally greater energy consumption.

3.4 Using Reconfigurable Caches for Instruction Reuse

To provide quantitative evidence of the benefits of reconfigurable caches, we focus on one representative application of such caches. This section evaluates the performance benefits from reconfigurable caches for *instruction reuse* for media processing benchmarks. Instruction reuse has been studied in detail for SPEC benchmarks [39, 70, 71, 107, 110, 111]; however, it has not been studied in detail for media processing. We anticipate that this technique will be effective for media processing applications due to the inherent repetitiveness and incremental gradation associated with the analog data types that correspond to media data. Furthermore, as we showed in Chapter 2, media applications are compute bound (versus memory bound) after the insertion of software prefetching. Instruction reuse coupled with reconfigurable caches allows us to improve computation performance using otherwise underutilized memory system resources.

Section 3.4.1 discusses our instruction reuse implementation. Section 3.4.2 summarizes our simulation environment, simulated system, and benchmarks. Section 3.4.4 presents the performance results.

3.4.1 Implementation of Instruction Reuse

Originally proposed instruction reuse buffer implementation

We use an instruction reuse buffer implementation similar to that proposed by Sodani and Sohi [110] and later refined by Molina et al. [81]. This scheme (referred to as S_v in [110]) detects reuse based on operand values and was shown to be the best performing scheme with large buffer sizes. In Sodani and Sohi's study, the processor is assumed to have a fixed-sized 12-ported reuse buffer and buffer sizes from 0.5KB to 12KB are considered. An instruction entry in the reuse buffer stores the source and destination operand values and part of the PC to identify the instruction. When an instruction is decoded, its source operand values are compared with those in the instruction reuse buffer entry corresponding to this instruction. If there is a match, the result is directly read from the buffer and the execution stage is bypassed. (Refer Section 2.3.1 for discussion of the pipeline stages.) The performance benefits from instruction reuse are thus achieved from reductions in the execution stages of the pipeline; the number of instructions issued per cycle continues to be bounded by the issue width of the processor. Load and store addresses as well as load values are stored in the instruction reuse buffer. To ensure consistency of data, on a store, the implementation in Sodani's study [110] requires a fully associative lookup of the reuse buffer for possible matching entries.

Instruction reuse buffer implementation with reconfigurable caches

When implementing an instruction reuse buffer as a partition of a reconfigurable cache, we need to make a few changes to the above design to address (i) the limited number of ports for the cache (4 cache ports vs. 12 ports in the instruction reuse buffer), (ii) the larger SRAM sizes and the inability to perform associative lookups, and (iii) increased latencies (we assume that the cache access takes 2 cycles as opposed to one cycle for a smaller dedicated instruction reuse buffer). Consequently, our implementation of the S_v scheme only stores values for arithmetic and logical instructions and addresses for memory instructions.

Other instruction values (including branch outcomes and load values) are not stored in the instruction reuse buffer. This reduces the number of accesses to the instruction reuse partition. For a further reduction, on a hit in the instruction reuse buffer, we do not update the state information for replacements. The buffer is however updated on misses. For each instruction, we store as many entries (sets of values) as fit in the cache line (with 64-byte cache lines, five entries can be stored and associatively checked for each instruction).

3.4.2 Simulation Framework

As in Chapter 2, we use the RSIM simulator [90] to evaluate the benefits from reconfigurable caches. Our simulated system is similar to the system studied in Chapter 2 except that we model an eight-way issue processor and scale the number of functional units and other structures suitably. (We don't expect the results to change qualitatively from the 4-way system we considered earlier.) We focus on a state-of-the-art out-of-order processor model that includes support for all the aggressive optimizations discussed in the last chapter including non-blocking loads and stores, speculative execution, media ISA extensions, memory disambiguation, and software prefetching. As before, the simulator supports the SPARC v9 ISA and Sun's VIS media ISA extensions. Tables 3.3 and 3.4 summarize the parameters used for the processor and memory subsystems in this chapter.

The base cache system uses a 128KB four-way associative first-level (L1) write-back data cache and a 1MB 4-way associative second-level (L2) write-back unified cache. The caches are non-blocking and allow support for multiple outstanding misses. At each cache, 12 miss status holding registers (MSHRs) reserve space for outstanding cache misses and combine a maximum of 8 multiple requests to the same cache line.

For the reconfigurable cache system, we partition the L1 data cache into two two-way associative partitions of 64KB each. As seen from Figure 3.2, the access time for this configuration is 1.63ns, 5% slower than a design without reconfigurability. One of the partitions is used as a conventional data cache while the other is used as an instruction

Processor parameters	
Processor speed	1 GHz
Issue width	8
Instruction window size	64
Functional units	
- integer arithmetic	4
- floating point	4
- address generation	4
- VIS multiplier	2
- VIS adder	2
Branch prediction	
- bimodal agree predictor size	2K
- return address stack size	32
Taken branches per cycle	1
Simultaneous speculated branches	16
Memory queue size	32

Table 3.3 Default processor parameters for reconfigurable cache characterization.

reuse buffer (as described in Section 3.4.1) for the entire run of all the benchmarks. (We choose the first-level data cache, as opposed to the first-level instruction cache, to minimize the complexity of routing the interconnects for instruction reuse. For other optimizations such as using reconfigurable caches for branch prediction, it may be simpler to partition the instruction caches.)

3.4.3 Applications Studied

Our benchmarks represent media processing on a variety of media data types, including images, video, audio, and speech. They include image and video source coding workloads as before, but also include speech and audio decoding and kernels from speech recognition and synthesis. Table 3.5 summarizes the benchmarks we study in this chapter. A brief description of the benchmarks is given below.

Memory hierarchy parameters	
Cache line size	64 bytes
L1 instr cache size	64KB
L1 instr cache associativity	2-way
L1 data cache size	128KB (base)
L1 data cache associativity	4-way (base)
L1 data cache request ports	4
L1 data cache hit time	2 ns
Number of L1 MSHRs	12
L2 cache size (on-chip)	1MB
L2 cache associativity	4-way
L2 request ports	1
L2 hit time (pipelined)	20 ns
Number of L2 MSHRs	12
Total contentionless memory latency for L2 misses	100 ns
Memory interleaving	4-way

Table 3.4 Default system parameters for reconfigurable cache characterization.

Image coding (*cjpeg* and *djpeg*). The image coding benchmarks are the same as the ones studied in Chapter 2 and use the non-progressive JPEG codec (Release 6a) available from the Independent JPEG group. More details on the JPEG source encoding and decoding is discussed in Section 2.1.

Video coding (*mpegenc* and *mpegdec*). The video coding benchmarks are similar to the benchmarks studied in Chapter 2 and use the MPEG2 version 1.1 codec available from the MPEG Software Simulation Group (MSSG). More details on the MPEG encoding and decoding is discussed in Section 2.1.

Speech and audio decoding (*speechdec* and *audiodec*). In this study, we study the GSM 06.10 speech decoding [29] and MPEG audio decoding [92, 51] applications.

The GSM codec we used is based on the publicly-available codec developed by Jutta Degener and Carsten Bormann at the Technische Universitat Berlin. Our benchmark implements a regular-pulse-excitation long-term-predictor (RPE-LTP) full-rate speech

Benchmark	Description (input)
cjpeg	JPEG encoding of 1024x630 3-band image (rose16.ppm), uses VIS and software prefetching
djpeg	JPEG decoding of 1024x630 3-band image (rose16.jpg), uses VIS and software prefetching
mpegdec	MPEG2 video decoding of video stream into YUV components (meil6v2rec.m2v), uses VIS and software prefetching
mpegenc	MPEG2 video encoding of four 352x240 frames (I-B-B-P) (meil6v2rec.yuv), uses VIS
speechdec	GSM speech decoding (clinton.pcm)
audiodec	MPEG-2 audio decoding (clinton.pcm)
spchrecog	Signal cepstral feature extraction in speech recognition (ex5_c1.wav)
spchsynth	Natural language processing in speech synthesis (test_data.in)

Table 3.5 Media processing benchmarks used for reconfigurable cache characterization.

transcoder. The key idea is to use linear predictive coding to model the human speech system as (1) an excitation source (larynx) that produce either a white source noise (unvoiced speech) or a train of pulses separated by a pitch period (voiced speech) and (2) a series of acoustic filters formed by the vocal tract (mouth, nose, etc.). The encoder divides the speech signal into short-term predictable parts, long-term predictable parts, and the remaining residual pulse. It then quantizes and encodes that pulse and parameters for the two predictors. The decoder (the benchmark we focus on in this work) reconstructs the speech by passing the residual pulse through the long-term prediction filter, and passes the output through the short-term predictor. The input stream we use is `clinton.pcm` from the UCLA MediaBench suite.

Our MPEG audio codec is based on version 3.9 of the MPEG-1 codec released by the MPEG/audio software simulation group. Unlike the GSM coder, the MPEG coder gets its compression without making assumptions about the nature of the audio source, instead exploiting perceptual properties of the human auditory system. The MPEG encoding application passes the input audio stream through a filter bank that divides the input stream into multiple sub-bands. The input audio stream simultaneously passes through a psychoacoustic model that determines the signal-to-mask ratio for each sub-band. Based on these ratios, code bits are apportioned for each of the subband signals to minimize the audibility of the quantized noise (auditory masking). Our MPEG audio decoding benchmark interprets the input bit stream and reconstructs the quantized sub-band values and finally transforms the set of subband values into a time-domain audio signal. We use the Layer II algorithm for our experiments. We use `clinton.pcm` from the UCLA MediaBench suite as the input stream.

Speech recognition and synthesis (*spchrecog* and *spchsynth*).

Speech recognition [105] is the process of converting an acoustic signal captured by microphone to a set of words which can be used as input for other applications. Speech recognition consists of two main processes – (1) representation of the speech signal and (2) modeling and classification of the produced representation. Our speech recognition benchmark focuses on the former. Specifically, our benchmark is based on the RASTA program available from the International Computer Science Institute, Berkeley. This benchmark uses perceptual linear prediction to parameterize the speech input based on cepstral co-efficients attributable to the shape of the vocal tract. We run the benchmark with the RASTA front-end filtering technique that allows for removal of additive noise and spectral distortion. We use the `ex5_c1.wav` input file from the UCLA MediaBench suite.

Speech synthesis [33] is the process of converting text to speech and consists of two parts - (1) the natural language processing, and (2) the digital signal processing. Our speech synthesis benchmark focuses on the former. Specifically, our benchmark is based on the alpha version of the FreeSpeech text-to-speech program with the MBROLA British English synthesis system released as part of the Festival project at the University of Edinburgh, Scotland. The program goes through several stages to convert English text to phonetic input for a synthesizer. First, input text is examined and classified into sentences made up of individual (content or function) words. A phonetic transcription is then constructed based on a set of letter-to-sound rules. Once the word level transcriptions are established, a sentence level prosodic contour is calculated. The phoneme list, a set of durations and a set of prosodic targets are then output in a format suitable as input for the MBROLA English synthesizer. We use the the file `test_data.in` containing the first 13 lines of the README file of the distribution.

All the benchmarks were modified to include a call to reconfigure the caches at the beginning of the execution.

3.4.4 Performance Results

Figure 3.5 summarizes the results for all the benchmarks. For each benchmark, four bars are shown representing (i) the base system with a conventional data cache (*base*), (ii) the base system with a reconfigurable L1 data cache with two partitions – one for conventional data cache and the other for instruction reuse (*IR*), (iii) the reconfigurable cache system with "infinite" ports and partition size for the instruction reuse partition (*IRideal*), (iv) the base conventional cache system with half the L1 cache size and associativity (*L1–*), i.e., with the L1 cache available in the reconfigurable cache configuration but without the benefit of the instruction reuse partition. The bars show execution cycles (or equivalently inverse of the IPC) normalized to the cycles (inverse of IPC) of the *base* configuration. Each bar is divided further into five components – busy cycles, functional unit (FU) stall cycles, read hit stall cycles, read miss stall cycles, and other stall cycles (e.g., instruction cache miss stalls). The busy and stall cycles are calculated using the following convention, similar to that used in the previous chapter. At every cycle, the fraction of instructions retired that cycle to the maximum retire rate is attributed to the busy time; the remaining fraction is attributed as stall time to the first instruction that could not be retired that cycle.

Comparing the *base* and *IR* configurations, we find that a realistic implementation of instruction reuse with reconfigurable caches achieves IPC improvements of 1.04X to 1.20X across our applications. The benefits are on reductions in the functional unit stall and read

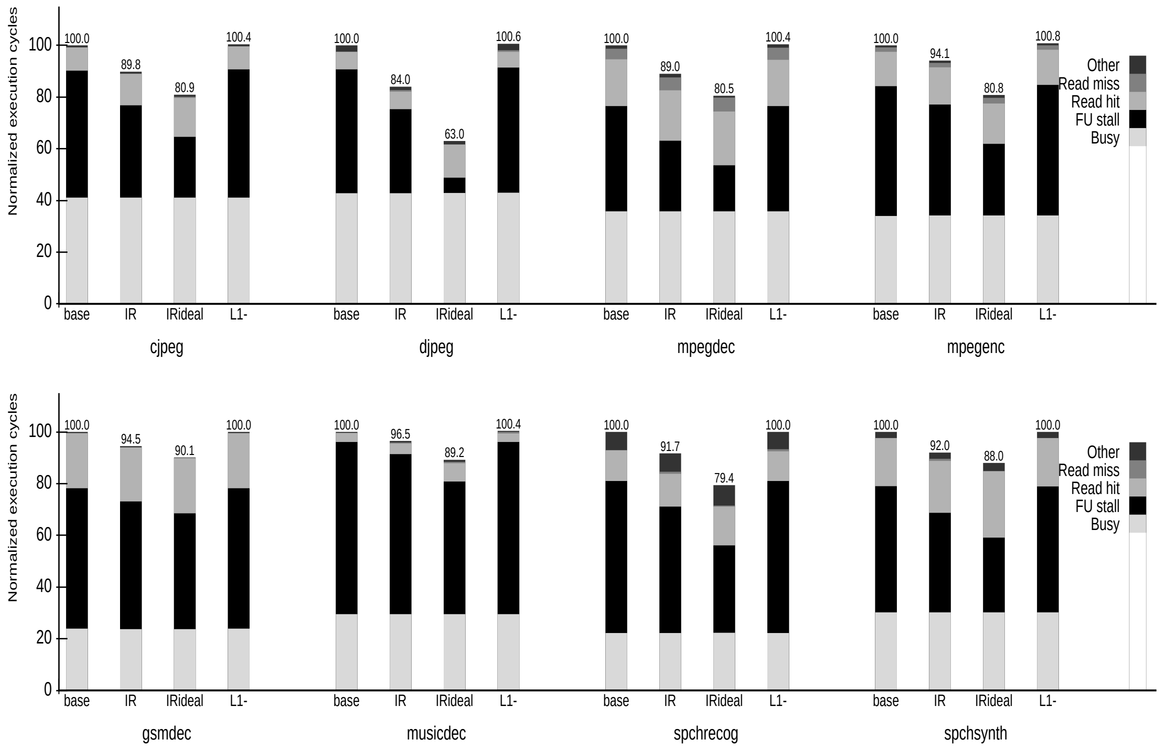


Figure 3.5 Performance benefits from using reconfigurable caches for instruction reuse.

hit stall components of the execution cycles, enabled by bypassing the execution stage due to hits in the instruction reuse buffer.

Comparing the *base* and *IRideal* systems, the ideal instruction reuse configuration with “infinite” resources achieves IPC improvements ranging from 1.11X to 1.60X. The differences between the *IRideal* configuration and the practical implementation of the instruction reuse buffer (*IR*) stem mainly from increased port contention for the SRAM partitions used for the instruction reuse buffer (all the SRAM partitions have the same number of ports as the original SRAM cache – in our experiments, four). Enhancing the *IR* model with additional filtering mechanisms (e.g., confidence based filtering [45]) may address the increased port contention and reduce the performance differences between the *IR* and *IRideal* models. However, this may involve extra hardware support. In this work, we restrict ourselves to reconfigurable cache organizations with marginal changes to the processor and do not study such aggressive implementations.

Finally, comparing the *base* and *L1–* results corroborates previous results that large caches are not critical for media processing workloads. Across all our applications, changing the first-level cache from a 4-way 128KB cache to a 2-way 64KB cache had marginal impact (less than 1%) on performance.

3.5 Related Work

Albonesi has concurrently proposed the use of "selective cache ways" to selectively disable parts of the data arrays of the cache to tradeoff performance for power conservation [5]. This work also proposes to use set-associativity based partitioning and addressing of the data arrays, and cache scrubbing for data consistency when partitions are shut down. These methods are similar to those used in our primary design. Our work, however, differs from Albonesi's work in several significant ways. First, in contrast to simply turning off some partitions, our work suggests using the partitions for alternate processor activities to enhance performance. This requires a more general cache design that allows multiple address inputs into and multiple data outputs out of the cache, requiring care to keep these multiple paths from contending with each other. Some of our design decisions may actually increase the power consumption in the cache arrays. Second, we perform a detailed timing analysis with the CACTI analytical model to determine the impact of our design on cache access time. Albonesi's evaluation is focused on power dissipation. Third, we evaluate the performance benefits of the reconfigurable cache organization for instruction reuse [110] for media processing. Thus, we show that cache storage can be used to improve computation speed for media processing. Albonesi focuses on shutting off parts of the cache to save power, and focuses on SPEC benchmarks.

Kim et al. have concurrently proposed a reconfigurable multi-function computing cache architecture [59] that can be used both as a functional unit and as a cache. The key difference between this approach and our reconfigurable cache structure is the granularity at which the reconfigurability operates. Specifically, our reconfigurable cache organization continues to use the cache transistors as higher-level SRAM storage, but allows them to be reused for any other processor and memory-system optimization that requires SRAM storage. In comparison, Kim et al.'s approach requires implementing the cache structure as a two-dimensional matrix of multi-bit look-up tables connected by a global "reconfigurable multiple bus network." Reconfigurability is achieved by reconnecting the look-up tables internally to implement functional units like DCT and convolution. Their results indicate that such a reconfigurable structure can achieve high performance improvements with reasonable increases in area (10-20%) and small impact on cache access time (1%-2%). However, these performance benefits are achieved with significantly more radical changes to the cache structure and require a more complicated programming model.

The Morph project [131] proposes integrating programmable logic throughout the memory hierarchy (processor, cache, and memory interconnection fabric) giving applications control over memory interleaving, cache organization, and caching policies. These reconfigurable logic resources allow the system to construct and manage an adaptive cache memory similar to our reconfigurable caches, but requires significantly more radical changes to the processor and cache design. Additionally, the key focus of the adaptive cache optimizations in Morph deal with the movement and placement of application data throughout the memory hierarchy. In comparison, as discussed in Section 3.1, our reconfigurable cache structure can be used with other processor optimizations as well.

Although we have focused mainly on reconfigurable cache organizations for general-purpose processors, they are applicable to digital signal processors as well. The recently announced Texas Instruments TMS320C62xx series of processors include support for memory systems that can be configured as cache or memory depending on the application. Similarly, the newly announced multiprocessor DSP architecture from Bell Labs and Lucent Technologies [16] allows the on-chip 8KB cache to be dynamically configured as instruction cache, data cache, or local buffer memory. Several other studies have also discussed the use of on-chip cache resources for software-controlled memory. John et al. [55] presents some results which indicate the performance potential of using portions of the cache as local memory, but do not address the mechanism to achieve that. Chiou et al. discuss a "malleable cache" [21] design that uses controlled line replacement to implement software control of cache structures. The Smart Memories project [74] exploits the inherent hierarchical design of SRAMs to implement a reconfigurable memory block that can be configured to implement different associativities of cache organization as well as implement scratch pad memories and local register files. As discussed in Section 3.1, reconfigurable caches can be used for these purposes as well. However, our reconfigurable cache organization is general enough to include other applications for the SRAM arrays including instruction reuse.

Other reconfigurable approaches like the Impulse Project [20], the Galileo project [15], the Stream Memory Controller [78], and other studies [54, 125] also propose adaptive cache designs to address the problem of inefficient cache usage. However, these approaches primarily focus on application-specific mechanisms to either (i) dynamically reorder memory accesses to exploit parallelism and locality, and/or (ii) change cache organization, replacement, and cache line size policies. Along the same lines, Albonesi discusses how the boundary between the first-level and second-level caches can be reconfigured dynamically to better explore the tradeoffs between access time and IPC [4]. Our reconfigurable cache approach, in comparison, addresses inefficient use of memory resources by *reusing* the cache transistors for other, possibly non-memory-system-related, optimizations.

Several recent studies have examined instruction reuse. While our work is the first to study the performance benefits from instruction reuse for media processing workloads, the key contribution of our work is in the idea of reconfigurable caches and not instruction reuse *per se*. Similarly, several recent studies have proposed new architectures targeted specifically for media processing applications (e.g., [66, 104]). As opposed to such special-purpose architectures, the focus of our work is on designing architectures that can adaptively reuse general-purpose features for different activities for different applications, including media processing applications.

3.6 Summary

This chapter proposes a new cache organization called reconfigurable caches. Reconfigurable caches provide the ability to divide the cache SRAM arrays into different partitions that can be used for different processor activities. These activities can benefit applications that could not otherwise exploit conventional caches. Our design requires very few modifications to conventional caches, exploiting the natural implementation of set-associative caches today. Detailed timing analysis using a modification of the CACTI model shows small impact on cache access time.

We suggest several applications of reconfigurable caches. To show quantitative benefits, we choose to evaluate instruction reuse for media processing as a representative application. Instruction reuse coupled with reconfigurable caches allows us to improve computation performance using otherwise underutilized memory system resources. For a simple design which uses one of the partitions of the reconfigurable cache for instruction reuse, while keeping the other partition for conventional caching, we achieve IPC improvements between 4% and 20%. Because the cache access time impact of reconfigurability is so low (5% for the organization used in the IPC simulations) it is likely that the overall microprocessor cycle time or the number of cycles required to access the cache will not be affected. Even if the microprocessor cycle time is increased by the cache access time increase, the net performance benefits still remain positive. Furthermore, comparisons with a more ideal implementation indicate that more aggressive implementations of instruction reuse can achieve higher performance benefits. Additionally, using more partitions for other activities can further improve the performance benefits from this organization. It is important to note that the benefits achieved in this work were with relatively small hardware and software changes to current general-purpose processors.

In the future, we anticipate that the paradigm of reusing on-chip storage for multiple processor activities can facilitate a number of architectural optimizations. Chapter 6 discusses some possible future research directions for extending this work.

Chapter 4

Database Workloads

This chapter presents a detailed simulation study of database workloads running on shared-memory multiprocessors based on next-generation out-of-order processors.

As discussed in Chapter 1, database workloads such as online transaction processing (OLTP) and decision-support systems (DSS) have quickly surpassed scientific and engineering workloads to become the largest market segment for multiprocessor servers. However, this dramatic change in the target market for shared-memory servers is yet to be fully reflected in the design of these systems. Most current system designs have been optimized to perform well on scientific and engineering workloads. For example, commodity processors are primarily optimized to perform well on the SPEC benchmark suite [113], and system designs are focused on scientific and engineering benchmarks such as STREAMS [76] and SPLASH-2 [128]. However, it is not clear if these design decisions are suitable for emerging database workloads. The goal of this chapter is to evaluate the effectiveness of current system features for database workloads and suggest architectural enhancements to improve the performance of future designs.

We use detailed simulation to study both online-transaction processing (OLTP) and decision-support systems (DSS) running on the commercial Oracle database engine (version 7.3.2). We present a thorough analysis of the benefits of techniques such as out-of-order execution and multiple issue in database applications, evaluate the tradeoffs between memory consistency models supported on these systems, and identify simple solutions that further optimize the performance of the more challenging OLTP workload.

Section 4.1 describes the workloads that we study. Sections 4.2 and 4.3 discuss the simulated architectures and simulation methodology. Sections 4.4 to 4.8 discuss the performance benefits of aggressive ILP techniques for online transaction processing and decision-support system workloads on shared-memory database servers. Sections 4.9 and 4.10 discuss the performance benefits from simple optimizations that can further improve the performance of the OLTP application. Section 4.11 discusses other related work relevant to the results presented in this chapter.

4.1 Applications Studied

We use the Oracle 7.3.2 commercial database management system as our database engine. Figure 4.1 illustrates the different components of Oracle. In addition to the server processes that execute the actual database transactions, Oracle spawns a few daemon processes that perform a variety of duties in the execution of the database engine. Two of these daemons, the database writer and the log writer, participate directly in the execution of transactions. The database writer daemon periodically flushes modified database blocks that are cached in memory out to disk. The log writer daemon is responsible for writing transaction logs to disk before it allows a server to commit a transaction.

Client processes communicate with server processes through pipes, and the various Oracle processes (i.e., daemons and servers) communicate through a shared memory region called the System Global Area (SGA). The SGA consists of two main regions - the block buffer area and the metadata area. The block buffer area is used as a memory cache of database disk blocks. The metadata area is used to keep directory information for the block buffer, as well as for communication and synchronization between the various Oracle processes. Additionally, all the Oracle processes also have their own private data segment called the program global area (PGA).

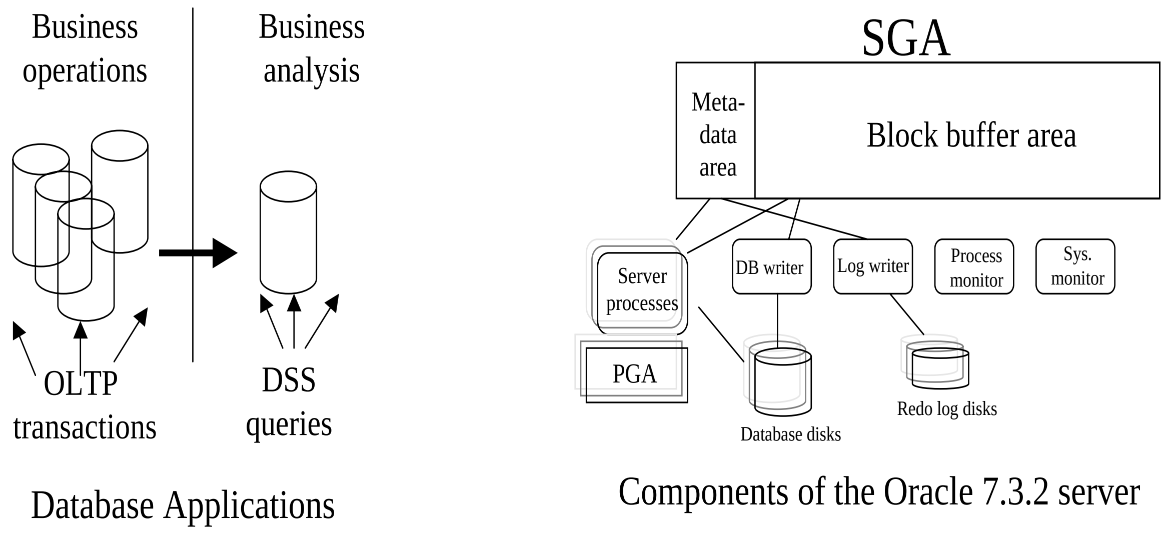


Figure 4.1 Major components of the Oracle server.

4.1.1 OLTP Workload

Our OLTP application is modeled after the TPC-B benchmark from the Transaction Processing Performance Council (TPC) [121]. TPC-B models a banking database system that keeps track of customers' account balances, as well as balances per branch and teller. Each transaction updates a randomly chosen account balance, which includes updating the balance of the branch the customer belongs to and the teller from which the transaction is submitted. It also adds an entry to the history table which keeps a record of all submitted transactions.

The application was extensively tuned in order to maximize transaction throughput and CPU utilization. For OLTP, we run Oracle in "dedicated mode," in which each client process has a dedicated Oracle server process to execute database transactions. The server process communicates with the client process through a Unix pipe. Each transaction consists of five queries followed by a commit request. To hide I/O latencies, OLTP runs are typically scheduled with multiple server processes per processor.

We chose to use TPC-B instead of TPC-C (the current official transaction processing benchmark from TPC) for a variety of reasons. First, TPC-B has much simpler setup requirements than TPC-C, and therefore lends itself better for experimentation through simulation. Second, our performance monitoring experiments with TPC-B and TPC-C show similar processor and memory system behavior, with TPC-B exhibiting somewhat worse memory system behavior than TPC-C [9]. As a result, we expect changes in processor and memory system features to affect both benchmarks in similar ways. Finally, it is widely acknowledged that actual customer database applications will typically show poorer performance than TPC-C itself. Stets et al. discuss these similarities and differences between TPC-B and TPC-C in greater detail [116].

4.1.2 DSS Workload

The DSS application is modeled after Query 6 of the TPC-D benchmark [122]. The TPC-D benchmark represents the activities of a business that sells a large number of products on a worldwide scale. It consists of several inter-related tables that keep information such as parts and customer orders. Query 6 scans the largest table in the database to assess the increase in revenue that would have resulted if some discounts were eliminated. The behavior of this query is representative of other TPC-D queries [9].

Since the time this study was performed, the transaction processing council has split the TPC-D benchmark into two separate benchmarks, TPC-R (business reporting) and TPC-H (ad-hoc querying). This split was in response to new technology from several database vendors to build structures that contain pre-computed query aggregates (similar to indexes) optimized for a business reporting environment. The TPC-R benchmark represents decision support queries in such environments. The TPC-H benchmark represents decision support environments where users don’t know which queries will be executed against a database system and consequently do not use auxiliary structures such as indexes and aggregates. To the best of our knowledge, the version of the database that we used does not support pre-computed query aggregates. Consequently, we expect our characterization results to most closely match the behavior of the TPC-H benchmark. (As of the date of this thesis, TPC-H continues to be the benchmark of choice for database vendors when reporting DSS workloads. NCR is the only vendor that has reported TPC-R results.)

For DSS, we used Oracle with the Parallel Query Optimization option, which allows the database engine to decompose the query into multiple sub-tasks and assign each one to an Oracle server process. The queries were parallelized to generate four server processes per processor (16 processes in a 4-processor system).

4.2 Simulated Architecture

As before, we use the RSIM simulator [90] to simulate a hardware cache-coherent non-uniform memory access (CC-NUMA) shared-memory multiprocessor system using an invalidation-based, four-state MESI directory coherence protocol.[8] To ensure practical simulation times, we only model a system with four nodes. Each node in our simulated system includes a processor, separate first level data and instruction caches, a unified second-level cache, a portion of the global shared-memory and directory, and a network interface. A split-transaction bus connects the network interface, directory controller, and the rest of the system node. The system uses a two-dimensional wormhole-routed mesh network.

The L1 data cache is dual-ported, and uses a write-allocate, write-back policy. The unified L2 cache is a fully pipelined, write-allocate write-back cache. In addition, all caches are non-blocking and allow up to 8 outstanding requests to separate cache lines. At each

[8]We use different simulated architectures and simulation methodologies for the the database and media processing workloads owing to (1) the differences in the systems available for the two markets, (2) the differences in the behavior of the two workloads, and (3) the constraints on the availability and simulation of commercial realistic applications. Specifically, our results with database workloads study the performance of Oracle 7.3.2 database engine compiled for the Alpha ISA. The simulations use a trace-driven methodology to model multiple server processes running on a shared-memory database server, but ensure that important operating system and I/O effects are carefully modeled. The media processing workloads use user-level execution-driven simulation to study media processing applications compiled for the SPARC ISA on uniprocessor general-purpose systems; virtual memory and operating system effects are not modeled because they are not critical for these workloads.

cache, miss status holding registers (MSHRs) [61] store information about the misses and coalesce multiple requests to the same cache line. All caches are physically addressed and physically tagged. The virtual memory system uses a bin-hopping page mapping policy with 8K page sizes, and includes separate 128-element fully associative data and instruction TLBs.

Our base system models an out-of-order processor with support for multiple issue, out-of-order instruction execution, non-blocking loads, and speculative execution. We use an aggressive branch prediction scheme that consists of a hybrid *pa/g* branch predictor for the conditional branches [117], a branch target buffer for the jump target branches, and a return address stack for the call-return branches. In the event of branch mispredictions, we do not issue any instructions from after the branch until the branch condition is resolved; our trace-driven methodology precludes us from executing the actual instructions from the wrong-path.

Tables 4.1 and 4.2 summarize the other important parameters used in our base system. To study the effect of the individual techniques as well as the relative importance of various performance bottlenecks, we vary many of these parameters in our experiments. Specifically, we study both in-order and out-of-order processor models, and the effect of instruction window size, issue width, number of outstanding misses, branch prediction, number of functional units, and cache size on the performance.

Both the in-order and out-of-order processor models support a straightforward implementation of the Alpha consistency model [32] (hereafter referred to as release consistency [RC] [43] for ease of notation), using the Alpha `MB` and `WMB` fence instructions to impose ordering at synchronization points. The out-of-order processor model also supports implementations of sequential consistency (SC) [63] and processor consistency (PC) [44], and optimized implementations for these consistency models. These are further described in Section 4.7.

Processor parameters	
Processor speed	1 GHz
Issue width	4 (default)
Instruction window size	64 (default)
Functional units	
- integer arithmetic	2
- floating point	2
- address generation	2
Branch prediction	
- conditional branches	*PA(4K,12,1)/g(12,12)*
- jmp branches	512-entry 4-way BTB
- call-returns	32-element RAS
Simultaneous speculated branches	8
Memory queue size	32

Table 4.1 Default processor parameters for database workload characterization.

4.3 Simulation Methodology

Due to the difficulty of running a commercial-grade database engine on a user-level simulator (such as RSIM), our strategy was to use traces of the applications running on a four-processor AlphaServer4100, and drive the simulator with those traces. This trace-driven simulation methodology is similar to that used by Lo et al. [72].

The traces were derived with a custom tool built using ATOM [112]. The workloads used to generate the traces were compiled with the maximum optimizations turned on with the Alpha compiler for an in-order Alpha 21164-based system. Only the Oracle server processes were traced since the many daemon processes have negligible CPU requirements. However, the behavior of the daemons with respect to synchronization and I/O operations was preserved in the traces. All blocking system calls were marked in the traces and identified as hints to the simulator to perform a context switch. The simulator uses these hints to guide context switch decisions while internally modeling the operating system scheduler. The simulation includes the latency of all I/O and blocking system calls. The values for

Memory hierarchy	
Cache line size	64 bytes
Number of L1 MSHRs	8
L1 data cache size (on-chip)	128 KB
L1 data cache associativity	2-way
L1 data cache request ports	2
L1 data cache hit time	1 cycle
L1 instruction cache size (on-chip)	128 KB
L1 instruction cache associativity	2-way
L1 instruction cache request ports	2
L1 instruction cache hit time	1 cycle
L2 cache size (off-chip)	8M
L2 cache associativity	4-way
L2 request ports	1
L2 hit time (pipelined)	20 cycles
Number of L2 MSHRs	8
Data TLB	128 entries, full-assos
Instruction TLB	128 entries, full-assos
Contentionless memory latencies	
Memory type	*Latency (in processor cycles)*
Local read	100
Remote read	160-180
Cache-to-Cache read	280-310

Table 4.2 Default memory system parameters for database workload characterization.

these latencies were determined by instrumenting the application to measure the effect of the system calls on an Alpha multiprocessor.

The trace also includes information regarding Oracle's higher-level synchronization behavior. The values of the memory locations used by locks are maintained in the simulated environment. This enables us to correctly model the synchronization between processes in the simulated environment and avoid simulating spurious synchronization loops from the trace-generation environment. Our results show that most of the lock accesses in OLTP were contentionless and that the work executed by each process was relatively independent of the order of acquisition of the locks. DSS shows negligible locking activity.

One trace file was generated per server process in the system. The total number of instructions simulated was approximately 200 million for both OLTP and DSS. Warmup transients were ignored in the statistics collection for both the workloads.

Scaling and Validation

We followed the recommendations of Barroso et al. [9] in scaling our workloads to enable tracing and simulation. Specifically, we carefully scaled down our database and block buffer sizes while continuing to use the same number of processes per processor as a full-sized database. We use an OLTP database with 40 branches and an SGA size over 900MB (the size of the metadata area is over 100MB). The DSS experiments use an in-memory 500MB database. The number of processes per CPU was eight for OLTP and four for DSS. Similar configurations have been used in several other studies [7, 8, 72].

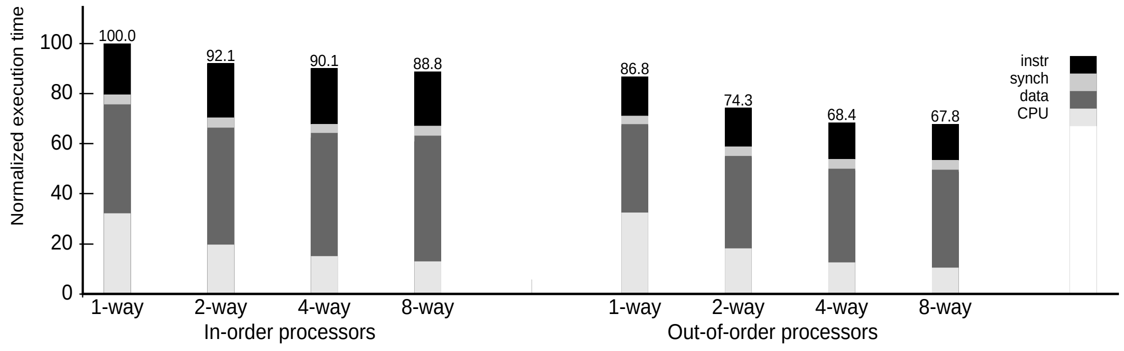
In the past, transaction processing applications were reported to be mainly I/O bound and to have a dominant component of their execution time in the operating system. Today, a modern database engine can tolerate I/O latencies and incurs much less operating system overhead; the operating system component for our tuned workloads (measured on the AlphaServer4100) was less than 20% of the total execution time for the OLTP workload and negligible for the DSS workload. Since our methodology uses user-level traces, we do not take into account the non-negligible operating system overheads of OLTP. (Note that, as discussed earlier, we do model the effect of the virtual memory system, the process scheduler, and the stall times for system calls.) However, as reported in Barroso et al. [9], the execution behavior of Digital Unix running this OLTP workload is very similar to the user-level behavior of the application, including CPI, cache miss ratios, and contributions of different types of misses. Therefore, we expect that the inclusion of operating system activity would not change the manner in which our OLTP workload is affected by most of the optimizations studied here.

Significant care was taken to ensure that the traces accurately reflect the application behavior, and that the simulated execution reproduces the correct interleaving of execution and synchronization behavior of the various processes. We configured our simulator to model a configuration similar to that of our server platform and verified that the cache behavior, locking characteristics, and speedup of the simulated system were similar to actual measurements of the application running on our server platform. We also modeled configurations similar to those used in previous studies[9, 72] and verified our statistics with those reported in those studies.

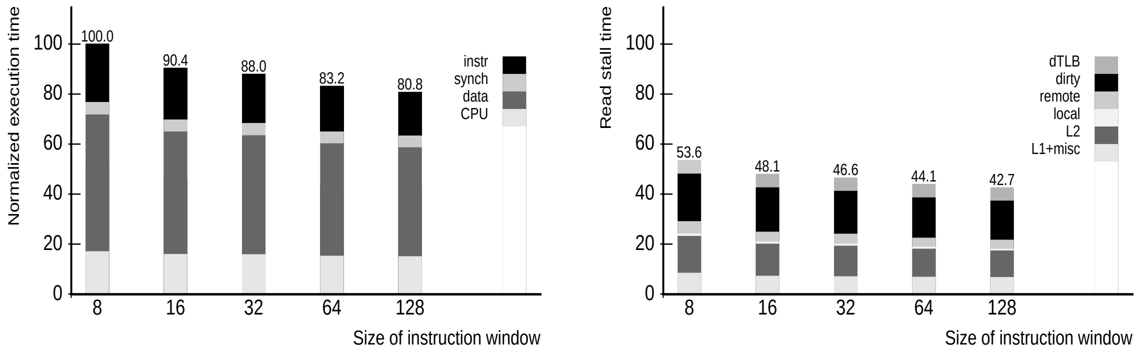
4.4 Performance Benefits from ILP Features

Figures 4.2 and 4.3 present our results for OLTP and DSS respectively. Part (a) of each figure compares multiprocessor systems with in-order and out-of-order processors with varying issue widths. Part (b) shows the impact of increasing the instruction window size for the out-of-order processor. Part (c) shows the impact of supporting multiple outstanding misses. The bars in each graph represent the execution time normalized to that of the leftmost bar in the graph. We factor out the idle time in all the results; the idle time is less than 10% in most cases. We further breakdown execution time into CPU (both busy and functional unit stalls), data (both read and write), synchronization, and instruction stall (including instruction cache and iTLB) components. Additionally, the bars on the right hand side in parts (b) and (c) show a magnification of the read stall time corresponding to the bars on the left hand side. The read stall time is divided into L1 hits plus miscellaneous stalls (explained below), L2 hits, local and remote memory accesses (serviced by memory), dirty misses (i.e., cache-to-cache transfers), and data TLB misses. The base results assume a release-consistent system, therefore there is little or no write latency. Section 4.7 discusses the performance of stricter consistency models.

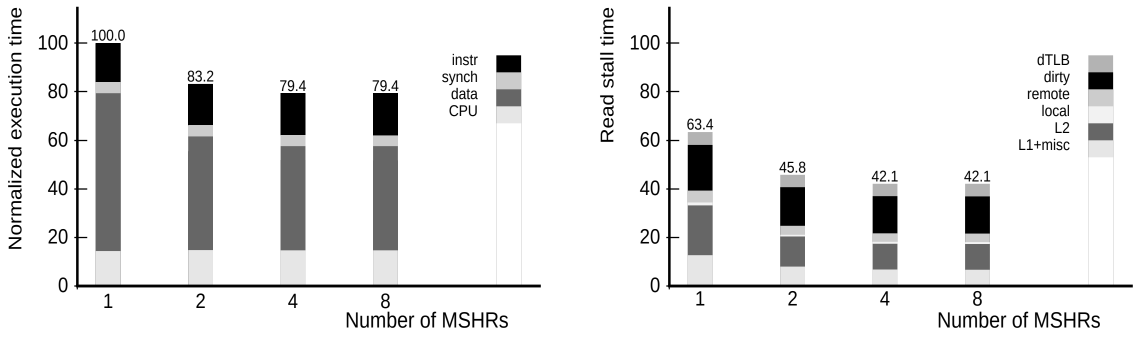
We use the same methodology as before to divide the execution time into its various stall components (discussed in Section 2.4). For memory instructions, there are extra stall cycles that may be attributed to the instruction in addition to the time spent in the execution stage: (i) time spent in the address generation stage that cannot be hidden due to data de-



(a) Effect of multiple issue and out-of-order execution (window size=64; maximum outstanding misses=8)

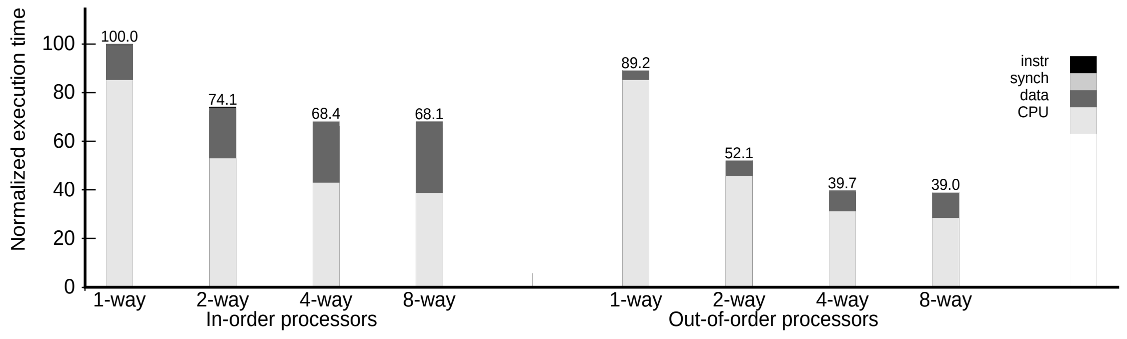


(b) Effect of varying instruction window size on out-of-order processors (issue width=4, maximum outstanding misses=8)

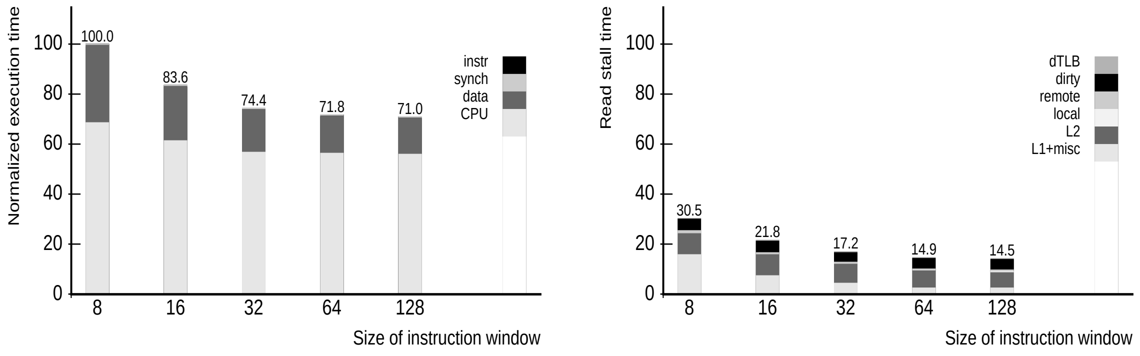


(c) Effect of varying multiple outstanding misses on out-of-order processors (window size=64, issue width=4)

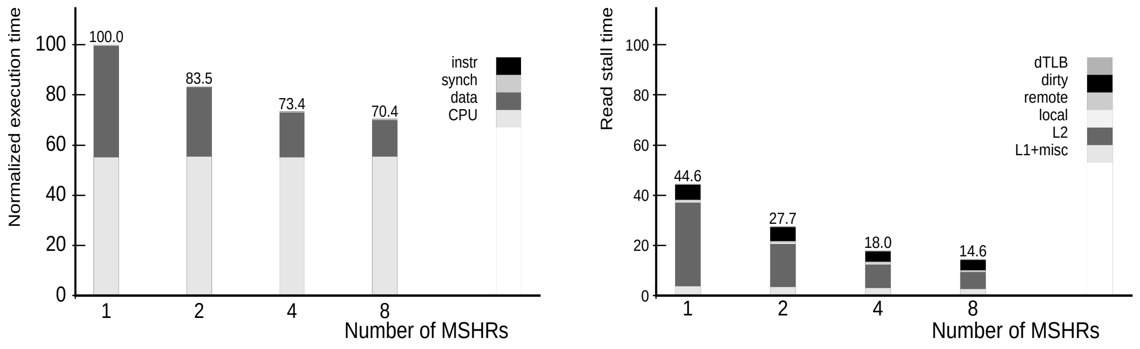
Figure 4.2 Impact of ILP features on OLTP performance.



(a) Effect of multiple issue and out-of-order execution (window size=64; maximum outstanding misses=8)



(b) Effect of varying instruction window size on out-of-order processors (issue width=4, maximum outstanding misses=8)



(c) Effect of varying multiple outstanding misses on out-of-order processors (window size=64, issue width=4)

Figure 4.3 Impact of ILP features on DSS performance.

pendence or resource contention, and (ii) pipeline restart time after a branch misprediction, instruction cache miss, or exception when the memory instruction is at the head of the window. While these stalls are included for all memory categories, their impact is particularly visible in the “L1+misc” component because the base L1 latency is only one cycle. Similar conventions have been adopted by many previous studies (e.g., [2, 89, 88, 101, 106]).

Overall, our results show that the OLTP workload is characterized by a significant L2 component due to its large instruction and data footprint. In addition, there is a significant memory component arising from frequent data communication misses. For OLTP, the local miss rates for the first-level instruction and data caches and the second level unified cache are 7.6%, 14.1% and 7.4% respectively. In comparison, the main footprint for the DSS workload fits in the large L1 caches (128K), and the memory component is much smaller relative to OLTP; DSS is more compute intensive and benefits from spatial locality on L2 misses. The local miss rates for DSS are 0.0% and 0.9% for the first-level instruction and data caches and 23.1% for the second level cache. These observations are consistent with those reported in previous studies [9, 72].

The results further indicate that support for multiple issue, out-of-order execution, and multiple outstanding loads provide significant benefits for OLTP and DSS, even though the benefits for OLTP are smaller in comparison to DSS. Most of the gains are achieved by a configuration with four-way issue, an instruction window of 32 to 64 entries, and a maximum of four outstanding cache misses (to unique cache lines). Interestingly, many current processors are in fact more aggressive than this configuration. For example, the HP-PA 8000 supports a fifty-six entry instruction window and ten outstanding misses. The Alpha 21264 supports an eighty entry instruction window and eight outstanding misses.

4.4.1 OLTP Workload

Multiple Issue

Going from single- to eight-way issue, in-order processors provide a 12% improvement in execution time while out-of-order processors provide a 22% improvement (Figure 4.2(a)). The benefits from multiple issue stem primarily from a reduction in the CPU component. Out-of-order processors allow more efficient use of the increased issue width. However, the benefits for in-order and out-of-order processors level off at two-way and four-way issue (respectively).

Out-of-order Execution

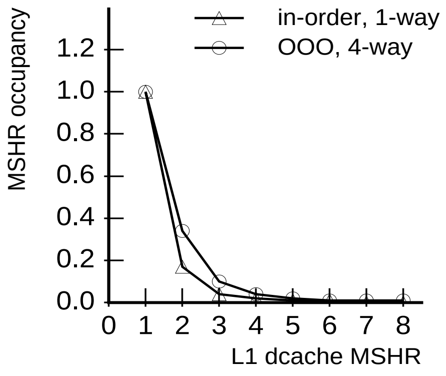
Comparing equivalent configurations with in-order and out-of-order processors in Figure 4.2(a), we see that out-of-order execution achieves reductions in execution time ranging from 13% to 24%, depending on the issue width. The combination of multiple issue and out-of-order execution interact synergistically to achieve higher performance than the sum of the benefits from the individual features.

The performance improvements due to out-of-order execution stem from reductions in the instruction and data stall components of execution time. The decoupling of fetch and execute stages in out-of-order processors enables part of the instruction miss latency to be overlapped with the execution of previous instructions in the instruction window. Similarly, the latency of a data load operation can be overlapped with other operations in the instruction window. In contrast, the amount of overlap in in-order processors is fundamentally limited since these systems require the processor to stall at the first data dependence that is detected.

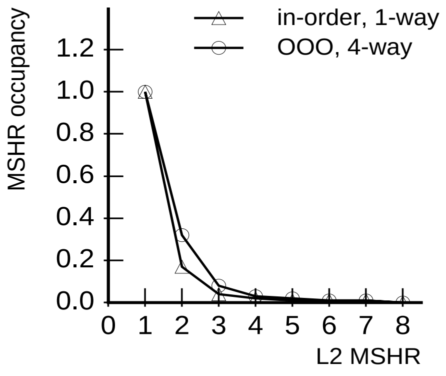
Increasing the instruction window size increases the potential for overlap in out-of-order processors. As seen from Figure 4.2(b), performance improves as the instruction window size is increased, but levels off beyond 64 entries. A large fraction of this improvement comes from the L2 cache hit component (the read stall time graph of Figure 4.2(b)).

Multiple Outstanding Misses

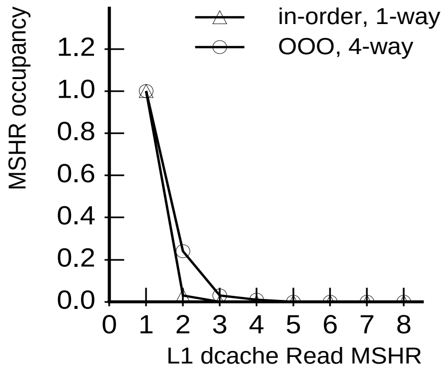
Figure 4.2(c) summarizes the impact of increasing the number of outstanding misses. The various bars show the performance as the number of MSHRs is increased. For OLTP,



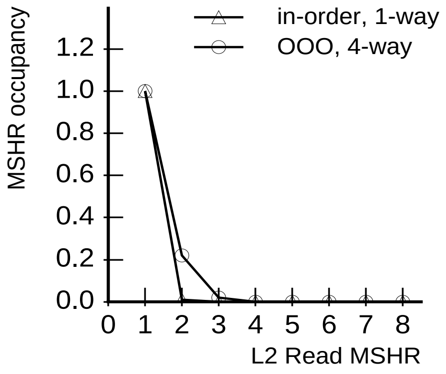
(a) L1 dcache overlap



(b) L2 overlap



(c) L1 dcache read overlap



(d) L2 read overlap

Figure 4.4 MSHR occupancy distributions for OLTP workloads.

supporting only two outstanding misses achieves most of the benefits. This behavior is consistent with frequent load-to-load dependences that we have observed in OLTP.

To further understand this result, Figures 4.4(a)-(d) display MSHR occupancy distributions. The distribution is based on the total time when at least one miss is outstanding, and plots the fraction of this time that is spent when at least *n* MSHRs are in use. Figures 4.4(a) and (b) present the distributions for all misses at the first-level data and second-level unified caches, while Figures 4.2(c) and (d) correspond to only read misses.

Figures 4.4(c)-(d) indicate that there is not much overlap among read misses, suggesting that the performance is limited by the data-dependent nature of the computation. By comparing Figures 4.4(a) and (b) (both read and write misses) with Figures 4.4(c)-(d) (read misses), we observe that the primary need for multiple outstanding misses stems from writes that are overlapped with other accesses. Overall, we observe that there is a small increase in the MSHR occupancy when going from in-order to out-of-order processors which correlates with the decrease in the read stall times seen in Figure 4.2(c).

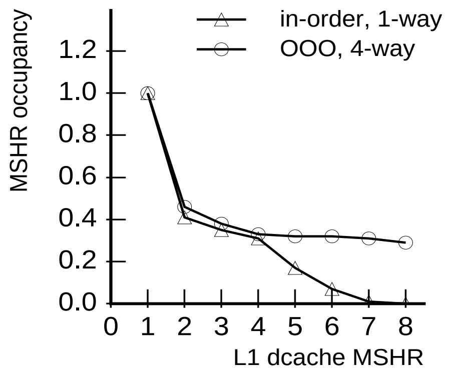
4.4.2 DSS Workload

Figure 4.3 summarizes the results for DSS. Overall, the improvements obtained from the ILP features are much higher for the DSS workload than for OLTP. This results from the compute-intensive nature of DSS that leads to little memory stall time. Clearly, the ILP features are more effective in addressing the non-memory stall times. Going from single issue to 8-way issue achieves a 32% reduction in execution time on in-order processors, and a 56% reduction in execution time on out-of-order processors (Figure 4.3(a)). Out-of-order issue achieves reductions in execution time ranging from 11% to 43% depending on the issue width. As with OLTP, there is synergy between out-of-order and multiple issue. Performance levels off for instruction window sizes beyond 32. Figures 4.2(c) and 4.3(c) indicate that the DSS workload can exploit a larger number of outstanding misses (4 compared to 2 in OLTP). Figures 4.5(a)-(d) indicate that this is mainly due to write overlap. The high write overlap arises because of the relaxed memory consistency model we use which allows the processor to proceed past write misses without blocking.

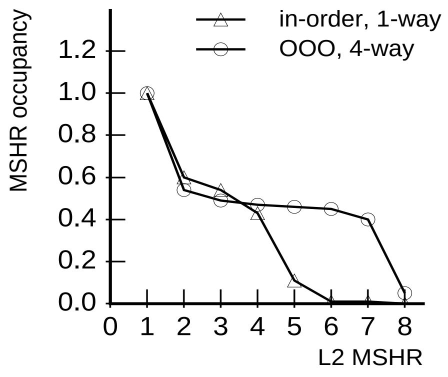
4.5 Limitations of the ILP Features

4.5.1 OLTP Workload

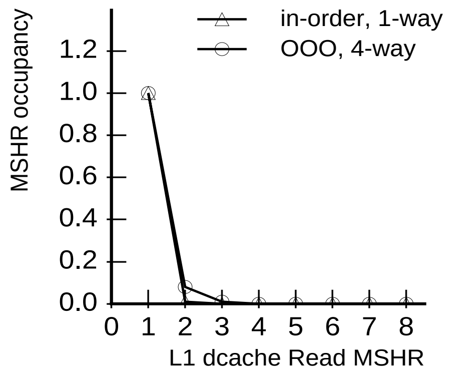
Although the aggressive ILP features discussed above significantly improve OLTP performance, the execution time is still dominated by various stall components, the most impor-



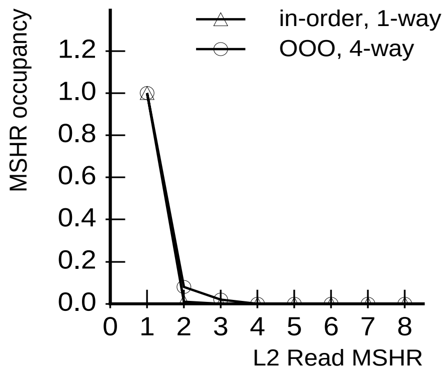
(a) L1 dcache overlap



(b) L2 overlap



(c) L1 dcache read overlap



(d) L2 read overlap

Figure 4.5 MSHR occupancy distributions for DSS workloads.

tant of which are instruction misses and dirty data misses. This leads to a low IPC of 0.5 on the 4-way out-of-order processor (henceforth used as the base system).

We next determine if enhancement of any processor features can alleviate the remaining stall components. Our results are summarized in Figure 4.6. The left-most bar represents the base 4-way out-of-order configuration; subsequent bars show the effect of infinite functional units, perfect branch prediction, and a perfect instruction cache. The last bar represents a system with twice the instruction window size (128 elements) along with in-

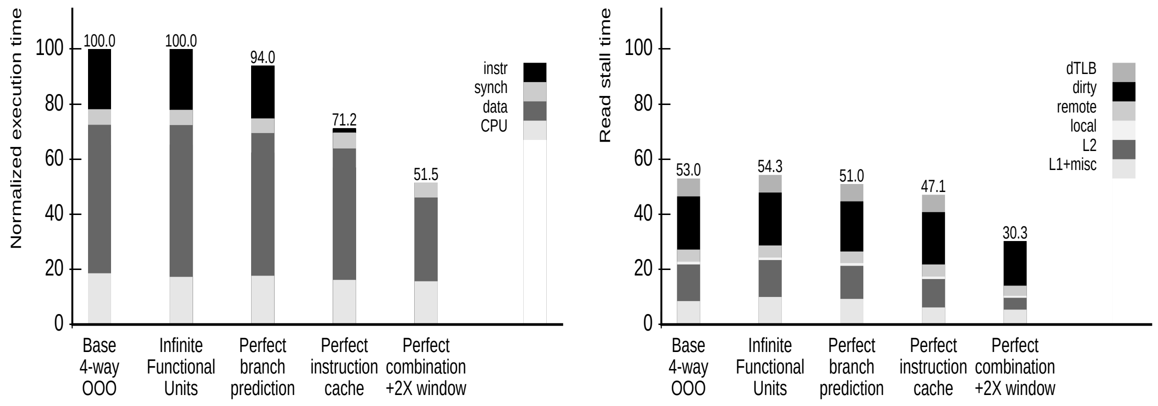


Figure 4.6 Factors limiting OLTP performance.

finite functional units and perfect branch prediction, instruction cache, and i- and d-TLBs (Figure 4.2(b) shows the performance of the system when the instruction window size alone is doubled).

The results clearly show that functional units are not a bottleneck. Even though OLTP shows a cumulative branch misprediction rate of 11%, perfect branch prediction gives only an additional 6% reduction in execution time. Frequent instruction misses prevent the branch prediction strategy from having a larger impact. Not surprisingly, an infinite instruction cache gives the largest gain, illustrating the importance of addressing the instruction stall time. Increasing the instruction window size on a system with infinite functional units, perfect branch prediction, perfect instruction cache, and perfect TLB behavior (rightmost bar in Figure 4.6) allows for more synergistic benefits. The L2 stall component is further diminished, leaving dirty miss latencies as the dominant component. Sections 4.9 and 4.10 discusses techniques to address both the instruction stall and the dirty miss components.

4.5.2 DSS Workload

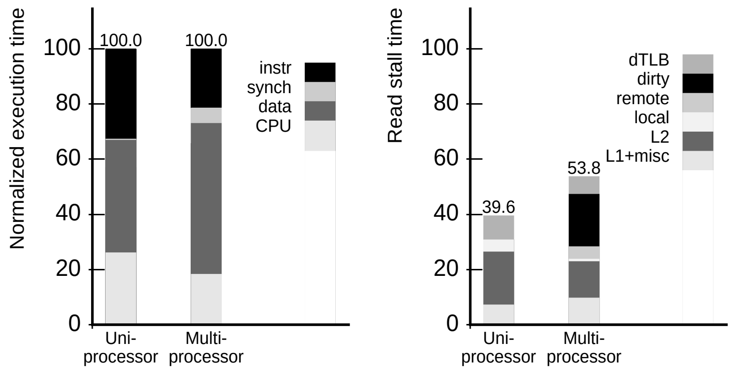
As discussed before, the DSS workload experiences very little stall time due to its highly compute-intensive nature and its relatively small primary working set. The IPC of the DSS workload on our base 4-way out-of-order system is 2.2. The main limitation in this application is the number of functional units. Increasing the number of functional units (to 16 ALUs and 16 address generation units; floating point units are not used by the workload) results in a 12% performance improvement. For our default configuration, improving other parameters like the branch prediction strategy used, the instruction cache size, and the TLB sizes do not make any significant impact.

4.6 Comparison with Uniprocessor Systems

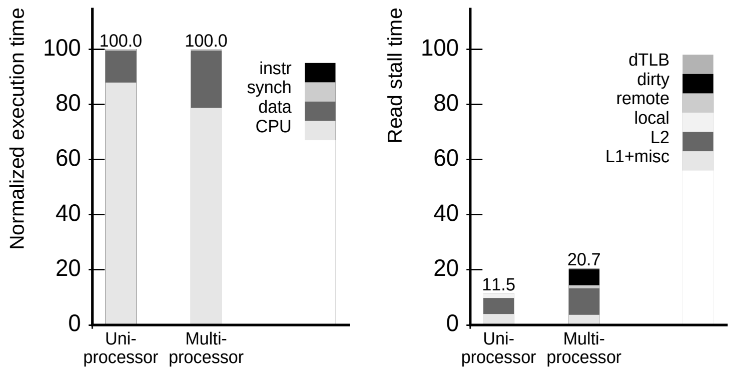
Our experiments show that uniprocessor systems achieve benefits quantitatively similar to multiprocessors from ILP features for both DSS and OLTP. However, it is interesting to compare the sources of stall time in uniprocessor and multiprocessor systems. Figure 4.7 presents the normalized execution times for our base 4-way uniprocessor and multiprocessor systems. For OLTP, the instruction stall time is a significantly larger component of execution time in uniprocessors since there are no data communication misses. For both workloads, multiprocessors bring about larger read components as expected.

4.7 Performance Benefits from ILP-Enabled Consistency Optimizations

Features such as out-of-order scheduling and speculative execution also enable hardware optimizations that enhance the performance of memory consistency models. These optimized implementations exploit the observation that the system must only *appear* to execute memory operations according to the specified constraints of a model.



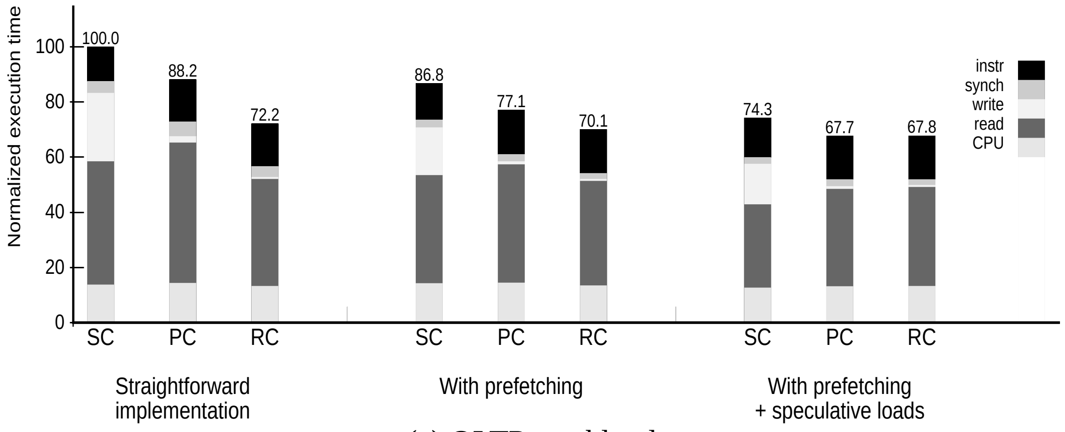
(a) OLTP workload



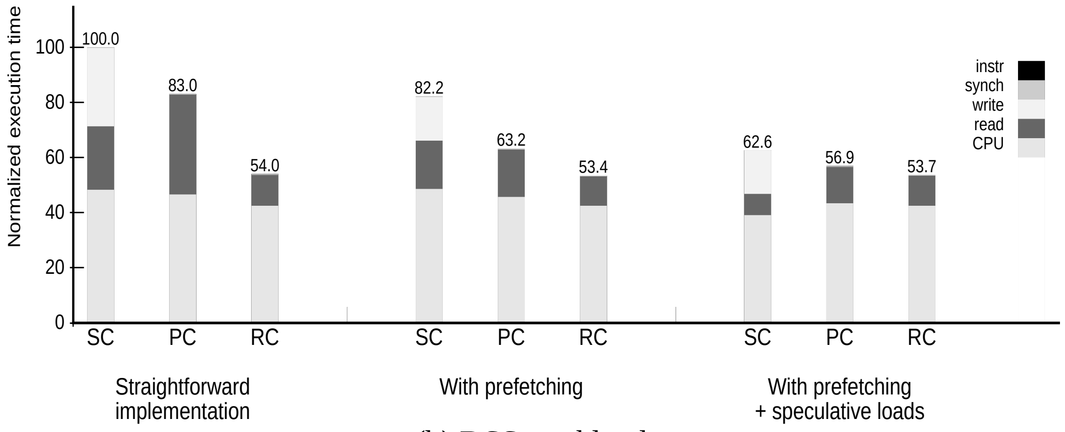
(b) DSS workload

Figure 4.7 Relative importance of components in uniprocessor and multiprocessor systems for database workloads.

The technique of *hardware prefetching* from the instruction window [41] issues non-binding prefetches for memory operations whose addresses are known, and yet are blocked due to consistency constraints. *Speculative load execution* [41] increases the benefits of prefetching by allowing the return value of the load to be consumed early, regardless of consistency constraints. The latter technique requires hardware support for detecting violations of ordering requirements due to early consumption of values and for recovering from such violations. Violations are detected by monitoring coherence requests and cache



(a) OLTP workload



(b) DSS workload

Figure 4.8 Performance benefits from ILP-enabled consistency optimizations for database workloads.

replacements for the lines accessed by outstanding speculative loads. The recovery mechanism is similar to that used for branch mispredictions or exceptions. Both of the above techniques are implemented in a number of commercial microprocessors (e.g., HP PA8000, Intel Pentium Pro, and MIPS R10000).

Figure 4.8 summarizes the performance of three implementations of consistency models – a straightforward implementation, another with hardware load and store prefetching, and a third that also uses speculative loads. The figure shows the performance of sequential consistency (SC), processor consistency (PC), and release consistency (RC) for each of the implementations. Execution times are normalized to the straightforward implementation of SC. The data stall component of execution time is further divided into read and write stall times.

Our results show that the optimizations have a large impact on the performance of the stricter models (SC and PC), and a relatively small impact on RC (as expected). While prefetching alone achieves some benefits, the data-dependent nature of computation allows for even greater benefits with speculative loads. For example, with both prefetching and speculative loads, the execution time of SC is reduced by 26% for OLTP and by 37% for DSS, bringing it to within 10% and 15% of the execution time of RC. In contrast, without these optimizations, RC achieves significant reductions in execution time compared to SC (46% for DSS and 28% for OLTP). Given that these optimizations are already included in several commercial microprocessors, our results indicate that the choice of the hardware consistency model may not be a key factor for out-of-order systems running database workloads (especially OLTP). In comparison, previous studies based on the same optimizations have shown a significant performance gap (greater than 15%) between SC and RC for scientific workloads [91]. The final choice of consistency models for these systems, however, will also depend on the impact of relaxed consistency models on compiler optimizations, which still remains an open research question [2].

4.8 Summary of ILP Benefits

Techniques such as multiple issue, out-of-order execution, and multiple outstanding loads provide significant benefits for both OLTP and DSS. The gains for DSS are more substantial as compared with OLTP. OLTP has a large memory system component that is difficult to hide due to the prevalence of dependent loads. Most of the benefits are achieved by a four-way issue processor with a window size of 32 to 64 and a maximum of four outstanding cache misses. Furthermore, ILP features present in current state-of-the-art processors allow optimized implementations of memory consistency models that significantly improve the performance of stricter consistency models (by 26-37%) for database workloads, bringing their performance to within 10-15% of more relaxed models.

Section 4.5 showed that the instruction stall time and stall time due to read dirty misses are two primary bottlenecks that limit the performance of our OLTP workload. The next two sections further analyze these bottlenecks and evaluate simple solutions to address them.

4.9 Addressing the Instruction Bottleneck

Our analysis of the instruction stall time in OLTP identifies two key trends. First, the instruction stall time is dominated by first-level cache misses that hit in the second-level cache. This stems from the fact that the instruction working set of OLTP is about 560KB which fits in the large 8M second-level cache, but overwhelms the 128K first-level cache. Second, a significant portion of the instruction references follow a streaming pattern with successive references accessing successive locations in the address space. These characteristics suggest potential benefits from adding a simple instruction stream buffer between the first and second level caches.

A *stream buffer* is a simple hardware-based prefetching mechanism that automatically initiates prefetches to successive cache lines following a miss [56]. To avoid cache pollution due to the prefetched lines, the hardware stores the prefetched requests in the stream buffer until the cache uses the prefetched data. In the event of a cache miss that does not hit in any of the entries in the stream buffer, the hardware flushes all the entries in its buffer and initiates prefetches to the new stream.

Our results show that a simple stream buffer is very effective in reducing the effects of instruction misses in OLTP. A 2-element instruction buffer is successful in reducing the miss rate of the base 4-way out-of-order system by almost 64%. A 4-element stream buffer reduces the remaining misses by an additional 10%. Beyond 4 elements, the stream buffer

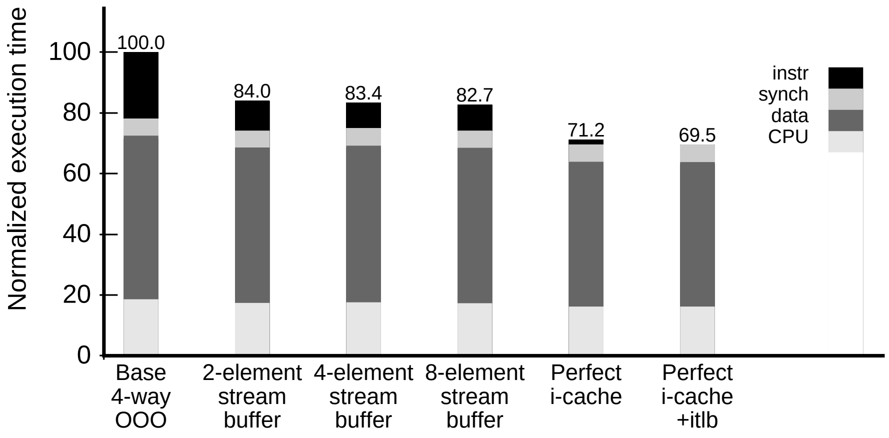


Figure 4.9 Addressing instruction misses in OLTP.

provides diminishing returns for two reasons. First, the misses targeted by the stream buffer decrease, since streams in OLTP are typically less than 4 cache lines long. Second, additional stream buffer entries can negatively impact performance by causing extra contention for second-level cache resources due to useless prefetches.

Figure 4.9 compares the performance of our base 4-way out-of-order system to systems including stream buffers of size 2, 4, and 8 elements. As an upper bound on the performance achievable from this optimization, we also include results for a system with a perfect instruction cache, and a system with a perfect instruction cache and perfect instruction TLB.

As shown in Figure 4.9, a 2- or 4-element stream buffer reduces the execution time by 16%, bringing the performance of the system to within 15% of the configuration with a perfect icache. Most of the benefits come from increased overlap of multiple instruction misses at the L1 cache. The improvement in performance from stream buffers is more pronounced in uniprocessor configurations where the instruction stall time constitutes a larger component of the total execution time. Our results for uniprocessors show that stream buffers of size 2 and 4 achieve reductions in execution time of 22% and 27% respectively compared to the base 4-way out-of-order system.

Further characterization of the instruction misses indicate that a large fraction of the remaining misses exhibit repeating sequences, though with no regular strides. Code re-alignment by the compiler, or a predictor that interfaces with a branch target buffer to issue prefetches for the right path of the branch could potentially target these misses. Our preliminary evaluation of the latter scheme indicates that the benefits from such predictors are likely to be limited by the accuracy of the path prediction logic and may not justify the associated hardware costs, especially when a stream buffer is already used.

An alternative to using a stream buffer is to increase the size of the transfer unit between the L1 and L2 caches. Our experiments with a 128-byte cache line size indicate that such an architectural change can also achieve reductions in miss rates comparable to the stream buffers. However, stream buffers have the potential to dynamically adapt to longer stream lengths without associated increases in the access latency, and without displacing other useful data from the cache.

We are not aware of any current system designs that include support for an instruction stream buffer between the L1 and L2 cache levels. Our results indicate that adding such a stream buffer can provide high performance benefits for OLTP workloads with marginal hardware costs.

4.10 Addressing the Data Communication Miss Bottleneck

As shown in Figure 4.7(a), read dirty misses (serviced by cache-to-cache transfers) account for 20% of the total execution time of OLTP on our base 4-way out-of-order system. In addition, dirty misses account for almost 50% of the total misses from the L2 cache. To better understand the behavior of these dirty misses, we performed a detailed analysis of the sharing patterns in Oracle when running our OLTP workload. Our key observations are summarized below.

First, we observed that 88% of all shared write accesses and 79% of read dirty misses are to data that exhibit a migratory sharing pattern. We use the following heuristic to identify migratory data [25, 114]. A cache line is marked as migratory when the directory receives a request for exclusive ownership to a line, the number of cached copies of the line is 2, and the last writer to the line is not the requester. Because our base system uses a relaxed memory consistency model, optimizations for dealing with migratory data such as those suggested by Stenstrom et al. [114] will not provide any gains since the write latency is already hidden. OLTP is characterized by fine-grain updates of meta-data and frequent synchronization that protects such data. As a result, data structures associated with the most actively used synchronization tend to migrate with the passing of the locks.

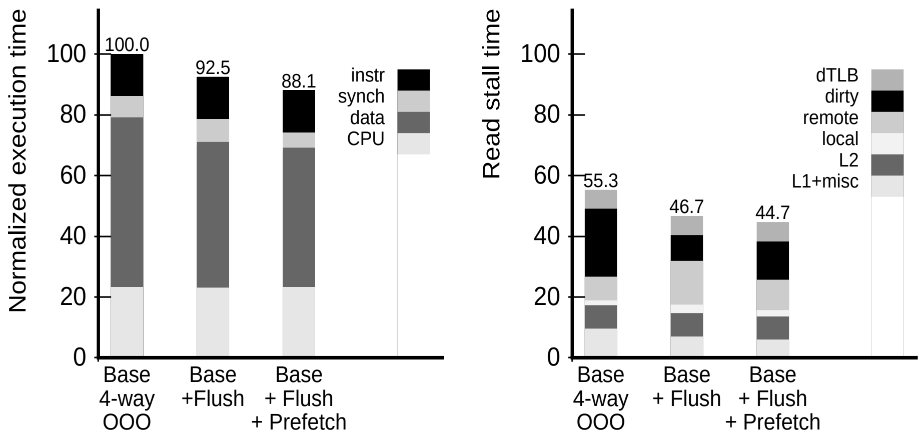
Second, additional characterization of the migratory misses show that they are dominated by accesses to a small subset of the total migratory data, and are generated by a small subset of the total instructions. On our base 4-way out-of-order system, 70% of the migratory write misses refer to 3% of all cache lines exhibiting migratory behavior (about 520 cache lines), and more importantly, 75% of the total migratory references are generated by less than 10% of all instructions that ever generate a migratory reference (about 100 unique instructions in the code). Finally, analysis of the dynamic occurrence of these instructions indicate that 74% of the migratory write accesses and 54% of the migratory read accesses occur within critical sections bounded by identifiable synchronization.

The above observations suggest two possible solutions for reducing the performance loss due to migratory dirty read misses. First, a software solution that identifies accesses to migratory data structures can schedule *prefetches* to the data, enabling the latency to be overlapped with other useful work. Support for such software-directed prefetch instructions already exists in most current processors.

Second, a solution that identifies the end of a sequence of accesses to migratory data can schedule a *“flush”* or *“check-in”* [47, 64] instruction or use “`WriteThrough`” [1, 109] instructions to update the memory with the latest values. Subsequent read requests can then be serviced at memory, avoiding the extra latency associated with a cache-to-cache transfer (for our system configuration, this can reduce the latency by almost 40%). We found that it is important for such flush or `WriteThrough` operations to keep a clean copy in the cache (i.e., not invalidate the copy) since the latency of subsequent read misses would otherwise neutralize the gains. This requires minor support in the cache-coherence protocol since the flush effectively generates an unsolicited "sharing writeback" message to the directory.

Given our lack of access to the Oracle source code, we could not correlate the migratory accesses to the actual data structures. Instead, we used our characterizations to identify the instructions that are likely to generate migratory data accesses to perform a preliminary study. Our experiments involve adding the appropriate prefetch and `WriteThrough` software primitives to the code around some of the key instructions.

Figure 4.10 summarizes our results. All results assume a 4-element instruction stream buffer given the importance of this optimization. The leftmost bar represents the base 4-way out-of-order system with a 4-element stream buffer. The second bar represents adding appropriate flush primitives at the end of specific critical sections. As shown in Figure 4.10, the flush primitive is successful in significantly reducing the impact of dirty misses on the total execution time, achieving a 7.5% reduction in execution time. (The increase in the local and remote components of read latency corresponds to the dirty misses that are converted to misses serviced by the memory.) As an approximate bound on this improvement, we selectively reduced the latency of all migratory read accesses (by 40%) to reflect service by memory; this gave a bound of 9% which is very close to the 7.5% improvement from the flush primitive. Finally, also adding prefetching for migratory data at the beginning of critical sections (rightmost bar) provides a cumulative reduction of 12% in execution time. Late prefetches and contention effects due to the combination of flush and prefetch appear to limit additional performance benefits. A realistic software solution that is aware of the data structures which cause the migratory accesses may be able to avoid both these prob-



(base assumes 4-element stream buffer)

Figure 4.10 Addressing communication misses in OLTP workloads.

lems. Such a software solution is particularly compelling given the amount of code tuning that is currently done by database vendors to improve performance.

4.11 Related Work

Our work performs the first detailed simulation study of database workloads on shared-memory multiprocessors based on state-of-the-art out-of-order processors. Our key contributions include providing a thorough analysis of the benefits of aggressive techniques such as multiple issue, out-of-order execution, non-blocking loads, and speculative execution in the context of database workloads, and identifying simple optimizations that can provide a substantial improvement in performance. Our work is also the first to study the performance of memory consistency models in the context of database workloads.

There are a number of studies based on the performance of out-of-order processors for non-database workloads(e.g., [42, 85, 89]). Most previous studies of databases are based on in-order processors [9, 27, 28, 34, 75, 94, 118, 119], and therefore do not address the benefits of more aggressive processor architectures. A number of the studies are limited to uniprocessor systems [27, 34, 72, 75]. As discussed in Section 4.4, data communication misses play a more dominant role in multiprocessor executions and somewhat reduce the relative effect of instruction stall times.

Another important distinction among the database studies is whether they are based on monitoring existing systems [3, 13, 18, 26, 27, 28, 57, 62, 118] (typically through performance counters) or based on simulations [7, 8, 34, 72, 75, 82, 94, 106, 119, 120]; one study uses a combination of both techniques [9]. Monitoring studies have the advantage of using larger scale versions of the workloads and allowing for a larger number of experiments to be run. However, monitoring studies are often limited by the configuration parameters of the system and the types of events that can be measured by the monitoring tools. For our own study, we found the flexibility of simulations invaluable for considering numerous design points and isolating the effects of different design decisions. Nevertheless, we found monitoring extremely important for tuning, scaling, and tracing our workloads and for verifying the results of our simulations.

The following describes the database studies based on out-of-order processors in more detail. Keeton et al. [57] present a monitoring study of the behavior of an OLTP workload (modeled after TPC-C) on top of Informix, using the performance counters on a quad Pentium Pro system. Similar to our study, this paper shows that out-of-order execution can achieve performance benefits on OLTP. Our work differs because we study several processor configurations of varying aggressiveness through detailed simulations. This enables us to quantitatively isolate the performance benefits from various ILP techniques as well as evaluate solutions to the various performance bottlenecks. We also evaluate the performance of DSS in addition to OLTP. We also study the tradeoffs between memory consistency models for database workloads on out-of-order processors.

Rosenblum et al. [106] present a simulation study that focuses on the impact of architectural trends on operating system performance using three workloads, one of which is TPC-B on Sybase. The study considers both uniprocessor and multiprocessor systems, with the multiprocessor study limited to in-order processors. The TPC-B workload used in this study is not representative because it exhibits large kernel and idle components primarily due to the lack of sufficient server processes. The paper does however make the observation that the decoupling of the fetch and execute units allows for some overlap of the instruction stall time in the uniprocessor system. Lo et al. [72] examine the performance of OLTP and DSS on a simultaneous multithreaded (SMT) uniprocessor system with support for multiple hardware contexts using the same simulation methodology as our study. Even though the SMT functionality is added on top of an aggressive out-of-order processor, the study primarily focuses on the effects of SMT as opposed to the underlying mechanisms in the base processor.

Following our work, a number of other papers have also studied database workloads in the context of out-of-order processors. Barroso et al. [8] discuss the performance benefits from chip-level integration on the performance of OLTP workloads. Another study by Barroso et al. [7] builds on our results on the performance benefits from aggressive ILP techniques to propose a new scalable architecture for OLTP and DSS workloads based on single-chip multiprocessing. Both these studies use a simulator that models both user-level and system code; however, their results comparing in-order and out-of-order processor models are very similar to those obtained in our work validating our assumptions to model only user-level code (discussed in Section 4.3). Other recent commercial publications([13], [26], [62], and [82]) also present some data on the performance of OLTP workloads on current state-of-the-art processors.

Ailamaki et al. [3] use performance counters to study four commercial database systems running simple queries used in DSS workloads (range-selection and join) on systems using a Pentium Xeon processor. Their overall results are qualitatively similar for all the four commercial database systems and show benefits comparable to those in our work for aggressive ILP techniques. However, in contrast to our results for TPC-D, their detailed division of execution time indicates a large fraction of stall time spent on instruction cache misses for their kernels. While the study does not discuss this in greater detail, the deeper pipeline in the Pentium Xeon system as well as the division of time methodology used with the performance monitoring counters could explain this discrepancy. Cao et al. characterize the TPC-D benchmark suite on a 4-way Pentium Pro server running commercial DBMS. Similar to our results, this study shows TPC-D queries having a relatively low CPI and negligible kernel time. Their results also show instruction fetch and L2 data cache misses as the major stall components [18]. Trancoso et al. also study DSS workloads. This study presents performance improvements from cache-aware query optimizers for a simple database query benchmark [120]. Bradford et al. [14] use RSIM to present an analysis of aggressive ILP techniques on several data mining workloads; their results are qualitatively similar to those presented in our work.

We are not aware of any work other than the original stream buffer work by Jouppi [56] that has studied the effect of stream buffers to alleviate the instruction miss bottleneck. This may partly be due to the fact that the standard SPEC, SPLASH, and STREAM benchmark suites do not stress the instruction cache significantly. Our evaluation differs from Jouppi's [56] in two key respects. First, we evaluate the impact of the stream buffer optimization on the execution time of a complex commercially-used database engine. Second, we perform our evaluation in the context of an aggressive out-of-order processor model that already includes support to hide part of the instruction stall latencies and show that the stream buffer can still provide a benefit.

Primitives similar to the "flush" primitive that we use in Section 4.10 have been proposed and studied by a number of other groups (e.g., [47, 64, 109]). Our "flush" primitive is most closely related to the "WriteThrough" primitive used by Abdel-Shafi et al. [1]. That study also showed that the combination of prefetching and "WriteThrough" could be used to achieve better performance improvements than using either of them alone (in the context of scientific applications). Following our work, two other recent studies discuss the performance benefits from prefetching and flushes for OLTP workloads. Friendly et al. [37] discuss the performance benefits from prefetching for OLTP workloads. Iyer et al. [53] identify location-sharing and migratory data to be the main contributors to the shared-access patterns in OLTP workloads.

Our study is based on systems with conventional cache hierarchies. A number of processors from Hewlett-Packard have opted for extremely large off-chip first level instruction and data caches (e.g., HP PA-8200 with up to 2MByte separate first level instruction and data caches), which may be targeting the large footprints in database workloads. These very large first level caches make the use of out-of-order execution techniques critical for tolerating the correspondingly longer cache access times.

Finally, our study evaluates the benefits of exploiting intra-thread parallelism in database workloads. Two previous studies have focused on exploiting inter-thread parallelism through the use of multithreaded processors [34, 72]. This approach depends on the fact that database workloads are already inherently parallel (either in the form of threads or processes) for hiding I/O latency. These studies show that multithreading can provide substantial gains, with simultaneous multithreading (SMT) [72] providing higher gains (as high as three times improvement for OLTP). Our study shows that DSS can benefit significantly from intra-thread parallelism (2.6 times improvement). The incremental gains from the addition of SMT are less significant in comparison [72]. Intra-thread parallelism is not as beneficial for OLTP (1.5 times improvement) due to the data dependent nature of the workload. In this case, SMT is more effective in hiding the high memory overheads. Other recent studies [126, 8, 7] have also discussed other approaches to improve OLTP performance such as larger on-chip caches, greater system level integration, and chip-multiprocessing.

4.12 Summary

In this chapter, we examined the performance of database workloads on shared-memory servers with state-of-the-art processors. We use detailed simulation to study both online-transaction processing (OLTP) and decision-support systems (DSS) running on the commercial Oracle database engine (version 7.3.2).

Our results show that current ILP-centric architectures are also very effective for database workloads. The combination of out-of-order execution and multiple instruction issue is effective in improving the performance of all our database workloads, providing factors of 1.5 and 2.6 performance gains over an in-order single issue processor for OLTP and DSS. Our results also show that speculative techniques that may be easily added to aggressive out-of-order processors are extremely beneficial for multiprocessor systems that support a stricter memory consistency model. For example, the execution time of a sequentially consistent system can be improved by 26% and 37% for OLTP and DSS workloads, bringing it to within 10-15% of more relaxed systems. Given that these techniques have been adopted in several commercial microprocessors, the choice of the hardware consistency model for a system does not seem to be a dominant factor for database workloads, especially for OLTP. The choice of consistency model for these systems only depends on the impact of relaxed consistency models on compiler optimizations and programming language support for relaxed consistency models.

The more challenging online transaction processing workloads exhibit radically different behavior compared to scientific/engineering applications. The key performance-limiting characteristics of these workloads are (i) large instruction footprints (leading to instruction cache misses) and (ii) frequent data communication (leading to cache-to-cache misses in the shared-memory system). We show that both these inefficiencies could be addressed with simple cost-effective optimizations. Stream buffers help alleviate instruction misses (approaching a perfect instruction cache within 15%). A combination of prefetching and producer-initiated communication directives address the communication misses. To the best of our knowledge, we are not aware of any current database server that includes all these optimizations.

Chapter 5

Conclusions

As computing demands have increased over the past few years, the workloads on general-purpose systems have also changed dramatically, with a greater emphasis on applications such as media processing and databases. However, in spite of the growing awareness of the importance of these workloads, prior to our work, there existed very little quantitative understanding of their behavior on general-purpose systems. Consequently, a number of fundamental questions still remained unanswered for these emerging workloads. Specifically, how effective are current processor and memory system features, designed in response to scientific and engineering workloads? What are the key performance bottlenecks that future system designs need to focus on for emerging workloads? How can we redesign architectures to achieve the higher performance requirements of future systems running these emerging workloads?

The goal of this dissertation was to address some of these questions. We studied several benchmarks representative of emerging media processing and database workloads identified through collaboration and consultation with application developers. Using a detailed simulator, RSIM, developed as part of this work, we characterized the performance of these workloads on state-of-the-art general-purpose systems and used the insights of this analysis to improve the performance of these systems. Our key high-level contributions include:

(i) providing the first detailed quantitative simulation-based studies of the performance of these workloads on systems using state-of-the-art processors

ii) proposing and evaluating cost-effective architectural solutions targeted at achieving the high performance requirements of future systems running these workloads.

The results from this dissertation can be organized into four parts discussed below.

5.1 Performance Characterization of Media Processing Workloads

The first part of our dissertation performed a detailed simulation study of several representative image and video applications to quantify the effectiveness of both processor and memory-system features found in state-of-the-art general-purpose processors.

Our results show that ILP-centric architectures are effective for media processing workloads. Conventional ILP features in current processors, including multiple issue and out-of-order execution, provide a factor of 2.3X to 4.2X performance improvement. Contrary to previous qualitative perceptions of these workloads [31], out-of-order execution is helpful in improving the performance of these workloads as well. Additionally, memory stall time is not an important bottleneck for all our benchmarks on the base ILP-based system.

The Sun VIS media ISA extensions directly target the compute bottleneck in the benchmarks and provide an additional 1.1X to 4.2X performance improvement. The benefits from the media ISA extensions are achieved with a much smaller increase in the die area compared to the ILP features. Our analysis also identifies several key limitations with the media ISA extensions (e.g., variable length data types, sequential and control-dependent code, VIS overhead) that can potentially be addressed to achieve higher performance benefits. With the improvements to the compute time due to VIS, the compute-memory balance shifts, with most of our image processing benchmarks now becoming memory bound.

The memory behavior of the media processing benchmarks is characterized by large working sets and streaming data accesses. Increasing the cache size has no impact on 8 of the benchmarks. The remaining benchmarks require relatively large cache sizes (dependent on the display sizes) to exploit data reuse, but derive less than 10 to 20% performance benefits even with the larger caches. Our results thus indicate that current trends towards large on-chip caches are likely to be ineffective for media processing workloads.

Software prefetching provides 1.4X to 2.5X performance improvement in the image processing benchmarks where memory is a significant problem. With the addition of software prefetching, the compute-memory balance of our benchmarks shifts again, with all our benchmarks reverting to being compute-bound.

5.2 Improving the Performance of Media Processing Workloads

The second part of our dissertation builds on the insights from the first part to design new architectures for improving the performance of media processing workloads. Specifically, we observe that while large on-chip caches are ineffective for most compute-intensive media workloads, a large fraction of on-chip transistors are currently devoted to caches because of their effectiveness for other classes of workloads.

We therefore propose a new cache organization, called reconfigurable caches, that allows flexibility in the use of on-chip transistors. Specifically, reconfigurable caches provide the ability to divide the cache SRAM arrays into different partitions that can be used for different processor activities. These activities can benefit applications that could not otherwise use conventional caches (e.g., instruction memoization, application-controlled memory, and prefetching buffers). Our design requires very few modifications to conventional caches, exploiting the natural implementation of set-associative caches today. Detailed timing analysis using a modification of the CACTI model shows small impact on cache access time.

We suggest several applications of reconfigurable caches. To show quantitative benefits, we choose to evaluate instruction reuse for media processing as a representative application. Instruction reuse coupled with reconfigurable caches allows us to improve computation performance using otherwise underutilized memory system resources. We find IPC performance benefits of between 4% and 20% from a reconfigurable cache with two partitions, one used for conventional caching and the other for instruction reuse. It is important to note that the benefits achieved in this work were with relatively small hardware and software changes to current general-purpose processors. Furthermore, comparisons with a more ideal implementation indicate that more aggressive implementations of instruction reuse can achieve higher performance benefits. Additionally, using more partitions for other activities can further improve the performance benefits from this organization. In the future, we believe that this paradigm of reusing on-chip storage for multiple processor activities can facilitate a number of other architectural optimizations.

5.3 Performance Characterization of Database Workloads

The next part of the dissertation examines the performance of database workloads on shared-memory servers with state-of-the-art processors. We use detailed simulation to study both online-transaction processing (OLTP) and decision-support systems (DSS) running on the commercial Oracle database engine (version 7.3.2).

Our results show that current ILP-centric architectures are also very effective for database workloads. Among database applications, online transaction processing (OLTP) workloads present the more demanding set of requirements for system designers. While DSS workloads are somewhat reminiscent of scientific/engineering applications in their behavior, OLTP workloads exhibit dramatically different behavior due to large instruction footprints and frequent data communication misses which often lead to cache-to-cache transfers. Our results reflect these characteristics, with our DSS workload achieving a 2.6 times improvement from out-of-order execution and multiple issue, and our OLTP workload achieving a more modest 1.5 times improvement.

We also found that speculative techniques that may be easily added to aggressive out-of-order processors are extremely beneficial for multiprocessor systems that support a stricter memory consistency model. For example, the execution time of a sequentially consistent system can be improved by 26% and 37% for OLTP and DSS workloads, bringing it to within 10-15% of more relaxed systems. Given that these techniques have been adopted in several commercial microprocessors, the choice of the hardware consistency model for a system does not seem to be a dominant factor for database workloads, especially for OLTP.

5.4 Improving the Performance of Database Workloads

The last part of the dissertation examines methods to improve the performance of more challenging OLTP workload. The key performance-limiting characteristics of these workloads are (i) large instruction footprints (leading to instruction cache misses) and (ii) frequent data communication (leading to cache-to-cache misses in the shared-memory system). We showed that both these inefficiencies could be addressed with simple cost-effective optimizations. Stream buffers help alleviate instruction misses (approaching a perfect instruction cache within 15%). A combination of prefetching and producer-initiated communication directives address the communication misses. To the best of our knowledge, neither of these architectural techniques are implemented in current database servers.

Chapter 6

Future Work

This chapter presents a brief overview of promising areas of future research that relate to the research addressed in this dissertation.

6.1 Extending and Exploiting Reconfigurability

This dissertation proposed the concept of a *reconfigurable cache* that allows the large on-chip transistor area devoted to caches to be flexibly used in different ways for different workloads. As discussed in Section 3.1, this includes look-up tables for optimizations like instruction reuse, value prediction, branch prediction, coherence prediction, memory address disambiguation, address prediction (for optimizations like hardware prefetching), and performance monitoring as well as flexibly-defined SRAM storage for memory-related optimizations like hardware and software prefetching or application-controlled memory. In addition, this paradigm of reusing on-chip storage for multiple processor activities has the potential to facilitate a number of different new optimizations. Some interesting areas of research are listed below.

- *More aggressive reconfiguration.* The benefits from reconfigurable caches can be further increased by aggressively using the cache partitions for multiple processor activities. Such a more general organization would require support for more frequent dynamic reconfiguration, discussed qualitatively in Chapter 3. Potential implementation challenges that need to be addressed include the heuristics for determining the size of the partitions allocated to each processor activity and the frequency of repartitioning.

A thorough exploration of the design space for such aggressive implementations will be valuable in understanding the performance potential of reconfigurable caches.

- *Real-time quality of service guarantees with reconfigurable caches.* Media applications typically place great emphasis on the perceived quality of the output and consequently require predictable real-time quality-of-service guarantees from the system. However, several features that enhance the performance of current general-purpose processors (*e.g.*, caching, speculative execution) do not allow such guarantees. It would be interesting to evaluate the possibility of developing systems that provide the functionality of these general-purpose features *without* compromising real-time predictability. For example, using parts of a reconfigurable cache for application-controlled memory can ensure deterministic latencies for key memory accesses, allowing a simple way to address unpredictability due to caching.
- *Integrating reconfigurable logic and reconfigurable caches.* Reconfigurable caches implement flexibility in the use of memory, but using high-speed custom hardware; other approaches to reconfigurability (*e.g.*, FPGA) use programmable logic to implement application-specific functions in hardware, but at the expense of slower circuit speeds and more complex synthesis techniques for programming. Recent advances in programmable logic suggest a possible future processor design with limited inclusion of programmable logic, co-existing with reconfigurable caches. Integrating reconfigurability in logic (current programmable logic) and memory (our proposed reconfigurable cache) is a very interesting open area of research, especially in the context of multi-granularity reconfigurability at both the logic and memory levels.

6.2 Other Optimizations for Emerging Applications

Compiler and programming language support for media applications. As discussed earlier, currently, we are not aware of any compilers that can automatically generate code enhanced for media ISA extensions from application C code. Our methodology in Section 2 summarizes the heuristics we used to enhance our workloads with VIS. It would be interesting to study the effectiveness of translating these heuristics into a compiler framework. Additional programming language directives may also be required to facilitate the compiling of media applications (e.g., packed data type specifications).

Increased system support for emerging applications. Our results in Chapters 2 and 4 demonstrate a number of interesting trends about the effectiveness of current architectural features. These trends open up the possibility for several of a number of interesting hardware/software optimizations. For example, these could include hardware support for new special-purpose instructions (e.g., bit-level arithmetic), support for unaligned accesses to the register file, support for MDMX-style large accumulators, etc. Similarly, there are a number of software optimizations like cache-aware algorithm restructuring (for media processing) and the better use of prefetching/flush hints and code restructuring (for database applications).

6.3 Other Emerging Applications and Appliances

This dissertation also focuses only on a representative subset of emerging applications restricted to media processing and database workloads. Some other recent emerging applications (e.g., wireless communication) have similar high computational requirements and may motivate the need for new architectural substrates. Similarly, this dissertation focuses only on general-purpose systems in PCs and servers (though all the ideas are applicable to other classes of systems as well). As discussed in Chapter 1, in addition to the dramatic changes in computing workloads, computing appliances have also moved beyond conventional PCs and servers to several new appliances (e.g., video cell phones, pocket computers, webTVs). In addition to execution time, many of these emerging appliances place greater emphasis on other metrics like power consumption, cost, and reliability. Developing architectures that achieve the constraints in these additional metrics along with high performance is likely to pose several exciting challenges.

In addition to higher compute requirements and greater emphasis on non-conventional performance metrics, emerging technologies that allow more transistors and greater on-chip system integration (e.g., memory, reconfigurable logic) allow for the possibility of newer architecture designs. These trends open up a number of fundamental research areas in computer architecture that are both intellectually satisfying and have high commercial value at all levels of the computing continuum (client, communication, and server). This dissertation focuses on a subset of these problems, namely architectures for desktop processors for media processing and processor and system design for database servers. However, there is still a lot of room for research. Our hope is that the ideas, insights and methodologies in this dissertation will help future developments in the area.

Bibliography

[1] Hazim Abdel-Shafi, Jonathan Hall, Sarita V. Adve, and Vikram S. Adve. An Evaluation of Fine-Grain Producer-Initiated Communication in Cache-Coherent Multiprocessors. In *Proceedings of the 3rd International Symposium on High-Performance Computer Architecture*, pages 204–215, February 1997.

[2] Sarita V. Adve, Vijay S. Pai, and Parthasarathy Ranganathan. Recent Advances in Memory Consistency Models for Hardware Shared Memory Systems. *Proceedings of the IEEE*, 87(3):445–455, March 1999.

[3] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 266–277. Morgan Kauffman, September 1999.

[4] David H. Albonesi. Dynamic IPC/Clock Rate Optimization. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 268–277, June 1998.

[5] David H. Albonesi. Selective Cache Ways: On-Demand Cache Resource Allocation. In *Proceedings of the 32nd Annual International Conference on Microarchitecture*, November 1999.

[6] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous Profiling: Where Have All the Cycles Gone? In *Proceedings of the Symposium on Operating System Principles*, October 1997.

[7] Luiz Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, and Ben Verghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Proceedings of the Annual International Symposium on Computer Architecture*, June 2000.

[8] Luiz Barroso, Kourosh Gharachorloo, Andreas Nowatzyk, and Ben Verghese. Impact of Chip-Level Integration on Performance of OLTP Workloads. In *Proceedings of the Sixth International Symposium on High Performance Computer Architecture*, January 2000.

[9] Luiz A. Barroso, Kourosh Gharachorloo, and Edouard D. Bugnion. Memory System Characterization of Commercial Workloads. In *Proceedings of the 25th International Symposium on Computer Architecture*, June 1998.

[10] Dileep Bhandarkar and Jason Ding. Performance Characterization of the Pentium Pro Processor. In *Proceedings of the Third International Symposium on High Performance Computer Architecture*, pages 288–297, Feb 1997.

[11] R. Bhargava et al. Evaluating MMX Technology Using DSP and Multimedia Applications. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, Dec 1998.

[12] Angelos Bilas et al. Real-time Parallel MPEG-2 Decoding in Software. In *Proceedings of the 11th International Parallel Processing Symposium*, April 1997.

[13] J. Borkenhagen and S. Storino. 5th Generation 64-bit PowerPC Compatible Commercial Processor Design. In *http:// www.rs6000.ibm.com/ resource/technology/pulsar.pdf*, September 1999.

[14] Jeff Bradford and Jose Fortes. Performance and Memory-Access Characterization of Data Mining Applications. In *Proceedings of the Workshop on Workload Characterization (held in cojunction with MICRO-32)*, pages 91–100, November 1998.

[15] Doug Burger, James R. Goodman, and Alain Kägi. Quantifying Memory Bandwidth Limitations of Current and Future Microprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 78–89, May 1996.

[16] Dave Bursky. Multiprocessor DSP Architecture Races to Deliver 1.6 Billion Multiply-Accumulates. In *Electronic Design, Volume 14, Number 13*, June 28 1999.

[17] David Callahan, Ken Kennedy, and Allan Porterfield. Software Prefetching. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[18] Qiang Cao, Pedro Trancoso, Josep-Lluis Larriba-Pey, Josep Torrellas, Robert Knighten, and Youjip Won. Detailed Characterization of a Quad Pentium Pro Server Running TPC-D . In *Proceedings of the International Conference on Computer Design*, October 1999.

[19] David A. Carlson et al. Multimedia Extensions for a 550MHz RISC Microprocessor. In *IEEE Journal of Solid-State Circuits*, 1997.

[20] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a Smarter Memory Controller. In *Proceedings of the Fifth Annual Symposium on High Performance Computer Architecture*, pages 70–79, January 1999.

[21] Derek Chiou, Prabhat Jain, Srinivas Devadas, and Larry Rudolph. Application-Specific Memory Management of Embedded Systems Using Software-Controlled Caches. Technical Report CSG-Memo 427, Massachussetts Institute of Technology, November 1999.

[22] Keith Cooper and Tim Harvey. Compiler-Controlled Memory. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1998.

[23] Jesus Corbal, Mateo Valero, and Roger Espasa. Exploiting a New Level of DLP in Multimedia Applications. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, November, 1999.

[24] MIPS Corporation. *MIPSV ISA Extension Manual*.

[25] Alan L. Cox and Robert J. Fowler. Adaptive Cache Coherency for Detecting Migratory Shared Data. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 98–108, May 1993.

[26] Zarka Cvetanovic. Analysis of Commercial and Technical Workloads on AlphaServer Platforms. In *First Workshop on Computer Architecture Evaluation using Commercial Workloads (in conjuntion with HPCA-4)*, 1998.

[27] Zarka Cvetanovic and Dileep Bhandarkar. Performance Characterization of the Alpha 21164 microprocessor using TP and SPECWorkloads. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 60–70, Apr 1994.

[28] Zarka Cvetanovic and Darrel D. Donaldson. AlphaServer 4100 performance characterization. *Digital Technical Journal*, 8(4):3–20, 1996.

[29] Jutta Degener. GSM 06.10 Lossy Speech Compression. Available at http://kbs/cs/tu-berlin.de/ jutta/toast.html, 1998.

[30] Advanced Micro Devices. *3DNow! Technology Manual*, 1998.

[31] Keith Diefendorff and Pradeep K. Dubey. How Multimedia Workloads Will Change Processor Design. In *IEEE Micro*, pages 43–45, Sep 1997.

[32] Digital Equipment Corporation. *Alpha Architecture Reference Manual*, 1992.

[33] Thierry Dutoit. *An Introduction to Text-to-Speech Synthesis*. Kluwer Academic Publishers, 1996.

[34] Richard J. Eickemeyer, Ross E. Johnson, Steven R. Kunkel, Mark S. Squillante, and Shiafun Liu. Evaluation of Multithreaded Uniprocessors for Commercial Applica-

tion Environments. In *Proceedings of the 21th Annual International Symposium on Computer Architecture*, pages 203–212, June 1996.

[35] Jennifer Eyre. Assessing General-Purpose Processors for DSP Applications. Berkeley Design Technology Inc. presentation, 1998.

[36] Jennifer Eyre and Jeff Bier. DSP Processors Hit the Mainstream. *IEEE Computer*, 1998.

[37] Dan Friendly and Mark Charney. Performance Analysis of Shadow Directory Prefetching for TPC-C. In *First Workshop on Computer Architecture Evaluation using Commercial Workloads (in conjuntion with HPCA-4)*, 1998.

[38] Sam Fuller. Motorola's AltiVec Technology - White Paper. http://www.mot.com/ SPS/PowerPC/teksupport/ teklibrary/papers/altivec_wp.pdf, 1998.

[39] Freddy Gabbay and Avi Mendelson. The Effect of Instruction Fetch Bandwidth on Value Prediction. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 272–281, 1998.

[40] D. Le Gall. MPEG: A Video Compression Standard for Multimedia Applications. *Communications of the ACM*, 34(4):46–58, April 1991.

[41] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages I:355–364, August 1991.

[42] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Hiding Memory Latency using Dynamic Scheduling in Shared-Memory Multiprocessors. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 22–33, May 1992.

[43] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable

Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[44] James R. Goodman. Cache Consistency and Sequential Consistency. Technical Report #61, SCI Committee, March 1989. Also available as Computer Sciences Technical Report #1006, Uni versity of Wisconsin, Madison, February 1991.

[45] Dirk Grunwald, Artur Klauser, Srilatha Manne, and Andrew Pleszkun. Confidence Estimation for Speculation Control. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 122–131, June 1998.

[46] John Hennessy. The Future of Systems Research. *IEEE Computer*, 32(8):27–33, August 1999.

[47] Mark D. Hill, James R. Larus, Steven K. Reinhardt, and David A. Wood. Cooperative Shared Memory: Software and Hardware Support for Scalable Multiprocessors. *TOCS*, 11(4):300–318, November 1993.

[48] Intel Corporation. *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference Manual*.

[49] International Organisation for Standardisation – ISO/IEC JTC1/SC29/WG11MPEG 98/N2457. *MPEG-4 Applications Document*, 1998.

[50] ISO/IEC IS10918-1, ITU-T T.81. *Digital Compression and Coding of Continuous-tone Still Images, Part 1: Requirements and guidelines*.

[51] ISO/IEC/JTC1/SC29/WG11 N MPEG96/July 1996. *MPEG-2: Generic Coding of Moving Pictures and Associated Audio Information*, 1996.

[52] Eiji Iwata and Kunle Olukotun. Exploiting Coarse-Grain Parallelism in the MPEG-2 Algorithm. In *Proceedings of the International Conference on Supercomputing (ICS)*, 1999.

[53] Ravi Iyer, G. Janakiraman, Raj Kumar, and Laxmi Bhuyan. A Trave-driven Analysis of Sharing Behavior in TPC-C. In *Proceedings of the Second Workshop on Com-*

puter Architecture Evaluation Using Commercial Workloads (held in conjunction with HPCA-5), January 1999.

[54] Lizy K. John. VaWiRAM: A Variable Width Random Access Memory Module. In *Proceedings of the International Conference on VLSI Design*, pages 219–224, January 1996.

[55] Lizy K. John, Raghuveer Reddy, Vijay Kammila, and Peter Maurer. Investigating the Use of Cache as a Local Memory. In *Proceedings of the International High Performance Computing Conference*, pages 117–122, December 1995.

[56] Norman P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings 17th Annual International Symposium on Computer Architecture*, pages 364–373, June 1990.

[57] Kimberly Keeton, David A. Patterson, Yong Qiang He, Roger C. Raphael, and Walter E. Baker. Performance Characterization of the Quad Pentium Pro SMP Using OLTP Workloads. In *Proceedings of the 25th International Symposium on Computer Architecture*, June 1998.

[58] Earl Killian. MIPS Extension for Digital Media with 3D. Slides presented at Microprocessor Forum, October 1996.

[59] Hue-Sung Kim, Arun K. Somani, and Akhilesh Tyagi. A Reconfigurable Multifunction Computing Cache Architecture. In *Proceedings of the International Conference on Field Programmable Gate Arrays (FPGA'2000)*, pages 85–94, 2000.

[60] L. Kohn et al. The Visual Instruction Set (VIS) in UltraSPARC. In *COMPCON Digest of Papers*, March 1995.

[61] David Kroft. Lockup-free Instruction Fetch/Prefetch Cache Organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 81–85, 1981.

[62] Kunkel, Armstrong, and Vitale. System Optimization for OLTP Workloads. *IEEE Micro*, May/June 1999.

[63] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers,* C-28(9):690–691, September 1979.

[64] A.R. Lebeck and D.A. Wood. Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture,* 1995.

[65] Chunho Lee et al. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture (MICRO-30)*, 1997.

[66] Corinna G. Lee and Mark G. Stoodley. Simple Vector Microprocessors for Multimedia Applications. In *Proceedings of the 31st Annual International Symposium on Microarchitecture (MICRO-31)*, 1998.

[67] Dennis C. Lee, Patrick J. Crowley, Jean-Loup Baer, Thomas E. Anderson, and Brian N. Bershad. Execution Characteristics of Desktop Applications on Windows NT. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 27–38, June 1998.

[68] Ruby B. Lee. Subword Parallelism with MAX-2. In *IEEE Micro*, volume 16(4), pages 51–59, August 1996.

[69] Ruby B. Lee and Michael D. Smith. Media Processing: A New Design Target. In *IEEE MICRO*, pages 6–9, Aug 1996.

[70] Mikki H. Lipasti and John P. Shen. Exceeding the Dataflow Limit via Value Prediction. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 226–237, 1996.

[71] Mikki H. Lipasti, Christopher B. Wilkerson, and John P. Shen. Value Locality and Load Value Prediction. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, 1996.

[72] Jack L. Lo, Luiz A. Barroso, Susan J. Eggers, Kourosh Gharachorloo, Henry M. Levy, and Sujay S. Parekh. An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors. In *Proceedings of the 25th International Symposium on Computer Architecture*, June 1998.

[73] R. Maher. Multimedia Instruction Set Extensions for a Sixth-Generation x86 Processor. In *Proceedings of the HOTCHIPS-VIII Conference*, 1996.

[74] Ken Mai, Tim Paaske, Nuwan Jayasena, Ron ho, William J. Dally, and Mark Horowitz. Smart Memories: A Modular Reconfigurable Architecture. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.

[75] Ann Marie Grizzaffi Maynard, Colette M. Donnelly, and Bret R. Olszewski. Contrasting Characteristics and Cache Performance of Technical and Multi-User Commercial Workloads. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 145–156, Oct 1994.

[76] John D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. In *IEEE Technical Committee on Computer Architecture Newsletter*, Dec 1995.

[77] Grant W. McFarland. *CMOS Technology Scaling and Its Impact on Cache Delay*. PhD thesis, Stanford Univerisity, 1997.

[78] S. A. McKee et al. Design and Evaluation of Dynamic Access Ordering Hardware. In *Proceedings of the 1996 International Conference on Supercomputing*, May 1996.

[79] Sun Microelectronics. The VIS Advantage: Benchmark Results Chart VIS Performance. Whitepaper #WPR-0012 (available at http://www.sun.com/ microelectronics/whitepapers), Oct 1996.

[80] MIPS Technologies, Inc. *R10000 Microprocessor User's Manual, Version 2.0*, December 1996.

[81] Carlos Molina, Antonio Gonzalez, and Jordi Tubella. Dynamic Removal of Redundant Computations. In *Proceedings of the ACM International Conference on Supercomputing*, June 1999.

[82] J.H. Moreno, M. Moudgill, J.D.Wellman, P. Bose, and L. Trevillyan. Trace-driven Performance Exploration of a PowerPC 601 OLTP Workload on Wide Superscalar Processors. In *First Workshop on Computer Architecture Evaluation using Commercial Workloads (in conjuntion with HPCA-4)*, 1998.

[83] Todd Mowry. *Tolerating Latency through Software-controlled Data Prefetching*. PhD thesis, Stanford University, 1994.

[84] Todd Mowry and Anoop Gupta. Tolerating Latency Through Software-Controlled Prefetching. *Journal of Parallel and Distributed Computing*, pages 87–106, June 1991.

[85] Basem A. Nayfeh, Lance Hammond, and Kunle Olukotun. Evaluation of Design Alternatives for a Multiprocessor Microprocessor. In *Proceedings of the 23rd International Symp. on Computer Architecture*, pages 67–77, May 1996.

[86] Huy Nguyen and Lizy Kurian John. Exploiting SIMD Parallelism in DSP and Multimedia Algorithms Using the AltiVec Technology. In *Proceedings of the International Conference on Supercomputing (ICS)*, 1999.

[87] Stuart Oberman et al. AMD 3DNow! Technology and the K6-2 Microprocessor. In *Proceedings of the HOTCHIPS-X Conference*, 1998.

[88] Vijay S. Pai, Parthasarathy Ranganathan, Hazim A.-Shafi, and Sarita Adve. The Impact of Exploiting Instruction-Level Parallelism on Shared-Memory Multiprocessors. *IEEE Transaction on Computers*, 48(2):218–226, February 1999.

[89] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. The Impact of Instruction Level Parallelism on Multiprocessor Performance and Simulation Methodology. In *Proceedings of the Third International Symposium on High Performance Computer Architecture*, pages 72–83, 1997.

[90] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. RSIM: An Execution-Driven Simulator for ILP-Based Shared-Memory Multiprocessors and Uniprocessors. In *Proceedings of the Third Workshop on Computer Architecture Education*, February 1997. Also appears in the IEEE TCCA Newsletter, October 1997.

[91] Vijay S. Pai, Parthasarathy Ranganathan, Sarita V. Adve, and Tracy Harton. An Evaluation of Memory Consistency Models for Shared-Memory Systems with ILP Processors. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 12–23, Oct. 1996.

[92] Davis Yen Pan. A Tutorial on MPEG/Audio Compression. *IEEE Multimedia*, 1995.

[93] Alex Peleg and Uri Weiser. MMX Technology Extension to the Intel Architecture. In *IEEE Micro*, volume 16(4), pages 51–59, Aug 1996.

[94] Sharon E. Perl and Richard L. Sites. Studies of Windows NT Performance using Dynamic Execution Traces. In *Proceedings of the Second Symposium on Operating System Design and Implementation*, pages 169–184, October 1996.

[95] Micheal Phillip et al. AltiVec Technology: Accelerating Media Processing Across the Spectrum. In *Proceedings of the HOTCHIPS-X Conference*, Aug 1998.

[96] Francisa Quintana, Jesus Corbal, Roger Espasa, and Mateo Valero. Adding a Vector Unit to a Superscalar Processor. In *Proceedings of the International Symposium on Supercomputing (ICS)*, June, 1999.

[97] Parthasarathy Ranganathan, Sarita V. Adve, and Norman P. Jouppi. Performance of Image and Video Processing with General-Purpose Processors and Media ISA Extensions. In *Proceedings of the 26th International Symposium on Computer Architecture (ISCA)*, 1999.

[98] Parthasarathy Ranganathan, Sarita V. Adve, and Norman P. Jouppi. Reconfigurable Caches and their Application to Media Processing. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA)*, 2000.

[99] Parthasarathy Ranganathan, Kourosh Gharachorloo, Sarita Adve, and Luiz Barroso. Performance of Database Workloads on Shared-Memory Systems with Out-of-Order Processors. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 307–318, 1998.

[100] Parthasarathy Ranganathan, Vijay S. Pai, Hazim Abdel-Shafi, and Sarita V. Adve. The Interaction of Software Prefetching with ILP Processors in Shared-Memory Systems. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 144–156, 1997.

[101] Parthasarathy Ranganathan, Vijay S. Pai, and Sarita V. Adve. Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap between Memory Consistency Models. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1997.

[102] Glenn Reinman and Norman P. Jouppi. CACTI 2.0 Beta. In *http://www.research.digital.com/wrl/people/jouppi/ CACTI.html*, 1999.

[103] Daniel S. Rice. High-Performance Image Processing Using Special-Purpose CPU Instructions: The UltraSPARC Visual Instruction Set. Master's thesis, Stanford University, 1996.

[104] Scott Rixner, William J. Dally, Ujval J. Kapasi, Brucek Khailany, Abelardo Lopez-Lagunas, Peter R. Mattson, and John D. Owens. A Bandwidth-Efficient Architecture for Media Processing. In *Proceedings of the 31st International Symposium on Microarchitecture (MICRO-31)*, 1998.

[105] Editor-in-chief Ronald A. Cole. *Survey of the State of the Art in Human Language Technology*. National Science Foundation and European Commission, 1996.

[106] Mendel Rosenblum, Edouard Bugnion, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. The Impact of Architectural Trends on Operating System Performance. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 285–298, 1995.

[107] Y. Sazeides and J. E. Smith. The Predictability of Data Values. In *Proceedings of the 30th Annual International Conference on Microarchitecture*, pages 248–258, 1997.

[108] Julien Sebot. A Performance Evaluation of Multimedia Kernels Using AltiVec Streaming SIMD Extensions. In *Proceedings of the Third Workshop on Computer Architecture Evaluation Using Commercial Workloads (held in conjunction with HPCA-6)*, January 2000.

[109] J. Skeppstedt and Per Stenstrom. A Compiler Algorithm that Reduces Read Latency in Ownership-Based Cache Coherence Protocols. In *International Conference on Parallel Architectures and Compilation Techniques*, 1995.

[110] Avinash Sodani and Guri Sohi. Dynamic Instruction Reuse. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 194–205, 1997.

[111] Avinash Sodani and Guri Sohi. An Empirical Analysis of Instruction Repetition. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 35–45, 1998.

[112] Amitabh Srivastava and Alan Eustace. ATOM: A System for Building Customized Program Analys is Tools. *Proceedings of the ACM SIGPLAN ‘94 Conference on Programming Languages*, March 1994.

[113] Standard Performance Council. *The SPEC95 CPU Benchmark Suite*. http://www.specbench.org, 1995.

[114] Per Stenstrom, Mats Brorsson, and Lars Sandberg. An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 109–118, May 1993.

[115] Per Stenstrom, Erik Hagersten, David J. Lilja, Margaret Martonosi, and Madan Venugopal. Trends in Shared Memory Multiprocessing. *IEEE Computer*, 1997.

[116] Rob Stets, Luiz Barroso, Kourosh Gharachorloo, and Ben Verghese. A Detailed Comparison of TPC-B Versus TPC-C. In *Proceedings of the Third Workshop on Computer Architecture Evaluation Using Commercial Workloads (held in conjunction with HPCA-6)*, January 2000.

[117] T.-Y.Yeh and Y.N.Patt. Alternative Implementations of Two-level Adaptive Branch Prediction. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992.

[118] Shreekant S. Thakkar and Mark Sweiger. Performance of an OLTP Application on Symmetry Multiprocessor System. In *Proceedings of the 17th Annual International Symposium on Computer Architecture,* pages 228–238, June 1990.

[119] Pedro Trancoso, Josep-L. Larriba-Pey, Zheng Zhang, and Josep Torrellas. The Memory Performance of DSS Commercial Workloads in Shared-Memory Multiproces-

sors. In *Third International Symposium on High-Performance Computer Architecture*, Jan 1997.

[120] Pedro Trancoso and Josep Torrellas. Cache Optimization for Memory-Resident Decision Support Commercial Workloads. In *Proceedings of the International Conference on Computer Design*, October 1999.

[121] Transaction Processing Performance Council. *TPC Benchmark B (Online Transaction Processing) Standard Specification*, 1990.

[122] Transaction Processing Performance Council. *TPC Benchmark D (Decision Support) Standard Specification*, Dec 1995.

[123] Marc Tremblay et al. VIS Speeds New Media Processing. In *IEEE Micro*, volume 16(4), pages 51–59, Aug 1996.

[124] D.M. Tullsen and S.J. Eggers. Effective Cache Prefetching on Bus-Based Multiprocessors. *ACM Transactions on Computer Systems*, 13(1):57–88, 1995.

[125] Alexander V. Veidenbaum, Weiyu Tang, Rajesh Gupta, Alexandru Nicolau, and Xiaomei Ji. Adapting Cache Line Size to Application Behavior. In *Proceedings of the International Conference on Supercomputing*, 1999.

[126] Ben Verghese, Luiz Barroso, and Kourosh Gharachorloo. Effectiveness of Off-Chip Caches for Commercial Applications. In *Proceedings of the 1999 Workshop on Scalable Shared Memory Multiprocessors (held in conjunction with ISCA-99)*, January 1999.

[127] Steve Wilton and Norman P. Jouppi. CACTI: An Enhanced Cache Access and Cycle Time Model. *IEEE Journal of Solid-State Circuits*, pages 677–687, 1996.

[128] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, June 1995.

[129] Chia-Lin Yang et al. Exploiting Instruction-Level Parallelism in Geometry Processing for Three Dimensional Graphics Applications. In *Proceedings of the 31st International Symposium on Microarchitecture (MICRO-31)*, 1998.

[130] Lixin Zhang, John B. Carter, Wilson C. Hsieh, and Sally A. McKee. Memory System Support for Image Processing. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, October 1999.

[131] X. Zhang, A. Dasdan, M. Schultz, R. Gupta, and A. Chien. Architectural Adaptation of Application-Specific Locality Optimizations. In *Proceedings of the 1997 IEEE International Conference on Computer Design (ICCD)*, 1997.

[132] Daniel F. Zucker. *Architecture and Arithmetic for Multimedia Enhanced Processors*. PhD thesis, Department of Electrical Engineering, Stanford University, June 1997.

[133] Daniel F. Zucker et al. An Automated Method for Software Controlled Cache Prefetching. In *Proceedings of the 31st Hawaii International Conference on System Sciences*, Jan 1998.