USING COMPRESSION TO IMPROVE CHIP MULTIPROCESSOR PERFORMANCE

by

Alaa R. Alameldeen

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN - MADISON

2006

Abstract

Chip multiprocessors (CMPs) combine multiple processors on a single die, typically with private level-one caches and a shared level-two cache. However, the increasing number of processors cores on a single chip increases the demand on two critical resources: the shared L2 cache capacity and the off-chip pin bandwidth. Demand on these critical resources is further exacerbated by latency-hiding techniques such as hardware prefetching. In this dissertation, we explore using compression to effectively increase cache and pin bandwidth resources and ultimately CMP performance.

We identify two distinct and complementary designs where compression can help improve CMP performance: Cache Compression and Link Compression. Cache compression stores compressed lines in the cache, potentially increasing the effective cache size, reducing off-chip misses and improving performance. On the downside, decompression overhead can slow down cache hit latencies, possibly degrading performance. Link (i.e., off-chip interconnect) compression compresses communication messages before sending to or receiving from off-chip system components, thereby increasing the effective off-chip pin bandwidth, reducing contention and improving performance for bandwidth-limited configurations. While compression can have a positive impact on CMP performance, practical implementations of compression raise a few concerns: (1) Compression algorithms have too high an overhead to implement at the cache level; (2) compression overhead can degrade performance; (3) the potential for compression on a CMP is unknown; (4) most benefits of compression can be achieved by hardware prefetching; and (5) the impact of compression on a balanced CMP design is not well understood.

In this dissertation, we make five contributions that address the above concerns. We propose a compressed L2 cache design based on a simple compression algorithm with a low decompression overhead. We develop an adaptive compression scheme that dynamically adapts to the costs and benefits of cache compression, and employs compression only when it helps performance. We show that cache and link compression both combine to improve CMP performance for commercial and (some) scientific workloads. We show that compression interacts in a strong positive way with hardware prefetching, whereby a system that implements both compression and hardware prefetching can have a higher speedup than the product of speedups of each scheme alone. We also provide a simple analytical model that helps provide qualitative intuition into the trade-off between cores, caches, communication and compression, and use full-system simulation to quantify this trade-off for a set of commercial workloads.

Acknowledgments

Thank God that I have received a lot of help and support from many people over the past few years. I cannot imagine making it this far without such support. It is with great pleasure that I thank and acknowledge those who contributed to my success, or as many of them as I can remember.

I am deeply grateful to my advisor, Prof. David Wood. He has been an outstanding mentor who provided me with a lot of support and guidance. I have learned a lot under his supervision, and not just about computer architecture. He taught me how to conduct research, guided me in writing research papers, and trained me on how to present my research to others. I am thankful that he gave me as much of his time as I asked for, and provided me with support when I needed it the most. On both a professional and a personal level, I greatly enjoyed being his student for all these years!

The University of Wisconsin is a great place to learn about computer science in general, and computer architecture in particular. I would like to thank many outstanding professors at UW-Madison who taught me a lot and helped me throughout my Ph.D. program. I feel very fortunate to have had the chance to work with and learn from Prof. Mark Hill, the other director of the Multifacet project. He is a great person to work with. I benefited a lot from his sound advice, excellent insight, and valuable feedback on my work, and especially my presentation skills. I would also like to thank Professors Jim Goodman, Jim Smith, Guri Sohi and Mikko Lipasti for building and maintaining an excellent computer architecture program. I thank Prof. Jim Goodman for teaching me about multiprocessors, for his strong support and encouragement, and for his useful feedback on my work. I also thank Prof. Guri Sohi who has been extremely helpful to me since my preliminary exam. I would also like to thank all my committee members: Professors Mark Hill, Mikko Lipasti, Jim Smith and Guri Sohi for their help, support, useful feedback, and flexibility in scheduling my preliminary and final oral exams. I thank them especially for encouraging and challenging me to improve my dissertation research. I would also like to thank Professors David Wood, Mark Hill, Guri Sohi, Jim Goodman, Mikko Lipasti and Jeff Naughton for taking the time to write reference letters for me.

I would also like to acknowledge many other UW-Madison faculty members. I thank Professors Mike Shulte, Charlie Fischer and Mary Vernon for their useful feedback on my research. I also thank Prof. David DeWitt who was the best instructor I had in graduate school, as well as all professors I was fortunate to learn from.

I received a lot of support from student members of the Multifacet project. Milo Martin was very helpful to me since the first day I joined the group. Milo and Dan Sorin were both good mentors to me. They both were the first to think about workload variability, and they pointed me in the right direction to continue that work. Carl Mauer has been a good friend, and I really appreciate his kindness towards me, as well as his helping me solve various technical problems. Anastassia Ailamaki was a tremendous help in getting me started on my database workloads work. I thank Brad Beckmann, my office-mate for the past year, for not complaining—at least to my face!—when I was bothering him with questions. I also thank him for helping me use his code to get started in studying hardware prefetching. Kevin Moore engaged me in many interesting discussions. Min Xu has been a source of tremendous technical help, and I benefited a lot from his knowledge. I also enjoyed listening to Mike Marty’s interesting opinions. Ross Dickson helped me quite a bit with the simulation infrastructure. I would like to acknowledge other former Multifacet members, as well as current members Luke Yen, Dan Gibson, Jayaram Bobba, Michelle Moravan, and Andy Phelps. Many thanks are due to project assistants Alicia Walley, Erin Miller, Carrie Pritchard and Caitlin Scopel for their help with project administration and proof-reading.

I’ve also had the opportunity to meet many excellent students in Madison. While I cannot possibly mention the names of all those who gave me moral support, help and encouragement, I would like to specifically thank a few people. My friend Ashraf Aboulnaga (currently Prof. Aboulnaga!) has helped me a lot during my first three years in Madison. I was glad I had the opportunity to work with him on a database research publication, much to my surprise! I am grateful to Craig Zilles, Adam Butts, Brian Fields, Carl Mauer, Brad Beckmann and Philip Wells for running the always-informative, often-entertaining architecture reading group, and to Allison Holloway for organizing the architecture students lunch. I also had a lot of interesting discussions with some of my colleagues here that I would like to thank: Trey Cain, Ravi Rajwar, Ashutosh Dhodapkar, Kevin Lepak, Shiliang Hu, Tejas Karkhanis, Collin McCurdy, Jichuan Chang, Gordie Bell, Sai Balakrishnan, Matt Allen, Koushik Chakraborty, and all others I might have forgotten. Trey Cain, Carl Mauer and I had a lot of enlightening discussions in our qualifying exam reading group. I would also like to specially thank Ravi Rajwar for inviting me to visit his group at Intel in 2003. I would like to acknowledge the following Wisconsin Computer Architecture Affiliates for their feedback on my work and for the many interesting discussions I had with them: Konrad Lai, Peter Hsu, C.J. Newburn, Greg Wright, Bob Zak, Shubu Mukherjee, and Quinn Jacobson.

My research required many hours of simulation time, and I would like to acknowledge those who helped make it possible. I thank Virtutech AB for supporting Simics and answering many of my technical questions, particularly Peter Magnusson, Andreas Moestedt and Ross Dickson. Many thanks are due to the Computer Systems Lab for all their support and help, especially John Perkins, David Parter and Paul Beebe. I would also like to thank the Condor project for their support.

My research has been financially supported in part by different sources for which I am grateful: the National Science Foundation with grants CCR-0324878, EIA-0205286, and EIA-9971256 and donations from IBM, Intel and Sun Microsystems.

On a personal level, I am certain I could not have made it this far without the support of my family and friends. I am forever indebted to my wife, Marwa, for supporting me during both the good days and the bad days. She has been a constant source of encouragement, caring, and inspiration for me. I am really grateful that she often preferred what is best for me over what is best for her, and I hope I can make it up to her. I am also deeply grateful to my father and mother for always encouraging me, pushing me forward, and keeping me motivated to finish my work. I could not have made it this far without their help, support, prayers, and kindness. I am also greatly indebted to them for not discouraging me from seeking to get my Ph.D. so far away from where they live. I would also like to thank my two sisters for their always thinking about me, keeping me in touch with their news, and for their continuous support and encouragement. I thank my uncle for his encouragement and support. I thank all members of my extended family, in Egypt and elsewhere, for keeping me in their prayers and for all their kindness.

I would also like to thank many of my Egyptian friends for staying in touch and for helping me through the past few years. Many thanks to Moataz Ali, Ayman El-Refaie, Ahmed Ayad, Ali Abdou, Essam El-Hanoony, Alaa Hassan, Ahmed Fahim, Adel Talaat, Khaled Kamal, Ahmad El-Guindy, Tamer Kadous, Essam Sharaf, Sherif Hatem, Amr Abdo, Amr Ali, Wael Abdelmegid, Ahmed Essmat, Ahmed Ghanem, Mohamed Eldawy, Mohamed Morsi, and all other friends in Madison that I might have forgotten to mention. Many thanks are also due to my friends in Egypt and elsewhere: Mohamed Samy, Mohamed Galal, Ahmed El-Mahdy, Ahmed Halal, Mohamed Mokbel, Sameh Elnikety, Khaled Diaa, Tamer Elsharnouby, Sameh Meligy, Tarek Khorshed, Moustafa Youssef, Moustafa Hammad, Ihab Ilyas, and all members of my graduation class in Alexandria. I would also like to acknowledge my alma mater, the University of Alexandria, and thank all professors there for teaching me about computer science and for their help in getting me started in graduate school. I would like to especially thank my Masters advisor Prof. Mohamed Ismail, and other people who wrote recommendation letters for me: Professors Nagwa El-Makky and Badr Aboulnasr.

I end just like I started by thanking God, for it is only by His blessings that I am able to accomplish this work or any other. He gave me the patience, the perseverance, the physical and the mental ability to do this. I pray He will continue to guide me in all my future endeavors.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

In today's information era, commercial applications—including on-line banking, airline reservations, web searching and browsing—have become essential to many aspects of everyday life. The increasing dependence on these multi-threaded, throughput-oriented applications drives the increasing demand for efficient, throughput-oriented computer systems. These workloads exhibit ample thread-level parallelism, which makes them suitable for running on multiprocessor systems.

Chip multiprocessors (CMPs) have become an increasingly appealing alternative to run such commercial workloads. The exponential increase in available on-chip transistors provides architects with the resources to build multiple processor cores and large shared caches on a single chip. However, given a fixed transistor (i.e., area) budget, designers must determine the "optimal" breakdown between cores and caches. This choice is not obvious, as the 2004 ITRS Roadmap [45] predicts that transistor performance will continue to improve faster than DRAM latency and pin bandwidth (26%, 10%, and 11% per year, respectively). The increasing number of processor cores on a single chip increases the demand on two critical resources: the shared cache capacity and the off-chip pin bandwidth. In this dissertation, we explore using compression to effectively increase these resources and ultimately overall system throughput. To achieve this goal, we identify two distinct and complementary designs where compression can help improve CMP performance: Cache Compression and Link Compression.

Cache compression stores compressed lines in the L2 cache, potentially increasing the effective cache size, reducing off-chip misses, and improving performance. Moreover, cache compression can also allow CMP designers to spend more transistors on processor cores. On the downside, decompression overhead can slow down cache hit latencies, which degrades performance for applications that would fit in an uncompressed cache. Such negative side-effects motivate a compression scheme that avoids compressing cache lines when compression is not beneficial.

Link (i.e., off-chip interconnect) compression compresses communication messages before sending to or receiving from off-chip system components. Link compression has the potential to increase the effective off-chip communication bandwidth, potentially reducing contention for pin bandwidth. Link compression can improve performance for applications that have a high demand for pin bandwidth, especially for bandwidth-limited configurations. Link compression can also shift the balance between cores and caches towards more cores. On the other hand, decompression overheads can degrade performance for workloads that are not bandwidth-limited.

In this dissertation, we propose using cache and link compression to improve the performance of chip multiprocessor systems. We introduce a simple compression scheme that is suitable for hardware compression of cache lines, and propose a compressed cache design based on that scheme. We develop an adaptive compression scheme that dynamically adapts to the costs and benefits of cache compression, and implements compression only when it helps performance. We propose and evaluate a CMP design that implements both cache and link compression. We show that compression interacts in a strong positive way with hardware prefetching, whereby a system that implements both compression and hardware prefetching can have a higher speedup than the product of the speedups due to either scheme alone. We derive a simple analytical model that can help provide qualitative intuition into the trade-off between cores, caches, communication and compression, and use full-system simulation to quantify this trade-off for a set of commercial workloads. While we focus in this dissertation on improving performance of commercial applications, we show that compression can also improve the performance of some (compressible) scientific applications.

In this chapter, we motivate why architects are currently building chip multiprocessors (Section 1.1). We discuss the technology and workload trends that guide CMP design (Section 1.2). We then discuss the role of compression in uniprocessor and CMP design (Section 1.3). We identify the main contributions of this dissertation (Section 1.4), and provide a roadmap for the remainder of this document (Section 1.5).

1.1 Why CMPs

The Need For More Throughput. In today's information era, commercial servers constitute the backbone of the global information and communication system infrastructure. Such servers run useful commercial applications that are essential to many aspects of everyday life such as banking, airline reservations, web searching and web browsing. As more people depend on these multi-threaded throughput-oriented applications, demand for more throughput is likely to increase for the forseeable future. Commercial servers must therefore improve their performance by providing more throughput to keep up with the application demand.

Commercial Server Design. Since commercial applications have abundant thread-level parallelism, commercial servers were designed as multiprocessor systems—or clusters of multiprocessors—to provide sufficient throughput. While traditional symmetric multiprocessors (SMPs) can exploit thread-level parallelism, they also suffer from a performance penalty caused by memory stalls due to cache misses and cache-to-cache transfers, both of which require waiting for long off-chip delays. Several researchers have shown that the performance of commercial applications, and database applications in particular, is often dominated by sharing misses that require cache-to-cache transfers [7, 14, 100]. To avoid these overheads, architects proposed several schemes to integrate more resources on a single chip. Barroso, et al., show that chip-level integration of caches, memory controllers, cache coherence hardware and routers can improve performance of online transaction processing workloads by a factor of 1.5 [16]. Simultaneous multithreading designs [39, 124] allow the processor to execute several contexts (or threads) simultaneously by adding per-thread processor resources. This approach also improves the performance of database applications compared to a superscalar processor with comparable resources [87]. The trend towards more integration of resources on a single chip is becoming more apparent in CMP designs where multiprocessor systems are built on a single chip.

Chip Multiprocessors (CMPs). As predicted by Moore’s law, the number of transistors on a single semiconductor chip has been increasing exponentially over the past 40 years [91]. Architects currently have enough transistors on a single chip that they can use to improve the throughput of multi-threaded applications. To achieve this goal, architects developed a system design in which multiprocessors are built on a single semiconductor chip [15, 51, 56, 68, 74, 90, 117, 128]. Chip multiprocessor (CMP) systems can provide the increased throughput required by multi-threaded applications while reducing the overhead incurred due to sharing misses in traditional shared-memory multiprocessors. A chip multiprocessor design is typically composed of two or more processor cores (with private level-one caches) sharing a second-level cache. CMPs in various forms are becoming popular building blocks for many current and future commercial servers. CMPs and multi-CMP systems have the potential to improve throughput for many multi-threaded applications.

1.2 Balanced CMP Design

Chip multiprocessors are becoming popular building blocks for commercial servers. However, an important question in CMP design is how to build a chip that can provide the best possible throughput for a given chip area. For a fixed transistor (i.e., area) budget, architects must decide on the “optimal” breakdown between cores and caches such that neither cores, caches nor communication is the only bottleneck. This choice is not obvious, since the number of transistors per chip is increasing at a much faster pace than DRAM latency or pin bandwidth, while software applications are demanding higher throughput every year. In this section, we discuss some technology and software trends that affect CMP design.

1.2.1 Technology Trends

Memory Wall. Over the past few decades, transistor performance has been improving at a much faster pace compared to memory performance. Wulf and McKee [132] show that the rate of improvement in microprocessor speed exceeds the rate of improvement in DRAM memory speed. While each is improving exponentially, the exponent for microprocessors is substantially larger than that for DRAMs, and the difference between diverging exponentials also grows exponentially. The 2004 ITRS Roadmap [45] predicts that transistor speed will continue to improve at a much faster annual rate (21%) over the next fifteen years compared to the rate of improvement in DRAM latency (10%). The trend toward increasingly deep pipelines [58, 60] further exacerbates this problem, increasing main memory latencies to hundreds of cycles.

Addressing the Memory Wall. The memory wall has been a problem for a long time, leading to many architectural enhancements that target hiding the memory latency. Thread-level speculation and multithreading [3, 5, 39, 108, 109], prefetching [26, 65, 66, 92, 102, 110], and runahead execution [37, 93] are among many schemes that target hiding memory access latencies by increasing memory-level parallelism. Value prediction targets reducing memory access latency by predicting load values that later have to be verified [86]. A more direct approach to hide memory latency is to avoid some cache misses by increasing the cache size. This can be achieved through cache compression [9, 24, 77, 139].

Pin Bandwidth Bottleneck. The presence of more processors on a single chip in CMPs can significantly increase demand on the off-chip pin bandwidth required for inter-chip and chip-to-memory communication. However, pin bandwidth is not improving at the same rate as transistor performance. According to the 2004 ITRS roadmap [45], the number of pins available per chip will increase at a rate of approximately 11% per year over the next fifteen years. This is a much lower rate than the predicted rate of increase in the number of transistors per chip, which is projected at 26% per year in the same span. This implies that the number of processor cores on a single chip will increase at a much faster rate compared to the number of communication pins available. Even though pins are expected to run at a higher frequency thus increasing the effective bandwidth, the on-chip frequency will increase at the same rate [45], which will offset the increase in pin frequency[1]. In addition, the cost per pin is predicted to decrease at a lower rate (~5%) than that of the increase in the number of pins [45]. This means that the overall cost of packaging will increase at a rate of approximately 5% a year. If no effort is made to reduce the cost further (or reduce the rate of increase in the number of pins), the overall packaging cost will double in the next fifteen years, a trend that is opposite to other design cost trends.

While the number of pins is one aspect of the pin bandwidth bottleneck problem, other factors also affect using on-chip signal pins to increase pin bandwidth. Increasing the number of communication pins requires increasing their pad and driver area [45]. Moreover, increasing the speed of each pin as a means to increase bandwidth also requires significant increases in area allocated to drivers. Overall, off-chip bandwidth appears to be a problem that will significantly increase for future CMP designs. The pin bandwidth bottleneck is a problem that can hinder the development of CMPs with a large number of cores. A good design balances demands for bandwidth against the limited number of pins and wiring area per chip [32]. In order to design CMPs—or multi-CMP systems—in which off-chip bandwidth is not a performance bottleneck, architects must find solutions to balance these systems by reducing their bandwidth requirements or increasing their pin bandwidth. To achieve these goals, several architectural, software, and technology proposals have been proposed to address pin bandwidth bottleneck.

Addressing Pin Bandwidth Bottleneck. Architectural and software proposals include increasing on-chip cache sizes, increasing area allocated for on-chip memory controllers, or using CMP-aware operating systems. Devoting more area for on-chip caches should decrease the required off-chip bandwidth at the expense of slowing down the increase in the number of on-chip processors. However, this solution

1. The 2004 ITRS roadmap predicts that the off-chip frequency is expected to increase at the same rate as processor frequency only for a small number of high-speed pins, which will be used with a large number of lower-speed pins to get the total off-chip bandwidth. This implies that while the off-chip latency will remain constant relative to processor frequency, the off-chip bandwidth will effectively decrease.

addresses only capacity and conflict misses, and not coherence misses. Devoting more chip area for on-chip memory controllers can increase pin bandwidth. For example, the Sun Niagara chip allocates four on-chip memory controllers to increase pin bandwidth beyond 20 GB/sec. [21]. However, the area allocated to memory controllers reduces the area used by cores and/or caches. CMP-aware operating systems can schedule threads that share a lot of data on the same chip to limit off-chip communications (i.e., similar to prior work on cache affinity process scheduling for SMPs [118, 126]). This can help systems that run different workloads at the same time, but provides less benefit for systems that run a homogeneous multi-threaded workload (e.g., OLTP).

Technology enhancements to address pin bandwidth bottleneck include modifying the memory interface as well as using different interconnect technologies. Fully-Buffered DIMM (FBDIMM) is a new technology designed to increase memory bandwidth and capacity [53]. FBDIMM uses a buffer as an interface between a memory controller and DRAM chips. The interface between the buffer and the memory controller is changed to a point-to-point serial interface (instead of a shared parallel interface). Such an interface change allows for a higher memory bandwidth per memory controller channel (nearly 7 GB/sec.) as well as a higher memory capacity. Optical interconnects are currently being pursued as a means to significantly increase pin bandwidth. The ITRS 2004 roadmap [45] identifies optical interconnects as one of the potential interconnect designs to succeed copper wires. Luxtera is currently designing optical links with a bandwidth per link greater than 10 Gb/sec. [103]. However, such technology requires significant changes to chip design and packaging [20].

Summary. The memory wall and pin bandwidth bottleneck are two technology trends whose impact is expected to increase over the next few years. Both trends can be addressed in a CMP design by increasing the chip area allocated to on-chip caches. This can reduce cache misses (thereby alleviating the impact of the memory wall), and can also reduce demand on off-chip pin bandwidth (thereby reducing the pin bandwidth bottleneck).

1.2.2 Workload Trends

Many current and future software applications have increasing throughput and computational demands. For example, systems at the top of the online transaction processing benchmark TPC-C performance list have improved throughput by more than 50% each of the last five years [119], with the current top performer achieving approximately 3.2 million transactions per minute. Media and gaming applications, which are increasing in popularity, also require increasing parallelism. World data doubles every three years and is now measured in billions of billions of bytes [36]. Intel predicts that software applications in the next decade and beyond will be more computationally intensive and can use more parallelism [75].

The above trends imply that current and future software applications demand more thread-level parallelism and computational power. Such demand can be satisfied by increasing the number of processor cores (or threads) on a CMP. So while technology trends favor allocating more area for shared caches, workload trends favor allocating more area for processor cores. CMP design has to balance the needs and requirements of software applications against technology limitations to build a system where none of the resources is the only bottleneck.

1.2.3 Balance in CMP Design

An important question in CMP design is how to use the limited area resources on chip to achieve the best possible system throughput for a wide range of applications. To achieve this goal, a CMP design has to balance cores, caches, and communications such that none of these resources is the only bottleneck. With few cores that cannot support enough threads, cores become a bottleneck and degrade system throughput. With too many cores and smaller caches, caches and/or pin bandwidth become a bottleneck and also degrade system throughput. Should the design center on caches, to hide DRAM latency and conserve pin bandwidth, or on cores, to maximize thread-level parallelism? The optimal balanced design point obviously lies somewhere between these two extremes.

Many hardware proposals —such as those we described in Section 1.2.1—address only one or two of the main technology and workload trends (i.e., memory wall, pin bandwidth bottleneck, and the increasing thread-level parallelism). Furthermore, some of these techniques reduce the impact of one at the expense of increasing the impact of another. For example, prefetching and thread-level speculation schemes can reduce the impact of the memory wall at the expense of increasing demand on pin bandwidth. In this dissertation, we show that compression can address all these requirements at the same time. Compression can increase the effective cache size at a small area cost and also increase the effective pin bandwidth. In effect, compression allows a CMP design where the optimal balanced design point has a larger effective cache size and pin bandwidth compared to a CMP design without compression.

1.3 Compression and Changing the Balance

In this dissertation, we advocate using compression to address constraints on cores, caches, and pin bandwidth. On-chip cache compression can increase the effective cache size without significantly increasing its area, thereby avoiding some off-chip misses. In addition, cache compression can potentially allow a design where more on-chip area is allocated to processor cores. Link compression can also reduce off-chip bandwidth demand for inter-chip and chip-to-memory communication, effectively increasing pin bandwidth. Both cache and link compression help achieve a balanced CMP system with higher throughput compared to a system without compression. We next describe how our proposed cache and link compression can affect uniprocessor and CMP design.

1.3.1 Cache Compression in Uniprocessors

We propose using cache compression to increase effective cache size, reduce off-chip misses and pin bandwidth demand, and ultimately improve system performance. Our proposed compressed cache design for a uniprocessor system is shown in Figure 1-1. We propose storing cache lines in a compressed format in the second level caches (and potentially memory) while leaving the L1 cache uncompressed. The benefits of this technique for many workloads are two-fold. Storing compressed cache lines can increase the effective cache size, potentially decreasing L2 miss rates and achieving better overall performance. In addition, transferring compressed data between the L2 cache and memory decreases the demand on pin bandwidth. We discuss this design in more detail in Chapter 3.

Unfortunately, cache compression also has a negative side effect, since L2 cache lines have to be decompressed before moving to the L1 cache or being used by the processor. This means that storing compressed lines in the L2 cache increases the L2 hit latency. While achieving a high compression ratio is important to increase the effective cache size, any cache compression algorithm should also have a small impact on L2 hit latency so as not to hinder performance in the common case. Most software-based compression algorithms are not suitable for low-latency hardware implementation. In addition, many hardware compression schemes that were previously proposed for memory compression have a significant relative decompression penalty when used for cache compression. To address this problem, we propose a simple low-latency cache compression algorithm that compresses cache lines on a word-by-word basis. Each word in a cache line is stored in a compressed form if it matches one of a few frequent patterns. Otherwise the word is stored in an

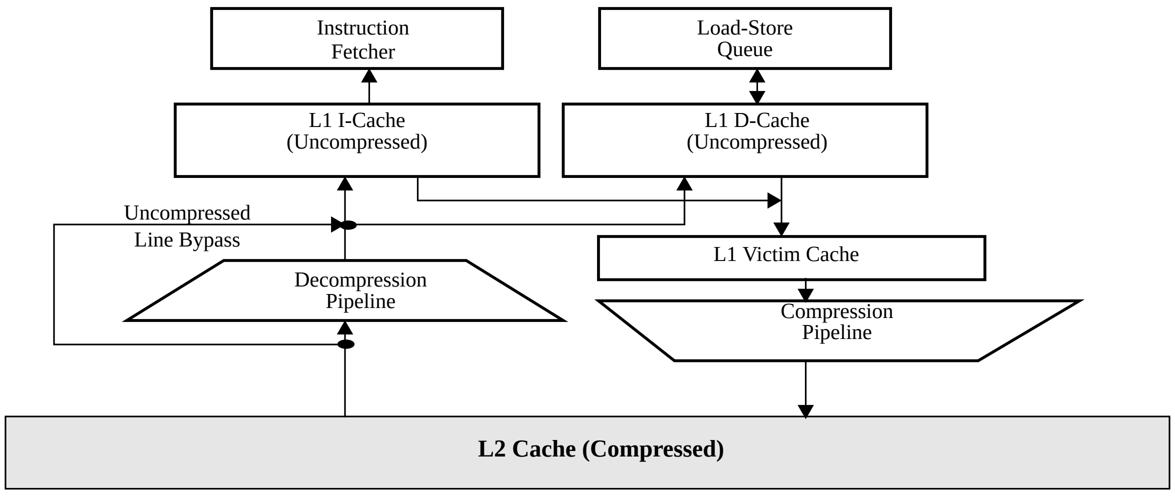


FIGURE 1-1. Compressed Cache Hierarchy in a Uniprocessor System

uncompressed form. We describe this simple hardware compression algorithm, Frequent Pattern Compression (FPC), in Chapter 3.

Even when using a simple hardware compression scheme with low decompression overhead, many workloads are still hurt by cache compression. For workloads whose working set sizes fit in an uncompressed cache, cache compression only serves to increase the L2 hit latency without having an impact on the L2 miss rate. In such cases, the cost of compression (i.e., increasing hit latency) outweighs its potential benefit (i.e., reducing miss rate), which may significantly hurt performance. To address this problem, we propose an adaptive cache compression scheme that uses the stack of the cache replacement algorithm [89] to determine when compression helps or hurts individual cache references. We use this cost and benefit information to implement a predictor that measures whether compression is helping performance (and therefore should be used for future cache lines) or hurting performance (and therefore should be avoided for future cache lines). This adaptive scheme achieves most of the benefits of always compressing while avoiding significant performance slowdowns when compression hurts performance. We describe this adaptive cache compression design in Chapter 4.

1.3.2 Cache Compression in Chip Multiprocessors

The increasing number of processor cores on a chip increases demand on shared caches and pin bandwidth. Hardware prefetching schemes further increase demand on both resources, potentially degrading performance. Cache compression addresses the increased demand on both of these critical resources in a CMP. In this dissertation, we propose a CMP design that supports cache compression, as shown in Figure 1-2. CMP cache compression can increase the effective shared cache size, potentially decreasing miss rate and improving system throughput. In addition, cache compression can decrease demand on pin bandwidth due to the decreased miss rate. We describe and evaluate our CMP compressed cache system in Chapter 5.

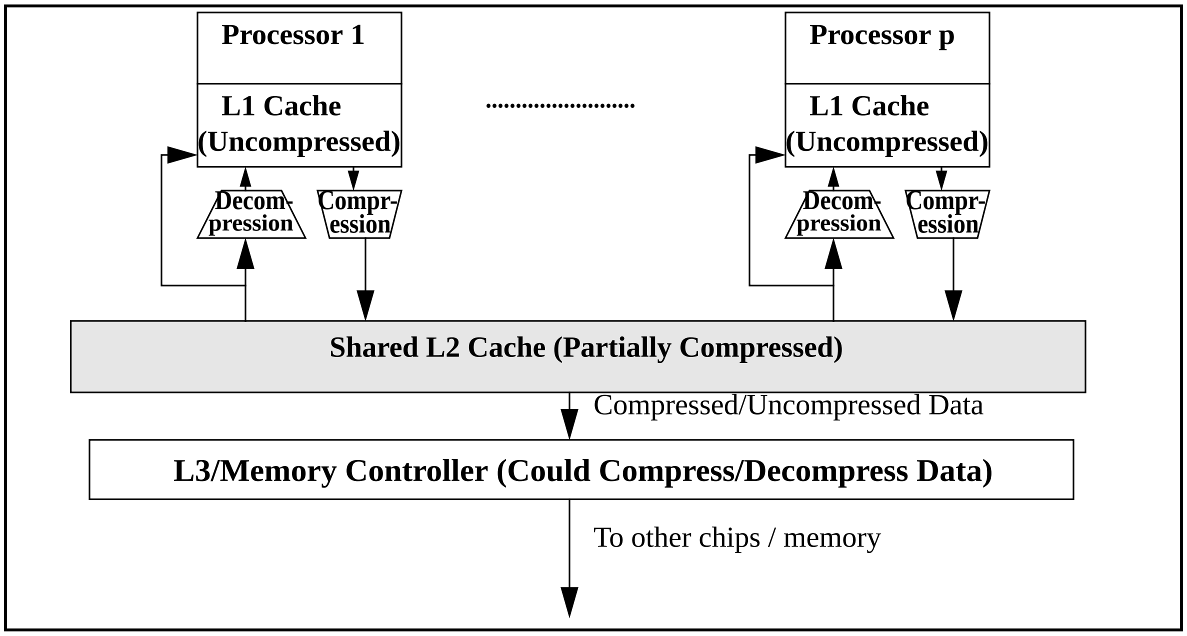


FIGURE 1-2. A Single-Chip p-core CMP with Compression Support

Due to the significant impact of the memory wall on performance, many existing uniprocessor and CMP systems implement hardware prefetching to tolerate memory latency [58, 68]. Prefetching is successful for many workloads on a uniprocessor system. For a CMP, however, prefetching further increases demand on both shared caches and pin bandwidth, potentially degrading performance for many workloads. This negative impact of prefetching increases as the number of processor cores on a chip increases. In Chapter 6, we show that cache compression can alleviate the increased demand on shared caches due to prefetching, leading to significant performance improvements. Combining compression with stride-based hardware prefetching can lead to speedups that exceed the product of speedups from either scheme alone.

1.3.3 Link Compression

CMP designs have limited off-chip bandwidth due to the chip's area and power limitations. Furthermore, limitations on both packaging area and the number of pins available on a chip exacerbate the pin bandwidth bottleneck. With a large number of processor cores on a CMP, limited pin bandwidth can lead to an unbalanced system. In addition, this pin bandwidth bottleneck can have a significant impact on performance due to increased queuing latencies for cache-to-cache inter-chip coherence requests as well as off-chip memory requests. For these reasons, future CMP designs should consider off-chip bandwidth a first order design constraint.

In any multiprocessor or CMP system, off-chip bandwidth is consumed by either address messages or data messages that are used to communicate between processors, multiprocessor chips, memory and/or directory. In this dissertation, we only target the bandwidth demand required for data messages. We propose using link compression to compress data messages before transferring to/from a CMP. We describe our CMP design with link compression support in Chapter 5.

Hardware prefetching schemes increase demand on pin bandwidth due to the increased volume of prefetch requests. Pin bandwidth demand increases significantly when prefetching's accuracy is low, leading to performance degradations due to increased queuing delays. Link compression alleviates the pin bandwidth demand increase due to prefetching, thereby turning significant performance losses due to prefetching into performance gains. We describe this positive interaction between compression and hardware prefetching in Chapter 6.

Both cache and link compression complicate the trade-off between cores, caches, and communication in a CMP. The optimal breakdown of a CMP area between cores and caches can change when a system supports compression. In Chapter 7, we use analytical modeling and simulation to study the trade-off between cores and caches for a fixed-area CMP, and the impact of compression on such trade-off. We show that an optimal, balanced design achieves a significantly higher throughput compared to unbalanced configurations. We show that cache and link compression can shift the optimal design to achieve higher throughput for many CMP configurations.

1.4 Thesis Contributions

In our view, the most important contributions of this dissertation are:

- **Frequent Pattern Compression (FPC).** We propose and evaluate a hardware-based compression scheme, Frequent Pattern Compression (FPC), that is suitable for compressing cache lines. We also propose a cache design based on this compression scheme (Chapter 3). Compared to other existing hardware-based compression schemes, FPC is less complex to implement in hardware, has a lower decompression overhead, and has a comparable compression ratio for cache lines.
- **Adaptive Cache Compression.** We develop an adaptive cache compression algorithm that dynamically adapts to the costs and benefits of cache compression (Chapter 4). This adaptive scheme achieves nearly all the benefit of cache compression when it helps, and avoids hurting performance when cache compression's overheads exceed its benefits.
- **CMP Cache and Link Compression.** We propose and evaluate a CMP design that supports both cache and interconnect (link) compression (Chapter 5). We show that cache compression improves performance by 5-18% for commercial workloads, and that link compression reduces their off-chip bandwidth demand by 30-41%. Both cache and link compression combine to improve commercial workloads' performance by 6-20%, and reduce their bandwidth demand by 34-45%.
- **Interactions Between Compression and Prefetching.** We study the interactions between cache compression and hardware-directed prefetching (Chapter 6). We show that the positive impact of hardware stride-based prefetching is significantly diminished for CMPs compared to uniprocessors, leading to performance degradations for some workloads. We show that compression and prefetching interact positively, leading to a combined speedup that is greater than the product of the speedups of prefetching alone and compression alone.

- **Model Balanced CMP Design.** We develop a simple analytical model that estimates throughput for different CMP configurations with a fixed area budget (Chapter 7). This model provides intuition into the trade-off between cores and caches, but makes many simplifying assumptions that significantly affect its accuracy. We use the model to qualitatively demonstrate the positive impact of cache and link compression on CMP throughput, and quantify these throughput improvements using simulation of commercial benchmarks.

1.5 Dissertation Structure

We begin this dissertation by discussing an overview of data compression and research efforts in hardware compression schemes (Chapter 2). In Chapter 3, we discuss our frequent pattern compression scheme (FPC), its hardware implementation and an evaluation of its main properties. We make minor changes to the compression scheme previously published as a technical report [10] by eliminating zero run-length encoding. We also provide a more thorough analysis of compression and decompression hardware and latencies. We further describe our compressed cache design which was first presented in our ISCA 2004 paper [9].

In Chapter 4, we show how compression can help some uniprocessor benchmarks while hurting others. This motivates our adaptive compression algorithm, which we describe and evaluate. The gist of this chapter was first published in our ISCA 2004 paper [9], but we extend the published work by analyzing the sensitivity of our adaptive compression scheme to different system parameters and discussing some of its limitations.

We describe our compressed cache and link CMP design in Chapter 5. In addition, we evaluate the performance of cache and link compression on an eight-core CMP, and its sensitivity to various design parameters. In Chapter 6, we study the interactions between compression and hardware prefetching. We define a terminology for such interactions, discuss different factors that cause positive and negative interactions, and evaluate such interactions on an eight-core CMP. We show that compression and prefetching interact in a strong positive way for many commercial and scientific applications. In Chapter 7, we present a simple analytical model that measures CMP throughput for a fixed chip area. We use this model to qualitatively evaluate optimal CMP configurations, and use simulation to quantitatively validate our model's conclusions. Chapter 8 concludes this dissertation and outlines some potential areas of future research.

Chapter 2

Compression Overview and Related Work

In this chapter, we present an overview of compression and related research. We first present a brief overview of data compression in Section 2.1. We discuss many hardware memory compression implementations in Section 2.2. We present an overview of prior work on cache compression (Section 2.3) and link compression (Section 2.4). We intend for this chapter to present an overview of related work and not as an exhaustive list for all prior research related to all contributions of this dissertation. In the next five chapters, we discuss related work that is relevant to each particular chapter's topic.

2.1 Compression Background

Data compression is a widely used technique that aims at reducing redundancy in stored or communicated data [84]. Compression has a wide variety of applications in software and hardware; including image compression [99], sparse data compression [115], web index compression [104, 143], main memory compression [73, 121], code compression [11, 23, 29, 81], and many applications for embedded processors [18, 82, 83, 136]. Some compression techniques are *lossless* where decompression can exactly recover the original data, while others are *lossy* where only an approximation of the original data can be recovered. Lossy compression is widely used in many applications where lost data do not affect their usefulness (e.g., voice, image, and video compression). In this dissertation, we only consider lossless compression since any single memory bit loss or change can affect the validity of results in most computer programs.

Data compression techniques provide a mapping from data messages (source data) to code words (compressed data). These techniques can be either static or dynamic [84]. Static techniques (e.g., Huffman coding[61]) provide a fixed mapping from data messages to code words. Dynamic (or adaptive) techniques can change that mapping over time. Dynamic techniques include adaptive Huffman algorithms [127], adaptive arithmetic coding [131], and Lempel-Ziv (LZ) coding [141, 142]. These dynamic techniques require only one pass on the input data (as compared to two for static Huffman encoding) [84]. Since dynamic techniques do not require knowing the data input beforehand, they are more widely used for hardware compression.

Most dynamic compression techniques operate on sequential inputs at a bit, byte, word or block granularity. However, several techniques where proposed to perform parallel compression [46, 48, 80, 111], where the input data can be partitioned and compressed in parallel. Parallel compression is better suited for fast hardware compression since parallel hardware circuitry can provide the necessary parallel speed. However, many of these parallel compression schemes achieve lower compressibility compared to their sequential counterparts. The success of parallel compression algorithms is measured by the compression and decompression speedup, as well as by the compression ratio they achieve. In this dissertation, we use the term ***compression ratio*** as the size of original uncompressed data divided by the size of data after compression.

The Lempel-Ziv (LZ) algorithm [141, 142] and its derivatives are currently the most popular class of lossless compression algorithms, and form the basis for many hardware implementations. LZ methods achieve higher compression ratios by parsing data input and defining source messages on the fly. The LZ algorithm consists of a rule for parsing strings of symbols from a finite alphabet into substrings whose length does not exceed a certain integer, and a coding scheme that maps these substrings sequentially into uniquely decipherable code words of fixed length [141]. Storer and Syzmanski [112] present a general model for data compression that encompasses LZ encoding, and discuss the theoretical complexity of encoding and decoding and the lower bounds on amount of compression obtainable. Franaszek, et al., present a parallel implementation of block-referential compression with lookahead, a technique that is similar to LZ77 but allows both backward and forward pointers to match locations [48].

2.2 Hardware Memory Compression

Dictionary-based vs. Significance-based Compression. Several researchers and hardware designers proposed hardware-based compression schemes to increase effective memory size. Most previous proposals in hardware cache or memory compression are hardware implementations of dictionary-based software compression algorithms (e.g., LZ77 [141]). Such hardware dictionary-based schemes depend mainly on (statically or dynamically) building and maintaining a per-block dictionary. This dictionary is used to encode words (or bytes) that match in the dictionary, while keeping words (bytes) that do not match in their original form with an appropriate prefix. Dictionary-based compression algorithms are effective in compressing large data blocks and files. Another class of compression algorithms, significance based compression, depend on the fact that most data types can be stored in a fewer number of bits than those used in the general case. We next discuss a few hardware memory compression implementations.

IBM's Memory Compression. IBM's Memory Expansion Technology (MXT) [121] employs real-time main-memory content compression that can be used to effectively double the main memory capacity without a significant added cost. MXT was first implemented in the Pinnacle chip [120], a single-chip memory controller. Franaszek, et al., described the design of a compressed random access memory (C-RAM), which formed the basis for the memory organization for the MXT technology, and studied the optimal line size for such an organization [47].

Data in main memory is compressed using a parallel version of the *Block-Referential Compression with Lookahead (BRCL)* compression algorithm**,** a derivative of the Lempel-Ziv (LZ77) sequential algorithm [141]. BRCL's parallel version, Parallel Block-Referential Compression with Directory Sharing, divides the input data block (1 KB in MXT) into sub-blocks (four 256-byte sub-blocks), and cooperatively constructs dictionaries while compressing all sub-blocks in parallel [48]. It decompresses data (with double clocking) at a speed of 8 bytes per cycle [121]. It depends on having long enough lines/pages to increase its overall compression ratio [48].

MXT is shown to have a negligible performance penalty compared to standard memory. In addition, memory contents for many applications and web servers can be compressed by a factor of two to one [2]. However, this scheme requires support from the operating system since it can change memory size and address mapping [1].

X-Match. Kjelso, et al., demonstrated that hardware main memory compression is feasible and worthwhile [73]. They used the X-Match hardware compression algorithm that maintains a dictionary and replaces each input data element (whose size is fixed at four bytes) with a shorter code when it matches with a dictionary entry. X-Match attempts to compress more data with a small dictionary by allowing partial matches of data words to dictionary entries [73]. Nunez and Jones [95] propose XMatchPRO, a high-throughput hardware FPGA-based X-Match implementation.

Other Hardware Memory Compression Designs. Ekman and Stenstrom [40] used our frequent pattern compression scheme (a significance-based compression algorithm which we discuss in detail in Chapter 3) to compress memory contents. Their compression scheme uses a memory layout that permits a small and fast TLB-like structure to locate the compressed blocks in main memory without a memory indirection. They arrange memory pages logically into a hierarchical structure with a different compressibility at each level.

Zhang and Gupta [137] introduce a class of common-prefix and narrow-data transformations for general-purpose programs that compress 32-bit addresses and integer words into 15-bit entities. They implemented these transformations by augmenting six data compression extension (DCX) instructions to the MIPS instruction set.

2.3 Cache Compression

Several researchers proposed hardware cache compression implementations that aim at increasing the effective cache size and reducing cache miss rate. These implementations apply known hardware compression algorithms to the contents of the L1 or the L2 cache. We next describe many of these cache compression proposals. We also describe significance-based compression schemes, which form the basis for our frequent pattern compression algorithm (Chapter 3).

Cache Compression and Related Designs. Lee, et al., propose a compressed memory hierarchy model that selectively compresses L2 cache and memory blocks if they can be reduced to half their original size [77, 78, 79]. Their selective compressed memory system (SCMS) uses a hardware implementation of the X-RL compression algorithm [73], a variant of the X-Match algorithm that gives a special treatment for runs of zeros. They propose several techniques to hide decompression overhead, including parallel decompression, selective adaptive compression for blocks that can be compressed to below a certain threshold, and the use of a decompression buffer to be accessed on L1 misses in parallel with an L2 access. Ahn, et al., propose several improvements on the X-RL technique that capture common values [4]. Chen, et al., propose a scheme that dynamically partitions the cache into sections of different compressibility using a variant of the LZ compression algorithm [24]. Hallnor and Reinhardt's Indirect-Index Cache [54] decouples index and line accesses across the whole cache, allowing fully-associative placement and the storage of compressed lines [55].

Frequent-Value-Based Compression. Yang and Gupta [134] found from analyzing SPECint95 benchmarks that a small number of distinct values represent a large fraction of accessed memory values. This value locality phenomenon enabled them to design energy-efficient caches [133] and data compressed caches [135]. In their compressed cache design, each line in the L1 cache represent either one uncompressed line or two lines compressed to at least half their original sizes based on frequent values [135]. Zhang, et al., designed a value-centric data cache design called the frequent value cache (FVC) [139], which is a small direct-mapped cache dedicated to holding frequent benchmark values. They showed that augmenting a direct mapped cache with a small frequent value cache can greatly reduce the cache miss rate. FVC represents a single dictionary for the whole cache, which increases the chances of a single word to be found and compressed with little space overhead. FVC designs are based on the observation that a few cache values are frequent and thus can be compressed to a fewer number of bits. However, a large FVC requires an increased decompression latency due to the increased FVC access time.

Significance-Based Compression. Significance-based compression is based on the observation that most data types (e.g., 32-bit integers) can be stored in a fewer number of bits compared to their original size. For example, sign-bit extension is a commonly implemented technique to store small integers (e.g., 8-bit) into 32-bit or 64-bit words, while all the information in the word is stored in the least-significant few bits.

In contrast with dictionary-based compression schemes (e.g., Lempel-Ziv), significance-based compression [23, 27, 28, 42, 69, 72] does not incur a per-line dictionary overhead. Hardware implementations of significance-based compression schemes can be simpler and faster when compared to dictionary-based schemes. Both of these properties make significance-based compression more suitable for the typically-short cache lines. As an example, Kim, et al., use a significance-based compression scheme to store compressed data in the cache and reduce cache energy dissipation [72]. However, compressibility can be significantly lower for long cache lines when compared to LZ-based compression. Since these significance-based compression algorithms were initially proposed to compress communication messages, we discuss such techniques in the next section.

2.4 Link Compression

Communication bandwidth compression (which we refer to in this dissertation as link compression) is used to reduce memory traffic and increase the effective memory bandwidth. Traffic can be reduced by "compacting" cache-to-memory address streams [42] or data streams [28]. Benini, et al., propose a data compression/decompression scheme to reduce memory traffic in general-purpose processor systems [19]. They propose storing uncompressed data in the cache, and compressing data on the fly when transferring it to memory. They also decompress memory-to-cache traffic on the fly. They use a differential compression scheme that is based on the assumption that it is likely for data words in the same cache line to have some bits in common [18].

Significance-based Link Compression. Many link compression proposals are variations of significance-based compression. Farrens and Park [42] make use of the fact that many address references—transferred between processor and memory—have redundant information in their high-order (i.e., most significant) portions. They cached these high-order bits in a group of dynamically allocated base registers and only transferred the low-order address bits in addition to small register indices (in place of the high-order address bits) between the processor and memory. Citron and Rudolph [28] use a similar approach for address and data words. They store common high-order bits in address or data words in a table and transfer only an index plus the low order bits between the processor and memory. Canal, et al., proposed a scheme that compresses data, addresses and instructions into their significant bytes while using two or three extension bits to maintain significant byte positions [23]. They use this method to reduce dynamic power consumption in a processor pipeline. Kant and Iyer [69] studied the compressibility properties of address and data transfers in commercial workloads, and reported that the high-order bits can be predicted with high accuracy in address transfers but with less accuracy for data transfers. Citron [27] utilized the low entropy of address and data messages to transfer compressed addresses and data off chip as a stopgap solution to reduce off-chip wire delay. He proposes transferring high-entropy data directly on the bus, while compacting low-entropy data into a fewer number of bits before sending them on the bus.

2.5 Summary

Hardware compression has been proposed and used to compress memory, caches and communication bandwidth. In this section, we described many hardware compression proposals. Compressions schemes used in hardware implementations are either dictionary-based or significance-based. Dictionary-based compression algorithms depend on building a dictionary and using its entries to encode repeated data values. Significance-based compression algorithms are based on the observation that most data types can be stored in a fewer number of bits compared to their original size. Dictionary-based algorithms typically have high compression ratios, while significance-based algorithms can have lower compression or decompression overheads.

In the next chapter, we propose a significance-based compression scheme that provides reasonable compressibility for the typically short cache lines with relatively fast compression and decompression hardware. This scheme, Frequent Pattern Compression (FPC), compresses a cache line on a word-by-word basis. For each word, FPC detects whether it falls into one of the patterns that can be stored in a smaller number of bits, and stores it in a compressed form with an appropriate prefix.

Chapter 3

Compressed Cache Design

As semiconductor technology continues to improve, the rising disparity between processor and memory speed increasingly dominates performance. Effectively using the limited on-chip cache resources becomes increasingly important as memory latencies continue to increase relative to processor speeds. Cache compression has been previously proposed to improve the effectiveness of cache memories [9, 24, 55, 77, 79, 135, 139]. Compressing data stored in on-chip caches increases their effective capacity, potentially reducing misses and improving performance. However, for complex hardware compression schemes, decompression overheads can offset compression benefits. For cache compression to be an appealing solution, it is necessary to develop hardware compression algorithms with low decompression overheads.

In this chapter, we propose and evaluate a simple significance-based compression scheme that has low compression and decompression overheads (Section 3.1). This scheme, Frequent Pattern Compression (FPC), compresses individual cache lines on a word-by-word basis by storing common word patterns in a compressed format accompanied by an appropriate prefix. This simple scheme provides comparable compression ratios to more complex schemes that have higher decompression overheads (Section 3.2). For a 64-byte cache line, compression can be completed in three cycles and decompression in five cycles, assuming 12 fanout-of-four (FO4) delays per cycle (Section 3.3).

In order to make use of FPC in cache compression, we propose a compressed cache design in which data is stored in a compressed form in the L2 caches, and in an uncompressed form in the L1 caches (Section 3.4). L2 cache lines are compressed to predetermined sizes to reduce decompression overhead, never exceeding their original size. Our compressed cache design, the Decoupled Variable-Segment Cache, builds on FPC.

This design decouples tag and data areas in the cache and divides the data area into compression units (segments). Using small 8-byte segments allows our cache design to store more compressed lines into the same space allocated for fewer uncompressed lines.

In this chapter, we make the following contributions:

- We propose a hardware compression scheme, Frequent Pattern Compression (FPC), that is suitable to compress cache lines in the L2 cache and beyond. We show that FPC achieves comparable compression ratios to more complex hardware schemes for a wide range of scientific and commercial applications.
- We propose a hardware implementation for FPC and estimate its complexity. We show that decompression of a 64-byte cache line can be performed in five cycles (or fewer), and compression in three cycles, assuming a 12 FO4 delays per cycle.
- We propose a compressed cache hierarchy that stores uncompressed lines in the L1 cache, and compressed lines in the L2 cache. We propose a compressed cache design, the Decoupled Variable-Segment Cache, that decouples tag and data areas in the cache and divides the data area into compression units (segments). We use our compressed cache design throughout this dissertation to evaluate compressed caches in uniprocessors and chip multiprocessors.

3.1 Frequent Pattern Compression (FPC)

In contrast with dictionary-based compression schemes, significance-based compression [23, 27, 28, 42, 69, 72] does not incur a per-line dictionary overhead, as we described in the previous chapter. In addition, compression and decompression hardware is faster than dictionary-based encoding and decoding. These properties make significance-based compression schemes more suitable for the typically-short cache lines. However, compressibility can be significantly lower for long cache lines.

TABLE 3-1. Frequent Pattern Encoding

Prefix	Pattern Encoded	Data Size
000	Zero	0 bits (no data stored)
001	4-bit sign-extended	4 bits
010	One byte sign-extended	8 bits
011	halfword sign-extended	16 bits
100	halfword padded with a zero halfword	The nonzero halfword (16 bits)
101	Two halfwords, each a byte sign-extended	The two bytes (16 bits)
110	word consisting of repeated bytes	8 bits
111	Uncompressed word	Original Word (32 bits)

In this dissertation, we propose and use a significance-based compression scheme, *Frequent Pattern Compression (FPC)* to compress cache lines. This scheme is based on the observation that some data patterns are frequent and are also compressible to a fewer number of bits. For example, many small-value integers can be stored in 4, 8 or 16 bits, but are normally stored in a full 32-bit word. These values are frequent enough to merit special treatment, and storing them in a more compact form can increase the cache capacity. Zeros are also frequent and merit special treatment, as in the X-RL scheme [73]. FPC is a compression algorithm that strives to achieve most of the benefits of dictionary-based schemes while keeping the per-line overhead at a minimum.

3.1.1 Frequent Patterns

Frequent Pattern Compression (FPC) compresses / decompresses on a cache line basis. Each cache line is divided into 32-bit words (e.g., 16 words for a 64-byte line). Each 32-bit word is encoded as a 3-bit prefix plus data. Table 3-1 shows the different patterns corresponding to each prefix.

Each word in the cache line is encoded into a compressed format if it matches any of the patterns in the first seven rows of Table 3-1. These patterns are: zero (an all-zero word), 4-bit sign-extended, one byte sign-extended, one halfword sign-extended, one halfword padded with a zero halfword, two byte-sign-extended halfwords, and a word consisting of repeated bytes (e.g. “0x20202020”, or similar patterns that can be used for data initialization). These patterns are selected based on their high frequency in many of our integer and commercial benchmarks. A word that doesn’t match any of these categories is stored in its original 32-bit format. All prefix values are stored at the beginning of the line to speed up decompression.

3.1.2 Segmented Frequent Pattern Compression (S-FPC)

To exploit compression, the L2 cache must be able to compress cache lines into a fewer number of bits compared to their original size. In theory, a cache line can be compressed into any number of bits. However, such designs add more complexity to cache management. In most practical cache designs, cache lines cannot occupy an arbitrary number of bits. In designing a practical compressed cache, selecting a specific base *segment* size is critical. Small segment sizes decrease fragmentation, therefore increasing compression ratios at the expense of a higher cache design complexity. The opposite is true for large segment sizes.

In our compressed cache design, the decoupled variable segment cache (Section 3.4), we selected a segment size of eight bytes. Each cache line can be stored as a group of one or more 8-byte segments. For example, a 64-byte line can be stored in 1-8 segments. A compressed line is padded with zeros until its size gets to be a multiple of the segment size. These extra zeros (that do not correspond to any prefixes) are ignored during decompression. While this approach doesn’t permit high compression ratios for some cache lines (e.g., all zero lines), it allows for a more practical and faster implementation of cache line accesses.

3.2 FPC Evaluation

We evaluate the FPC scheme in terms of its achieved compression ratio compared to other compression schemes. We show compression results for our frequent patterns, and demonstrate that zeros are the most frequent. We also analyze the performance of segmented compression, and show the sensitivity of compression ratios to FPC’s base segment size.

3.2.1 Workloads

To evaluate the FPC scheme against alternative schemes, we used several multi-threaded commercial workloads from the Wisconsin Commercial Workload Suite [6]. We also used eight of the SPEC 2000 benchmarks [114], four from the integer suite (SPECint2000) and four from the floating point suite (SPECfp2000). All workloads run under the Solaris 9 operating system. These workloads are briefly described in Table 3-2. We ran multiple simulation runs for each benchmark, and measured compression

TABLE 3-2. Workload Descriptions

Online Transaction Processing (OLTP). DB2 with a TPC-C-like workload. The TPC-C benchmark models the database activity of a wholesale supplier, with many concurrent users performing transactions. Our OLTP workload is based on the TPC-C v3.0 benchmark using IBM's DB2 v7.2 EEE database management system. We use a 5 GB database with 25,000 warehouses stored on eight raw disks and an additional dedicated database log disk. We reduced the number of districts per warehouse, items per warehouse, and customers per district to allow more concurrency provided by a larger number of warehouses [6]. There are 16 simulated users, and the database is warmed up for 100,000 transactions.
Java Server Workload: SPECjbb. SPECjbb2000 is a server-side java benchmark that models a 3-tier system, focusing on the middleware server business logic. We use Sun's HotSpot 1.4.0 Server JVM. Our experiments use two threads and two warehouses, a data size of ~44 MB, and a warmup interval of 200,000 transactions.
Static Web Serving: Apache. We use Apache 2.0.43 for SPARC/Solaris 9, configured to use pthread locks and minimal logging as the web server. We use SURGE [13] to generate web requests. We use a repository of 20,000 files (totalling ~500 MB), and disable Apache logging for high performance. We simulate 400 clients each with 25 ms think time between requests, and warm up for 50,000 requests.
Static Web Serving: Zeus. Zeus is another static web serving workload driven by SURGE. Zeus uses an event-driving server model. Each processor of the system is bound by a Zeus process, which is waiting for web serving event (e.g., open socket, read file, send file, close socket, etc.). The rest of the configuration is the same as Apache (20,000 files of ~500 MB total size, 400 clients, 25 ms think time, 50,000 requests for warmup).
SPEC2000. We use four integer benchmarks (bzip, gcc, mcf and twolf) and four floating point benchmarks (ammp, applu, equake, and swim) from the SPECcpu2000 set to cover a wide range of compressibility properties and working set sizes. We use the first reference input for each benchmark. We warm up caches of each benchmark run for 1 billion instructions.

statistics periodically every 300 million cycles for each simulation run. For each data point in our results, we present the average and the 95% confidence intervals of these multiple measurements to account for both time and space variability [8].

3.2.2 Compression Ratio

To evaluate the success of our compression scheme, we estimated the compressibility properties of our set of benchmarks. For each simulation run, we computed compression statistics for the whole L2 cache contents every 300 million cycles (after a warm-up interval). Assuming variable length cache lines that can occupy any number of bits, we compared the compression ratio from our Frequent Pattern Compression scheme (FPC) with two other memory compression schemes:

- The X-RL algorithm [73] used in some compressed cache implementations [77, 78, 79].
- The Block-Referential Compression with Lookahead (BRCL) scheme [48], applied to cache lines. This scheme is a serialized implementation of the parallel compression scheme used for memory compression in the IBM MXT technology [121]. The compression ratio of this scheme presents an upper bound on the compressibility of its parallel version.

We also compare against the “Deflate” algorithm used in the gzip unix utility, which combines an LZ-variant implementation with Huffman encoding of codewords in the dictionary. For this algorithm, we run the gzip utility on the whole cache snapshot file (as opposed to 64-byte lines individually compressed by the other three schemes). The “Deflate” algorithm is used to provide a practical bound on compressibility of dictionary-based schemes for arbitrarily long cache lines.

Figure 3-1 compares compression ratios for the four compression schemes. While FPC is faster to implement in hardware, it provides comparable compression ratios to the dictionary-based X-RL and BRCL, and even approaches gzip for some benchmarks. For some floating point benchmarks (applu and swim),

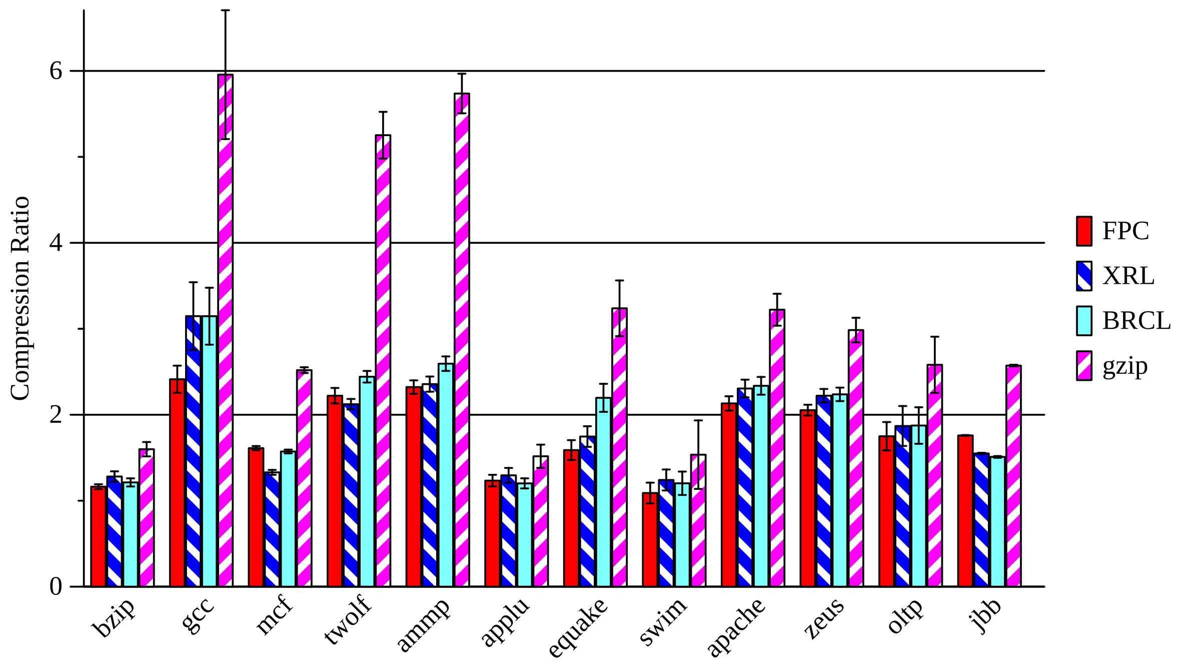


FIGURE 3-1. Compression ratios (original size / compressed size) for FPC, XRL, BRCL and gzip

none of the hardware schemes achieve a high compression ratio, and even gzip's compression ratio was low. We attribute this to the low compressibility of floating point numbers, since we only consider lossless compression schemes. FPC's compression ratio is higher than 2.0 for five of the twelve benchmarks. We note that the wide confidence intervals for some benchmarks' compression ratios (e.g., gcc) is due to these benchmarks exhibiting different phase behavior during the benchmarks' runtime. Cache lines are significantly more compressible in some program phases compared to other (less compressible) phases.

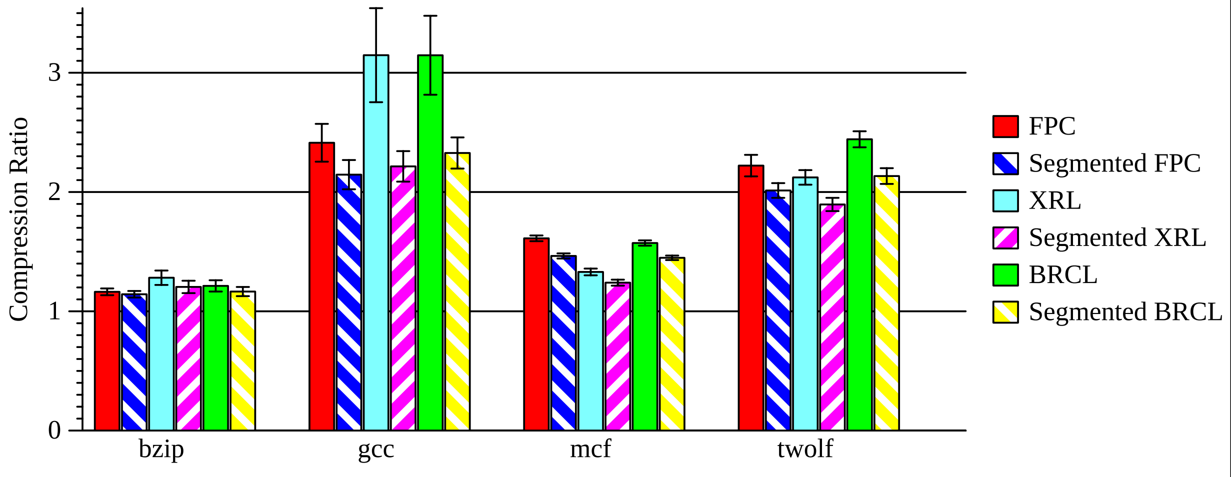
As we discussed in the previous section, cache lines cannot occupy any arbitrary number of bits in most practical cache designs. Restricting the compressed line sizes to a certain subset of all possible lengths (as we do in our segmented design) partially reduces compressibility. To assess the loss in compressibility, we compare the compression ratio from our Segmented Frequent Pattern Compression scheme (Segmented-FPC) against the compression ratio from the Frequent Pattern Compression scheme assuming variable-length lines are possible (FPC). We also compare against segmented and ideal versions of the XRL and BRCL. Figure 3-2 shows the compression ratios for our benchmarks.

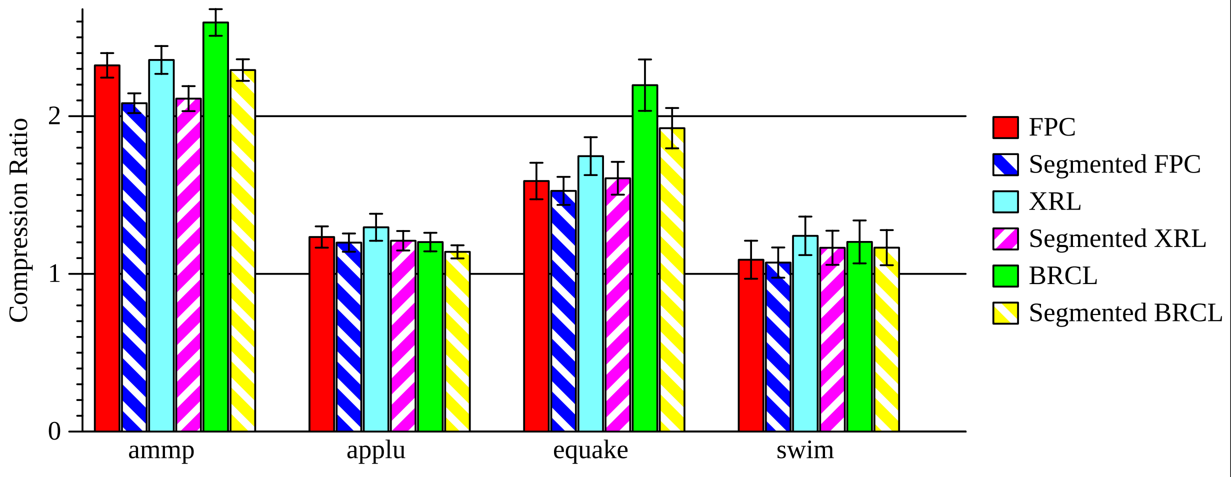
We note that the simple scheme (Segmented-FPC) achieves most of the compression benefit from variable-length lines of the ideal version (FPC). Segmented-FPC has compression ratios of 1.15-2.15 for the four SPECint2000 benchmarks, 1.07-2.08 for the four SPECfp2000 benchmarks, and 1.61-1.98 for the four commercial benchmarks. We note that OLTP had a low compression ratio since its data is randomly generated. A real OLTP application would have much less randomness, and thus have a higher compression ratio. In addition, our FPC scheme only targets data and not code which constitutes a large fraction of cache lines in commercial benchmarks, especially OLTP.

Compression ratios are, on average, higher for integer benchmarks compared to floating point benchmarks. For example, only 3.3% of all cache lines across our *swim* snapshots are compressible. However, some benefit is still possible for floating point benchmarks with a high percentage of zero words. This low compression ratio is because floating point numbers (except zero) rarely fit any of the frequent patterns. However, low floating point compression ratios is not only limited to the FPC scheme. Lossless compression remains a hard problem for floating point data, even for more complex software schemes that depend on knowing the specific application domain of floating point data (e.g., compression of floating-point geometry data [64], fluid dynamics data compression using wavelets [122], or JPEG image compression [125]).

3.2.3 Which Patterns Are Frequent?

Frequent Pattern Compression (FPC) is based on the observation that some word patterns are more frequent than others. We experimented with cache snapshots for our different benchmarks to come up with a reasonable set of frequent patterns (described in Table 3-1). Figure 3-3 shows the relative frequency of incompressible words, zero words and words compressible to 4, 8 and 16 bits. The 4-, 8-, and 16-bit patterns are present with various frequencies across our integer and commercial benchmarks. Unfortunately,





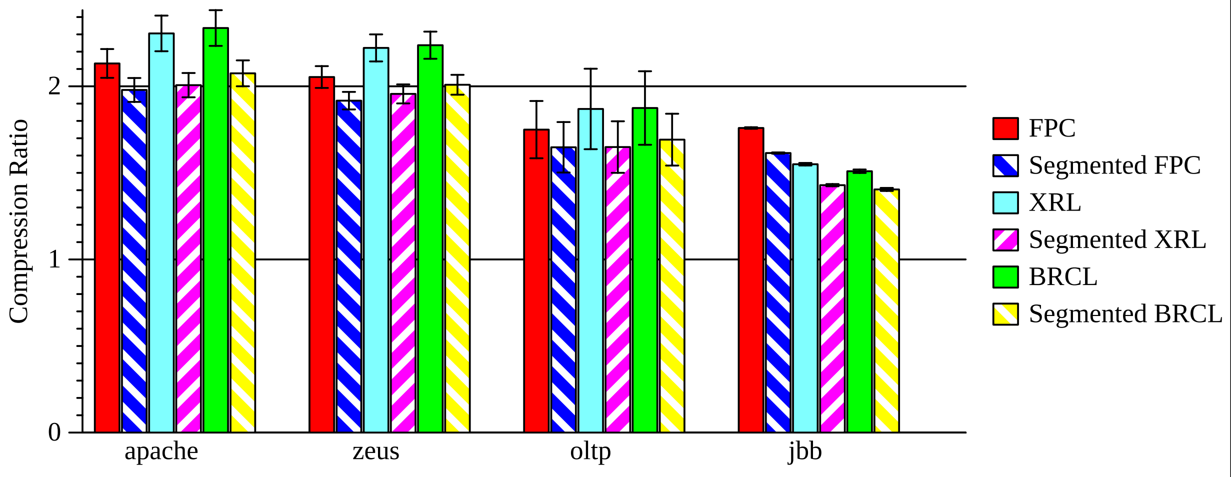


FIGURE 3-2. Compression ratios for segmented and variable-length FPC, XRL and BRCL. The three graphs show ratios for SPECint, SPECfp and commercial benchmarks

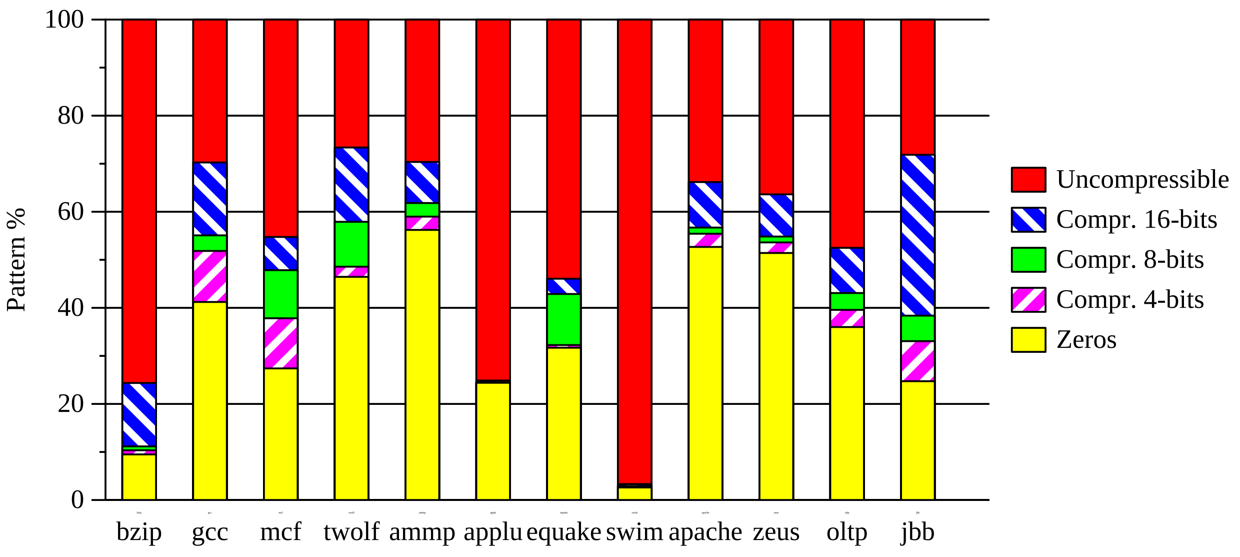


FIGURE 3-3. Frequent Pattern Histogram

most of the words in floating point benchmarks are incompressible with FPC, since its patterns are mainly integer patterns.

As Figure 3-3 demonstrates, zero words are the most frequent compressible pattern across all benchmarks. For five benchmarks (gcc, twolf, ammp, apache, zeus), at least 40% of all cache words are zeros. This causes some compression techniques (e.g., X-RL) to specifically optimize for runs of zeros. Figure 3-4 shows the average number of zeros in a zero run for our set of benchmarks. The figure shows that only four benchmarks (ammp, apache, zeus, oltp) have an average of 4 or more zeros in a zero run. In developing the FPC scheme, we had two options to compress zeros: either to have a prefix for each zero word with no data, or to encode zero runs with a single prefix and save the length of the run in the data part corresponding to that prefix. However, the savings that could be achieved using the second alternative (zero run encoding) would be mostly lost due to segmentation in S-FPC. Furthermore, the first alternative—which we selected—also allows implementations with a lower compression and decompression overheads.

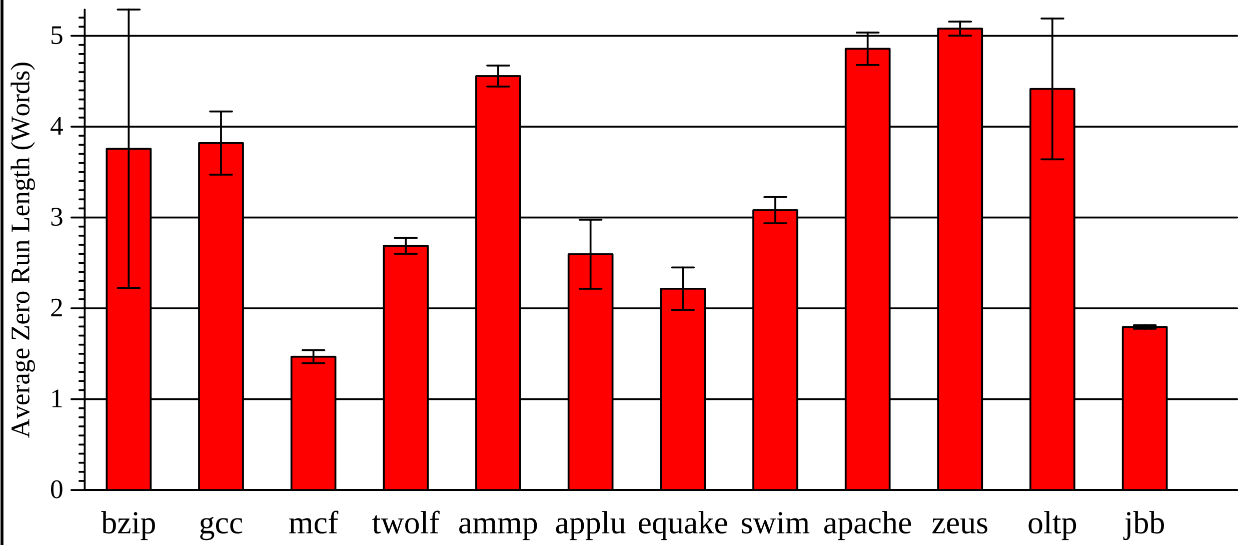


FIGURE 3-4. Average number of words in a zero run for our ten benchmarks. Large confidence intervals are due to the variability of cache contents between different time samples

3.2.4 Sensitivity to Segment Size

In designing a practical compressed cache implementation, selecting a specific base segment size is an important design decision. A compressed line can only be stored in a size that is an integer multiple of the base segment size. Selecting a small segment size would likely decrease the amount of fragmentation, which allows for higher compression ratios. However, a small segment size increases cache design complexity. The opposite is true for large segment sizes. Cache design should balance the trade-off between these two conflicting design constraints. We selected a base segment size of 8-bytes (i.e., up to 8 segments for 64-byte lines) in the Segmented FPC design as a reasonable design point.

Figure 3-5 shows the sensitivity of FPC to the base segment size. The four bars for each benchmark represent compression ratios if we divide up a 64-byte cache line into two segments (i.e., 32-byte segments), four (16-byte segments), eight (8-byte segments, which is the same as Segmented-FPC in Figure 3-2), and 64 (1-byte segments). Our 8-byte-segment design increases the compression ratio by 4-59% vs. 32-byte

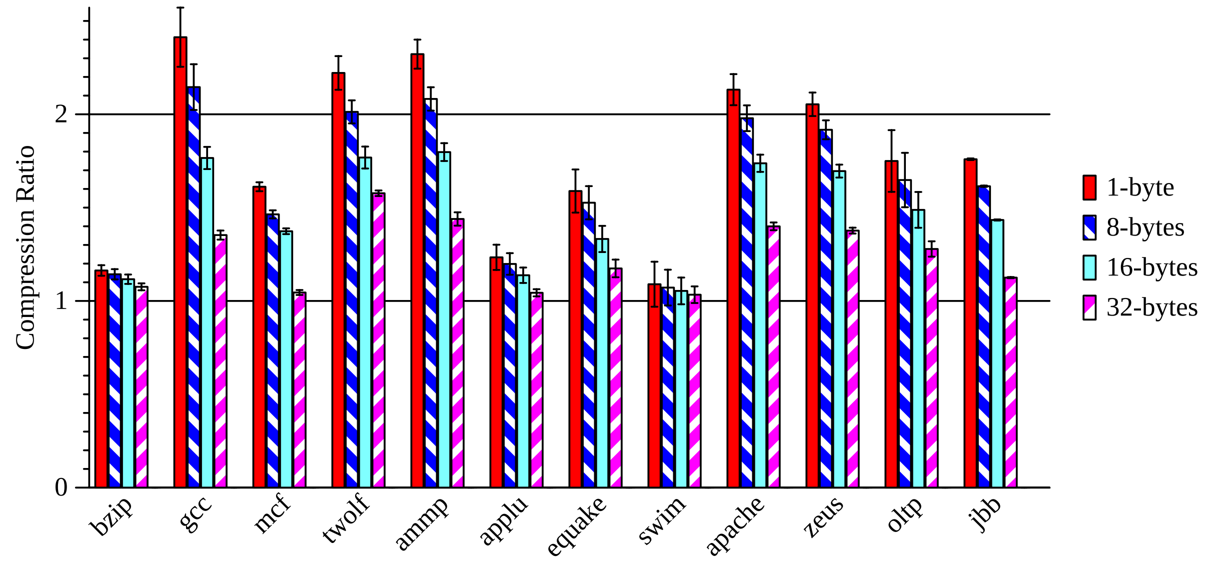


FIGURE 3-5. FPC Compression ratios for segment sizes (1 byte to 32 bytes)

segments, and 2-22% vs. 16-byte segments. Compared to 1-byte segments, our 8-byte segment design has a 2-12% lower compression ratio. This data shows that our 8-byte segment selection is a reasonable design point.

We characterize our benchmarks further by showing the percentage of cache lines (across all snapshots for each benchmark) that can be compressed into 1-8 segments (Figure 3-6). The figure shows that different benchmarks favor different sizes. For example, nearly 97% of *swim* lines are uncompressed (8 segments), while nearly 55% of *mcf* lines can be compressed to six segments. For completeness, we show a more detailed distribution in Figure 3-7, demonstrating the cumulative distribution of compressed cache line sizes (1-512 bits) assuming ideal FPC compression (i.e., 1-bit segments) for our twelve benchmarks.

3.3 FPC Hardware Implementation

Frequent Pattern Compression is an appealing compression scheme for cache lines because it achieves relatively high compression ratios with a small overhead for compression and decompression. In this section,

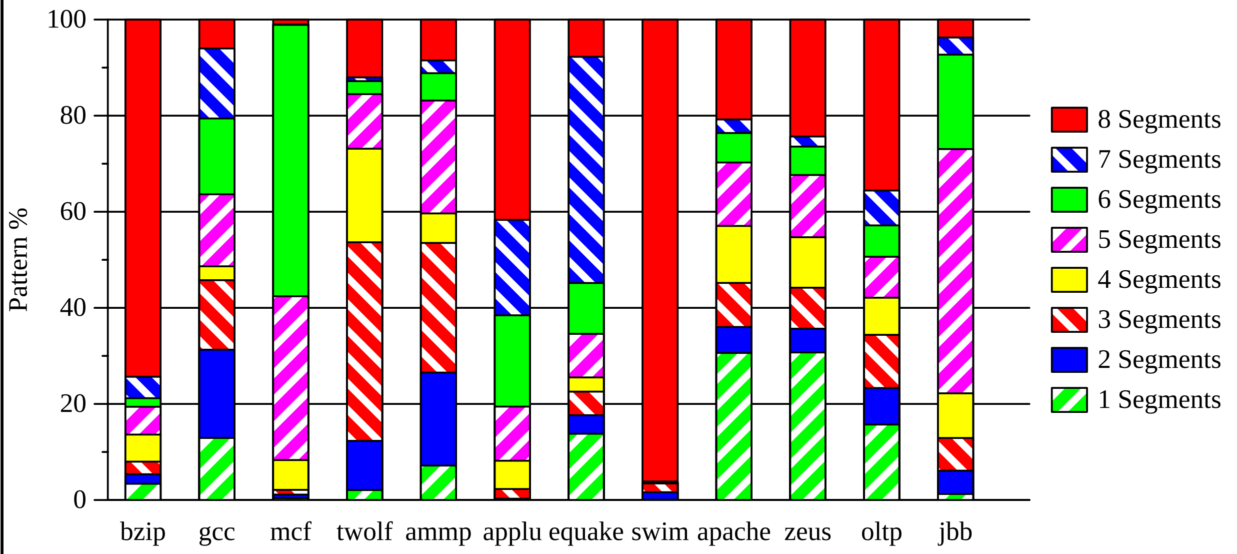


FIGURE 3-6. Segment Length Histogram: Percentage of cache lines compressed into 1-8 Segments

we present high-level and gate-level designs of FPC's compression and decompression hardware, and estimate their circuit delay.

3.3.1 High-Level Design

We propose a compressed cache design in which data is stored uncompressed in the level-1 caches and compressed in the level-2 caches (Section 3.4). This helps reduce many of the costly L2 cache misses that hinder performance, while not affecting the common case of an L1 hit. However, such a design adds the overhead of compressing or decompressing cache lines when moved between the two levels. FPC allows a relatively fast implementation of both of these functions.

Compression. Cache line compression occurs when data is written back from the L1 to the L2 cache, or when data is fetched from memory to the L2. A cache line is compressed easily using a simple circuit that checks each word (in parallel) for pattern matches. If a word matches any of the seven compressible patterns, a simple encoder circuit is used to encode the word into its most compact form. If no match was found, the whole word is stored with the prefix '111'.

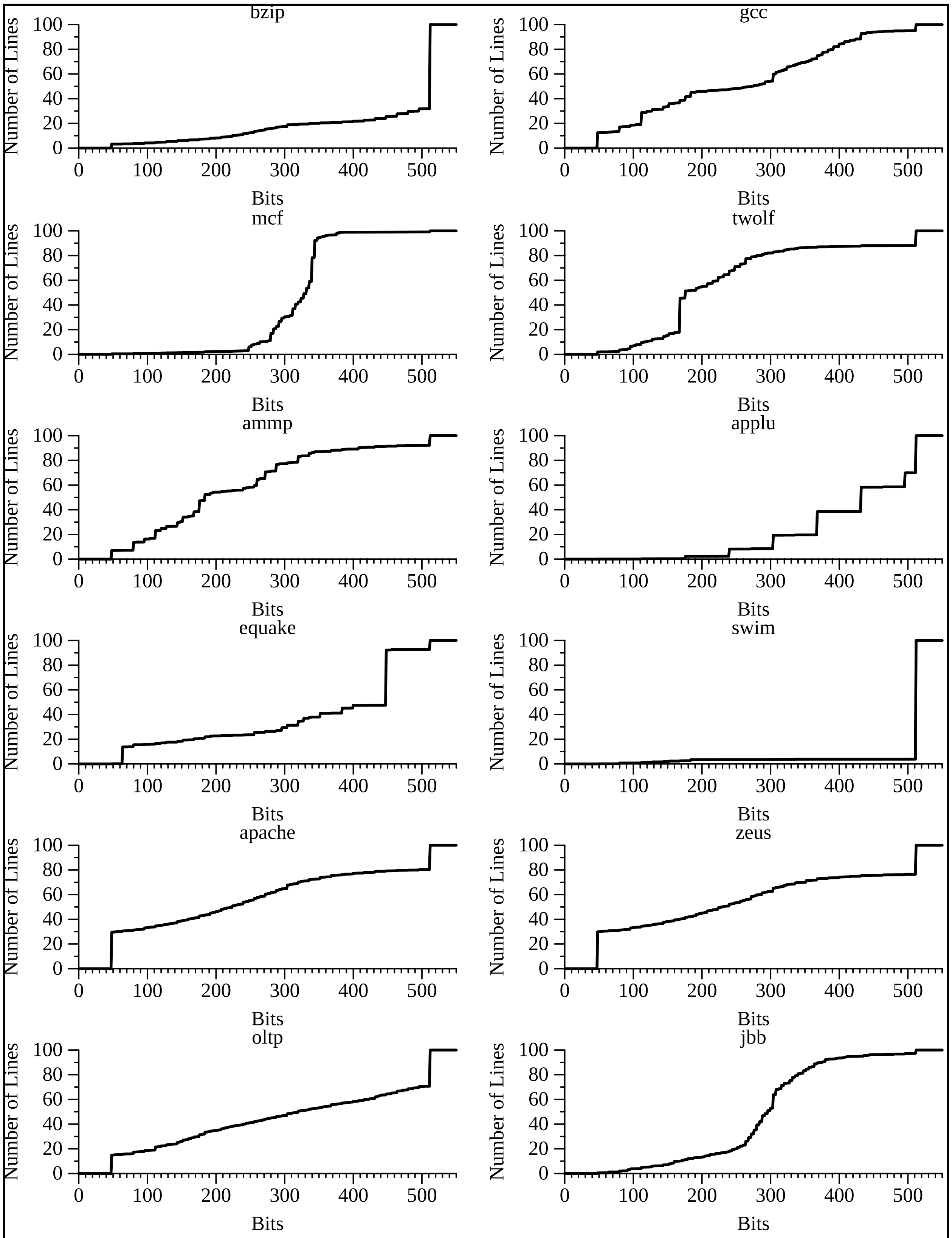


FIGURE 3-7. Cumulative Distribution of Compressed Line Lengths (1 to 512 bits)

Cache line compression can be implemented in a memory pipeline, by allocating three pipeline stages on the L1-to-L2 write path (one for pattern matching, and two for gathering the compressed line). A small victim cache that contains a few entries in both compressed and uncompressed form can be used to hide the compression latency on L1 writebacks. However, since compression is off the critical path, we do not consider compression latency a first order design constraint.

Decompression. Cache line decompression occurs when data is read from the L2 to the L1 cache. This is a frequent event for most benchmarks whose working sets do not fit in the L1 cache. Decompression latency is critical since it is directly added to the L2 hit latency, which is on the critical path. Decompression is a slower process than compression, since compressed lengths for all words in the line (except the last) have to be added. Each prefix is used to determine the length of its corresponding encoded word and therefore the starting location of all the subsequent compressed words. Figure 3-8 presents a schematic diagram for a five-stage hardware pipeline that can be used to decompress 64-byte cache lines. Each pipeline stage is 12 FO4 delays or less, assuming the parallel resources required are available for the parallel adder, shift register and pattern decoder. Assuming one processor cycle requires 12 FO4 gate delays, this means that the decompression latency is limited to five processor cycles.

3.3.2 Gate-Level Design

In order to estimate circuit delays for the compression and decompression circuits, we constructed a gate-level design of their critical paths. We used the method of logical effort [113] to estimate FO4 gate delays for each circuit.

Compressor. We present the first stage of the compression pipeline in Figure 3-9. The input for this circuit is the 32-bits of an uncompressed word. The output is the three prefix bits that encode the word's pattern (as specified in Table 3-1). The critical path for this circuit passes through a 16-input 3-output AND gate, a

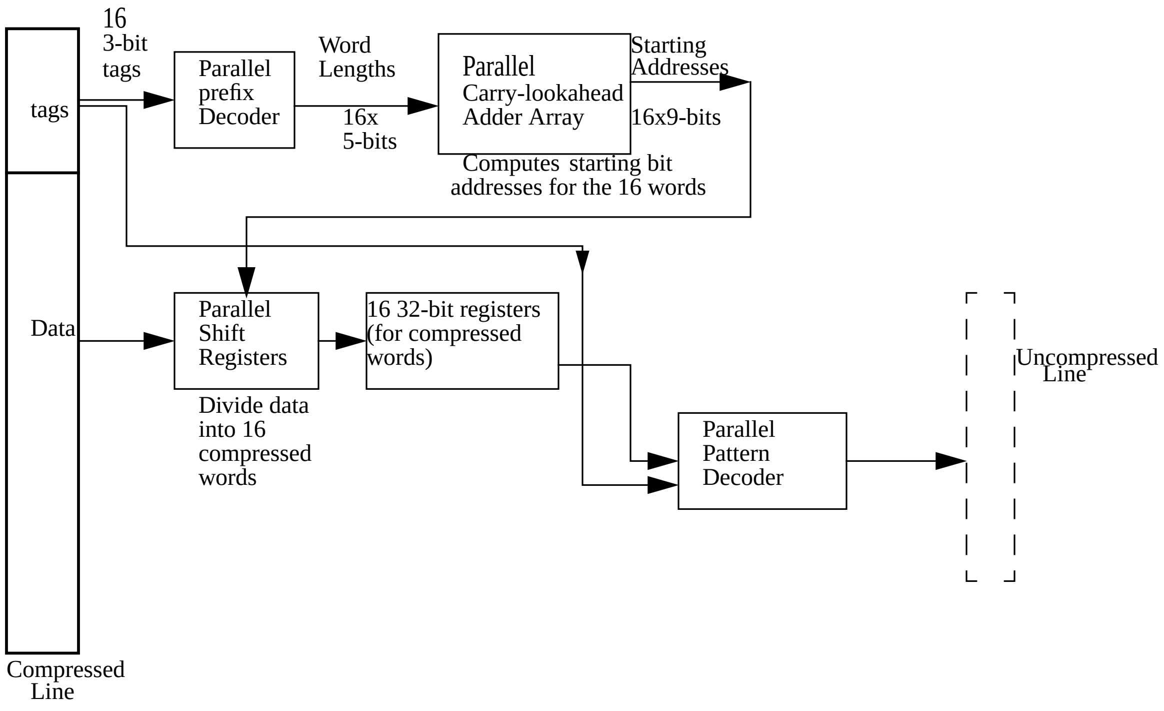


FIGURE 3-8. Cache line decompression pipeline for a 64-byte (16-word) cache line

This is a five-stage pipeline used to decompress a compressed cache line, where each stage contains 12 FO4 gate delays or less. The first pipeline stage (containing the parallel prefix decoder) decodes the prefix array to determine the length (in bits) of each word. The second and third stages (Parallel carry-lookahead adder array) compute the starting bit address for each data word by adding the length fields of the preceding words in a hierarchical fashion. The fourth stage (parallel shift registers) contains 16 registers each of which is shifted by the starting address of its word (in 4-bit increments). The fifth and last stage contains the pattern decoder, which decodes the content of each 32-bit register into an uncompressed word according to its corresponding prefix.

2-input 1-output AND GATE, a 2-input 2-output OR gate, a 1-input 1-output inverter, a 7-input 2-output AND gate and a 4-input 1-output OR gate, from input bits 16-31 through pattern #5 (two sign-extended halfwords) to the output of Prefix0 or Prefix2. Using the method of logical effort, delay in a logic gate is computed using:

$$d = g \, . \, h + p \tag{3.1}$$

where g is the gate’s logical effort, h is the gate’s electrical effort, and p is the gate’s parasitic delay [113]. We use the g, h and p assumptions for various gates from Sutherland, et al. ([113]):

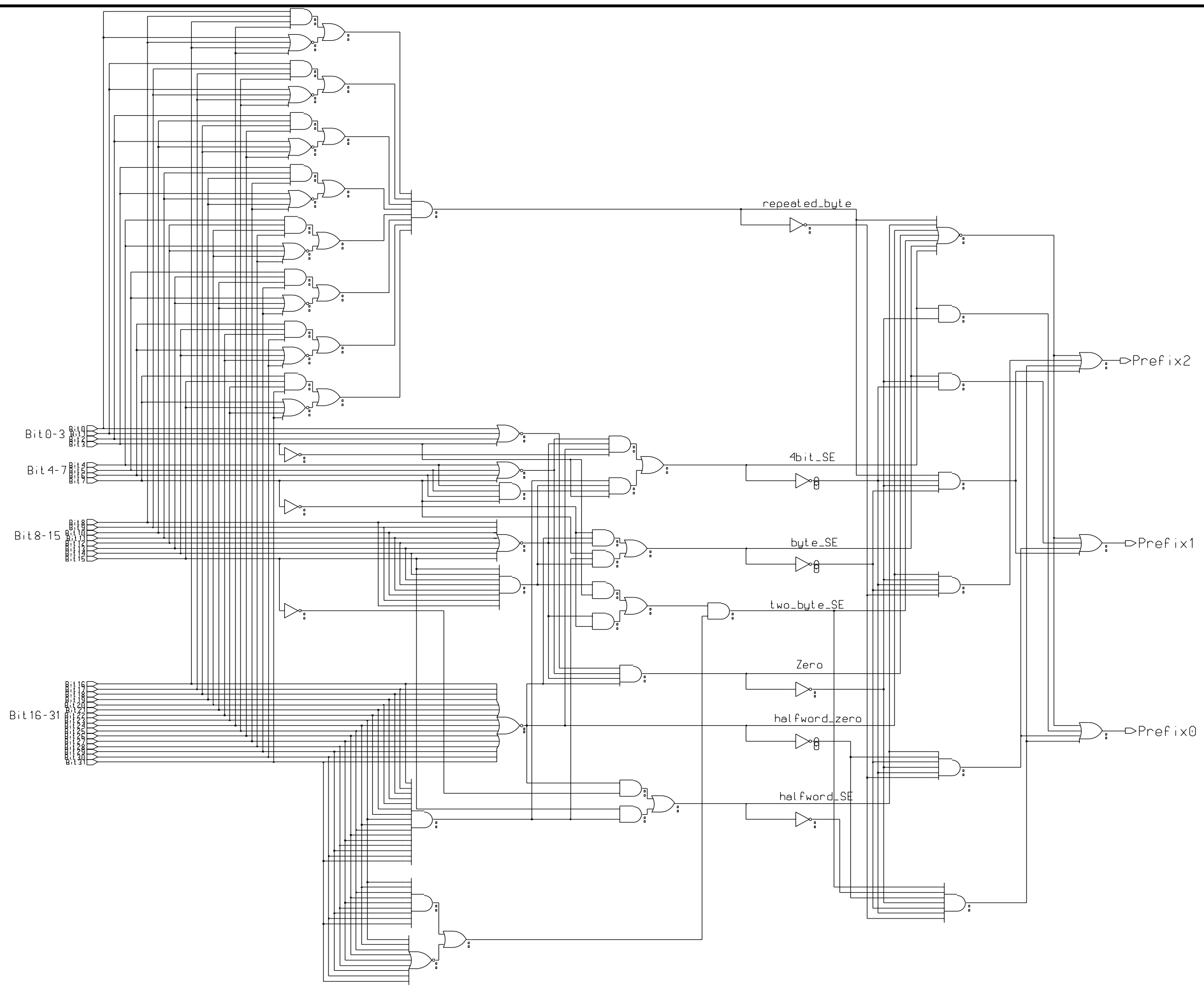


FIGURE 3-9. First pipeline stage of compression circuit

This circuit obtains the pattern prefix for a 32-bit word. For compression, parallel instances should be used to obtain the prefix of all words in a cache line in parallel

p(INV) = 0.6 delay units, g(INV) = 1 (3.2)

p (NAND(n,1)) = n . p(INV), g(NAND(n,1)) = (n+2)/3 (3.3)

p(NOR(n,1)) = n . p(INV), g(NOR(n,1)) = (2n + 1)/3 (3.4)

p(XOR(2,1)) = 4 . p(INV), g(XOR(2, 1)) = 4 (3.5)

$$p(\mathrm{MUX}(n,1)) = 2 \,.\, n \,.\, p(\mathrm{INV}),\ g(\mathrm{MUX}(n,1)) = 2 \qquad (3.6)$$

$$h = C_{out}/C_{in} \text{ (Output capacitance divided by input capacitance)} \qquad (3.7)$$

The total delay for the above compression circuit is computed from the following equation (obtained from the critical path Figure 3-9):

$$\text{Total\_delay} = d(\mathrm{AND}(16,3)) + d(\mathrm{AND}(2,1)) + d(\mathrm{OR}(2,2)) + d(\mathrm{INV}(1,1)) + d(\mathrm{AND}(7,2)) + d(\mathrm{OR}(4.1)) \quad (3.8)$$

Values for different gate delays can be directly computed from Equations 3.1-3.7. For example, the delay for AND(16,3) can be computed from:

$$d(\mathrm{AND}(16,3)) = d(\mathrm{NAND}(16, 1)) + d(\mathrm{INV}(1, 3)) = (16+2)/3 * 1/16 + 16*0.6 + 1 * 3/1 + 0.6 = 13.575 \quad (3.9)$$

We omit the details for the remaining gates. Summing up all gate delays, we get Total_delay = 35.254 delay units.

The fanout-of-four delay using the same assumptions is:

$$d(\mathrm{INV}(1,4)) = 1 * 4/1 + 0.6 = 4.6 \qquad (3.10)$$

So the above circuit has a 35.254/4.6 = 7.66 fanout-of-four delays (or less than 8 FO4 delays). This can be implemented in one clock cycle for most processes. We are assuming 12 FO4 delays per clock cycle, which is close to the estimated per-cycle delay for current processes [60]. The second pipeline stage, not presented here, is used to gather significant bits (0, 4, 8, 16 or 32 bits based on the prefix) from all words in the cache line into a single compressed word, using multiplexers and barrel shifters. This can be pipelined in two single-clock pipeline stages.

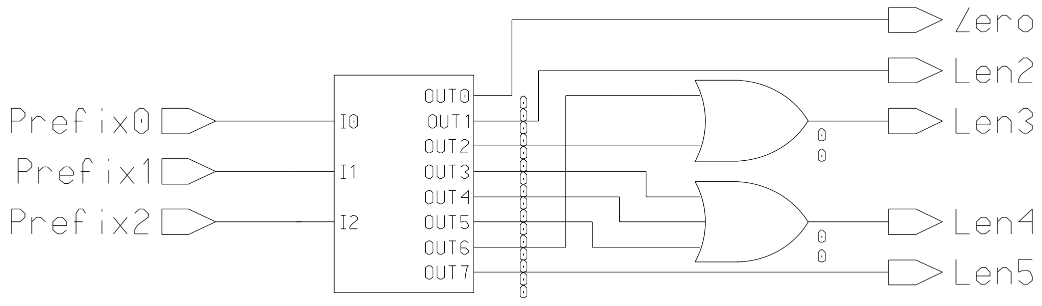


FIGURE 3-10. First stage of the decompression pipeline: Parallel Prefix Decoder

For each prefix in the line header, we use the decoder to figure out the length of the corresponding word.

Decompression. We present different stages of the decompression pipeline in Figure 3-10, Figure 3-11, Figure 3-12 and Figure 3-13.

Figure 3-10 illustrates part of the first stage in the decompression pipeline (the parallel prefix decoder), where we compute the compressed word length of each word in the cache line from its prefix. This is a fairly simple circuit with the critical path going through an inverter, a three input one output AND gate (both inside the 3-to-8 decoder), and a three input one output OR gate. Using the method of logical effort, the total delay for this circuit is 9.733 delay units, or less than 3 FO4 delays (i.e., less than one cycle of 12 FO4 delays).

In Figure 3-11, we show the critical path for second and third stages of the decompression pipeline, where the computed word lengths are used to compute the starting location for each word. This stage consists of a series of carry lookahead adders that compute the starting bit address of each word in the cache line. The critical path shown in the circuit is that required to compute the starting address of the last word (i.e., word #15) in the 16-word cache line. This critical path goes through the following gates: XOR(2,4), AND(4,1), OR(4,1), XOR(2,5), AND(5,1), OR(5,1), XOR(2,6), AND(6,1), OR(6,1), XOR(2,7), AND(7,1) and

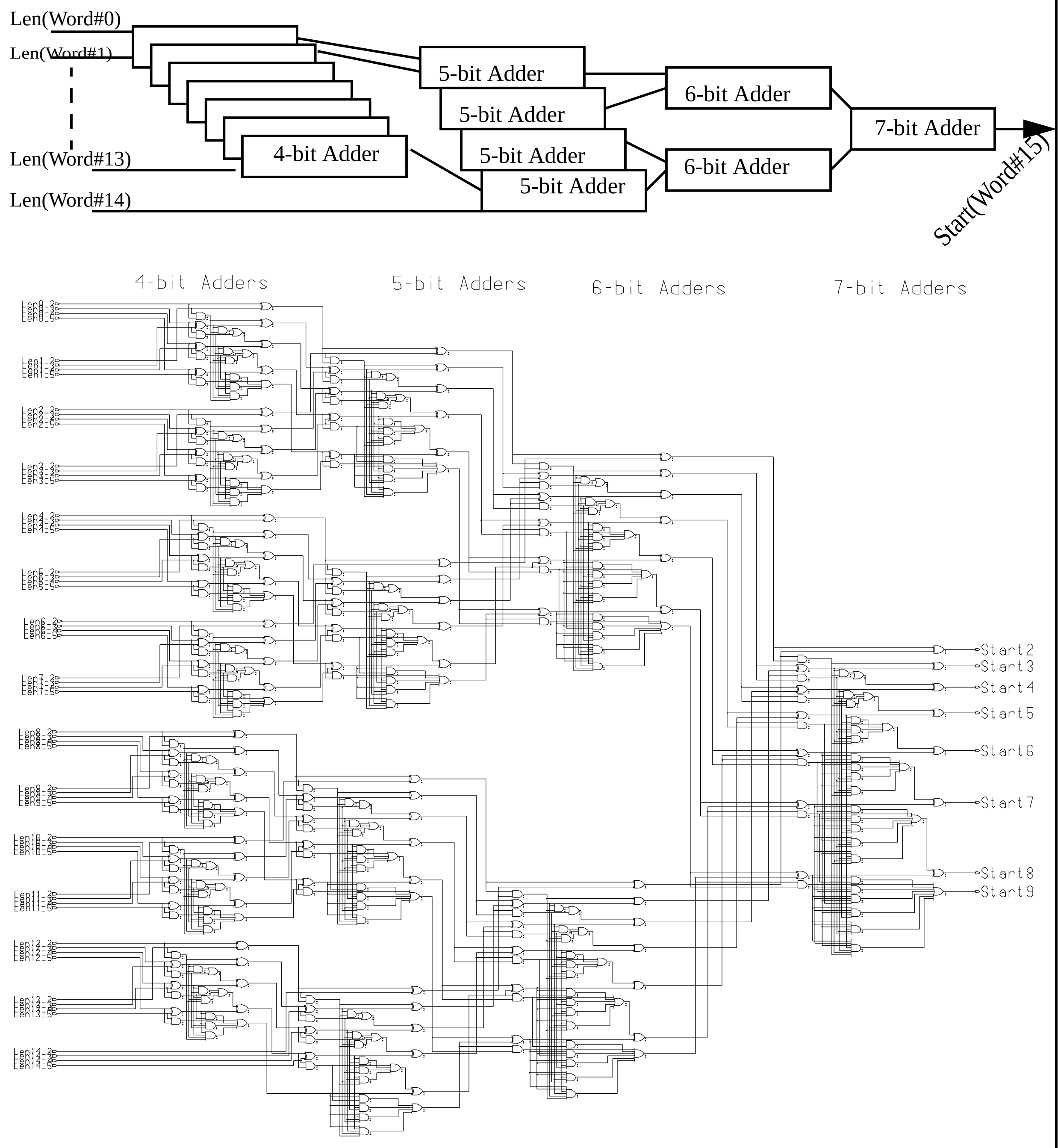


FIGURE 3-11. Critical path for second and third stages of the decompression pipeline. High-level design is shown at the top, and gate-level design at the bottom

This circuit uses the compressed word lengths (output of Figure 3-10) to compute the starting bit address of each word in the cache line. A multi-stage carry lookahead adder network computes the starting bit addresses of all words. The above circuit shows the path for computing the starting address of the last word (word 15).

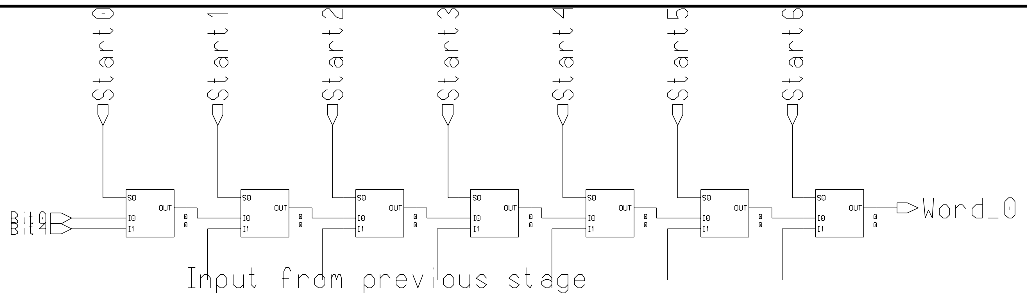


FIGURE 3-12. Fourth stage of the decompression pipeline (parallel shifter)

For each word starting bit location (obtained from the output of Figure 3-11), we shift the cache line right by a the corresponding number of four-bit nibbles using a barrel-shifter design. The above circuit is repeated for all line bits. The critical path for this stage is seven 2-to-1 multiplexers. Due to the large number of multiplexers in the parallel shifter circuit (512x7 array of 2-to-1 multiplexers), it is uneconomical to replicate this circuit for all cache words. A more appealing alternative is to have two copies of the above circuit, and time-multiplex the shift operations for eight words on each circuit.

OR(7,1). Using the method of logical effort, the total delay for this circuit is 97.56 delay units, or less than 22 FO4 delays. This is less than two cycles assuming a 12 FO4 delay per cycle.

In Figure 3-12, we demonstrate the fourth stage in the decompression pipeline, where the compressed words are extracted by shifting the cache line bits in 4-bit increments based on the starting locations computed at the end of the third stage. The critical path of this circuit is the series of seven 2-to-1 multiplexers. Again using the method of logical effort, the total delay is 40.6 delay units, or less than 9 FO4 delays.

In Figure 3-13, we present part of the fifth stage of the decompression pipeline used to compute the decompressed words given their compressed format and the word prefix. The critical path for this circuit is the 8-to-1 multiplexer, with a delay of 9.85 delay units (or less than 3 FO4 delays) according to the method of logical effort.

We could optimize this circuit further by combining the fourth and fifth pipeline stages into one, thus using four cycles for decompression instead of five. However, we chose not to optimize the circuit designs specifically for speed or power. Our objective was to illustrate that decompression can be done in five cycles or fewer, which is faster than other hardware schemes such as X-RL or BRCL. It is possible to optimize the

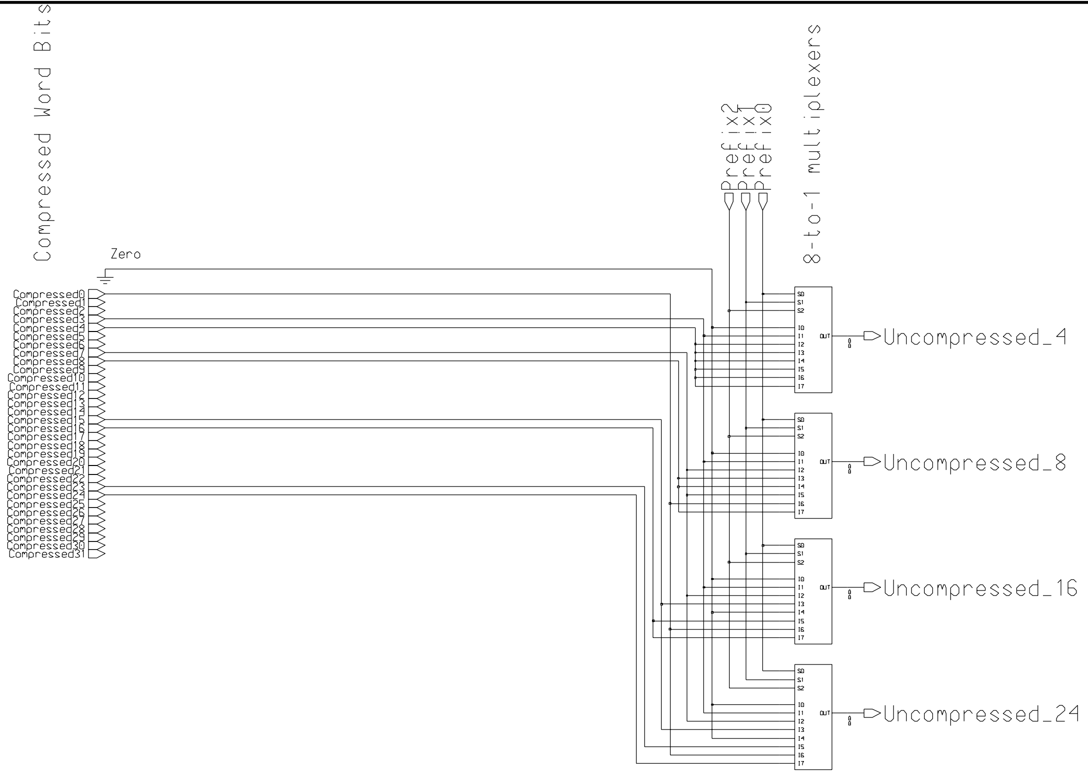


FIGURE 3-13. Part of the fifth stage of the decompression pipeline (Parallel Pattern Decoder)

We show the circuits used to compute bits 4, 8, 16 and 24 of the decompressed word.

decompression pipeline for power by using non-parallel resources with time multiplexing in pipeline stages with a lot of slack (e.g., the first and the last). This low decompression overhead is particularly important for use in our compressed cache design, as we show in the next section. We analyze the sensitivity of a compressed cache's performance to decompression latency in the next chapter.

3.4 Decoupled Variable-Segment Cache

In this section, we propose a two-level cache hierarchy consisting of uncompressed L1 instruction and data caches, and an optionally compressed L2 unified cache [9]. We evaluate the performance of our compressed cache design in the next chapter. While many of the mechanisms and policies we develop can be

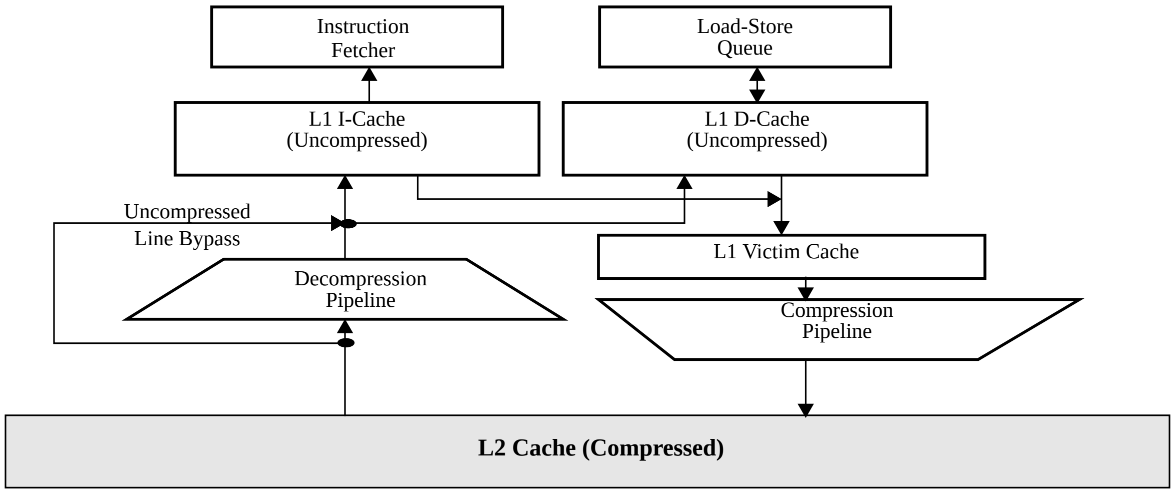


FIGURE 3-14. Compressed Cache Hierarchy

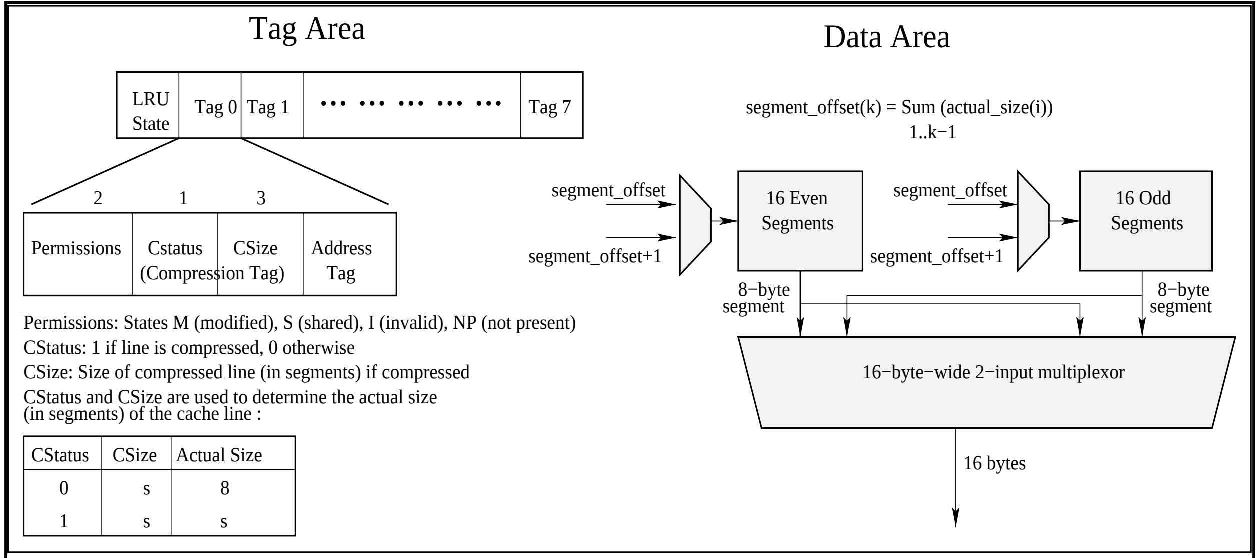
adapted to other cache configurations (e.g., three-level hierarchies), we only consider two-level hierarchies in this dissertation.

Figure 3-14 illustrates the proposed cache hierarchy. L1 instruction and data caches store uncompressed lines, eliminating the decompression overhead from the critical L1 hit path. This design also completely isolates the processor core from the compression hardware. The L1 data cache uses a writeback, write allocate policy to simplify the L2 compression logic. On L1 misses, the controller checks an uncompressed victim cache in parallel with the L2 access. In addition to its normal function, the victim cache acts as a rate-matching buffer between the L1s and the compression pipeline [79]. On an L2 hit, the L2 line is decompressed if stored in compressed form. Otherwise, it bypasses the decompression pipeline. On an L2 miss, the requested line is fetched from main memory. We assume compressed memory, though this is largely an orthogonal decision. The L1 and L2 caches maintain inclusion. Lines are allocated in the L2 cache on L1 replacements and writebacks, L1 misses (that also miss in the L2), and L1 or L2 prefetches. For design simplicity, we assume a single line size of 64-bytes for all caches.

To exploit compression, the L2 cache must be able to pack more compressed cache lines than uncompressed lines into the same space. One approach is to decouple the cache access, adding a level of indirection between the address tag and the data storage. Seznec's decoupled sector cache does this on a per-set basis to improve the utilization of sector (or sub-block) caches [105]. Hallnor and Reinhardt's Indirect-Index Cache (IIC) decouples accesses across the whole cache, allowing fully-associative placement, a software managed replacement policy, and (recently) compressed lines [54, 55]. The recently proposed V-Way cache [98] adopts a similar decoupled approach. Lee, et al.'s selective compressed caches use this technique to allow two compressed cache blocks to occupy the space required for one uncompressed block [77, 78, 79]. Decoupled access is simpler if we serially access the cache tags before the data. Fortunately, this is increasingly necessary to limit power dissipation [71].

In our design, we use a similar approach to previous proposals by decoupling the cache access, adding a level of indirection between the address tag and the data storage. We also use more tags than (uncompressed) cache lines to support storing compressed lines. Pomerene, et al. [97], used a similar scheme in a shadow directory with more address tags than data blocks to improve upon LRU replacement.

We show our compressed cache design in Figure 3-15. Each set is 8-way set-associative, with a compression information tag stored with each address tag. The data array is broken into eight-byte segments, with 32 segments statically allocated to each cache set. Thus, each set can hold no more than four uncompressed 64-byte lines, and compression can at most double the effective capacity. Each line is compressed into 1-8 eight-byte segments, with eight segments being the uncompressed form. The compression tag indicates i) the compressed size of the line (CSize) and ii) whether or not the line is stored in compressed form (CStatus). A separate cache state indicates the line's coherence state, which can be any of M (modified), O (Owned), E (Exclusive), S (shared), I (invalid). We maintain the compression tag even for invalid lines since we use these tags in our adaptive compression policy (discussed in the next chapter).


Tag Area
Data Area
LRU State
Tag 0
Tag 1
Tag 7
2
1
3
Permissions
Cstatus (Compression Tag)
CSize
Address Tag
Permissions: States M (modified), S (shared), I (invalid), NP (not present)
CStatus: 1 if line is compressed, 0 otherwise
CSize: Size of compressed line (in segments) if compressed
CStatus and CSize are used to determine the actual size (in segments) of the cache line :
segment_offset(k) = Sum (actual_size(i)) 1..k−1
segment_offset
segment_offset+1
16 Even Segments
16 Odd Segments
8−byte segment
16−byte−wide 2−input multiplexor
16 bytes

CStatus	CSize	Actual Size
0	s	8
1	s	s

FIGURE 3-15. A single set of the decoupled variable-segment cache

Data segments are stored contiguously in address tag order. That is, the offset for the first data segment of line k is:

$$\text{segment\_offset}(k) = \sum_{i=1}^{k-1} \text{actual\_size}(i) \tag{3.11}$$

A line's actual size is determined by the compression tag (Figure 3-15) and the eight segment offsets are computed in parallel with the address tag match using 5-bit parallel adders (similar to the circuit in Figure 3-11, though much smaller). On an address tag match, the segment offset and actual length are used to access the corresponding segments in the data array. The array is split into banks for even and odd segments, allowing two segments (16 bytes) to be fetched per cycle regardless of the alignment [52].

An L1 replacement writes back dirty lines to the L2 cache, where it finds a matching address tag since we maintain inclusion. An L2 fill can also replace a line in the L2 cache. If the new line's compressed size is the same as the replaced line (or smaller), this writeback or fill is trivial. However, if the new size is larger, the cache controller has to allocate space in the set. This may entail replacing one or more L2 lines or compacting invalid lines to make space. More than one line may have to be replaced if the newly allocated line is larger than the LRU line plus the unused segments. In this case, we replace at most two lines by replacing the LRU line and searching the LRU list to find the least-recently-used line that ensures we have enough space.

Compacting a set requires moving tags and data segments to maintain the contiguous storage invariant. This operation can be quite expensive, because it may require reading and writing all the set's data segments. For this reason, compaction is deferred as long as possible and is never needed on a read (e.g., L1 fill) access. In the next chapter, we evaluate the impact of compaction on the number of bits read/written for our compressed cache design. With a large L1 victim cache and sufficient L2 cache banks, compaction can have a negligible impact on performance.

A decoupled variable-segment cache adds relatively little storage overhead. For example, consider a 4-way, 4 MB uncompressed cache with 64-byte lines. Each set has 2048 data bits, in addition to four tags. Each tag includes a 24-bit address tag, a 2-bit LRU state, and a 3-bit permission, for a total of 4*(24+2+3)=116 bits per set. Our scheme adds four extra tags, increases the LRU state to three bits and adds a 4-bit compression tag per line. This adds 116+8*1+8*4=156 bits per set, which increases the total cache storage by approximately 7%. For an 8-way 4 MB cache, the overhead per set is 312 bits, also approximately 7%.

Chapter 4

Adaptive Cache Compression

Cache compression increases the effective cache size at the expense of increasing cache hit latency for compressed lines. On the one hand, compression can potentially help some applications by eliminating many off-chip misses. On the other hand, applications that fit in an uncompressed cache can be hurt by decompression overhead.

In this chapter, we develop an adaptive policy that dynamically balances the benefits of cache compression against its overheads. We use the cache replacement algorithm's stack depth [89] and compression information to determine whether compression (could have) eliminated a miss or incurs an unnecessary decompression overhead (Section 4.1). Based on this determination, we develop an adaptive policy that updates a single global saturating counter. This counter predicts whether to allocate future cache lines in compressed or uncompressed form (Section 4.2).

We evaluate our adaptive cache compression policy using full-system simulation of a uniprocessor system and a range of benchmarks (Section 4.3). We show that compression can improve performance for some memory-intensive workloads by 2-34%. However, always using compression hurts performance for low-miss-rate benchmarks—due to unnecessary decompression overhead—degrading performance by 4-16%. By dynamically monitoring workload behavior, the adaptive policy achieves comparable benefits from compression, while avoiding most of the performance degradation for benchmarks that are hurt by compression (Section 4.4). We analyze the sensitivity of compressed cache performance to different memory system parameters (Section 4.5). We conclude by discussing limitations of our adaptive compression scheme (Section 4.6) and presenting relevant related work (Section 4.7).

This chapter makes the following contributions:

- We show that cache compression in a uniprocessor system can help the performance of some benchmarks while hurting the performance of others.
- We propose a scheme that uses the stack of a cache replacement algorithm [89] to identify whether compression helps or hurts each individual cache reference.
- We propose an adaptive prediction scheme that dynamically adapts to a benchmark's behavior or system configuration. This adaptive scheme compresses cache lines only when compression helps. We show that our adaptive compression scheme performs similar to the best of the two extremes: *Always Compress* and *Never Compress*.

4.1 Cost/Benefit Analysis

In this section, we analyze the costs and benefits associated with cache compression. We present a simple analytical model for compression's costs and benefits, and discuss how we classify different cache accesses according to whether compression helped, hurt or did not affect each access.

4.1.1 Simple Model

While compression helps eliminate long-latency L2 misses, it increases the latency of the (usually more frequent) L2 hits. Thus, some benchmarks (or benchmark phases) will benefit from compression, but others will suffer. For a simple, in-order blocking processor, L2 cache compression will help if the benefit of compression due to avoiding some L2 misses is greater than the cost due to increasing L2 hit latency for compressed lines:

$$\text{Avoided L2 Misses} \times (\text{L2 Miss Penalty} - \text{L2 Hit Latency}) > \text{Penalized L2 Hits} \times \text{Decompression Penalty} \tag{4.1}$$

Where penalized L2 hits are those that unnecessarily incur the decompression penalty. Rearranging terms yields:

$$\frac{\text{Penalized L2 Hits}}{\text{Avoided L2 Misses}} < \frac{(\text{L2 Miss Penalty} - \text{L2 Hit Latency})}{\text{Decompression Penalty}} \tag{4.2}$$

For a 5-cycle decompression penalty and 400-cycle L2 miss penalty, compression wins if it eliminates at least one L2 miss for every 400/5=80 penalized L2 hits (or a ratio of less than 80 penalized hits per avoided miss). While this may be easily achieved for memory-intensive commercial workloads, smaller workloads—whose working set size fits in an uncompressed L2 cache—may suffer performance degradation. We note, however, that this model might not be accurate for more complex processors that use various latency hiding techniques (e.g., out-of-order execution and prefetching).

Ideally, a compression scheme should compress data when the benefit (i.e., avoided misses) outweighs the cost (i.e., penalized L2 hits). We next describe how we classify cache accesses according to the cost or benefit of compression, and use that information in the next section to update a compression predictor.

4.1.2 LRU Stack and the Classification of Cache Accesses

The key insight underlying our adaptive compression policy is that the LRU stack depth [89] and compressed line sizes determine whether compression helps or hurts a given reference. As an example, Figure 4-1 illustrates the LRU stack of a single cache set, where a stack depth of 1 indicates the most recently used line. In our decoupled variable-segment cache design, only the top half of the stack (i.e., the most recently used four lines) would be in the cache without compression. Lines in the bottom half only exist in the cache because of compression. Based on the cache lines in Figure 4-1, we next classify cache references (hits and misses) according to whether compression helps or hurts these references.

Classification of hits:

- A reference to Address A hits at stack depth 1. Because the set can hold four uncompressed lines and the LRU stack depth is less than or equal to four, compression provides no benefit. Conversely, since the data is stored uncompressed, the reference incurs no decompression penalty as we bypass the decompression pipeline for uncompressed lines. We call this case an *unpenalized hit*.
- A reference to Address C hits at stack depth 3. Compression does not help, since the line would be present even if all lines were uncompressed. Unfortunately, since the block is stored in compressed form, the reference incurs an unnecessary decompression penalty. We call this case a *penalized hit*.
- A reference to Address E hits at stack depth 5. As only the top four lines would have been in the cache without compression, this reference is a hit only because of compression. In this case, compression has eliminated a miss that would otherwise have occurred. We call this case an *avoided miss*.

Stack Depth	Address Tag	CStatus	CSize (Segments)	Permissions
1	A	Uncompressed	2	Modified
2	B	Uncompressed	8	Modified
3	C	Compressed	4	Modified
4	D	Compressed	3	Modified
5	E	Compressed	2	Modified
6	F	Compressed	7	Modified
7	G	Uncompressed	5	Invalid
8	H	Uncompressed	6	Invalid

FIGURE 4-1. A cache set example

Address tags are shown in LRU order (Address A is the most recent). The first six tags corresponds to lines in the cache, while the last two correspond to evicted lines (Permissions = Invalid). Addresses C, D, E and F are stored in compressed form.

Classification of misses:

- A reference to Address G misses in the cache, but matches the address tag at LRU stack depth 7. The sum of the compressed line sizes at stack depths 1 through 7 totals 29. Because this is less than 32 (the total number of data segments per set), this reference misses only because one or more lines at stack depths less than 7 are stored uncompressed (i.e., Address A could have been stored in two segments). Since compression could have helped avoid a miss, we call this case an *avoidable miss*.
- A reference to Address H misses in the cache, but matches the address tag at LRU stack depth 8. However, this miss cannot be avoided because the sum of compressed sizes for stack depths 1-8 exceeds the total number of segments available (i.e., 35 > 32). Similarly, a reference to Address I does not match any tag in the cache set, so its LRU stack depth is greater than 8. We call both of these cases an *unavoidable miss*.

While we assume LRU replacement in this dissertation, any stack algorithm—including random [89]—will suffice. Moreover, the stack property only needs to hold for lines that either do or might have fit due to compression (e.g., LRU stack depths 5–8 in the example of Figure 4-1). We can use any arbitrary replacement policy for the top four elements in the “stack.”

In our adaptive compression scheme, the cache controller uses the LRU state and compression tags to classify each L2 reference. The avoidable miss calculation can be implemented using a five-bit parallel adder with 8:1 multiplexors on the inputs to select compressed sizes in LRU order. To save hardware, a single carry-lookahead adder can be time-multiplexed, since gathering compression information is not on the critical path, and the data array access takes longer than the tag access. We use the above classification of hits and misses to monitor the actual effectiveness of cache compression. We next describe our adaptive predictor that uses this information to dynamically determine whether to store a line in a compressed or uncompressed form.

4.2 Compression Predictor

Like many predictors, the adaptive compression policy uses past behavior to predict the future. Specifically, the L2 cache controller uses the classification in the previous section to update a global saturating counter—called the Global Compression Predictor (GCP)—to aggregate the recent history of compression benefit minus cost. On a penalized hit (a compression cost), the controller biases against compression by subtracting the decompression penalty. On an avoided or avoidable miss (a compression benefit or potential benefit), the controller increments the counter by the (unloaded) L2 miss penalty. To reduce the counter size, we normalize these values to the decompression latency, subtracting one and adding the miss penalty divided by decompression latency (e.g., 400 cycles / 5 cycles = 80).

The L2 controller uses the GCP when allocating a line in the L2 cache. Positive values mean compression has been helping eliminate misses, so the L2 controller stores the newly allocated line in compressed form. Negative values mean compression has been penalizing hits, so the controller store the line uncompressed. All allocated lines—even those stored uncompressed—must run through the compression pipeline to calculate their compressed size, which is used in the avoidable misses calculation.

The size of the saturating counter determines how quickly the predictor adapts to workload phase changes. The results in this dissertation use a single global 19-bit counter that saturates at 262,143 or -262,144 (approximately 3300 avoided or avoidable misses). Using a large counter means the predictor adapts slowly to phase changes, preventing short bursts from degrading long-run behavior. On the other hand, a small counter can quickly identify phase changes. Section 4.5.8 examines the impact of workload phase behavior on the predictability of cache compression. We next evaluate the performance of this global adaptive compression policy.

4.3 Evaluation

We present an evaluation of adaptive compression on a dynamically-scheduled out-of-order uniprocessor system. We use full-system simulation of commercial workloads and a subset of the SPECcpu2000 benchmarks. The workloads we use in this chapter are the same as those of Section 3.2.1. We ran warmed up the benchmarks as specified in the previous chapter. We then ran apache, zeus, oltp, and jbb for 3000, 3000, 300 and 20000 transactions, respectively. We also ran SPEC2000 benchmarks for one billion instructions after the warm interval. We summarize our simulated system parameters next.

4.3.1 System Configuration

We evaluated the performance of our compressed cache designs on a dynamically-scheduled SPARC V9 uniprocessor using the Simics full-system simulator [88], extended with the Multifacet General Execution-driven Multiprocessor Simulator [130]. Our target system is a superscalar processor with out-of-order execution. Table 4-1 presents some of our basic simulation parameters.

4.3.2 Three Compression Alternatives

To evaluate the effectiveness of adaptive compression, we compare it with two extreme policies: *Never* and *Always*. *Never* models a standard 8-way set associative L2 cache design, where data is never stored compressed. *Always* models a decoupled variable-segment cache (Section 3.4) that always stores compressible data in compressed form. Thus *Never* strives to reduce hit latency, while *Always* strives to reduce miss rate. *Adaptive* uses the policy described in Section 4.2 to utilize compression only when it predicts that its benefits outweigh decompression overheads.

TABLE 4-1. Uniprocessor Simulation Parameters

L1 Cache Configuration	Split I & D, each 64 KB 4-way set associative with LRU replacement, 64-byte line, 3-cycle access time
L2 Cache Configuration	Unified 4 MB (unless otherwise specified), 8-way set associative with LRU replacement for both compressed and uncompressed caches (compressed caches have double the number of sets), 64-byte lines,
L2 Cache Hit Latency	Uncompressed caches have a 10-cycle bank access latency plus a 5-cycle wiring delay. Compressed caches add a 5-cycle decompression overhead for compressed lines.
Memory Configuration	4 GB of DRAM, 400 cycles access time with a 50 GB/sec. memory bandwidth, 16 outstanding memory requests (including prefetches).
Processor Configuration	4-wide superscalar, 11-stage pipeline—Pipeline stages: fetch (3), decode (4), schedule (1), execute (1 or more), retire (2).
IW/ROB	64-entry instruction window, 128-entry reorder buffer.
Branch Predictors	4 KB YAGS direct branch predictor [38], a 256-entry cascaded indirect branch predictor [35], and a 64-entry return address stack predictor [67].

4.4 Compression Performance

In this section, we present performance results for the three compression alternatives: *Never*, *Always* and *Adapt*. For each data point in our results, we present the average and the 95% confidence interval of multiple simulations to account for space variability [8]. Our runtime results for commercial workloads represent the average number of cycles per transaction (or request), whereas runtime results for SPEC benchmarks represent the average number of cycles per instruction (CPI). We evaluate cache miss rates, performance and the effect of compression on the number of bits read and written (as an indirect measure of dynamic power).

4.4.1 Cache Miss Rate

Using compression to increase effective cache capacity should decrease the L2 miss rate. Figure 4-2 presents the average miss rates for our set of benchmarks. The results are normalized to *Never* to focus on the benefit of compression, but the absolute misses per 1000 instructions for *Never* are included at the bottom. Both *Always* and *Adaptive* have lower or equal miss rates when compared to *Never* with one exception, ammp. In the next section, we show that ammp's *Adaptive* policy predicts that no compression compares favorably to compression. When *Adaptive* predicts no compression, the effective L2 cache configuration is a 4 MB, 4-way set associative cache, as compared to a 4 MB, 8-way set associative cache for *Never*. This difference in associativity accounts for the small increase in the miss rate for *Adaptive* when compared to *Never* in ammp (about 0.2%).

Not surprisingly, the commercial benchmarks achieve substantial benefits from compression, reducing the miss rates by 4–18%. Some other benchmarks achieve significant reductions in miss rate (e.g., mcf's miss rate decreases by 15%). Benchmarks with small working sets (e.g., twolf) get little or no miss rate reduction from compression. The four floating-point benchmarks, despite very large working sets, do not benefit

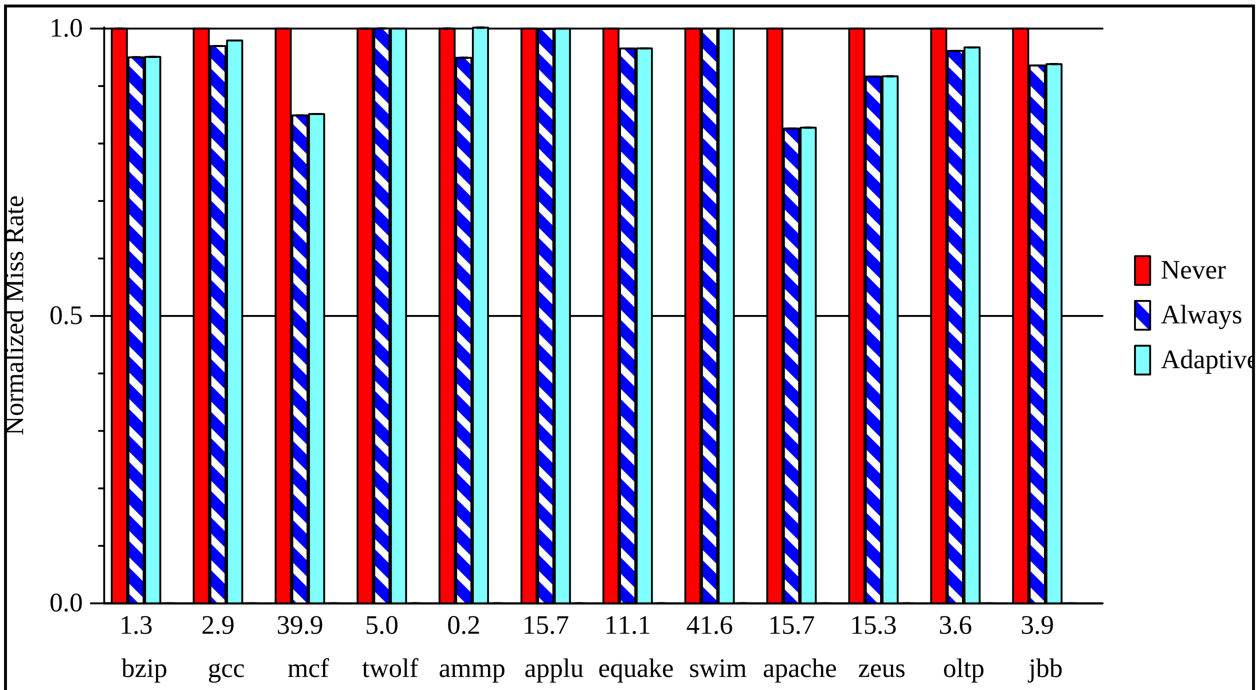


FIGURE 4-2. L2 cache miss rates (misses per thousand instructions) for the three compression alternatives. Miss rates are normalized to the *Never* miss rate (shown at the bottom)

from compression (except for ~4% for equake) due to the poor compression ratios that our compression algorithm achieves for floating-point data.

4.4.2 Performance

The ultimate objective of adaptive cache compression is to achieve performance that is comparable to the best of *Always* and *Never*. Reducing the cache miss rate, as *Always* does for some benchmarks, may be outweighed by the increase in hit latency. Figure 4-3 presents the simulated runtime of our twelve benchmarks, normalized to the *Never* case. Most of the benchmarks that have substantial miss rate reductions under *Always* also improve runtime performance (e.g., a speedup of 18% for apache, 7% for zeus, and 34% for mcf[1]). However, the magnitude of this improvement depends upon the absolute frequency of misses. For example, jbb and zeus have similar relative miss rate improvements, but since zeus has more than four times as many misses per instruction, its performance improvement is greater. On the other hand, bench-

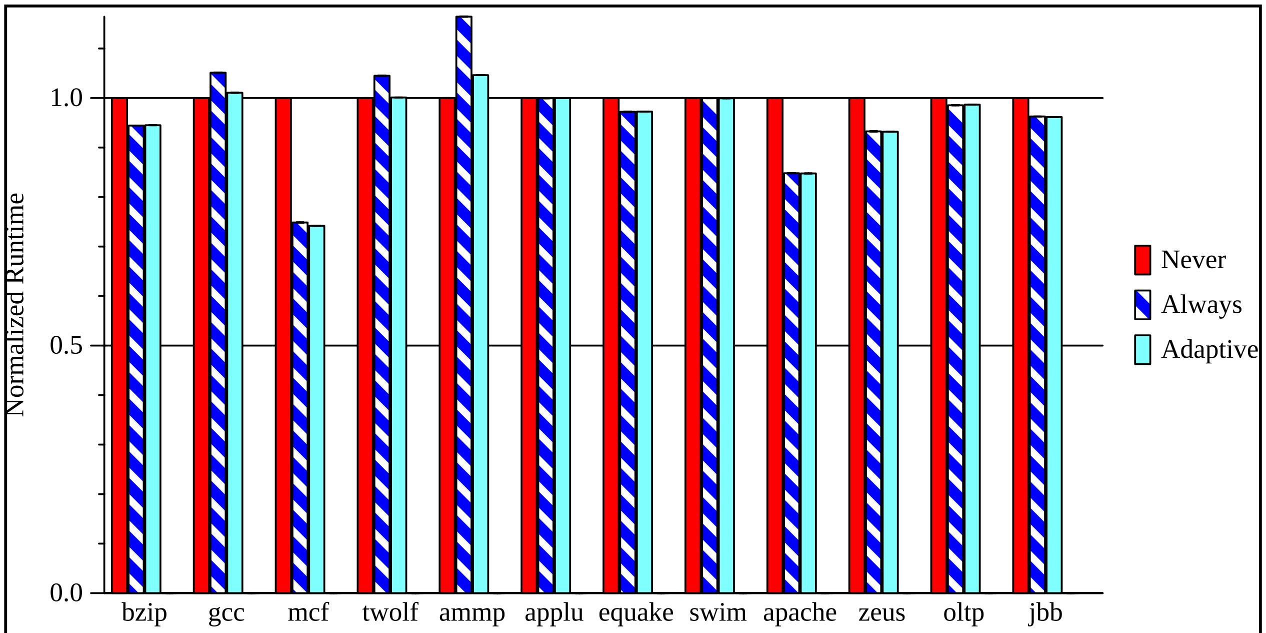


FIGURE 4-3. Runtime for the three compression alternatives, normalized to the *Never* runtime

marks with smaller working sets (e.g., gcc, twolf, ammp) do not benefit from bigger cache capacity. For example, *Always* degrades performance compared to *Never* by 16% for ammp, 5% for gcc and 4% for twolf.

Figure 4-3 also shows that *Adaptive* achieves the benefit of *Always* for benchmarks that benefit from compression. In addition, for benchmarks that do not benefit from compression, it degrades performance by less than 4% compared to *Never.* The 4% is in the case of ammp due to the lower associativity for *Adaptive* compared to *Never.*

In summary, while some memory-intensive benchmarks benefit significantly from compression, other benchmarks receive little benefit or even degrade significantly. For our benchmarks, *Adaptive* achieves the best of both worlds, improving performance by using compression when it helps, while not hurting performance (except marginally) when compression does not help. We next discuss the impact of *Always* and *Adaptive* on the number of bits read and written from/to the L2 cache.

4.4.3 Bit Activity level

Our adaptive cache compression scheme specifically targets being close in performance to the best of *Always* or *Never.* However, it does not specifically adapt to other system aspects such as power or cache bandwidth. While doing a complete study of the impact of cache compression on power is beyond the scope of this dissertation, we present a study of bits read and written from/to the L2 cache as an indirect measure of dynamic power. This study doesn't take into account the power savings due to avoiding cache misses.

1. We note that some benchmarks (e.g., mcf in this section, and others in later sections) have super-linear speedups that are higher than expected from the reduction in miss rate. This is attributed to compression decreasing the miss rates to a degree where the number of outstanding misses (i.e., number of MSHRs) is not a bottleneck. A miss rate of ~40 per 1000 instructions for mcf mean a near 100% utilization of MSHRs, since each miss penalty is 400 cycles. Compression helps reduce miss rates to a point where utilization is significantly lower.

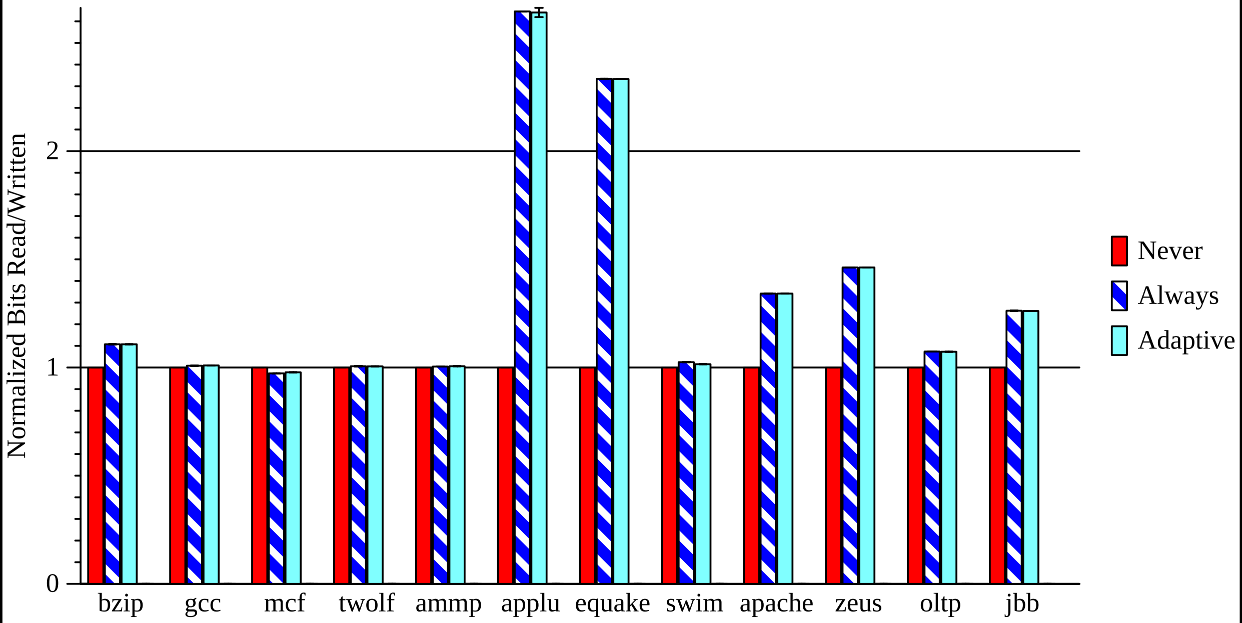


FIGURE 4-4. Bits read and written from/to the L2 cache for the three compression alternatives, normalized to the *Never* case

Figure 4-4 shows the number of bits read and written from/to the L2 cache for the three compression alternatives, normalized to *Never.* Compression can increase the number of bits read and written since allocations may require repacking, i.e., reading all lines in a cache set and writing an aligned, compact version of it back. This increases the number of bits read and written compared to *Never* for many benchmarks, by up to 164% for applu and 133% for equake. Other than these two benchmarks, only apache, zeus and specjbb show increases of more than 10% in bits read/written (46% for zeus).

We further studied the percentage of cache allocates in which repacking is needed. Figure 4-5 shows such percentage for *Always*. The percentage of repacks required is at its highest for twolf and ammp, but their impact on power (in terms of number of bits read/written) is minimal since the absolute number of cache allocations is small. The two benchmarks with the most increase in bits read/written are applu and equake, whose repack percentages are 37% and 36%, respectively. However, since the absolute number of allocations is large, the impact of such percentage on bits read/written is significant.

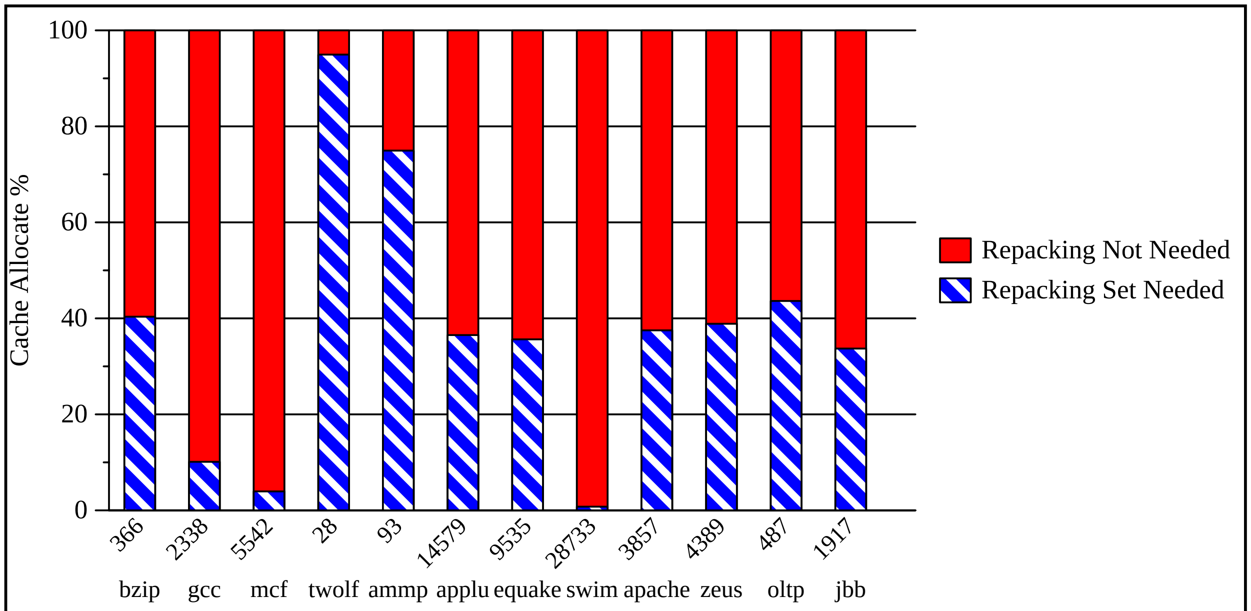


FIGURE 4-5. Percentage of cache allocations that require repacking for *Always*

The absolute number of allocations (in thousands) is shown at the bottom

Our adaptive compression policy did not take the metric of cache set repacking into account when updating the predictor. However, the policy can be slightly modified to bias against compression when a repacking event occurs. In this case, the predictor can be decremented by the number of cycles required to read, modify and write a whole cache set (normalized to decompression penalty). While we did not implement such policy in this dissertation, we anticipate it will cause applu and equake to adapt to *Never*, avoiding such high bit activity level. Furthermore, this policy will not affect benchmarks that benefit from compression since the benefit clearly outweighs the cost (even when adding the cost of repacking) in all such benchmarks.

4.5 Sensitivity Analysis

The effectiveness of cache compression depends upon the interaction between a workload's working-set size and the caches' sizes and latencies. Adaptive cache compression is designed to dynamically adjust its compression decisions to approach the performance of the better of the two static policies *Always* and

Never. In this section, we investigate how well *Adaptive* adjusts to changes in L1 and L2 cache sizes and associativities, memory latency, decompression latency, prefetching, cache line size and benchmark phases. We vary a single parameter in each of the following subsections while keeping the remaining parameters constant. We focus on three benchmarks where compression helps (mcf, apache, zeus) and three where compression hurts (ammp, gcc, twolf).

4.5.1 L1 Cache Size and Associativity

The effectiveness of L2 cache compression depends on the overhead incurred decompressing lines on L2 hits. Since the L1 cache filters requests to the L2, the L1 cache size impacts this overhead. As the L1 size (or associativity) increases, references that would have hit in the L2 can be satisfied in the L1. Thus the decompression overhead tends to decrease. Conversely, as the L1 size (or associativity) decreases, the L2 incurs more penalized hits due to the increased number of L1 misses. Figure 4-6 illustrates this trade-off for a 4 MB L2 cache, assuming a fixed L1 access latency.

For our set of benchmarks and configuration parameters, increasing L1 size has very little impact on the relative benefit of compression. Only mcf’s performance is slightly degraded (by approximately 1%) for *Always* with the smaller L1 cache compared to the original 128K L1 cache. The figure also shows that L1 associativity has little impact on the performance of compressed caches.

4.5.2 L2 Cache Size

Cache compression works best when it can increase the effective L2 cache size enough to hold a workload’s critical working set. Conversely, compression provides little or no benefit when the working set is either much larger or much smaller than the L2 cache size. Figure 4-7 illustrates this phenomenon by presenting normalized runtimes for various L2 cache sizes, assuming a fixed L2 access latency.

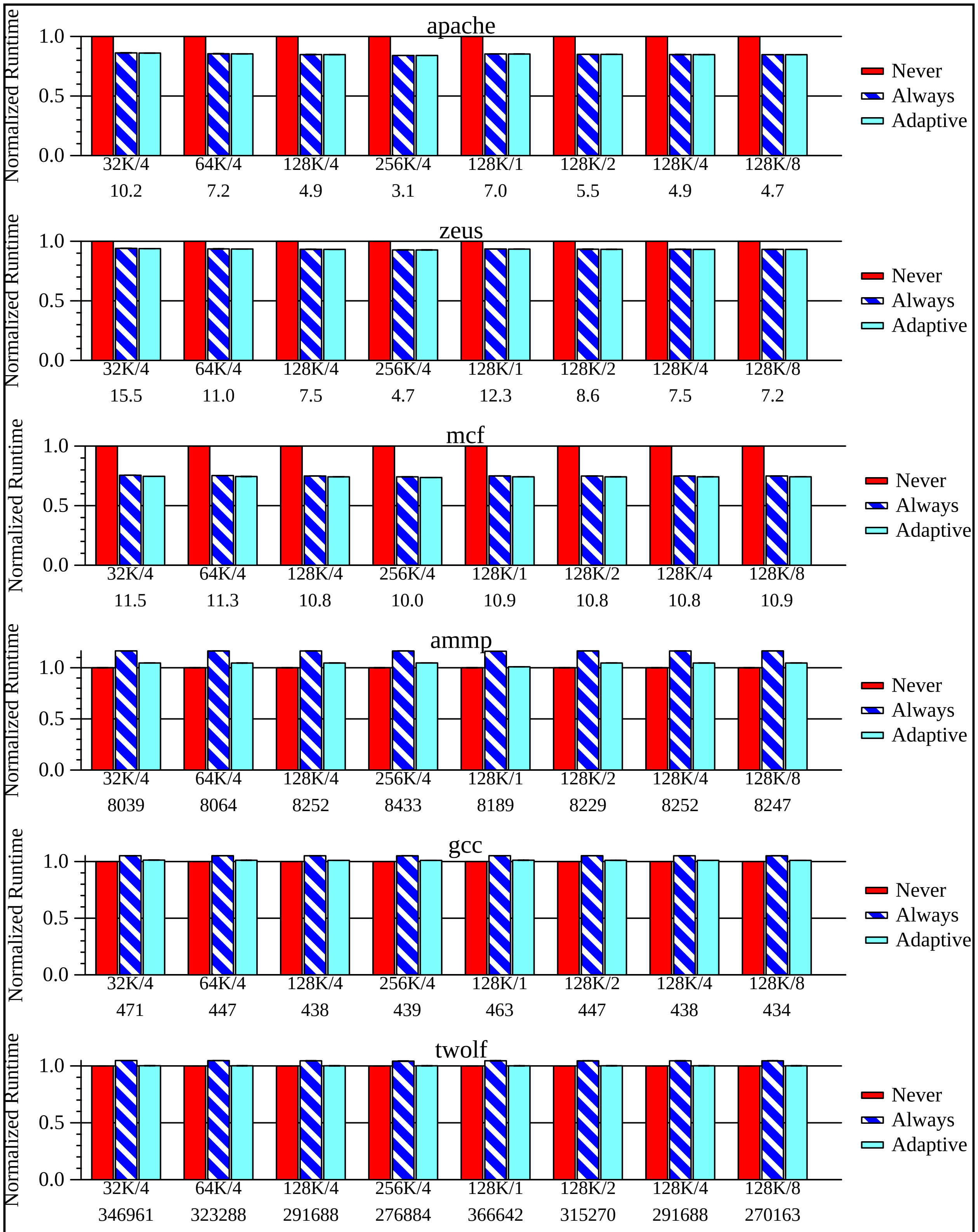


FIGURE 4-6. Sensitivity to L1 cache size and associativity of the three compression alternatives. The number of penalized hits per avoided miss for *Always* is shown at the bottom

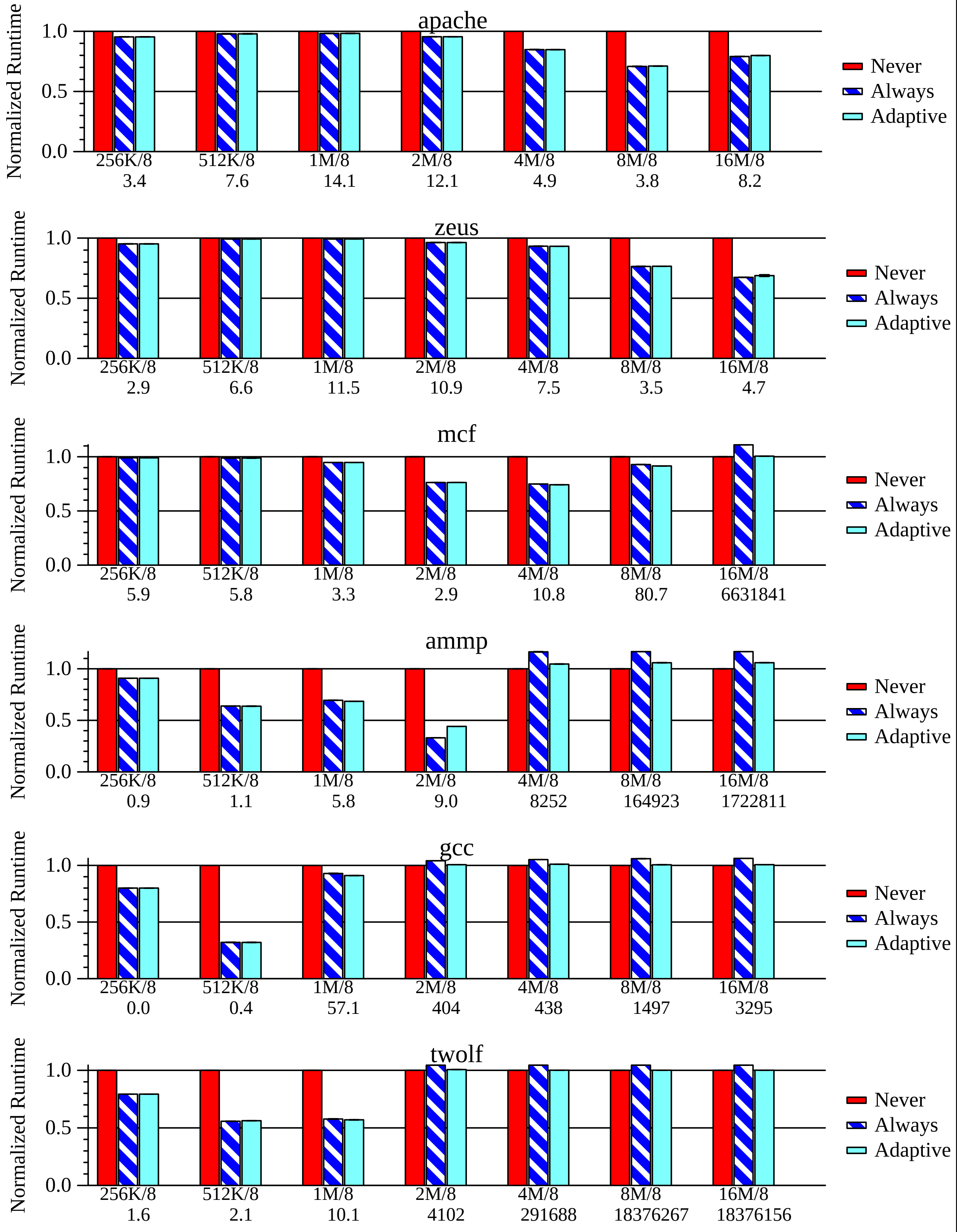


FIGURE 4-7. Sensitivity to L2 cache size of the three compression alternatives. The number of penalized hits per avoided miss for *Always* is shown at the bottom

For benchmarks that were hurt by compression for a 4 MB L2 cache (ammp, gcc and twolf), compression helps performance for smaller cache sizes. This is due to compression allowing the L2 cache to hold more data (e.g., compression allows gcc to hold an average of ~1 MB in a 512 KB L2, resulting in more than a 3x speedup). However, compression hurts performance for larger cache sizes, since compression increases the hit latency but doesn't significantly increase the effective cache size. At the other extreme, mcf, apache and zeus benefit more from compression for larger caches (2 to 16 MB), since the working set is too large to fit in the smaller cache sizes, even with compression. For all cases, *Adaptive* adapts its behavior to match the better of *Always* and *Never.*

4.5.3 L2 Cache Associativity

Our cache compression proposal helps performance when it is able to increase the effective cache size. Since we are comparing to an uncompressed cache with the same associativity as a fully-compressed cache, L2 associativity plays an important role in the performance of cache compression. For caches with low associativities (e.g., 2-way set associative caches), a compressed cache has—in effect—a lower associativity than an uncompressed cache. For example, we compare a 1-2 way set-associative compressed cache to a 2-way uncompressed cache. Such associativity can be too low and can cause many more conflict misses for most benchmarks. When cache associativity is high, reduction in associativity has a lower impact on the performance of compressed caches.

Figure 4-8 illustrates the impact of L2 cache associativity on the relative performance of *Always* and *Adaptive* compared to *Never.* For benchmarks that are helped by compression (on the left hand side), compression's benefit increases for higher associativities. For example, mcf has a speedup of 70% for 16-way set associative caches compared to *Never.* However, that trend does not hold for high associativities for benchmarks that are hurt by compression (on the right hand side).

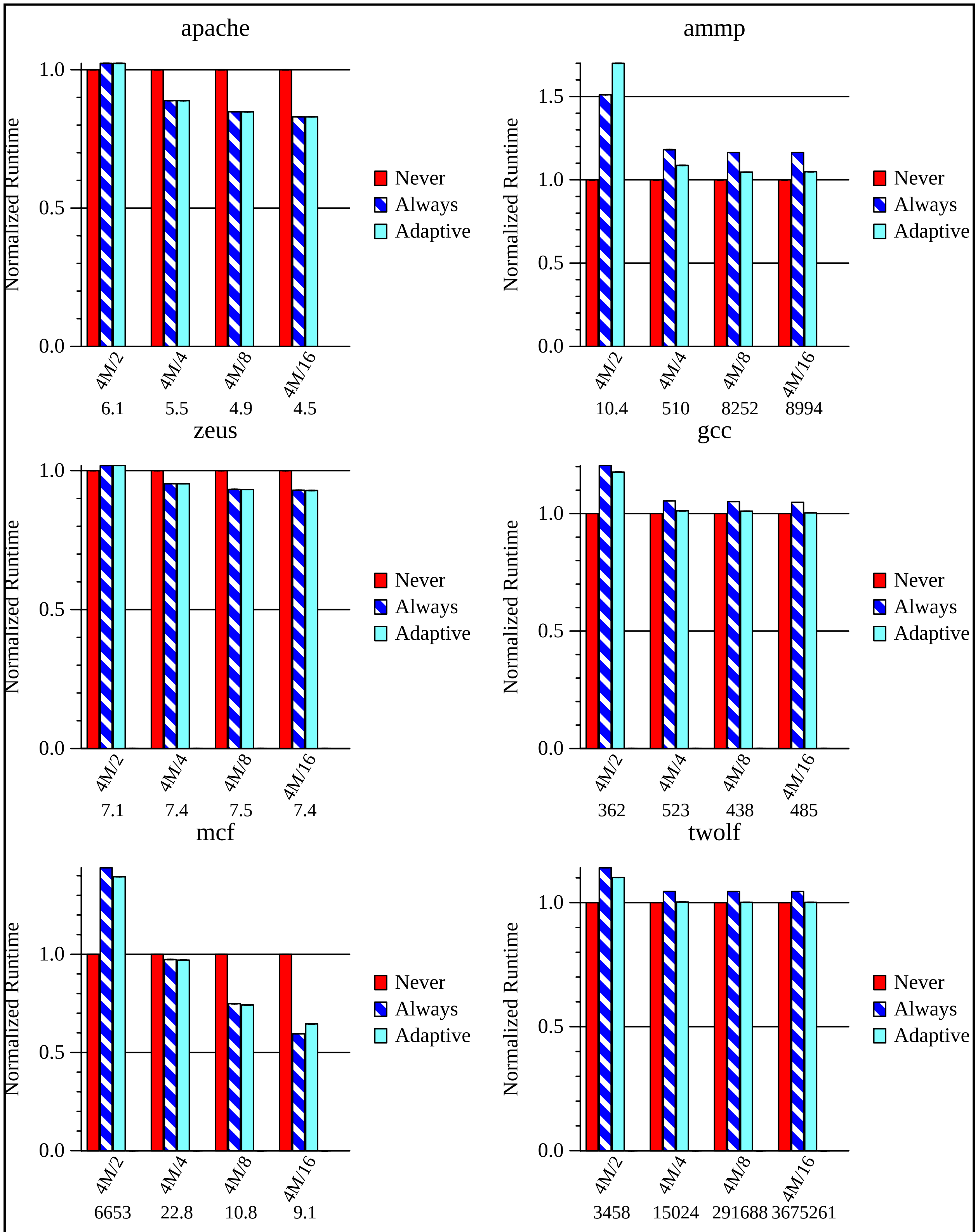


FIGURE 4-8. Sensitivity to L2 cache associativity of the three compression alternatives. The number of penalized hits per avoided miss for *Always* is shown at the bottom

Figure 4-8 also clearly shows that compression is ineffective when the associativity is low (e.g., 2-way). Even for benchmarks that are helped by compression for 8-way set associative caches, compression hurts for 2-way caches. This is due to the fact that compressed caches decrease the effective associativity, causing an increase in conflict misses that exceeds the decrease in misses due to compression. This is exacerbated for benchmarks that are hurt by compression. The extreme example in our experiment is ammp, in which *Always* increases the number of misses by 7%. Combined with decompression overheads, this causes a 51% slowdown compared to *Never*. Even worse, *Adaptive* adapts to *Never*, thus negating the performance gains due to the increased cache size, while still increasing the number of misses due to reducing associativity. The net increase in misses for *Adaptive* is 27% compared to *Never*, which causes a relative slowdown of 70%. This shows a weakness in our adaptive model which only considers avoided misses due to compression, and not additional misses due to reduced associativity caused by our compressed cache design. We do not consider reduced associativity since our decoupled variable-segment cache can only hold half the uncompressed lines per cache set, so *Adaptive* is comparing the performance of a 2-way compressed cache to that of a direct-mapped uncompressed cache in this case.

4.5.4 Memory Latency

As semiconductor technology continues to improve, processors may use faster clocks and deeper pipelines. A consequence of this trend is that cache and memory latencies are likely to increase (in terms of processor cycles), potentially decreasing the relative compression overhead (i.e., decompression latency) and increasing the potential benefit (i.e., eliminating longer latency misses). On the other hand, with the trend towards faster clocks slowing down, memory speeds may get to be smaller (in terms of cycles) compared to what they are now. We analyze the sensitivity of cache compression to both higher and lower memory latencies in Figure 4-9.

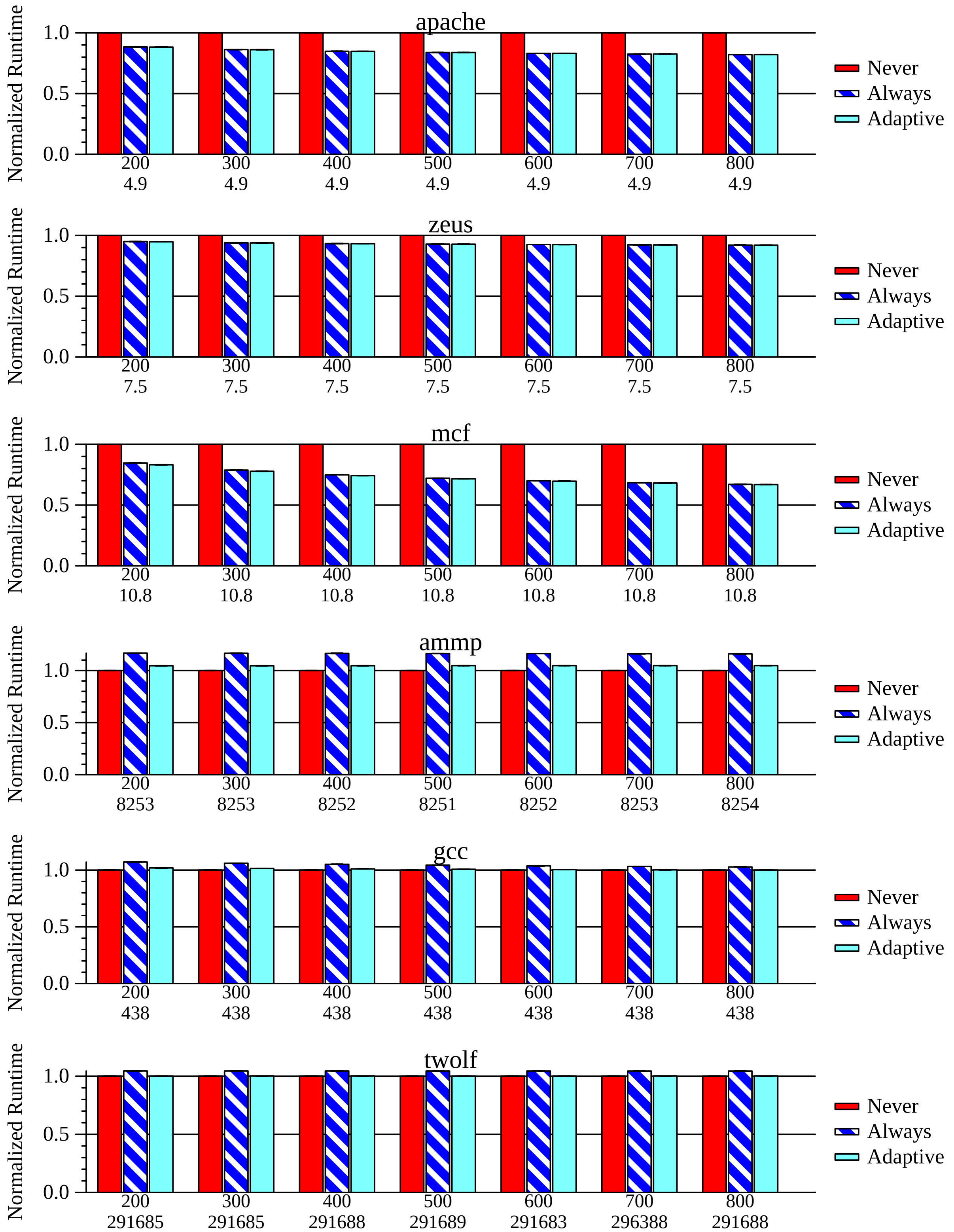


FIGURE 4-9. Sensitivity to memory latency of the three compression alternatives. The number of penalized hits per avoided miss for *Always* is shown at the bottom

For benchmarks that benefit from cache compression, performance benefits are smaller for the shorter memory latencies, and are larger for the longer memory latencies. In mcf, for example, compression speeds up performance by only 18% for a 200-cycle memory latency, and by 49% percent for a 800-cycle memory latency. On the other hand, performance of benchmarks that are hurt by compression varies slightly with a change in memory latency (e.g., less than a 2% swing in relative performance for *Always* in ammp). This phenomenon is due to the fact that such benchmarks have very few misses, and therefore penalized hits dominate performance. We note also that in all cases, *Adaptive* adapts its behavior to match the better of *Always* and *Never* (again with the notable exception of ammp where it is 4% slower than *Never*).

4.5.5 Decompression Latency

All cache compression schemes are highly sensitive to the decompression latency. Larger decompression latencies decrease the appeal of cache compression by increasing the effective access latency. On the other hand, smaller decompression latencies increases the benefits due to compression. We study the sensitivity of the three compression alternatives to decompression latency in Figure 4-10, where decompression latencies vary from 0 to 25 cycles.

For benchmarks that benefit from compression, speedup increases—as expected—when the decompression penalty is low and decreases when it is high. In mcf, *Always* speeds up performance by 40% for a 0-cycle decompression overhead, but by only 19% for a 25-cycle overhead. On the other hand, benchmarks that are hurt by compression are penalized more for *Always* when the decompression latency is high (e.g., ammp's performance is slowed down by 83% for a 25-cycle overhead, compared to 16% for a 5-cycle overhead). However, small changes in decompression latency do not have a significant impact on performance, since we simulate an out-of-order processor that can partly hide cache latency. Figure 4-10 also

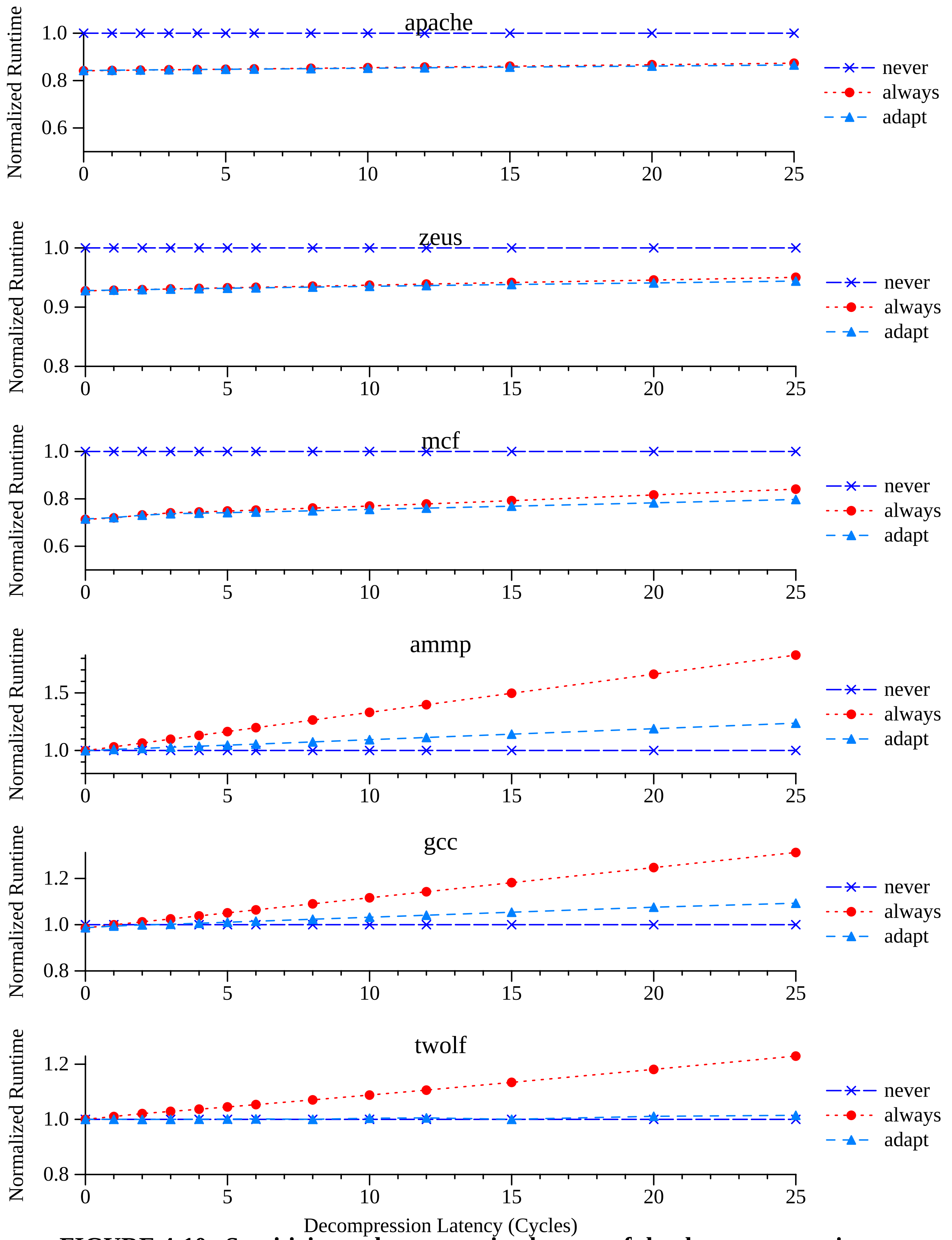


FIGURE 4-10. Sensitivity to decompression latency of the three compression alternatives. Decompression latency ranges from 0 cycles (perfect) to 25 cycles

show that *Adaptive* adjusts to changes in decompression latency, and therefore achieves performance comparable to the better of *Always* and *Never*.

4.5.6 Prefetching

Hardware prefetching is a technique that is used in many modern processors to hide memory latency. Hardware-directed stride-based prefetchers make use of repeatable memory access patterns to avoid some cache misses and tolerate cache miss latency [26, 96]. Current hardware prefetchers [58, 116, 117] observe the unit or fixed stride between two cache misses, then verify the stride using subsequent misses. Once the prefetcher reaches a threshold of strided misses, it issues a series of prefetches to the next level in the memory hierarchy to reduce or eliminate miss latency. Since compression is another technique used to hide memory latency, we studied the sensitivity of compression speedups to whether prefetching was implemented or not. We present a more detailed study for chip multiprocessors in Chapter 6.

We implemented a strided L1 and L2 prefetching strategy based on the IBM Power 4 implementation [116, 117] with some minor modifications, which we discuss in more detail in Chapter 6. Figure 4-11 shows how prefetching impacts the performance benefit due to compression. For most benchmarks, the only apparent difference in performance is the increase in the number of penalized hits per avoided miss when prefetching is implemented. For example, apache's penalized hits per avoided miss increases from 4.9 to 9.0 with both L1 and L2 prefetching. This is expected since prefetching avoids some of the misses that could have been avoided by compression, thus reducing compression's share of avoided misses. However, the relative speedup of cache compression is almost the same compared to *Never* regardless of whether prefetching is implemented.

The only exception to the above observation is gcc, where prefetching causes *Always* and *Adaptive* to achieve a small speedup as compared to a slowdown in the case of *Never*. This is cause by L1 prefetching triggering L2 fill requests from memory at a higher rate compared to gcc's original miss rate. Prefetching

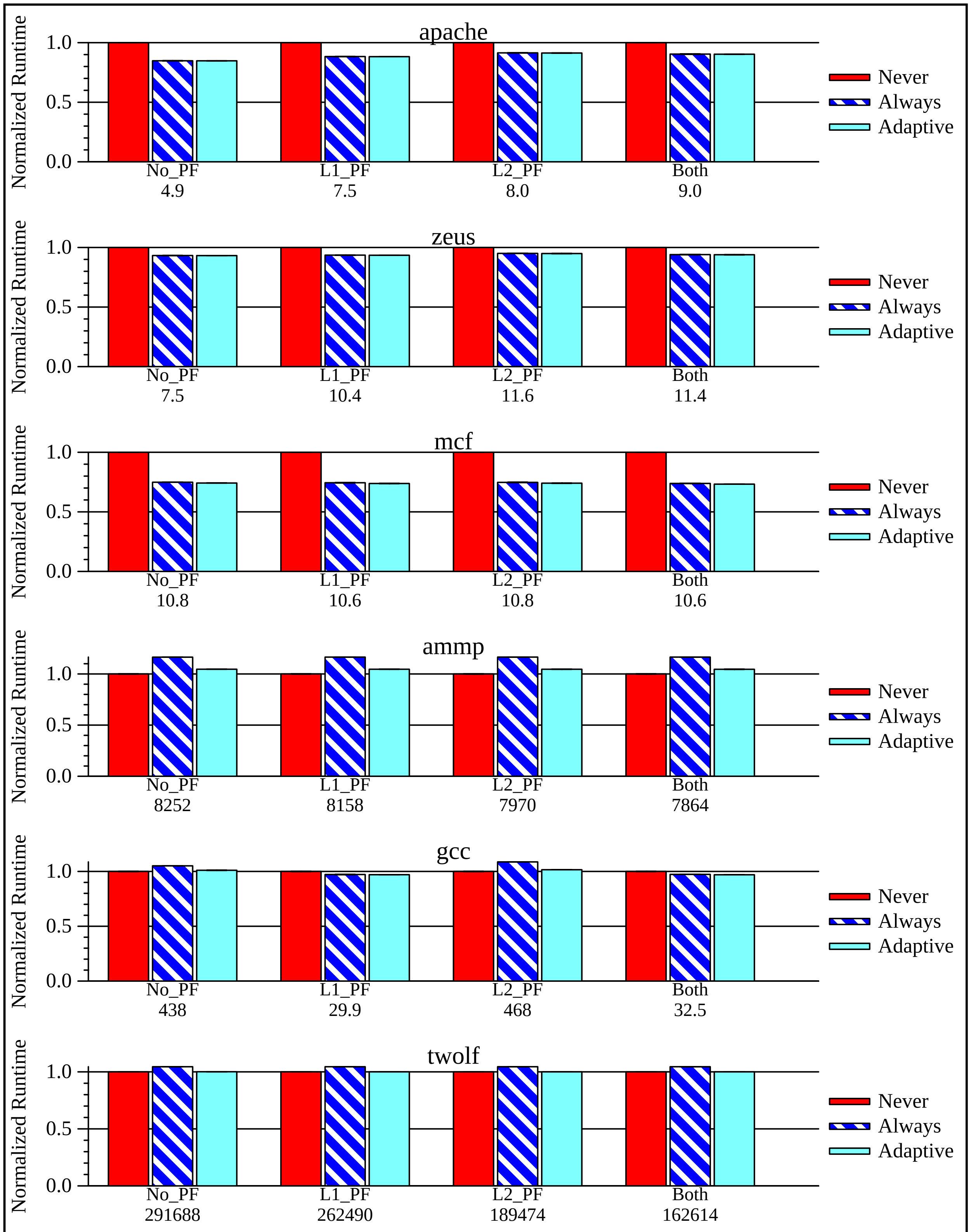


FIGURE 4-11. Sensitivity of compression benefit to L1 and L2 prefetching. The number of penalized hits per compression-avoided miss is shown at the bottom

therefore increases the L2 cache footprint (or working set size) of gcc to greater than 4 MB. Since compression can increase the effective cache size, it can alleviate some of the increased demand on the L2 cache and therefore speed up gcc's performance for *Always* and *Adaptive* compared to *Never.* We discuss similar interactions between compression and hardware prefetching in Chapter 6.

4.5.7 Cache Line Size

We evaluated the performance of cache compression for 64-byte cache lines. However, some modern processors have longer lines in their L2 cache as a technique to tolerate memory latency at the expense of increasing demand on memory bandwidth. Shorter cache lines increase the number of misses, while longer cache lines increase the required chip-to-memory bandwidth. We studied the sensitivity of cache compression's performance to various cache lines sizes (16, 32, 64, 128 and 256) in Figure 4-12.

For most benchmarks, compression provides a bigger benefit for the small cache line sizes. For example, twolf shows a 68% speedup due to compression for 16-byte line sizes. This is because compression significantly reduces the absolute number of misses, thus improving performance.

Longer cache lines do not follow a single trend regarding the impact of compression on performance. In some cases, longer lines tend to decrease the performance benefit because of compression. For example *Always* achieves only a 5% speedup for 128-byte lines, and a 6% slowdown for 256-byte lines in the mcf benchmark. However, this comes at the expense of increased bandwidth (more than 5x from 64 to 256-byte lines). In some other cases (e.g., apache and zeus), longer cache lines increase the performance benefit due to compression (e.g., 22% for 128-byte lines and 39% for 256-byte lines in zeus, compared to 7% for 64-byte lines). This is because the absolute number of misses is lower for larger cache line sizes, which inflates the relative improvement due to compression for these benchmarks. In all cases, *Adaptive* adapts its behavior to match the better of *Always* and *Never* (again except for long cache lines in ammp).

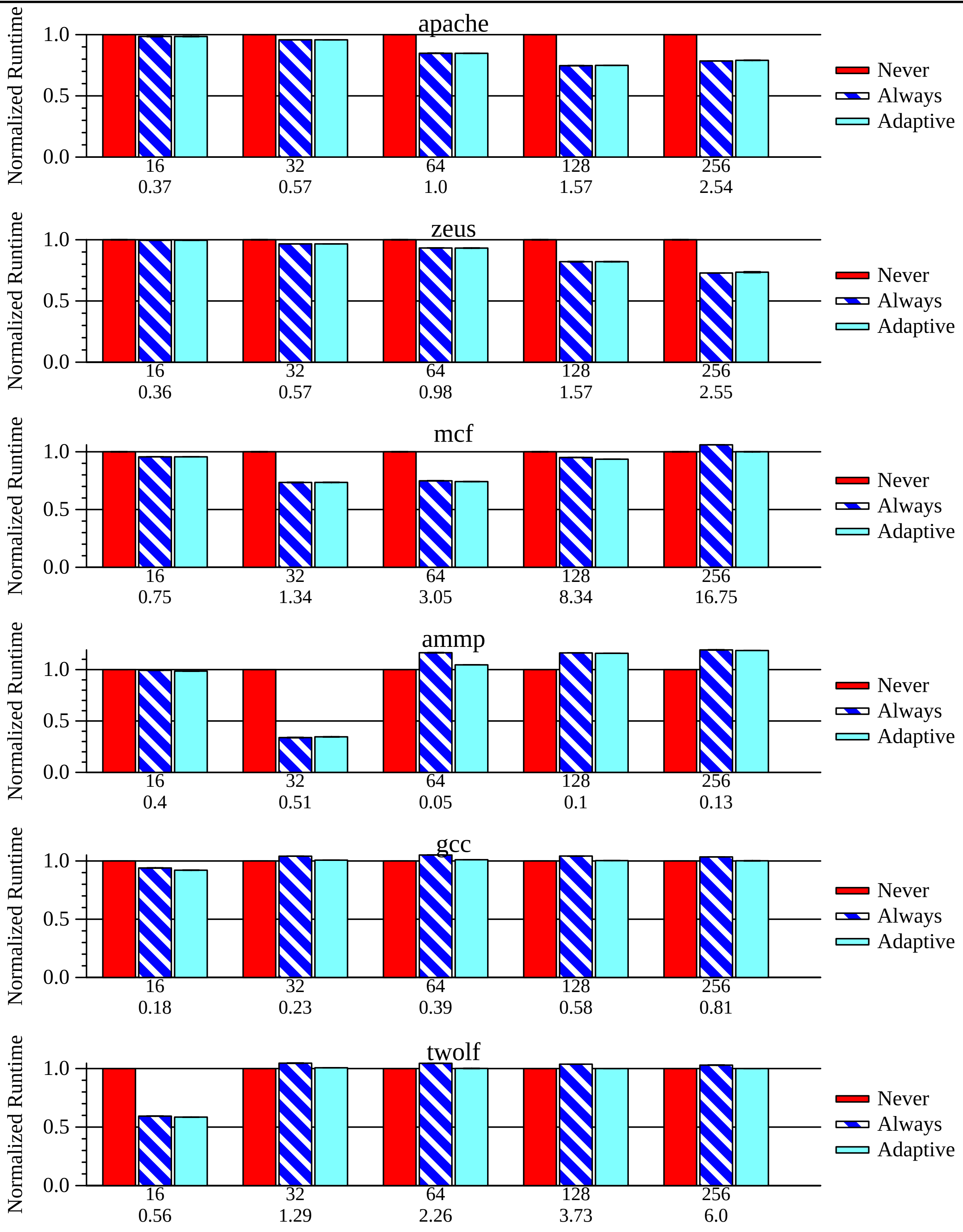


FIGURE 4-12. Sensitivity to cache line size of the three compression alternatives. We assume almost infinite off-chip bandwidth available

The actual bandwidth (in GB/sec.) required for *Never* is shown at the bottom

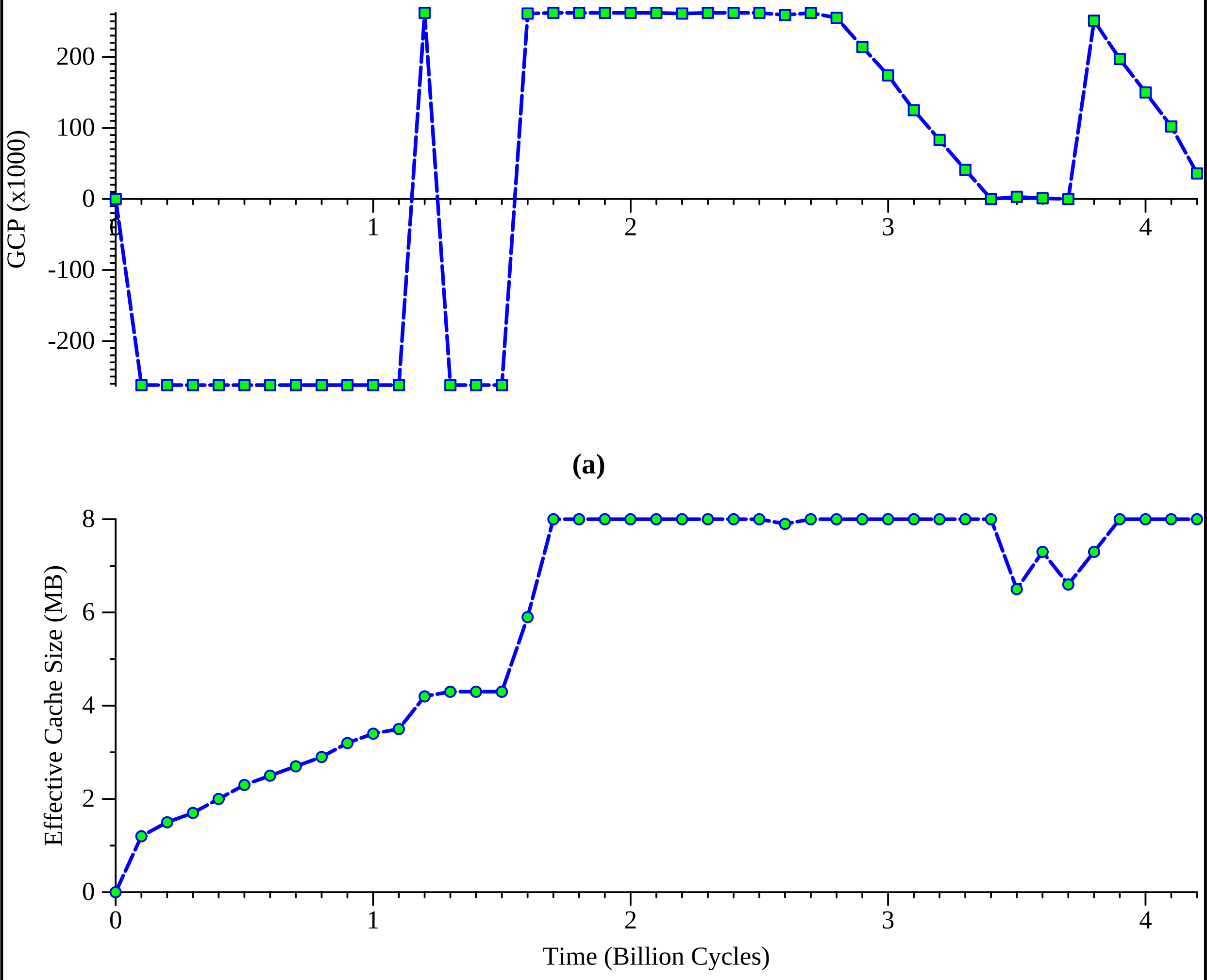


FIGURE 4-13. Phase Behavior for gcc with adaptive compression

(a) Changes in the Global Compression Predictor (GCP) values over time;
(b) Changes in effective L2 cache size over time

4.5.8 Benchmark Phases

Many benchmarks exhibit phase behavior [106], and a benchmark’s working set size may change between different phases. Such changes can affect the *Adaptive* policy, since the past (the previous phase) may not be a good predictor of the future (the next phase). On the other hand, adaptive compression can outperform both *Always* and *Never* for benchmarks with a changing working set size where neither extreme policy is the best all the time.

For our set of benchmarks, gcc had the most recognizable phase behavior. Figure 4-13 shows the changes over time of the Global Compression Predictor values (Figure 4-13 (a)) and the effective cache size (Figure 4-13 (b)) for a two-billion instruction run of gcc. It is clear from the figure that gcc has two distinct phases: A phase with a working set that is smaller than 4 MB (up to approximately 1.6 billion cycles), and a second phase with a working set bigger than 4 MB. In the first phase, *Adaptive* adapts to *Never* (since values of GCP are below zero) to avoid decompression overheads associated with *Always*. In the second phase, however, *Adaptive* adapts to *Always*, and the effective cache size shoots up to 8 MB to accommodate the bigger working set size. As a result, *Adaptive* outperforms both *Always* and *Never* for gcc, although by small margins (3% and 1%, respectively).

4.6 Discussion and Limitations

Evaluation of adaptive compression shows that our adaptive compression scheme adapts to benchmark behavior, adapting to *Never* when compression hurts performance, and adapting to *Always* when compression helps performance. With a few exceptions, *Adaptive* provides performance that is very close to the better of *Always* and *Never*, and outperforms both in some cases (Section 4.5.8). However, this adaptive policy is a simple, global scheme with room for improvement. In this section, we discuss some of the possible improvements and limitations for adaptive compression.

4.6.1 Possible Extensions

Since our adaptive compression scheme is a simple, global scheme, it has room to improve predictor accuracy. We list some of the possible improvements below.

- We can build a distributed predictor where each cache bank has its own saturating counter that is used to make compression predictions for that bank. This has the potential to help performance if different cache banks show different compression characteristics (i.e., different ratios for penalized hits per avoided miss). The number of bits per predictor should be adjusted to the ratio between the number of bits in the global predictor to the number of cache banks since each instance of the predictor is likely to be updated fewer times. In the general case, however, it is unlikely that such a distributed predictor will have a significant impact on performance. On the other hand, it may be necessary to build such a distributed predictor to avoid wiring delay between the global predictor and different banks.
- A smaller, simpler predictor can be attached to each cache set. This can help performance since different cache sets are likely to have different compression characteristics. While compression can help reduce miss rate for some cache sets, it can also hurt by increasing miss rates for other sets. However, a per-set predictor can significantly increase the overhead for each cache set in terms of area, power and bandwidth. The area and power increases are due to the additional predictor bits, as well as the circuitry required to update all predictors. Updates to multiple predictors may also place additional constraints on cache bank bandwidth.
- For heterogeneous multi-threaded applications, different applications might have different working set sizes that can be affected by compression in different (and sometimes opposite) ways. Having separate predictors for different threads can improve performance by avoiding compression for threads that are hurt by it, and compressing data for threads that benefit from it. However, it is not clear that such a prediction strategy is going to be effective since all threads share the same cache. The compression policy of one thread can affect the performance of another thread (e.g., a thread that is hurt by compression can store uncompressed lines where the space used by such lines could have been used by another thread that benefits from compression). In addition, such a predictor will require maintaining thread information (e.g.,. thread IDs) for L2 cache lines.

In order to estimate the potential benefits possible from any of these extensions, we studied a pseudo-ideal compression scheme in the next section.

4.6.2 Ideal Compression

Our adaptive compression scheme maintains a single predictor for the whole L2 cache. Similar to branch predictors, the accuracy of our global compression predictor can be improved, as discussed in the previous section. However, it is not clear that even a perfect predictor will lead to much improvement in performance. In this section, we study the limitations of a perfect compression predictor.

An ideal compression predictor would base its cache line allocation predictions on perfect knowledge of future accesses for every cache set. Unfortunately, simulating such a predictor requires prohibitive simulation time, since each prediction needs to compare the outcome of simulation for both cases (i.e., allocating a compressed or an uncompressed cache line). Another easier-to-implement upper bound on performance

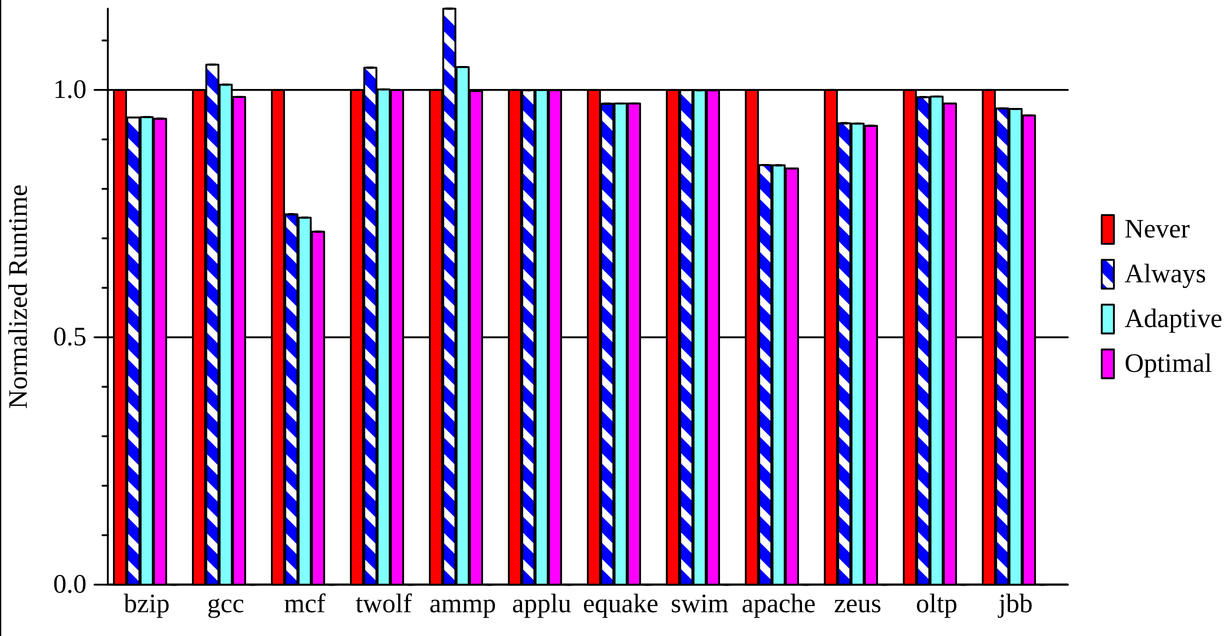


FIGURE 4-14. Normalized performance of the three compression alternatives (*Never, Always, Adaptive*), compared to an unrealistic pseudo-perfect compression scheme *"Optimal"* that is similar to *Always* with no decompression overhead

of such a perfect predictor is the performance of *Always* with zero decompression overhead. Figure 4-14 compares the performance of our three compression alternatives with such pseudo-perfect compression scheme.

Figure 4-14 shows that our adaptive compression policy achieves performance that is very close to the unrealistic pseudo-perfect upper bound. The *"Optimal"* scheme achieves a speedup of 0-4.6% over *Adaptive* across all benchmarks. Only three benchmarks show improvements that are greater than 2% with *"Optimal"* over *Adaptive* (4.6% for ammp, 4% for mcf, and 2.4% for gcc). For the remaining benchmarks, the pseudo-perfect compression scheme had an insignificant impact on performance. Based on these results, we find it hard to justify the additional hardware, area, power and bandwidth required for a more accurate compression prediction scheme.

4.6.3 Limitations

Our adaptive compression scheme is built on top of a decoupled variable-segment L2 cache. Its performance is limited by the limitations of our cache design. First, we assume Frequent Pattern Compression (FPC) as our compression algorithm. Second, we are limited to double the number of lines for each cache set compared to an uncompressed cache. Third, we assume only a two-level cache hierarchy. However, we can also apply adaptive compression when these limitations are removed. We discuss such designs next.

Fully-associative caches. The indirect-indexed cache and the V-Way cache are examples of recent proposals for fully-associative cache implementations [54, 55, 98]. For such schemes, the global replacement policy is responsible for classifying different types of cache misses (Section 4.1.2). Unfortunately, these proposals do not maintain accurate replacement stack information. However, approximate information can be used to update the compression predictor since it does not need to be completely accurate. For example, the generational replacement algorithm in the Indirect Index Cache can be extended by adding more pools to accommodate compressed cache lines [54]. An access to higher-priority pools can be considered a penalized hit (for compressed lines), whereas an access to lower-priority pools can be considered an avoided miss. Another coarse approximation can be implemented for the reuse replacement algorithm used in the V-Way cache [98]. In this case, an access to a line whose reuse counter value is in the top half (e.g., 2 or 3 for 2-bit reuse counters) can be considered a penalized hit. An access to a line with a reuse counter value in the bottom half is considered an avoided miss with some probability that corresponds to the probability of evicting this line without compression. However, further study for these schemes is needed to estimate their effectiveness.

Different compression algorithms. The implementation of adaptive compression is straight-forward with a different compression algorithm. The only parameter that needs adjustment is the decompression penalty. For compression algorithms with variable decompression penalties, a variable latency can be taken into consideration in updating the compression predictor. Alternatively, an average can be used as an estimate decompression penalty to minimize the predictor update overhead (since predictor updates can be coarsely accurate).

Deeper cache hierarchies. Our adaptive compression scheme is implemented at the second-level cache in a two-level cache hierarchy. We believe that compression at the L1 level has a high overhead and a negative impact on performance. However, adaptive compression can be implemented at any other level of the cache or memory hierarchy where compression can help or hurt performance. Adaptive compression has been proposed for virtual memory systems as we discuss in the next section.

4.7 Related Work

Adaptive compression has been previously proposed for virtual memory management schemes. In such systems, portions of main memory are compressed (and thus called compression caches) to avoid I/O operations caused by page faults. Douglis observes that different programs need compressed caches of different sizes [34]. He implements a simple adaptive scheme that dynamically split main memory pages between uncompressed and compressed portions. Both portions compete for the LRU page in memory. However, he biases allocating a new page towards the compression cache. Cortes, et al., classify reads to the compression cache according to whether they were caused by swapping or prefetching, and propose optimized mechanisms to swap pages in/out [30].

Wilson, et al., propose dynamically adjusting the compressed cache size using a cost/benefit analysis that compares various target sizes, and takes into account the compression cost vs. the benefit of avoiding I/Os [129]. Their system uses LRU statistics of touched pages to compare the costs and benefits of target sizes, and adjusts the compression cache size on subsequent page accesses. However, the adaptive compression scheme we propose in this chapter is different since it is not restricted to specific compressed-cache quotas. Freedman [49] optimizes the compression cache size for handheld devices according to the energy costs of decompression vs. disk accesses.

Chapter 5

Cache and Link Compression for Chip Multiprocessors

Chip multiprocessor caches experience greater capacity demand compared to uniprocessor caches since they are shared among multiple processors. Such high demand can increase cache miss rates and contention for the limited off-chip pin bandwidth. A CMP design should balance processor cores, shared caches, and off-chip pin bandwidth so that no single resource is the only bottleneck. In this chapter, we explore using cache and off-chip interconnect (link) compression to more efficiently utilize the shared cache and communication resources on a CMP.

Compression increases the effective cache capacity (thereby reducing off-chip misses) and increases the effective off-chip bandwidth (reducing contention). On an 8-processor CMP with no prefetching, we show that L2 cache compression improves commercial workloads' performance, but has little benefit for scientific workloads. We also show that adding link compression greatly reduces pin bandwidth demand for most of our workloads.

We first present our Chip Multiprocessor design with cache compression support in Section 5.1. We then motivate interconnect compression and discuss how to implement it on a CMP (Section 5.2). We evaluate our compressed CMP design for an 8-core CMP with commercial and SPEComp workloads (Section 5.3). We show that cache compression improves performance by 5-18% for commercial workloads, and that link compression reduces off-chip bandwidth demand for most workloads by 17-41% (Section 5.4). We study the sensitivity of our results to various system parameters in Section 5.5. We summarize our results in Section 5.6.

In this chapter, we make the following contributions:

- We extend our compressed cache design to CMPs. We also propose a CMP design that supports link compression.
- We show that cache compression helps increase the effective CMP shared cache size, reduce miss rate and improve performance for commercial workloads.
- We show that link compression greatly reduces off-chip pin bandwidth demand for commercial and (some) scientific workloads, potentially improving performance.

5.1 C-CMP: A CMP with Compression Support

As we discussed in Chapter 1, semiconductor technology trends continue to exacerbate the wide gap between processor and memory speeds as well as the increasing gap between on-chip transistor performance and the available off-chip pin bandwidth. Both of these trends favor allocating more of the on-chip transistors to caches. On the other hand, throughput-oriented commercial workloads place an increasing demand on the processor resources of a system to sustain their increasing transaction processing rates.

With technology trends favoring more cache area and workload trends favoring more processing resources, cache compression provides an appealing alternative to achieve the best of both worlds. A compressed cache system has the potential to increase the effective cache capacity, thereby reducing off-chip misses. Having fewer misses leads to improved performance, power savings, and decreased demand for off-chip pin bandwidth. We next describe our design for a CMP with compression support.

5.1.1 C-CMP Design

Our base design is a *p*-processor CMP with single-threaded cores. Most of our design parameters are an extrapolation of a next-generation CMP chip loosely modeled after IBM's Power5 [68] and Sun's Niagara [74], except that our chip only has single-threaded cores. Each processor has a private L1 I-cache and a private L1 D-cache. The shared L2 cache is divided into *b* banks, and cache line addresses are mapped into banks based on the least significant address bits.

We extend the base design to include compression support for both the caches and the interconnect. Figure 5-1 summarizes our proposed CMP system. For all processor cores, private L1 instruction and data caches store uncompressed lines, eliminating the decompression overhead from the critical L1 hit path. On a hit to a compressed L2 line, the line is decompressed before going to the L1 cache. A hit to an uncompressed L2 line bypasses the decompression pipeline. On an L2 miss, the requested line is fetched from main memory or directory and, if compressed, runs through the decompression pipeline on its way to the L1. For design simplicity, we assume a single line size for all caches. We use an MSI-based coherence protocol between the L1's and the shared L2, and an MOESI-based protocol between L2 caches in a multi-CMP system (though we only studied systems with a single CMP in this dissertation). The L2 cache maintains strict inclusion and has full knowledge of on-chip L1 sharers via individual bits in its cache tag. L1

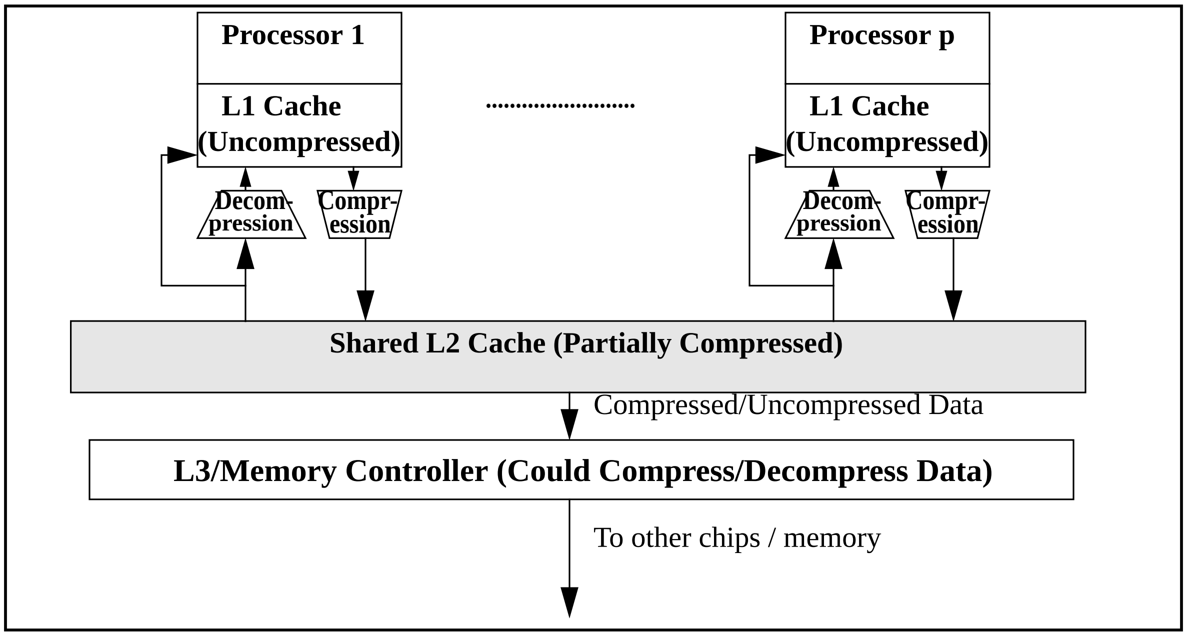


FIGURE 5-1. A Single-Chip *p*-core CMP with Compression Support

caches are write-back caches, and only communicate with memory through the shared L2 cache where inter-chip coherence is maintained. We next describe our support for cache compression.

5.1.2 Support for Cache Compression

In our design, we implement each bank of the shared L2 cache as a decoupled variable-segment cache (as we described in Chapter 3). We extend our uniprocessor design from previous chapters to target a CMP. We evaluated an 8-way set associative L2-cache with a compression tag stored with each address tag. The compression tag indicates the compressed size of the line and whether or not the line is stored in compressed form. The data area is broken into eight-byte segments, with 32 segments statically allocated to each cache set. Thus, each set can hold no more than four uncompressed 64-byte lines, and compression can at most double the effective capacity. Each line is compressed into between one and eight segments, with eight segments being the uncompressed form. We maintain inclusion between the L1 cache contents and the shared L2 cache. We also use the Frequent Pattern Compression scheme (Chapter 3) to compress individual cache lines into multiples of these eight-byte segments. Our decoupled variable-segment cache adds relatively little storage overhead to the cache area, approximately 7% for a 4MB cache (Chapter 3).

While compression helps eliminate long-latency L2 misses, it also increases the latency of the more frequent L2 hits. For benchmarks (or benchmark phases) that have working sets that fit in an uncompressed cache, compression only degrades performance. We therefore use the adaptive cache compression scheme (Chapter 4) to dynamically adapt to workload behavior, and compress only when the benefit of compression exceeds its cost. However, in our evaluation, none of our benchmarks suffered a significant degradation due to compression, since their working sets did not fit in our uncompressed cache.

Adaptive compression only considers compression costs from avoiding off-chip misses and compression costs from penalized hits to compressed lines. It does not take into account other compression benefits such as the reduction in on-chip and off-chip bandwidth demand. However, our simple adaptive model did not negatively impact performance for our benchmarks since they all adapted to the *Always-Compress* policy. We next describe link compression and its role in reducing off-chip bandwidth demand.

5.2 Link Compression

5.2.1 Technology Trends

While CMP systems can increase commercial workloads' throughput, a CMP design inherently increases the amount of off-chip bandwidth required (per-chip) for inter-chip and chip-to-memory communication compared to a uniprocessor system. Without any optimizations to reduce off-chip bandwidth, adding more processors on a chip will increase the amount of data transferred for communication with main memory (and maintaining memory consistency between chips in a CMP-based multiprocessor system). This problem is exacerbated by hardware-directed prefetching schemes that target increasing the memory-level parallelism and reducing off-chip demand misses [22].

As we discussed in Chapter 1, the 2004 ITRS roadmap [45] predicts that the number of pins available per chip for high performance processors will increase at a rate of approximately 11% per year till 2009. This is a much lower rate than the 26% predicted as the annual rate of increase in the number of transistors per chip. These trends imply that the number of processor cores on a single chip could increase at a much faster rate compared to the number of communication pins available. Huh, et al. [62] identified pin bandwidth as a potential limiting factor for CMP performance. Furthermore, Kumar, et al. [76] identified on-chip interconnect bandwidth as a first-order CMP design concern.

Overall, the increasing demand on off-chip bandwidth appears to be a problem that will significantly get worse for future CMP designs, unless emerging technologies (e.g., optical interconnects) evolve quickly. A balanced CMP design balances demands for bandwidth against the limited number of pins and wiring area per chip [32]. In order to reduce such bandwidth demand, an obvious solution is to increase the on-chip cache size to reduce off-chip misses. Unfortunately, this comes at the expense of reducing on-chip processor area, thereby reducing potential throughput.

In this dissertation, we propose using cache and link compression to reduce the off-chip bandwidth demand for inter-chip and chip-to-memory communication. Cache compression helps as it reduces the cache miss rate, thus eliminating some off-chip accesses. Link compression helps by compressing both outgoing and incoming communication messages, which also reduces a workload's bandwidth demand.

5.2.2 On-Chip Link Compression

To make use of the reduction in off-chip bandwidth due to compression, the L3/memory controller—the on-chip part of the memory controller—must be able to send and receive compressed messages to/from the chip. In addition, the off-chip memory controller must be able to send/receive compressed messages and understand the format required for compressed messages. Off-chip messages should support compressed formats, and the memory controller should be equipped to handle compressed lines.

Our design uses a message format that is similar to the segment format of the decoupled variable-segment cache. We also use the same FPC compression algorithm to compress cache lines (if they were not already compressed). Each data message that originally included a complete cache line is transferred in 1-8 submessages (flits), each containing an 8-byte segment. The message header contains a length field indicating the number of segments in the line. In our evaluation, we assume that no messages or flits are lost or corrupted during transmission. However, this constraint can be easily relaxed by applying standard flow control and error detection mechanisms. ECC codes can be maintained on a per-line basis whereby an error in a single segment requires the retransmission of all segments of a line.

Assuming compressed memory, the off-chip memory controller combines flits of the same cache line and stores the combined cache line to physical memory. However, this requires having an additional bit per line to indicate whether the line is compressed, possibly encoded in the line's error-correcting code [101]. If memory is uncompressed, the off-chip memory controller must have the capability to compress lines on their path from main memory to the chip, and decompress compressed data messages sent from the chip to memory. Using the simple frequent pattern compression scheme greatly reduces the latency overhead due to compression and decompression, as exploited by Ekman and Stenstrom [40].

Link compression affects both bandwidth and latency in a CMP. Link compression increases the effective pin bandwidth, which can significantly improve performance for bandwidth-limited benchmarks. On the other hand, the impact of link compression on latency is not significant, especially when a system implements cache compression. We summarize the impact of link compression on latency for outgoing (i.e., from chip to memory/other chips) and incoming messages as follows:

- For outgoing compressed cache lines transferred off-chip, no additional compression overhead is incurred.
- Outgoing uncompressed cache lines are not compressed if compression does not save bandwidth (i.e., if their compressed size has the same number of segments as an uncompressed line). When such lines are compressible, however, a compression penalty is added to the memory writeback latency which has little effect since it is off the critical path.
- Incoming compressed data from memory or other system CMPs can be directly filled into the compressed L2 cache. When such data is requested by the L1 cache or the processor, a decompression overhead is incurred, which is on the critical path. However, such overhead is relatively small (5 cycles in our design) relative to the memory access penalty (typically measured in hundreds of cycles).

5.2.3 Memory Interface

Memory compression has previously been proposed to increase the effective memory capacity and reduce overall system cost. For example, IBM’s Pinnacle chip [120] implements the IBM Memory Expansion Technology (MXT) [121]. In MXT, memory management hardware dynamically allocates main memory

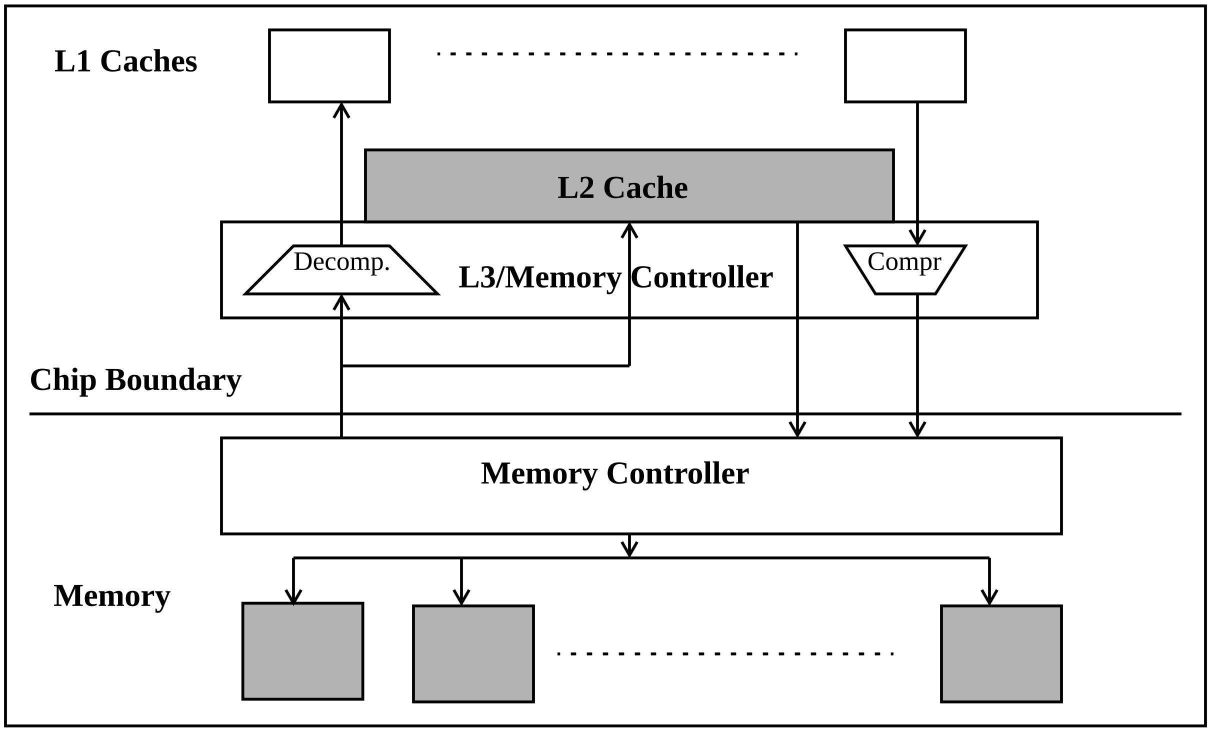


FIGURE 5-2. Link Compression on a CMP

Components that store compressed data are shaded (i.e., L2 cache and memory). Cache lines are compressed before sending to memory, and L1 fills have to be compressed. This figure assumes compressed memory. If memory is not compressed, compression and decompression circuits have to be added to the off-chip memory controller.

storage in small 256-byte sectors in order to support variable-size compressed data with minimal fragmentation. The compressed memory is divided into two logical structures: the Sector Translation Table and the sectored memory. This scheme requires support from the operating system [1].

Since our focus in this dissertation is on cache and link compression rather than memory compression, we use a simpler scheme that does specifically target increasing the effective memory capacity. We propose storing our 64-byte cache lines in either uncompressed or compressed form in memory, with a bit encoded into the ECC to indicate whether the corresponding line is compressed or uncompressed [101]. In Figure 5-2, we show a CMP with link compression support that assumes compressed memory. An alternative scheme to use is the memory compression scheme recently proposed by Ekman and Stenstrom [40] that also uses the same FPC compression algorithm to store cache lines in memory. Neither scheme uses the more complex compression algorithms that have higher in-memory compression ratios. However, both schemes have the advantage of being transparent to software. Cache and link compression can also be used with the MXT scheme with additional overheads and complexity due to the difference in compression algorithms and granularities between caches and memory.

5.3 Evaluation

We present an evaluation of cache and link compression using an 8-core CMP, where each core is a single-threaded dynamically-scheduled out-of-order processor with private L1 caches. We use full-system simulation of commercial workloads and a subset of the SPEComp benchmarks. We next describe our base system configuration and the workloads we use in our evaluation.

5.3.1 Base System Configuration

We evaluate the performance of our compressed cache and link designs on an 8-core CMP with SPARC V9 processors and a 5 GHz clock. We use the Simics full-system simulator [88], extended with GEMS [130] (a detailed memory system timing and out-of-order processor simulator). Table 5-1 presents our basic simulation parameters. All parameters are the same as those in Chapter 4 except for CMP-specific parameters and the lower, more realistic pin bandwidth. Our base system is modeled as a future generation CMP inspired by IBM’s Power5 [68] and Sun’s Niagara [74], except that our CMP only has single-threaded cores. Different base system parameters are based on our speculation of future CMP parameters. We use this base system to demonstrate the impact of cache and link compression in Section 5.4. However, we also study the sensitivity of our results to various parameters in Section 5.5.

TABLE 5-1. CMP Simulation Parameters

Processor Cores	Eight processors, each a single-threaded core with private L1 caches.
Private L1 Caches	Split I & D, each 64 KB 4-way set associative with LRU replacement, 64-byte lines, 3-cycle access time, 320 GB/sec. total on-chip bandwidth (from/to L1's).
Shared L2 Cache	Unified 4 MB, composed of eight 512KB banks, 8-way set associative (uncompressed) or 4-8 way set associative (compressed) with LRU replacement, 64-byte lines, 15 cycle uncompressed hit latency (includes bank access latency), 20 cycles compressed hit latency (15 + 5 decompression cycles).
Memory Configuration	4 GB of DRAM, 400 cycles access time with 20 GB/sec. chip-to-memory bandwidth, each processor can have up to 16 outstanding memory requests.
Processor Model	Each processor is an out-of-order superscalar processor with a 5 GHz clock frequency.
Processor Pipeline	4-wide fetch and issue pipeline with 11 stages (or more): fetch (3), decode (4), schedule (1), execute (1 or more), retire (2).
IW/ROB	64-entry instruction window, 128-entry reorder buffer.
Branch Prediction	4 KB YAGS direct branch predictor [38], a 256-entry cascaded indirect branch predictor [35], and a 64-entry return address stack predictor [67].

5.3.2 Workloads

To evaluate compression alternatives, we use several multi-threaded commercial workloads from the Wisconsin Commercial Workload Suite [6]. We also use four benchmarks from the SPEComp2001 suite [12]. All workloads run under the Solaris 9 operating system. These workloads are briefly described in Table 5-2. Commercial workloads are the same as those in the previous chapters except for multiprocessor-related parameters (e.g., the number of users or threads). We selected workloads that cover a wide range of compressibility properties, miss rates, and working set sizes. For each data point in our results, we present the average and the 95% confidence interval of multiple simulations to account for space variability [8]. Our runtime results for commercial workloads represent the average number of cycles per transaction (or request). For SPEComp benchmarks, our runtime results represent the average number of cycles required to complete the main loop.

TABLE 5-2. Workload Descriptions

Online Transaction Processing (OLTP): DB2 with a TPC-C-like workload. The TPC-C benchmark models the database activity of a wholesale supplier, with many concurrent users performing transactions. Our OLTP workload is based on the TPC-C v3.0 benchmark using IBM's DB2 v7.2 EEE database management system. We use a 5 GB database with 25,000 warehouses stored on eight raw disks and an additional dedicated database log disk. We reduced the number of districts per warehouse, items per warehouse, and customers per district to allow more concurrency provided by a larger number of warehouses. We simulate 128 users, and warm up the database for 100,000 transactions before taking measurements for 100 transactions.
Java Server Workload: SPECjbb. SPECjbb2000 is a server-side java benchmark that models a 3-tier system, focusing on the middleware server business logic. We use Sun's HotSpot 1.4.0 Server JVM. Our experiments use 1.5 threads and 1.5 warehouses per processor (12 for 8 processors), a data size of ~44 MB, a warmup interval of 200,000 transactions and a measurement interval of 2,000 transactions.
Static Web Serving: Apache. We use Apache 2.0.43 for SPARC/Solaris 9, configured to use pthread locks and minimal logging as the web server. We use SURGE [13] to generate web requests. We use a repository of 20,000 files (totalling ~500 MB), and disable Apache logging for high performance. We simulate 400 clients per processor (3200 clients for 8 processors), each with 25 ms think time between requests. We warm up for ~2 million requests before taking measurements for 500 requests.
Static Web Serving: Zeus. Zeus is another static web serving workload driven by SURGE. Zeus uses an event-driving server model. Each processor of the system is bound by a Zeus process, which is waiting for web serving event (e.g., open socket, read file, send file, close socket, etc.). The rest of the configuration is the same as Apache (20,000 files of ~500 MB total size, 3200 clients, 25 ms think time, ~2 million requests for warmup, 500 requests for measurements).
SPEComp. We use four benchmarks from the SPEComp2001 benchmark suite [12]: 330.art, 324.apsi, 328.fma3d, and 314.mgrid. We used the ref input set, and fast-forwarded each benchmark till the beginning of the main loop. We warmed up caches for approximately 2 billion instructions, and measured till the end of the loop iteration. Since these benchmarks are multi-threaded, we use a work-related metric rather than IPC to address workload variability [8]. Loop completions seemed the best choice [17], since simulating the whole benchmark takes a prohibitive period of time.

5.4 Cache and Link Compression Performance

In this section, we compare the relative benefits of cache and link compression. We monitor their separate and combined effects on miss rates, off-chip bandwidth and performance for our base 8-core, 4 MB L2 configuration. We simulated the following configurations: No compression, cache compression only, link compression only, and both cache and link compression. Neither L1 nor L2 prefetching is implemented for results in this section. Cache compression implements our adaptive cache compression policy (Chapter 4) which reverts to *Always* for all benchmarks.

5.4.1 Workload Compressibility

A compression scheme is successful if it can significantly increase the effective cache size. We use *compression ratio*s of different benchmarks as indicators of a compression algorithm's success, as we discussed in Chapters 1 and 3. As in Chapter 3, we define the *compression ratio* of a cache snapshot as the quotient of the effective cache size divided by the uncompressed size of 4 MB. We computed compression ratios as an average of compression ratios for periodic snapshots taken every 50 million cycles during a benchmark's runtime. This method is similar to how we computed compression ratios for uniprocessor benchmarks in Chapter 3, except for a shorter sampling interval.

Our eight workloads show a wide range of compression ratios (Table 5-3). Compression ratios for commercial benchmarks were relatively high, ranging from 1.36 to 1.8. Zeus's compression ratio is 1.8, which translates into approximately 7.2 MB of effective cache size (compared to the 4 MB base case). However, SPEComp benchmarks showed smaller gains, with compression ratios ranging from 1.01 to 1.19. This can be attributed to the fact that the simple compression scheme we chose does not perform as well for floating point data, for which lossless compression remains a hard problem even for more complex compression schemes (as we discussed in Chapter 3).

TABLE 5-3. Compression Ratios for a 4MB cache for commercial and SPEComp benchmarks

Benchmark	**apache**	**zeus**	**oltp**	**jbb**	**art**	**apsi**	**fma3d**	**mgrid**
Compr. Ratio	1.74	1.80	1.36	1.48	1.15	1.01	1.19	1.02

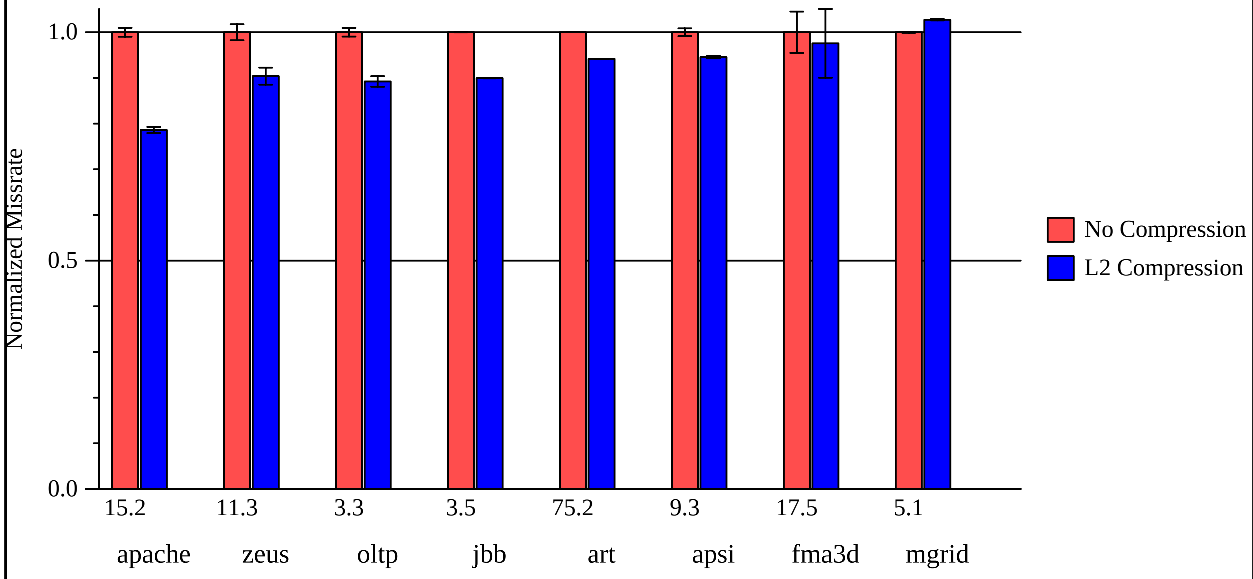


FIGURE 5-3. Cache miss rates normalized to miss rate without compression

Misses per 1000 instructions for the no compression case are shown at the bottom.

5.4.2 Reduction in Cache Misses

Compression increases the effective cache size for most benchmarks. With such increases in cache size, commercial benchmarks show a reduction in cache miss rates ranging from 10-22% (Figure 5-3). For SPEComp benchmarks, the maximum miss rate reduction was 6% due to their lower compression ratios. We note that mgrid has a 3% higher miss rate with compression due to the decrease in associativity (4-way vs. 8-way) that is not offset by the increase in cache size.

5.4.3 Bandwidth Reduction

We define *pin bandwidth demand* as a workload's utilized pin bandwidth on a system with an infinite available pin bandwidth. Without prefetching, our set of commercial workloads is not bandwidth-limited, as we demonstrate in Figure 5-4. The average bandwidth demand ranges from 5.0 GB/sec. for oltp to 8.8 GB/sec.

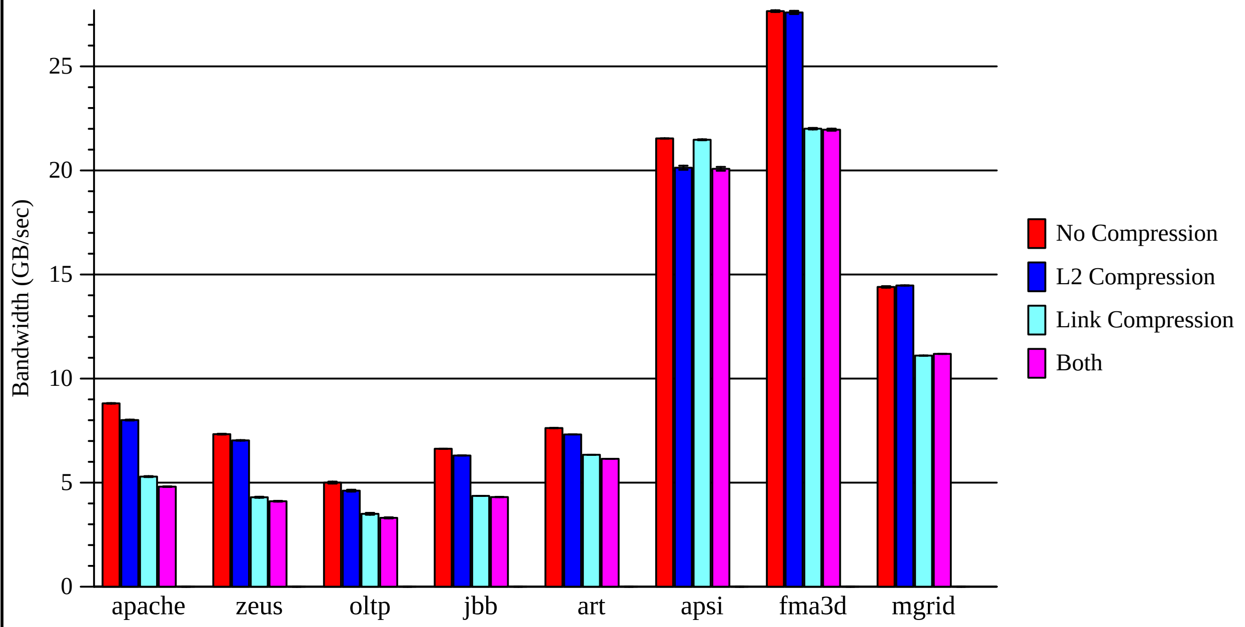


FIGURE 5-4. Pin Bandwidth demand for all benchmarks (in GB/sec.) for compression alternatives

Pin bandwidth demand is based on a CMP with infinite available bandwidth.

for apache, which are much lower than the available 20 GB/sec. pin bandwidth in our baseline system. For SPEComp benchmarks, however, bandwidth demand is high, ranging from 7.6 GB/sec. for art to 27.7 GB/sec. for fma3d.

Figure 5-4 presents the bandwidth demand for our benchmarks with no compression, only cache compression, only link compression, and both types of compression. Link compression can achieve up to 41% reduction in off-chip bandwidth (for zeus), a significant reduction for bandwidth-limited configurations. Link compression achieves a 34-41% reduction in off-chip bandwidth for the four commercial benchmarks, and 17-23% reduction for three of the four SPEComp benchmarks. Only apsi, whose compression ratio is 1.01, fails to achieve a significant bandwidth reduction. The combination of cache and link compression achieves a 35-45% reduction in off-chip bandwidth for commercial benchmarks, and a 7-23% reduction for SPEComp benchmarks.

We note from Figure 5-4 that the impact of cache compression alone on bandwidth reduction is sometimes smaller than expected given the compression ratios in Table 5-3. This can be attributed to the fact that cache compression affects two terms in the bandwidth equation (assuming a blocking in-order processor):

$$BandwidthDemand(bytes/sec) = (bytes/miss) \times (misses/instr) \times (instructions/cycle) \times (cycles/sec) \tag{5.1}$$

Cache compression reduces the number of *misses/instr*, which should reduce bandwidth demand. However, such reduction in miss rate also improves the number of *instructions/cycle*, which increases the bandwidth demand. Both of these effects almost offset for many benchmarks, and so our results show that cache compression has little impact on bandwidth reduction. For our benchmarks, cache compression alone led to a decrease of up to 9% in bandwidth demand (for apache), but bandwidth is not significantly reduced for most benchmarks.

Compared to cache compression, link compression reduces the *bytes/miss* term of the above equation, and has minor impact on the other terms except for systems with high contention. Therefore, link compression always leads to a reduction in bandwidth demand for compressible benchmarks, as we show in Figure 5-4.

5.4.4 Performance

Cache compression achieves a significant reduction in miss rates, especially for commercial benchmarks (Section 5.4.2). For these benchmarks, cache compression has a significant impact on performance. Figure 5-5 shows that cache compression alone can speed up performance of our base, 8-core 4 MB L2 system by 5-18% for our four commercial workloads. However, it doesn't perform as well for the less-compressible SPEComp benchmarks (0-4% speedup). For 20 GB/sec. bandwidth, link compression has an impact on performance only for benchmarks with high off-chip bandwidth requirements (up to a 23%

speedup for fma3d). The combined speedup of cache and link compression is slightly higher than that of cache compression alone (except for fma3d where link compression shows a significant speedup).

These results show speedups by cache and link compression for a system that does not implement hardware prefetching. However, many current systems have some sort of hardware prefetching implemented. For such systems to consider implementing compression, we need to study the impact of compression on the performance of such systems. We also need to study whether compression complements the performance benefits due to prefetching and vice-versa. We study the interactions between compression and hardware prefetching in the next chapter.

5.5 Sensitivity Analysis

As with many architectural enhancements, the performance impact of cache and link compression is affected by changes in system configuration parameters. In this section, we investigate how compression's performance is affected by changes in L1 and L2 cache sizes and associativities, memory latency, and pin

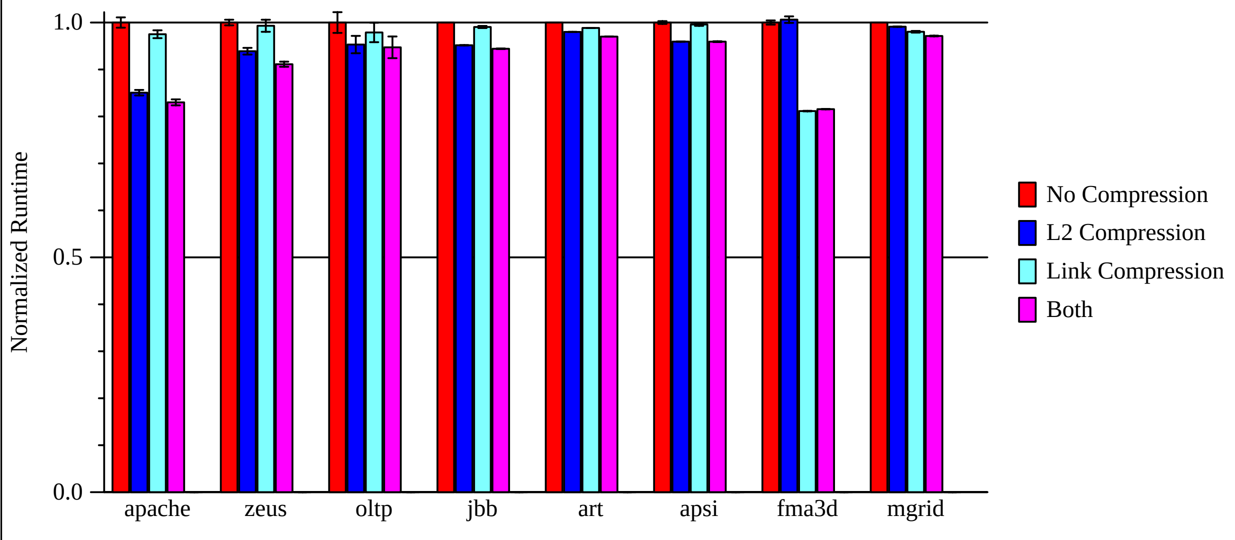


FIGURE 5-5. Normalized runtime for the four compression alternatives (relative to no compression)

bandwidth. We vary a single parameter in each of the following subsections while keeping the remaining parameters constant. In the next few subsections, we demonstrate that:

- Increasing L1 cache size or associativity slightly increases the performance benefit due to compression.
- Increasing L2 cache size has a mixed effect on the performance benefit due to compression, increasing the benefit for some benchmarks while reducing it for others.
- Increasing L2 cache associativity increases the performance benefit due to cache compression.
- Increasing memory latency increases the performance benefit due to cache compression. Decreasing memory latency leads to increasing bandwidth demand, therefore increasing the performance benefit of link compression.
- Increasing available pin bandwidth significantly diminishes the performance gains due to link compression.

5.5.1 L1 Cache Size and Associativity

The effectiveness of L2 cache compression depends on the overhead incurred by decompressing lines on L2 hits. Since the L1 filters requests to the L2, the L1 size impacts this overhead (similar to the uniprocessor case in the previous chapter). As the L1 cache size (or associativity) increases, some references that would have missed in the L1 and hit in the L2 will instead hit in the L1. This leads to a lower number of penalized hits to the compressed L2 cache, thereby reducing the decompression overhead. We illustrate this trade-off for commercial workloads (Figure 5-6) and for SPEComp benchmarks (Figure 5-7). These results assume a 4 MB L2 cache and a fixed L1 access latency.

For our set of benchmarks and configuration parameters, increasing L1 size or associativity has a noticeable effect on the relative benefit of cache compression for zeus, jbb and apsi. For these benchmarks, the

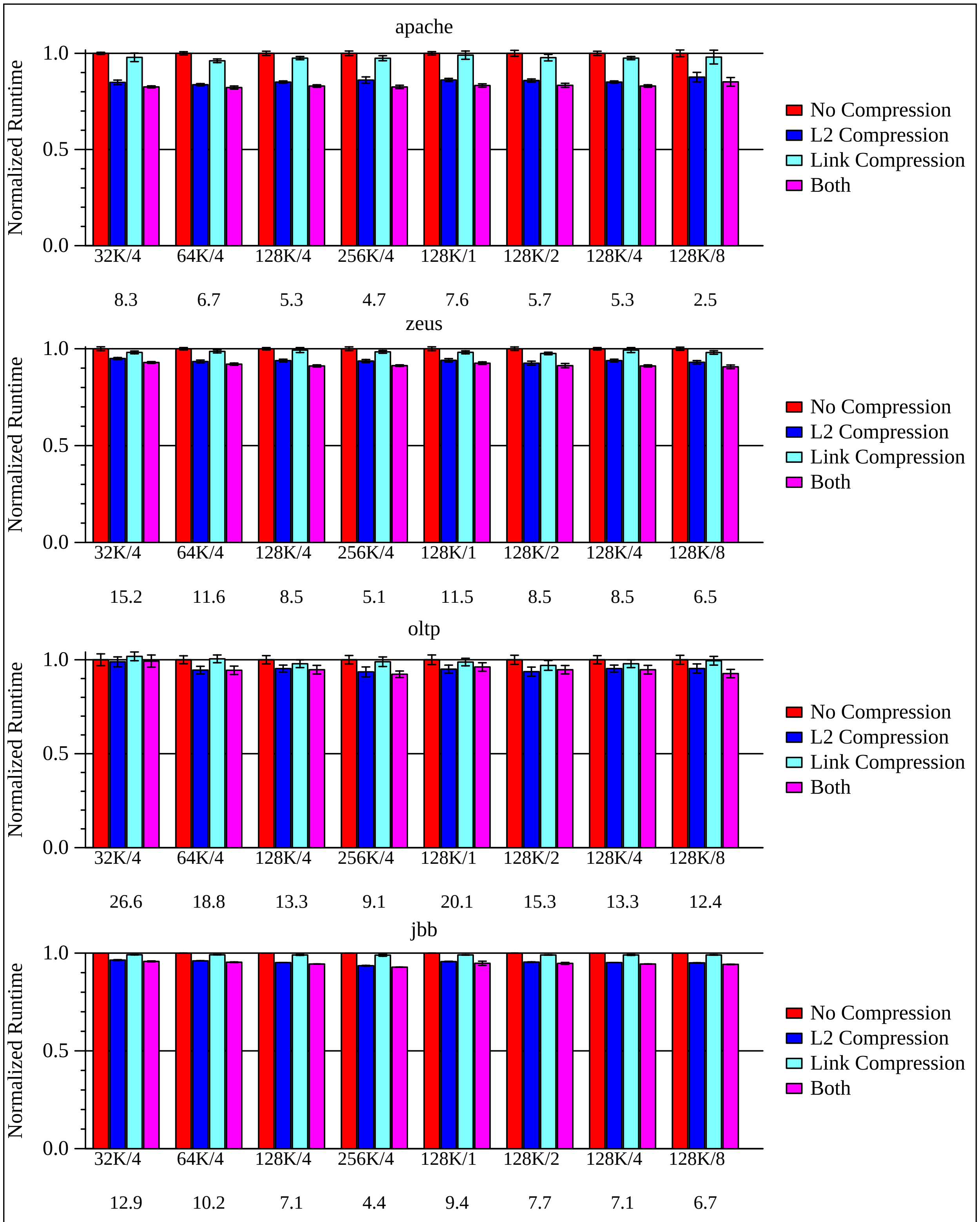


FIGURE 5-6. Sensitivity to L1 cache size and associativity for commercial benchmarks

The number of penalized hits per avoided miss for cache compression is shown at the bottom.

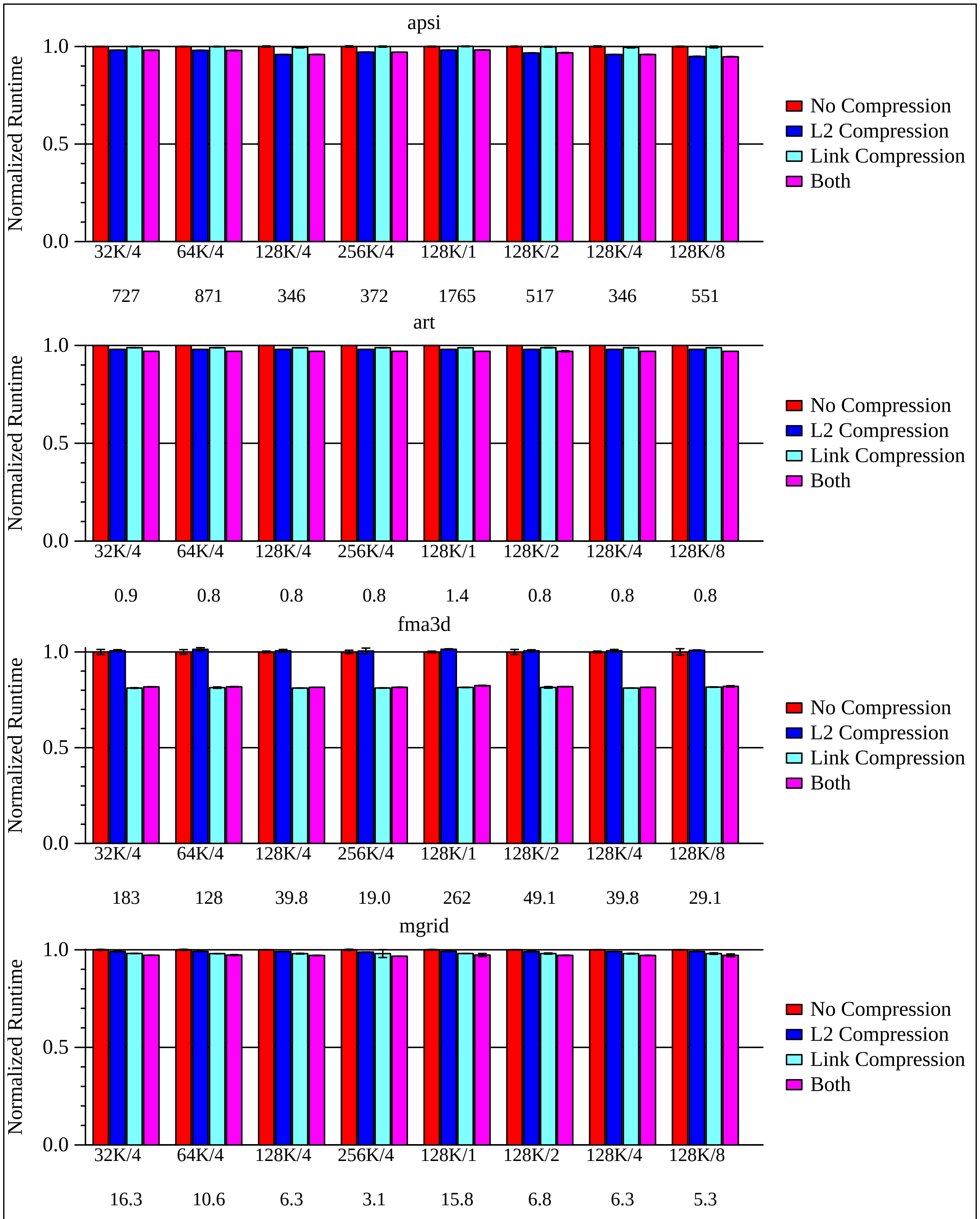


FIGURE 5-7. Sensitivity to L1 cache size and associativity for SPEComp benchmarks

The number of penalized hits per avoided miss for cache compression is shown at the bottom.

speedup due to compression increased slightly when the L1 cache size or associativity increased. For other benchmarks the L1 configuration had little impact on the performance of cache compression. For all benchmarks, the performance of link compression was not affected by changes in the L1 cache configuration.

We also note from Figure 5-7 that apsi shows a performance improvement due to compression despite its high ratio of penalized hits per avoided miss. This is caused by our adaptive compression implementation that adapts to compression only when it helps. For the most part, the ratio is high and our adaptive compression algorithm adapts to *Never*. For some short intervals, however, *Always* is the better policy and adaptive compression implements it.

5.5.2 L2 Cache Size

Cache compression works best when it can increase the effective L2 size enough to hold a workload's critical working set. For chip multiprocessors, working set sizes of workloads are typically much larger than those for uniprocessors since multiple processors share the L2 cache. We show the relative performance of cache and link compression as the L2 cache size changes for commercial workloads (Figure 5-8) and SPEComp benchmarks (Figure 5-9).

For commercial workloads, L2 cache size slightly increases compression's benefits for apache, and slightly reduces these benefits for jbb and oltp, due to the different patterns of change for penalized hits per avoided miss (explained later in this section). For SPEComp benchmarks, L2 cache size had little impact on the relative speedup of cache and link compression. We note that for some benchmarks, increasing the cache size did not have much impact on performance after a certain size. For example, increasing the cache size beyond 8 MB did not significantly affect apsi's performance. On the other hand, many SPEComp benchmarks did not show significant performance improvements for small L2 cache sizes, but their runtimes

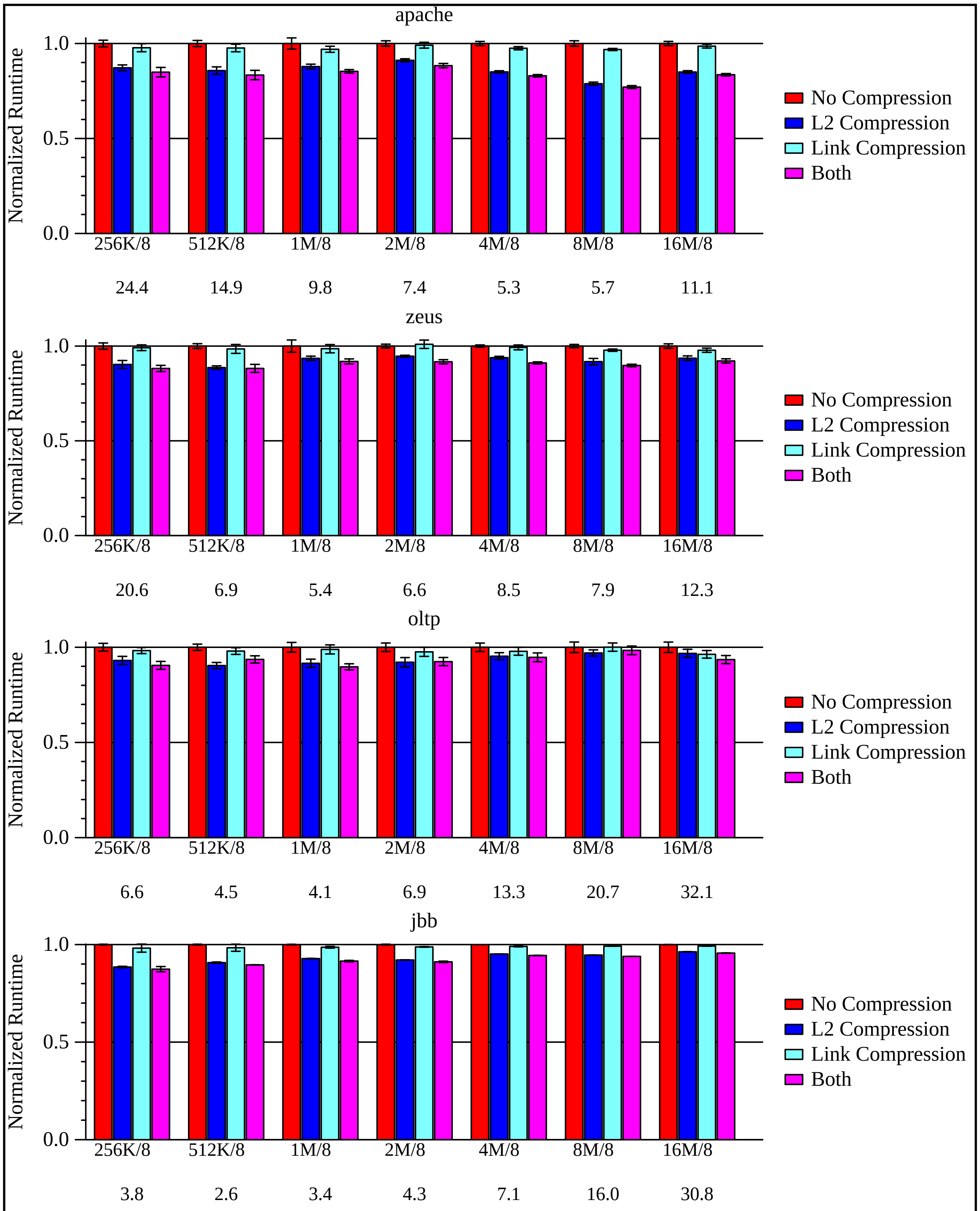


FIGURE 5-8. Sensitivity to L2 cache size for commercial benchmarks

The number of penalized hits per avoided miss for cache compression is shown at the bottom.

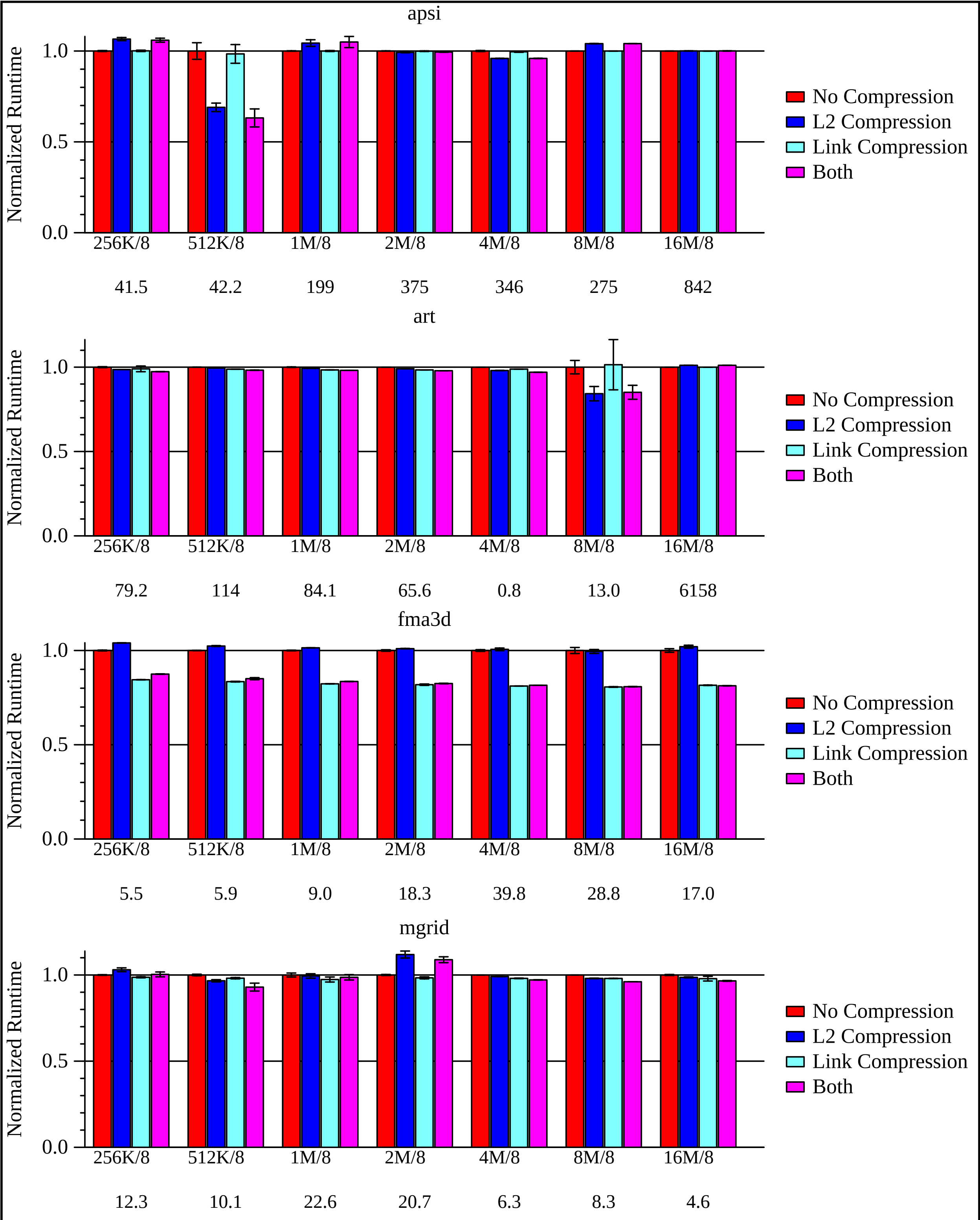


FIGURE 5-9. Sensitivity to L2 cache size for SPEComp benchmarks

The number of penalized hits per avoided miss for cache compression is shown at the bottom.

decreased dramatically once a certain L2 size was reached. For example, art’s absolute performance (not shown in the figure) achieved more than a 2x speedup when the cache size doubled from 4 to 8 MB.

We also note from both figures that the number of penalized hits per avoided miss follows two distinct trends in two sets of benchmarks. For some benchmarks (e.g., apsi and mgrid), the number of penalized hits per avoided miss steadily increases as the L2 cache size increases. For other benchmarks (e.g., apache), the number of penalized hits per avoided miss decreases at the beginning until it reaches a minimum then increases again. The first trend is intuitive, since a bigger cache size tends to increase the number of L2 hits, thus increasing the number of penalized L2 hits. A bigger cache also reduces the number of L2 misses, thereby reducing the number of misses avoidable by compression. Both effects, i.e., increasing the numerator and reducing the denominator, lead to a steady increase in the ratio of penalized hits per avoided miss. The same factors affect the increase in the number of penalized hits per avoided miss for medium and large cache sizes in the second trend (i.e., the decrease then increase in the number of penalized hits per avoided miss). For small cache sizes, however, the number of misses avoided by compression is low since the cache size is much smaller than the benchmarks’ working set sizes. This leads to a higher ratio of penalized hits per avoided miss, since the numerator dominates that ratio with a smaller denominator.

5.5.3 L2 Cache Associativity

Cache compression increases the effective L2 associativity compared to an uncompressed cache. In our experiments, however, we compare against an uncompressed L2 cache that has the same associativity as the maximum associativity of a compressed cache. Therefore our results tend to favor uncompressed caches for benchmarks that are not compressible, since such caches have double the effective associativity as a compressed cache. We illustrate the relative speedup of cache and link compression as the L2 associa-

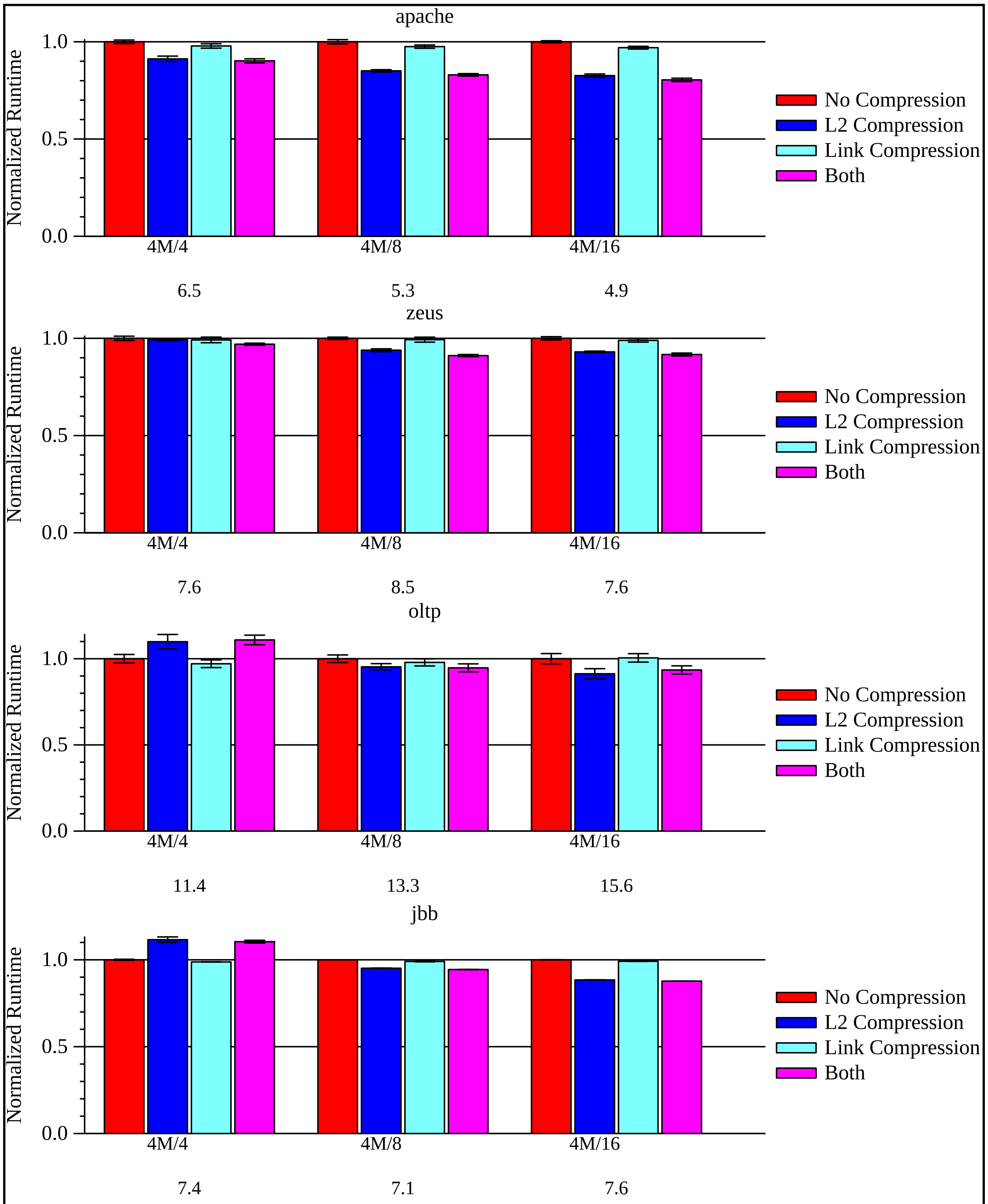


FIGURE 5-10. Sensitivity to L2 cache associativity for commercial benchmarks

The number of penalized hits per avoided miss for cache compression is shown at the bottom.

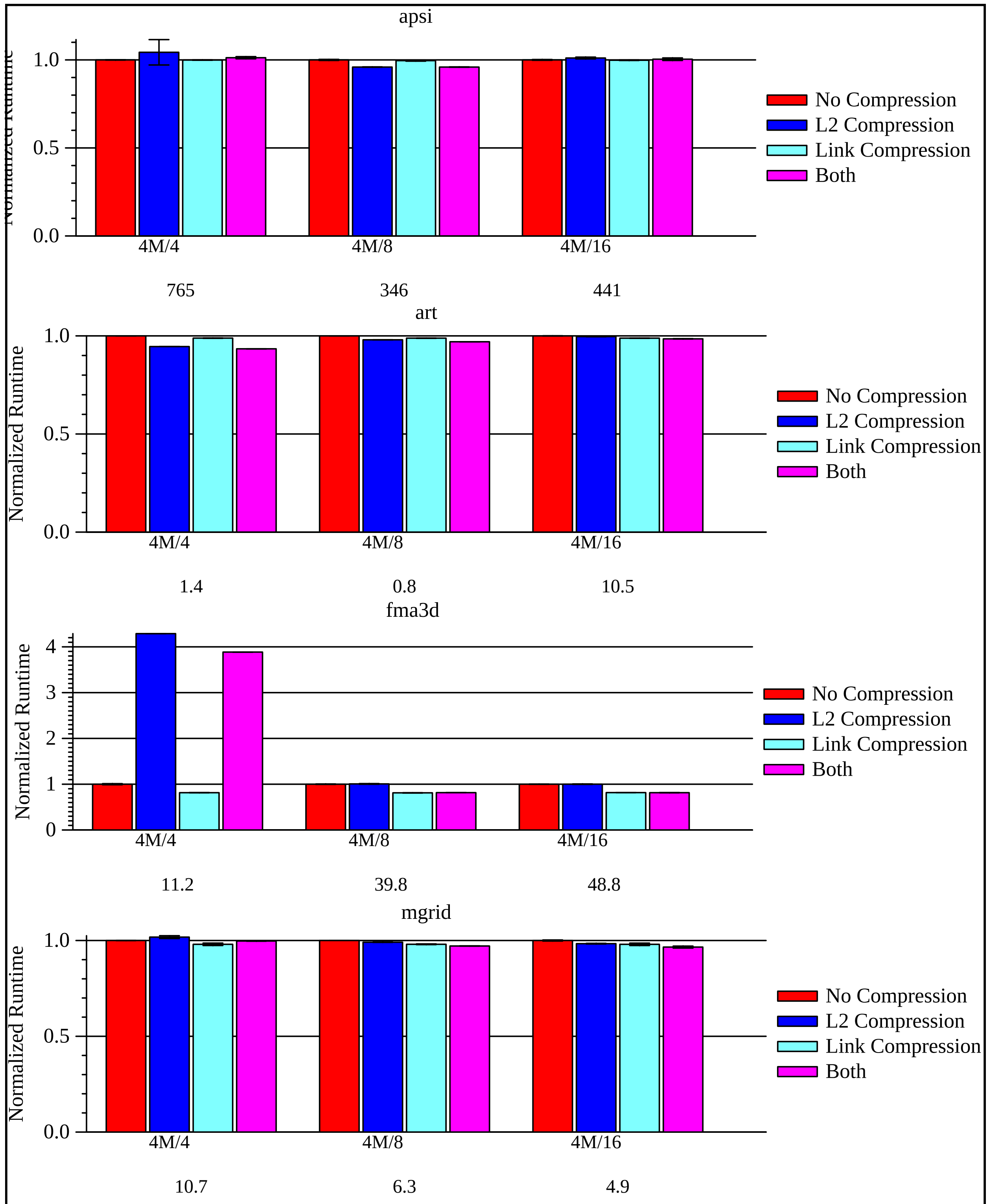


FIGURE 5-11. Sensitivity to L2 cache associativity for SPEComp benchmarks

The number of penalized hits per avoided miss for cache compression is shown at the bottom.

tivity increases for commercial workloads (Figure 5-10) and SPEComp benchmarks (Figure 5-11). We assume the same L2 bank access latency for all configurations.

We note from both figures that L2 associativity has little or no impact on the performance of link compression. For cache compression, however, the speedup due to compression increases when the L2 cache associativity increases. The extreme example in our commercial workloads is jbb, where adaptive cache compression has a 12% slowdown for a 4-way cache but achieves a 13% speedup for a 16-way cache. Fma3d represents the extreme case for SPEComp benchmarks, with more than a 4x slowdown for 4-way caches, while achieving a 23% speedup for a 16-way cache. The reduction in associativity for compressed caches has a significant impact on performance for lower associativity caches (e.g., 4-way), while such effect is greatly diminished for caches with medium and high associativities. We also note that the speedup of cache compression significantly increases for 16-way caches, where apache achieves a speedup of 21% compared to an uncompressed cache.

5.5.4 Memory Latency

We analyze the sensitivity of cache compression to higher and lower memory latencies for commercial workloads (Figure 5-12) and SPEComp benchmarks (Figure 5-13). We varied memory latencies between 200 and 800 cycles. As intuitively expected, the relative speedup of cache compression increases for slower memory, since compression avoids more costly misses. Perhaps less intuitive, however, is the impact of smaller memory latencies on the relative speedup of link compression. Fma3d, mgrid, art and apache show significantly greater speedups due to link compression when the memory latency decreases (e.g., 200 cycles). For example, link compression alone achieves a 26% speedup for fma3d when the memory latency is 200 cycles, compared to a 3% speedup when latency is 800 cycles. This is caused by the fact that a lower memory latency significantly increases the *instructions/cycle* term in Eq 5.1 (Section 5.4.3) without affecting any of the equation’s other terms. This leads to a significant increase in pin bandwidth

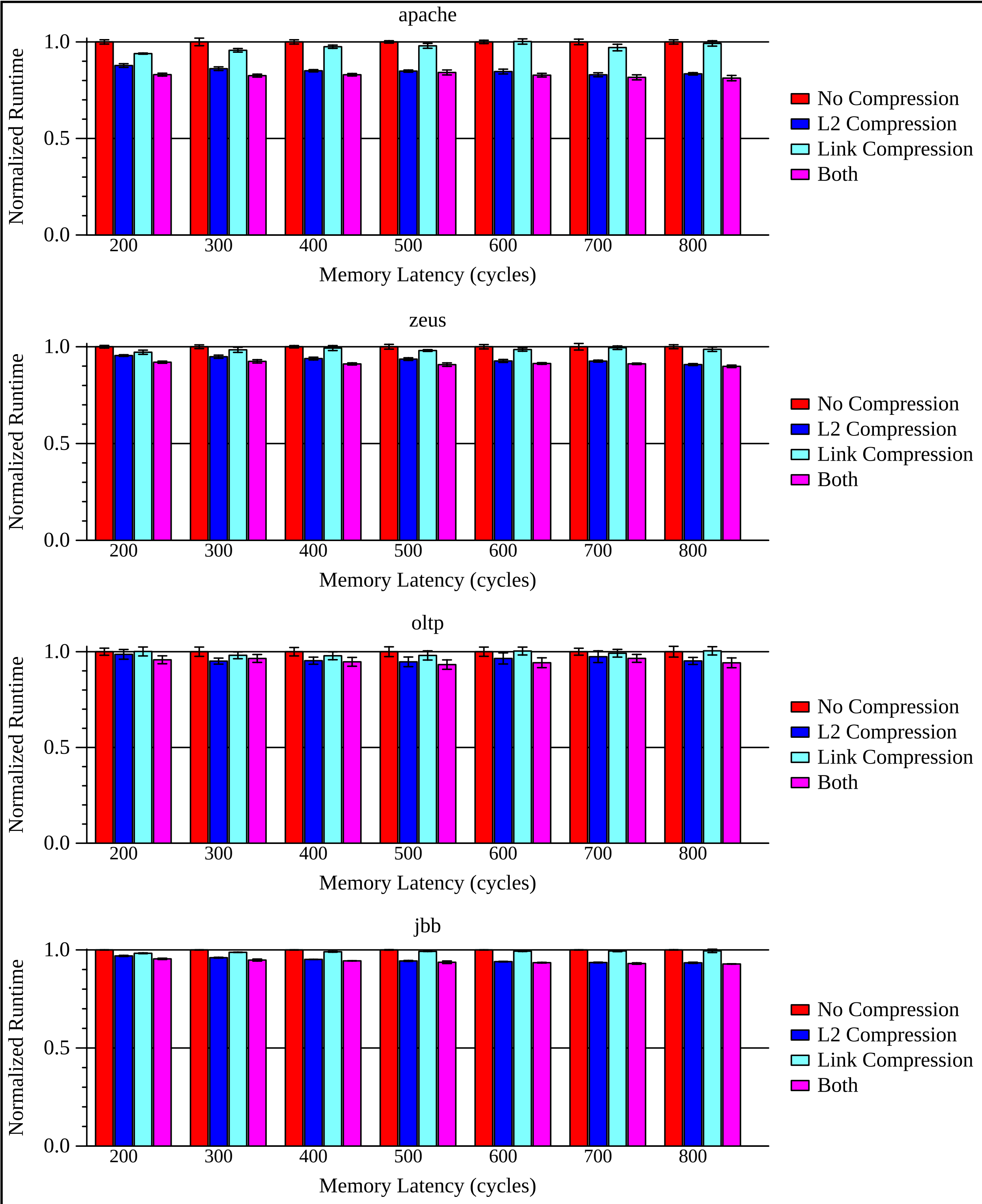


FIGURE 5-12. Sensitivity to memory latency for commercial benchmarks

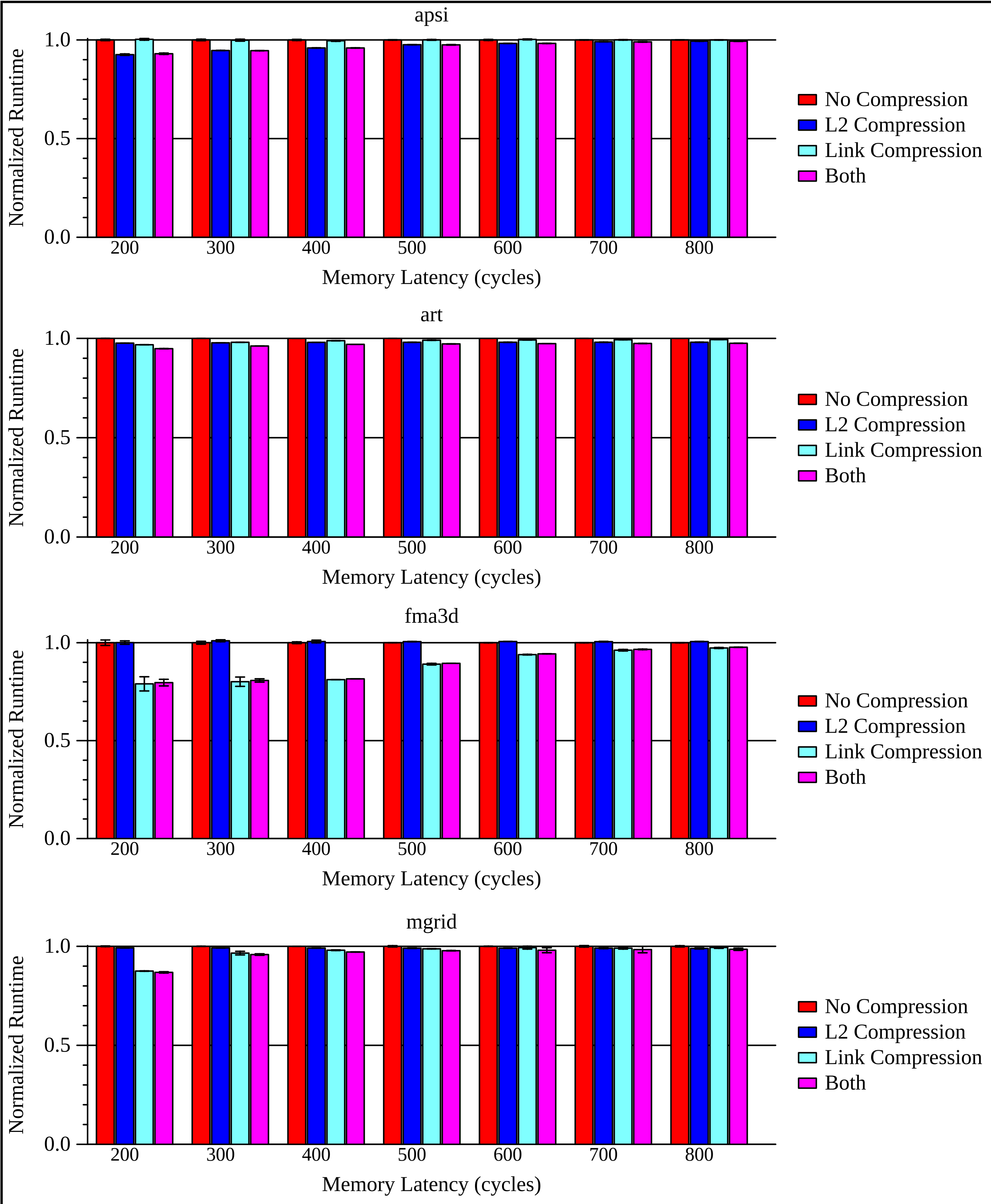


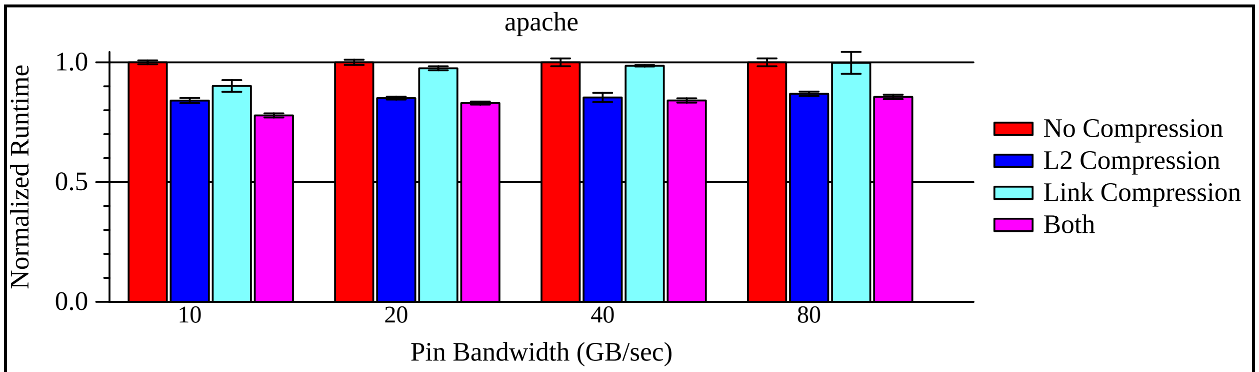
FIGURE 5-13. Sensitivity to memory latency for SPECcomp benchmarks

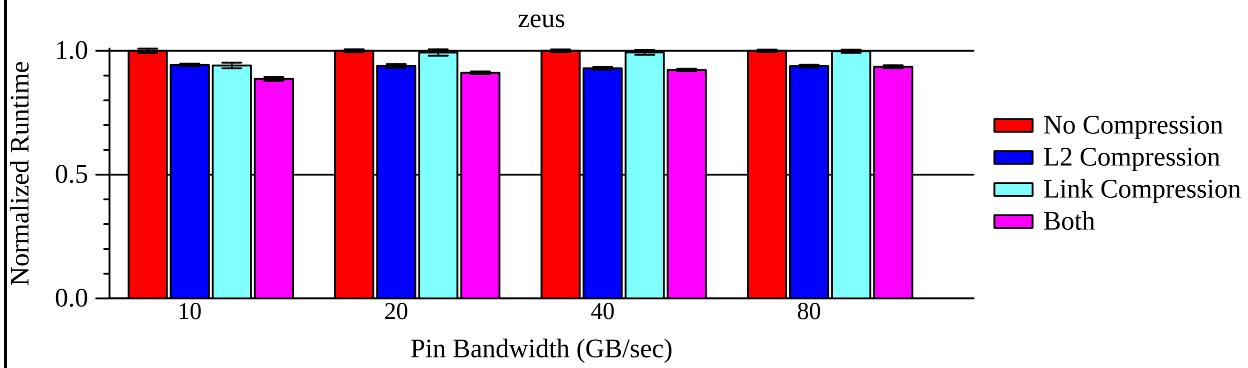
demand, causing more benchmarks to be bandwidth-limited. Since link compression helps offset the increase in pin bandwidth demand, its impact on performance increases.

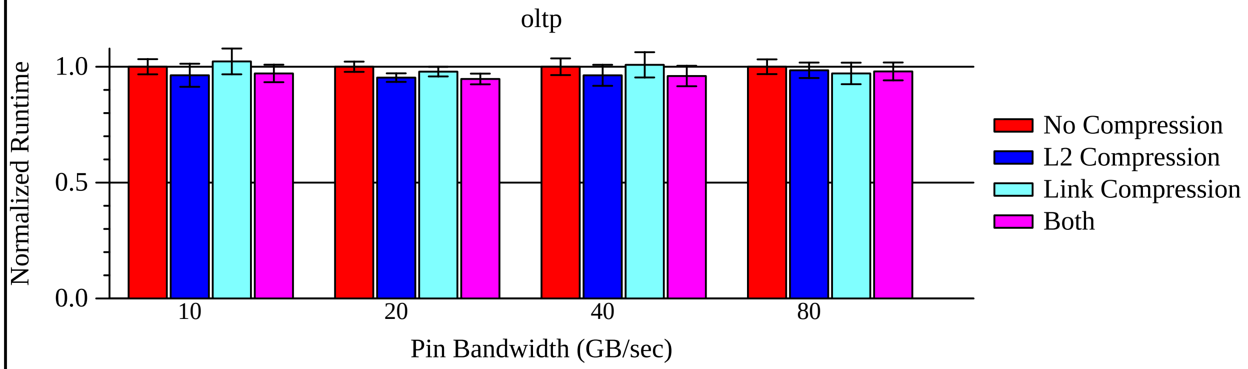
5.5.5 Pin Bandwidth

Link compression's and (in part) cache compression's performance gains are caused by their impact on pin bandwidth demand. Link compression directly decreases pin bandwidth demand, while cache compression does so indirectly by decreasing off-chip misses (Section 5.4.3). In this section, we analyze the impact of pin bandwidth changes on the relative speedups of cache and link compression for commercial workloads (Figure 5-14) and SPEComp benchmarks (Figure 5-15). We simulated systems with pin bandwidth ranging from 10 GB/sec. to 80 GB/sec.

The speedup of cache compression did not change significantly with pin bandwidth, since cache compression affects bandwidth by changing two terms in Eq. 5.1 in opposite directions, as we discussed in Section 5.4.3. Overall, cache compression achieved almost the same speedup regardless of pin bandwidth. As expected, the impact of pin bandwidth on the speedup of link compression is more obvious. For 10 GB/sec., link compression—and therefore the combination of cache and link compression—achieves significant speedup for many benchmarks, up to a 26% speedup for fma3d, 22% for mgrid, and 11% for apache. However, such speedup almost completely disappears for the 40 and 80 GB/sec. configurations. This trend demonstrates that the speedup of link compression is closely correlated to how close the benchmark is to the pin bandwidth saturation point. When the bandwidth available is much higher than the pin bandwidth demand, the impact of link compression is significantly diminished. However, such significant increases in available pin bandwidth are unlikely to happen for larger chip multiprocessors (as we discussed in Section 5.2.1) unless optical interconnect technologies develop and mature quickly. The implementation of prefetching can also significantly increase bandwidth demand, as we discuss in the next chapter. We also







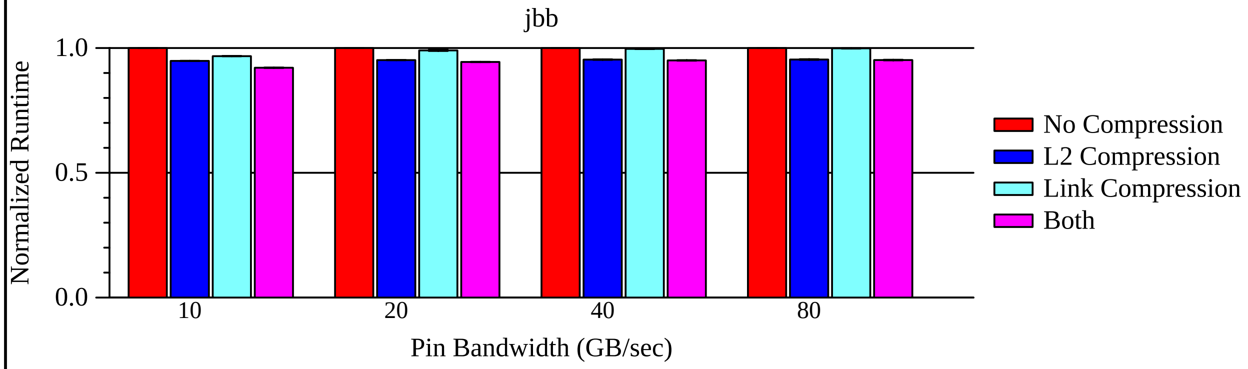


FIGURE 5-14. Sensitivity to pin bandwidth for commercial benchmarks

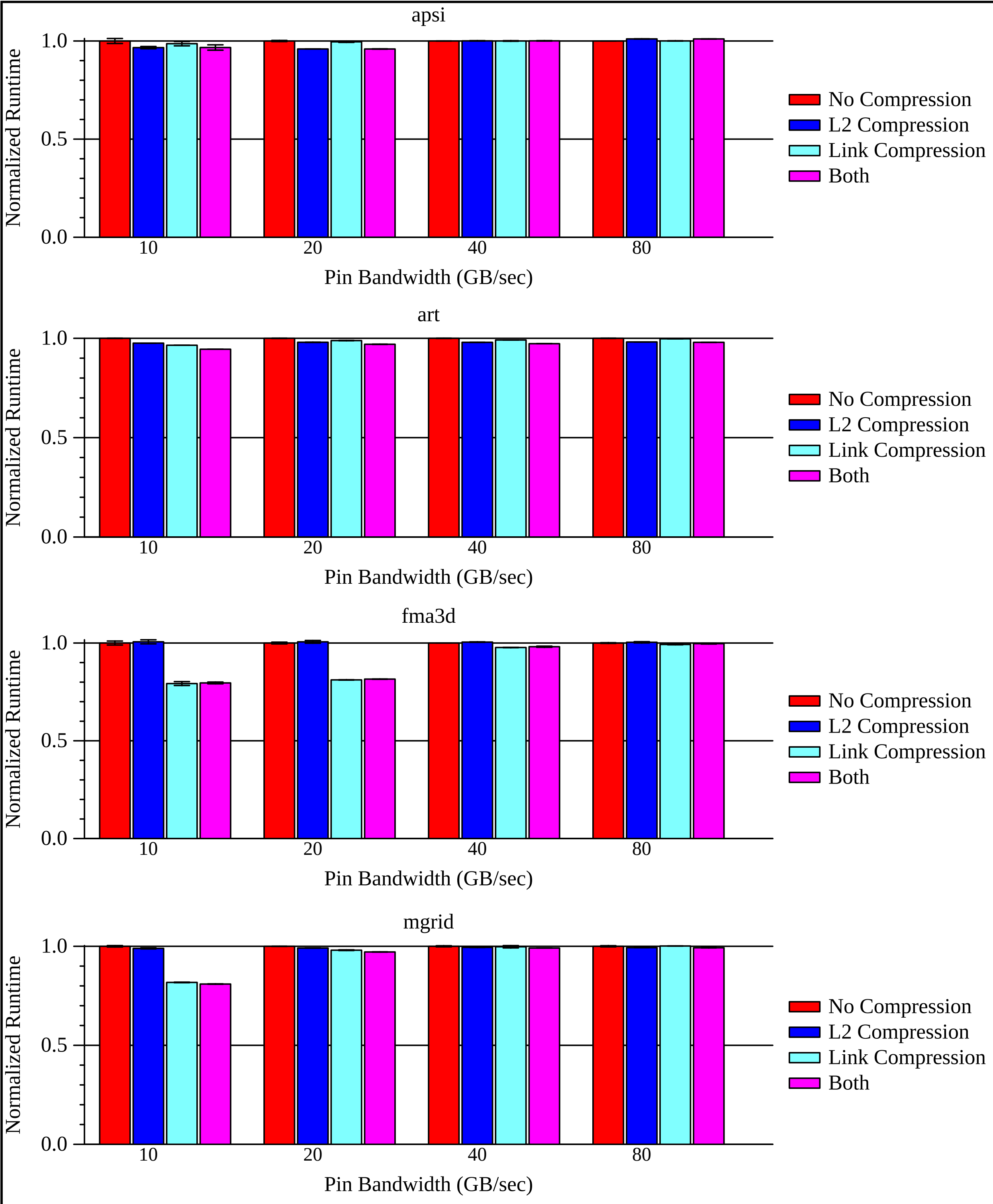


FIGURE 5-15. Sensitivity to pin bandwidth for SPEComp benchmarks

note that link compression can be a cheaper alternative with lower power consumption compared to other techniques that can increase pin bandwidth.

5.6 Summary

Chip multiprocessor design requires balancing three critical resources: number of processors, on-chip cache size, and off-chip pin bandwidth. In this chapter, we explored using cache and interconnect (link) compression to effectively increase cache size and pin bandwidth resources and ultimately overall system throughput. Cache compression increases the effective capacity of the shared cache (by 36-80% in our commercial benchmarks), thus reducing off-chip misses and improving performance. Link compression increases the effective off-chip communication bandwidth for most workloads by 17-41%, reducing possible contention. However, link compression did not have a significant impact on performance except for one benchmark (fma3d). We showed that both cache and link compression combine to improve performance of commercial benchmarks, and that link compression can achieve significant speedups for bandwidth-limited SPEComp benchmarks.

Many current CMP systems implement hardware prefetching. On the one hand, hardware prefetching can significantly increase pin bandwidth demand, thereby increasing compression's relative benefits. On the other hand, prefetching can also avoid misses that could be avoided by compression, thereby reducing compression's benefits. We study the interactions between compression and hardware stride-based prefetching in the next Chapter.

Chapter 6

Interactions Between Compression and Hardware Prefetching

In the previous chapter, we showed that compression is an appealing alternative for Chip Multiprocessors that can avoid many off-chip misses and reduce off-chip pin bandwidth demand. Many current CMP systems implement some form of hardware-directed prefetching to help tolerate increasingly long memory latencies [58, 68]. For uniprocessors, stride-based hardware prefetching improves performance by 2-74% for the commercial workloads used in this dissertation. Unfortunately, prefetching also increases a workload's working set size and memory traffic, therefore increasing demand for caches and pin bandwidth. Since processors share both of these critical resources in a CMP, the benefit of prefetching decreases dramatically. On a 16-processor CMP with the same uniprocessor cache size and pin bandwidth, stride-based prefetching can degrade performance by up to 35% (Section 6.1). In this chapter, we show that compression and prefetching interact positively, leading to a combined speedup that is greater than the product of the speedup of prefetching alone and compression alone.

After motivating how compression can interact positively with prefetching (Section 6.1), we define the terminology we use to study the interactions between any two hardware schemes (Section 6.2). We discuss our simulation parameters and prefetching implementation (Section 6.3). We study the interactions between compression and L2 prefetching (Section 6.4), L1 prefetching (Section 6.5) and both (Section 6.6). We analyze the sensitivity of these results to available pin bandwidth (Section 6.7) and CMP configurations (Section 6.8). We then discuss some related work (Section 6.9) and summarize our results (Section 6.10).

In this chapter, we make the following main contributions:

- We present quantitative evidence that stride-based prefetching improves performance of CMPs far less than it does for uniprocessors, even degrading performance for some workload and system configurations.
- We show that compression and prefetching interact in strongly positive ways, resulting in a combined performance improvement of 10-51% for seven of our eight benchmarks on an 8-processor CMP.
- We show that the combined improvement from both prefetching and compression significantly exceeds the product of individual improvements for most benchmarks, with positive interactions of 13-22% for half of our benchmarks.
- We analyze different factors that cause positive and negative interactions between compression and L1 prefetching or L2 prefetching.

6.1 Motivation

Compression is an appealing alternative for CMP systems that helps increase the effective on-chip cache capacity and effective off-chip pin bandwidth for compressible benchmarks. Cache compression increases the effective cache capacity, thereby reducing some off-chip misses. Link compression (directly) and cache compression (indirectly) increase the effective off-chip bandwidth, potentially reducing contention for pin bandwidth. Both cache and link compression combine to improve performance for most benchmarks at a small hardware cost. However, decompression overheads increase the L2 cache hit latency, which reduces some of compression's performance gains. Overall, we showed in the previous chapter that compression can improve performance for many commercial and some scientific benchmarks.

Many current systems implement hardware-based prefetching to address the increasing gap between processor and memory speeds [58, 68, 117]. Many such implementations are variations of hardware-directed stride-based prefetching [26]. Hardware prefetching improves CMP performance by hiding L2 access

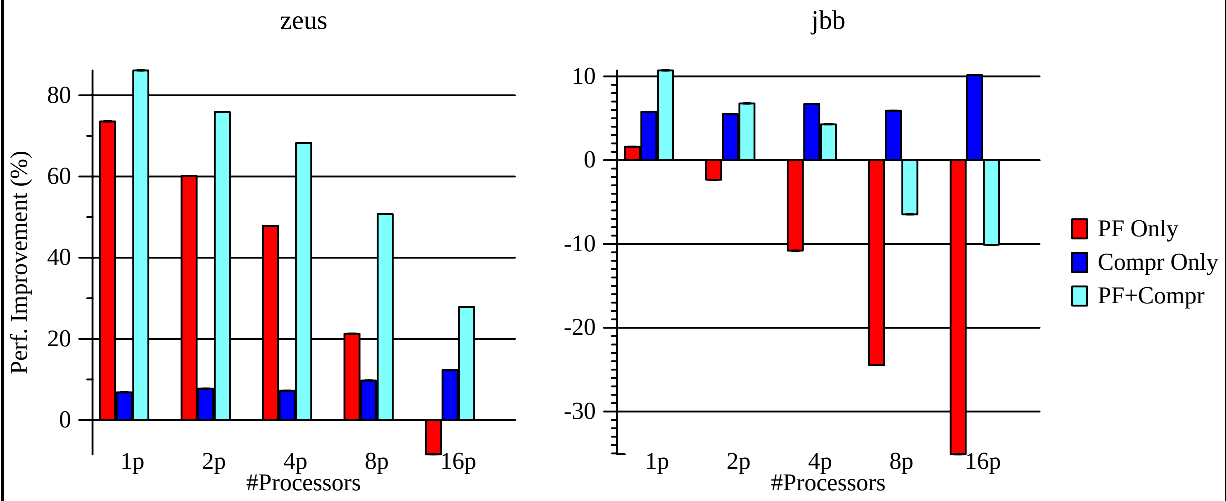


FIGURE 6-1. Performance improvement (%) for two commercial benchmarks for different uniprocessor and CMP configurations

Performance improvement is shown compared to a base case of no compression or prefetching. The three bars represent performance improvement due to stride-based prefetching alone, cache and link compression alone, and both prefetching and compression. All configurations have a 4 MB (shared) L2 cache and a 20 GB/sec. available off-chip bandwidth.

latency (L1 prefetching) and avoiding some off-chip misses or tolerating memory latency (L2 prefetching). Unfortunately, hardware stride-based prefetching can significantly increase demand on cache banks and off-chip pin bandwidth. If hardware prefetching's accuracy is low, it can increase a workload's working set size and therefore increase cache pollution. CMPs exacerbate both problems since more processors share cache and pin bandwidth resources.

For CMP systems that implement prefetching, an important question is whether prefetching achieves most of the benefits of compression, which makes implementing compression less appealing. Alternatively, compression and prefetching can help offset each other's disadvantages, leading to a combined design that outperforms either scheme alone. Figure 6-1[1] shows that for a uniprocessor, hardware stride-based

1. We discuss this figure and other benchmarks in more detail in Section 6.8.

prefetching achieves 74% and 2% performance improvements for zeus and jbb, respectively. For a 16-processor CMP with the same cache size and pin bandwidth, however, stride-based prefetching degrades performance by 8% and 35%, respectively. For both benchmarks, the benefit of stride-based prefetching decreases as the number of processor cores grows, eventually degrading performance.

In this chapter, we show that compression and prefetching interact positively, leading to a combined speedup that equals or exceeds the product of the two individual speedups for most of our benchmarks. A 16-processor CMP with both prefetching and compression has a speedup of 28% for zeus, and only a 10% slowdown for jbb (Figure 6-1). This is in contrast with slowdowns of 8% and 35%, respectively, with prefetching alone. Such combined performance reflect positive interactions (i.e., greater than expected performance improvements) between compression and prefetching of 24% and 26%, respectively. We present terminology to quantify positive and negative interactions in the next section.

6.2 Terminology

In order to understand the interactions between prefetching and compression, we use the following terminology derived from Fields, et al.’s interaction cost definition [44]. For an architectural enhancement A (e.g., L2 compression), we define its speedup for a certain workload, *Speedup(A)*, as the workload’s runtime on a base system (without A) divided by the workload’s runtime on the same system with enhancement A. For two architectural enhancements A and B (e.g., link compression and L2 prefetching), we define the combined speedup of the base system with both enhancements as:

$$Speedup(A, B) = Speedup(A) \times Speedup(B) \times (1 + Interaction(A, B)) \tag{6.1}$$

When *Interaction(A,B)* is positive, the speedup of the two enhancements together exceeds the product of individual speedups. We call this case a *positive interaction* between A and B. When *Interaction(A,B)* is negative, the speedup of the combined system is less than that of the product of individual speedups, and we call this case a *negative interaction* between A and B. We use these definitions to quantify the interactions between compression and prefetching combinations.

6.3 Evaluation

We evaluated the interactions between compression and prefetching using the same set of benchmarks of Chapter 5, and using the same base system configuration. We next describe the hardware stride-based prefetchers we use to study their interactions with compression.

6.3.1 Strided Prefetching

Hardware-directed stride-based prefetchers make use of repeatable memory access patterns to avoid some cache misses and tolerate cache miss latency [26, 96]. Current hardware prefetchers [58, 116, 117] observe a unit or a fixed stride between two cache misses, then verify the stride using subsequent misses. Once the prefetcher reaches a threshold of strided misses, it issues a series of prefetches to the next level in the memory hierarchy to reduce or eliminate miss latency.

We implemented a strided L1 and L2 prefetching strategy to study the interactions between L2 compression and hardware prefetching. We based our prefetching scheme on the IBM Power 4 implementation [116, 117] with some minor modifications. Each processor has three separate prefetchers for the L1I, L1D and L2 caches. Each prefetcher contains three separate 32-entry filter tables, and an 8-entry stream table. The three filter tables detect positive unit stride, negative unit stride, and non-unit stride access patterns, respectively. Once a filter table entry detects four fixed-stride misses, the prefetcher allocates the miss stream to an entry in its 8-entry stream table (i.e., each prefetcher can prefetch lines from up to 8 streams detected using its 96 filter table entries). Upon allocation, the L1I or L1D prefetcher launches 6 consecutive prefetches along the stream to compensate for the L1 to L2 latency, while the L2 prefetcher launches 25 consecutive prefetches to memory to compensate for the memory latency. Each prefetcher issues prefetches for both loads and stores because our target system uses an L1 write-allocate protocol supporting sequential consistency (unlike Power 4) [17]. We also model separate L2 prefetchers per processor rather than a single shared prefetcher to reduce stream interference [17], and we allow L1 prefetches to trigger L2 prefetches. We evaluate hardware-based strided prefetching and its interactions with cache and link compression in the next few sections.

6.3.2 Hardware Stride-Based Prefetching Characteristics

Table 6-1 presents the characteristics of hardware stride-based prefetching for the L1I, L1D and L2 prefetchers using the following metrics:

$$PrefetchRate = \textit{Prefetches per thousand instructions} = \frac{TotalPrefetches \times 1000}{TotalInstructions} \tag{6.2}$$

$$Coverage(\%) = \frac{PrefetchHits}{PrefetchHits + DemandMisses} \times 100\% \tag{6.3}$$

TABLE 6-1. Prefetching Properties for Different Benchmarks

Benchmark	L1 I Cache			L1D Cache			L2 Cache		
	Pf rate	Coverage	Accuracy	Pf rate	Coverage	Accuracy	Pf rate	Coverage	Accuracy
apache	4.9	16.4%	42.0%	6.1	8.8%	55.5%	10.5	37.7%	57.9%
zeus	7.1	14.5%	38.9%	5.5	17.7%	79.2%	8.2	44.4%	56.0%
oltp	13.5	20.9%	44.8%	2.0	6.6%	58.0%	2.4	26.4%	41.5%
jbb	1.8	24.6%	49.6%	4.2	23.1%	60.3%	5.5	34.2%	32.4%
art	0.05	9.4%	24.1%	56.3	30.9%	81.3%	49.7	56.0%	85.0%
apsi	0.04	15.7%	30.7%	8.5	25.5%	96.9%	4.6	95.8%	97.6%
fma3d	0.06	7.5%	14.4%	7.3	27.5%	80.9%	8.8	44.6%	73.5%
mgrid	0.06	15.5%	26.6%	8.4	80.2%	94.2%	6.2	89.9%	81.9%

Where a prefetch hit is defined as the first reference to a prefetched block, excluding partial hits where prefetched blocks are still in flight.

$$Accuracy(\%) = \text{Percent of Accurate Prefetches} = \frac{PrefetchHits}{TotalPrefetches} \times 100\% \qquad (6.4)$$

Table 6-1 shows the different prefetching properties of commercial and SPEComp benchmarks on an 8-processor CMP[2]. Commercial benchmarks issue many more L1 instruction prefetches (up to 13.5 per 1000 instructions for oltp), while the number of instruction prefetches for SPEComp benchmarks is negligible. L1 instructions' prefetching accuracy is not high since prefetch streams are initialized after recognizing four fixed-stride cache line accesses, typically larger than most basic blocks. On the other hand, the number of prefetches and the accuracy of L1 data and L2 prefetching is much higher for SPEComp benchmarks, which is expected since their data access patterns are more predictable compared to commercial benchmarks.

In the next two sections, we study the interactions between compression and L2 prefetching alone, and its interactions with L1 prefetching alone. We separate the effects of both types of prefetching to demonstrate the different positive and negative interaction factors between compression and prefetching. In Section 6.6, we show the combined effect of both L1 and L2 prefetching and its interactions with compression.

6.4 Interactions Between Compression and L2 Prefetching

In this section, we study the interactions between compression and L2 prefetching. We simulated different combinations of cache/link compression and L2 prefetching for our eight benchmarks. We present results from configurations that implement L2 prefetching—and not L1 prefetching—to isolate the impact of L2

2. Uniprocessors have higher L1D and L2 coverage for commercial benchmarks since they have less thread contention. Other prefetching properties for commercial benchmarks do not differ significantly from those of Table 6-1. We did not study uniprocessor versions of SPEComp benchmarks in this dissertation.

prefetching alone. In the next few subsections, we show the following positive and negative interactions between compression and L2 prefetching:

- L2 prefetching significantly increases pin bandwidth demand for most benchmarks. Compression (mostly link compression) helps alleviate such increase in pin bandwidth demand, a positive interaction between compression and L2 prefetching.
- L2 prefetching significantly increases many benchmarks' working set size (or cache footprint). Cache compression alleviates this increase by increasing the effective cache size, a positive interaction.
- Cache compression and L2 prefetching avoid some of the same L2 misses. These common avoided misses cannot be counted towards improving the combined performance of prefetching and compression. According to our terminology in Section 6.2, this constitutes a negative interaction between compression and L2 prefetching.

6.4.1 Bandwidth Demand

Figure 6-2 shows the bandwidth demand of prefetching and compression combinations, normalized to the case of no compression or prefetching. L2 prefetching alone increases off-chip bandwidth demand for our benchmarks by 17-178%. Combining prefetching with L2 and link compression achieves a significant off-chip bandwidth demand reduction across all benchmarks except apsi, a *positive interaction* between the two techniques. For example, while zeus has an 82% bandwidth demand increase due to L2 prefetching, bandwidth demand increases by only 3% when combining it with L2 and link compression (compared to the base case of no compression or prefetching). Apache's 64% bandwidth demand increase due to prefetching turns into a 7% reduction when both compression and prefetching are combined. Compression in apsi does not achieve a significant reduction in bandwidth over L2 prefetching (only 3%) since its compression ratio is low (Chapter 5).

6.4.2 Classification of L2 Misses

Figure 6-3 presents a classification of L2 misses according to whether they are avoidable by L2 prefetching only, L2 compression only, both, or neither. The figure shows six classes of accesses (from the bottom up): misses that could not be avoided by either L2 prefetching or L2 compression, misses that could be avoided by L2 compression and not L2 prefetching, misses that could be avoided by L2 prefetching and not L2 compression, misses that could be avoided by either L2 compression or L2 prefetching, extra L2 prefetches that could not be avoided by L2 compression, and extra L2 prefetches that could be avoided by L2 compression. The 100% line represents the total misses in the case of no compression or prefetching. The figure presents approximate data that we obtained from comparing cache miss profiles across simulations of different configurations, and using set theory and the theory of inclusion and exclusion to obtain cardinalities of different sets of accesses.

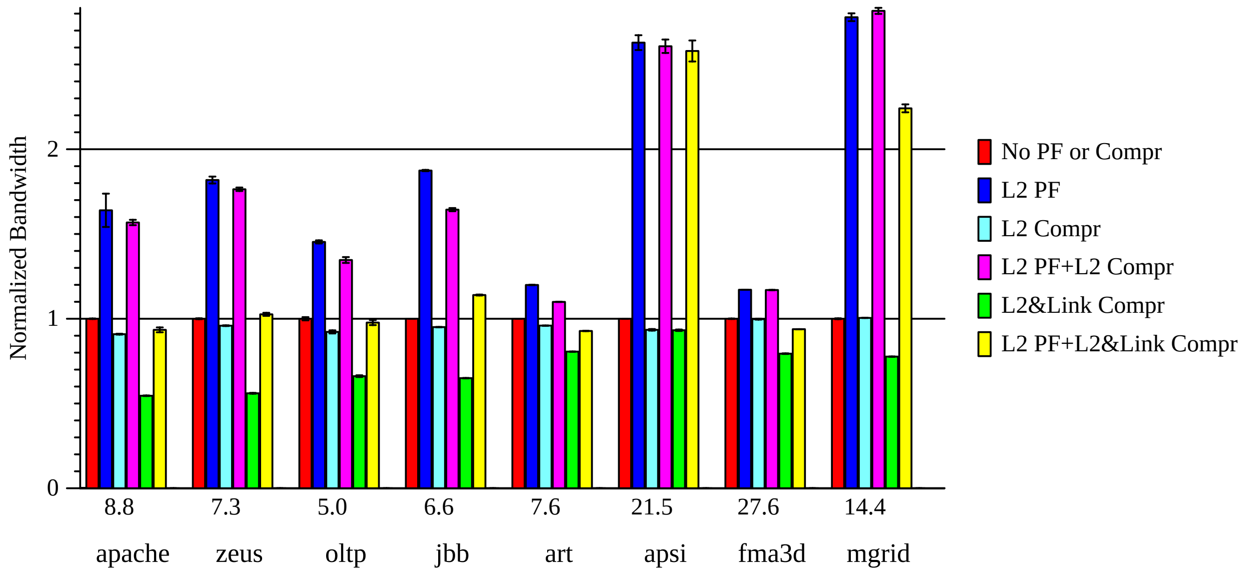


FIGURE 6-2. Normalized off-chip bandwidth demand for L2 prefetching and compression combinations

Bandwidth demand is obtained as the bandwidth utilized with an infinite available pin bandwidth. For each benchmark, bandwidth demand is normalized to the bandwidth demand with no prefetching or compression, shown at the bottom (in GB/sec.).

Figure 6-3 demonstrates that L2 prefetching succeeds in avoiding many misses in SPEComp benchmarks, while L2 compression is not as successful. For commercial benchmarks, both prefetching and compression avoid some L2 misses. We also note the following two sources of interaction between compression and prefetching:

Negative Interaction: Misses avoided by Both. Figure 6-3 shows that there is an intersection between the sets of misses avoided by L2 compression and those that could be avoided by L2 prefetching (i.e., misses that could be avoided by either technique). This intersection is a negative interaction factor between the two techniques, since they can be accounted for once when computing *Speedup(Compression, Prefetching)* in Eq. 6.4. However, this set only represents a small fraction of the total number of misses (8% for apache,

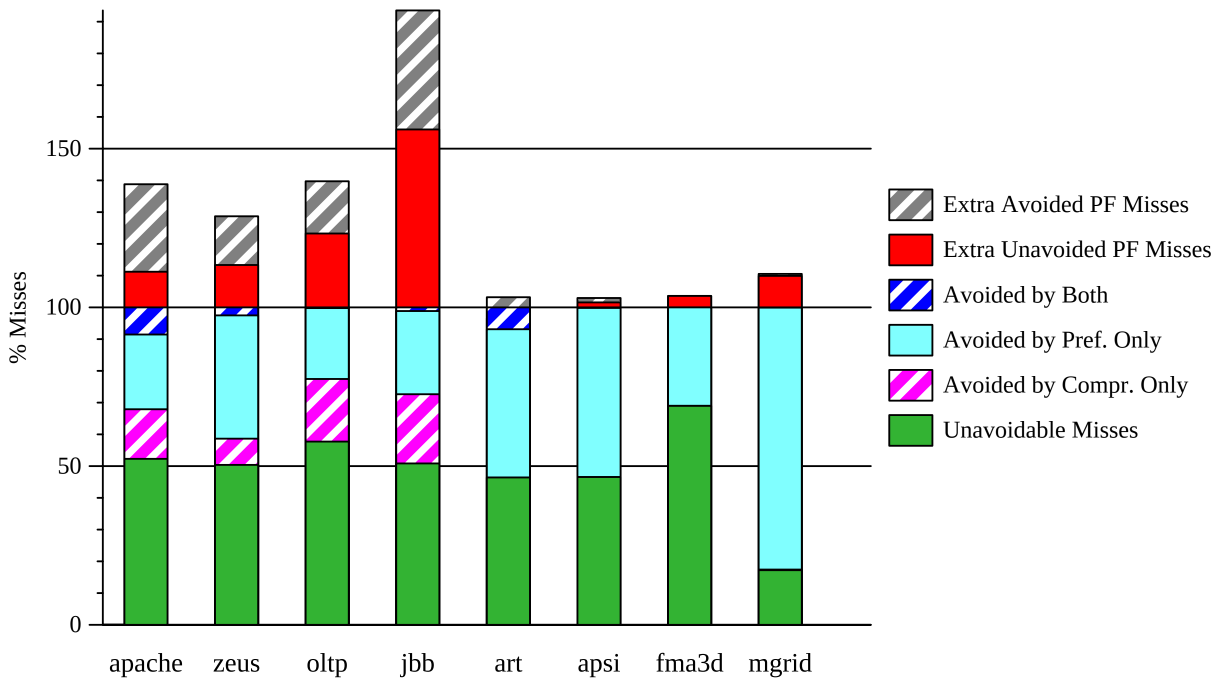


FIGURE 6-3. Breakdown of L2 cache misses and prefetches

The figure shows for each benchmark (from the bottom up): unavoidable misses, misses avoided by compression and not prefetching, misses avoided by prefetching and not compression, and misses avoided by both compression and prefetching. The 100% line represent the total misses for no compression or prefetching. We show extra prefetches (both avoided and unavoided by compression) above the 100% line.

7% for art, and 3% or less for all other benchmarks). We attribute this small intersection to the fact that L2 compression and L2 prefetching target different sets of misses: While L2 compression mainly targets conflict and capacity misses, L2 strided prefetching targets misses that follow a strided pattern. The two sets of misses, while partially overlapping, are largely orthogonal.

Positive Interaction: Prefetching Misses Avoided by Compression. Figure 6-3 also shows that compression avoids some prefetches when their data can fit in a compressed cache, removing some of the additional bandwidth due to prefetching. While this fraction is negligible for SPEComp benchmarks where prefetching is more accurate, it is significant for commercial workloads. Prefetching, in effect, increases a workload's working set size (or cache footprint), and compression helps by increasing the effective cache size to tolerate that increase in cache footprint.

6.4.3 Performance

Figure 6-4 shows normalized runtimes for our eight benchmarks for different combinations of compression and L2 prefetching, relative to the base case of no compression or prefetching. Table 6-1 presents speedups and interaction coefficients between different combinations. L2 prefetching alone speeds up all benchmarks (except jbb and fma3d) by 1-28%. These speedups are higher for SPEComp benchmarks compared to the commercial benchmarks except for zeus. Zeus shows the highest performance improvement of all commercial benchmarks since L2 prefetching avoids a larger percentage of all L2 misses, as we showed in Section 6.4.2.

Jbb suffers from a 16% slowdown because its L2 prefetching accuracy is much lower (at 32.4%, Table 6-1) than all other benchmarks, which leads to many additional misses since prefetches replace useful lines in the L2 cache. Fma3d with no compression or prefetching is already bandwidth limited for our 20 GB/sec. pin bandwidth base configuration as we showed in Chapter 5, and prefetching increases bandwidth demand leading to a performance slowdown.

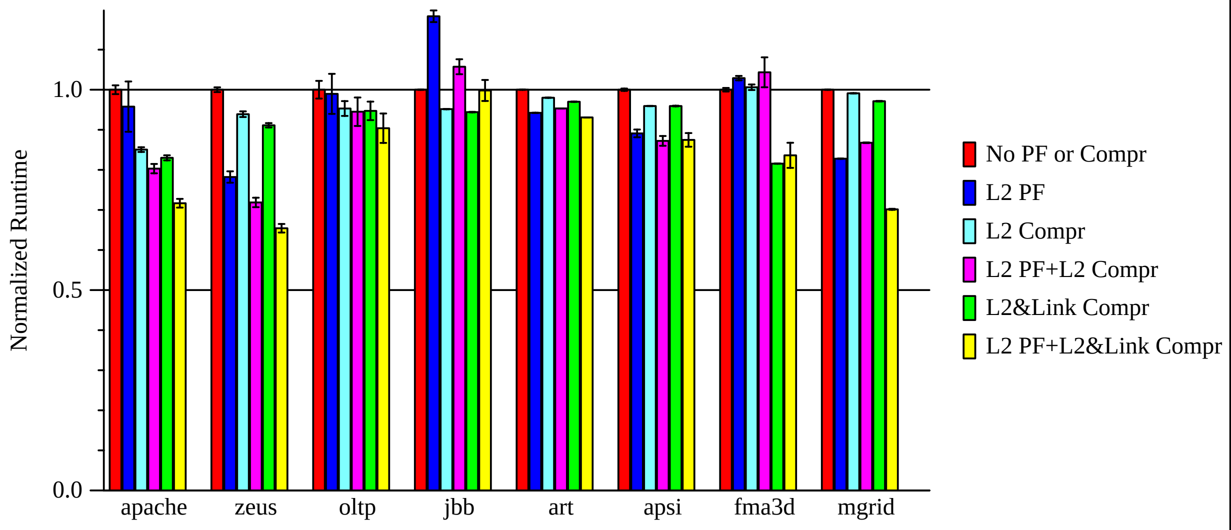


FIGURE 6-4. Performance of combinations of L2 Prefetching and compression

Performance is normalized to the case of no prefetching or compression.

TABLE 6-1. Speedups and Interactions between L2 Prefetching and Compression

Benchmark	**apache**	**zeus**	**oltp**	**jbb**	**art**	**apsi**	**fma3d**	**mgrid**
Speedup (L2 PF)	4.4%	27.9%	1.0%	-15.5%	6.1%	12.2%	-2.8%	20.8%
Speedup (L2 C)	17.6%	6.5%	4.9%	5.1%	2.0%	4.2%	-0.6%	0.9%
Speedup (L2 PF, L2 C)	24.5%	39.1%	5.8%	-5.4%	4.9%	14.6%	-4.2%	15.3%
Speedup (L2+Link C)	20.5%	9.7%	5.6%	5.9%	3.1%	4.2%	22.6%	2.9%
Speedup (L2 PF, L2+Link C)	39.5%	52.8%	10.6%	0.2%	7.4%	14.3%	19.6%	42.5%
Interaction(L2 PF, L2C)	1.4%	2.2%	-0.2%	6.5%	-3.1%	-2.0%	-0.8%	-5.4%
Interaction (L2 PF, L2+Link C)	**10.9%**	**8.9%**	3.7%	**11.9%**	-1.8%	-2.3%	0.3%	**14.6%**

6.4.4 Interaction Between L2 Prefetching and Cache Compression

When combining cache compression and L2 prefetching, we get a multiplicative speedup effect compared to the speedup of either technique alone. Table 6-1 shows the interaction between L2 prefetching and cache compression (sixth row). The interaction factor is negligible for most benchmarks as the positive and negative interactions offset. When the negative interaction factor is higher than the positive interaction factor, the speedup of the combined configuration is less than the product of individual configurations. This is the case for SPEComp benchmarks since prefetching is highly accurate, and compression avoids part of the same misses avoided by prefetching. On the other hand, most commercial applications show a positive interaction factor. This is because cache compression helps avoid some of the “additional prefetches” caused by L2 prefetching (Figure 6-3), as well as reducing off-chip bandwidth demand (Figure 6-2). Both of these positive interaction effects lead to a larger benefit from prefetching than if compression was not implemented. In apache, for example, the speedup of L2 prefetching alone is 4.4%, and that of L2 compression alone is 17.6%, and the speedup of the combination of L2 prefetching and L2 compression is 24.5% (slightly higher than 1.044*1.176=1.228 or a 22.8% speedup). Eq. 6.4 shows that this is a positive interaction coefficient of 1.4%. This is because the positive interaction (of compression avoiding some additional prefetches and prefetching-induced misses) outweighs the negative interaction of having both schemes target some of the same misses (Figure 6-3).

6.4.5 Link Compression Impact

In Chapter 5, we showed that link compression alone provided little benefit for all benchmarks except fma3d. However, link compression increased the speedup of the combination of L2 prefetching and L2 compression for apache and zeus by more than 14% (compare bars 6 and 4 in Figure 6-4). This is because link compression helps reduce the increase in off-chip bandwidth demand due to prefetching, thus reducing interconnect contention and increasing prefetching speedup. The combination of L2 prefetching and cache plus link compression helps achieve a speedup of 11-53% for commercial benchmarks except jbb, and 7-43% for SPEComp benchmarks.

6.4.6 Summary

The combination of L2 prefetching and compression achieves a significant speedup (0-53%) for scientific and commercial benchmarks. This combined speedup is affected by positive and negative interaction factors. Positive interaction factors include: link compression reducing contention because of prefetching, and cache compression avoiding some of the additional prefetches and misses caused by prefetching. A negative interaction factor is that both compression and prefetching can help avoid some of the same misses. Combining L2 prefetching and compression achieves speedups that are higher—in all but two benchmarks—than the product of their individual speedups.

6.5 Interactions Between Compression and L1 Prefetching

In this section, we study the interaction between compression and L1 prefetching. We simulated different combinations of cache/link compression and L1 prefetching for our eight benchmarks. We present results from configurations that implement L1 prefetching—and not L2 prefetching—to isolate the impact of L1 prefetching alone. In the next few subsections, we show the following two positive interactions between compression and L1 prefetching:

- L1 prefetching significantly increases pin bandwidth demand for many benchmarks since it can initialize L2 fill requests from memory. Compression (mostly link compression) helps alleviate such increase in pin bandwidth demand, a positive interaction between compression and L1 prefetching.
- L1 prefetching helps tolerate decompression overhead for some L2 hits penalized by cache compression, a positive interaction.

6.5.1 L1 Prefetching Bandwidth Demand

When L1 prefetching is implemented without L2 prefetching, L1 prefetches that miss in the L2 cache initialize L2 fill requests from memory. This leads to an increase in pin bandwidth demand that can be significant for many benchmarks. Figure 6-5 shows the off-chip bandwidth demand for different combinations of L1 prefetching and compression. L1 Prefetching alone increases the off-chip bandwidth demand by 22-51% for commercial benchmarks, and by 12-133% for SPEComp benchmarks. This increase in bandwidth demand is less than that of L2 prefetching (Figure 6-2). When combining L1 prefetching with cache and link compression, however, all benchmarks (except the incompressible apsi) show a significant reduction in bandwidth demand compared to L1 prefetching alone. This constitutes a *positive interaction* between L1 prefetching and compression.

6.5.2 Impact on L2 Hit Latency

Cache compression increases L2 hit latency, since L2 hits to compressed cache lines suffer a decompression overhead (five cycles in our evaluation). We expect that L1 prefetching would decrease such overhead

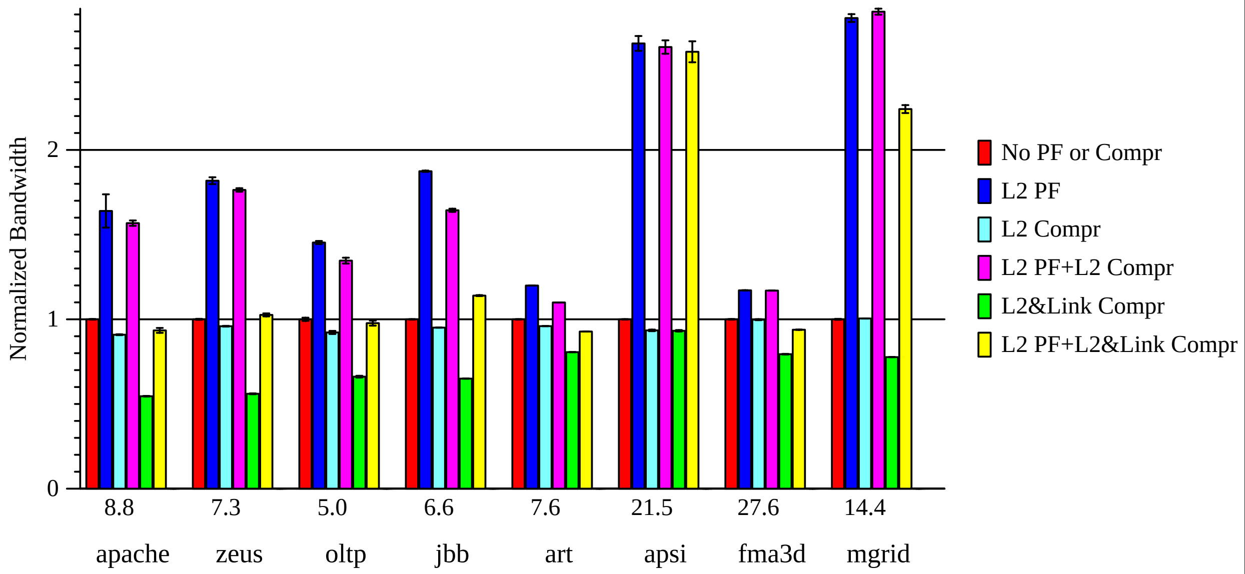


FIGURE 6-5. Off-chip bandwidth demand for L1 prefetching and compression combinations

Bandwidth demand is obtained as the bandwidth utilized with an infinite available pin bandwidth. For each benchmark, bandwidth demand is normalized to the bandwidth demand with no prefetching or compression, shown at the bottom (in GB/sec.).

due to prefetching compressed lines before they are needed, thereby reducing the impact of the decompression overhead. Figure 6-6 shows the average L2 cache hit latency for different combinations of L2 compression and L1 prefetching. L2 compression increases the average L2 hit latency by 1.2-3.7 cycles for compressible benchmarks.

Surprisingly, L1 prefetching does not decrease the L2 hit latency for most benchmarks. This is because we do not count hits in the L1 due to L1 prefetches as L2 hits, so L1 prefetching reduces the total number of L2 hits while increasing the number of L1 hits. Since L1 prefetching increases the demand on the L2 cache, this leads to an increased contention for L2 bank ports and a slight increase in L2 hit latency (up to 0.7 cycles).

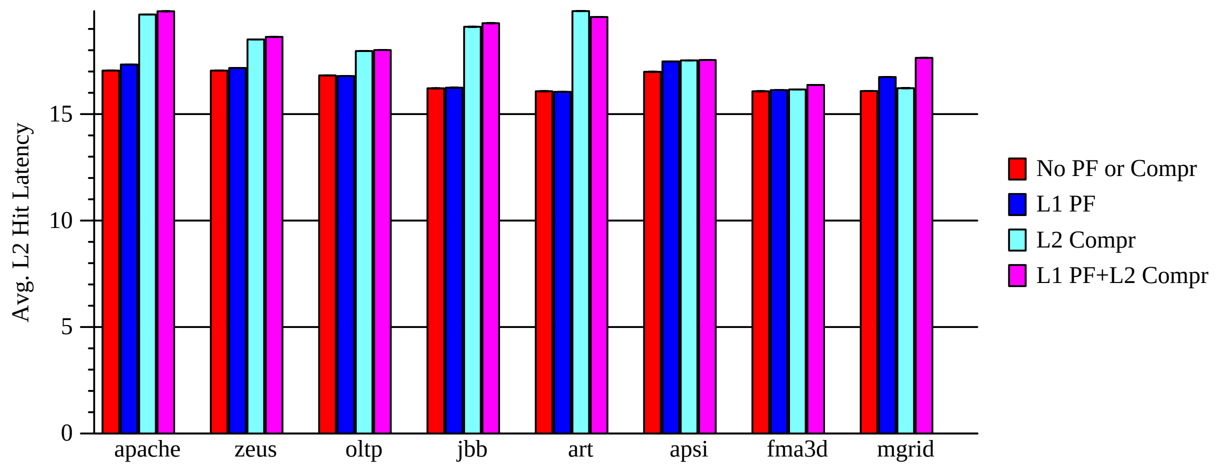


FIGURE 6-6. L2 hit latency for combinations of L2 compression and L1 prefetching

Average L2 hit latency is measured in cycles from fill to use. We do not count completed L1 prefetches (that hit in the L1) as L2 hits.

TABLE 6-2. Percentage of penalized hits avoided by L1 prefetching

Benchmark	**apache**	**zeus**	**oltp**	**jbb**	**apsi**	**art**	**fma3d**	**mgrid**
%Penalized Hits	2.4	6.2	0.4	0.9	1.9	0.9	11.4	0.4

L1 prefetching helps reduce the decompression overhead by prefetching compressed lines to the L1 cache, a *positive interaction*. We show the percentage of penalized hits that are avoided by L1 prefetching in Table 6-2. However, this percentage is small for most benchmarks due to the low L1 prefetching accuracy and coverage for compressible commercial benchmarks, and the incompressibility of most lines in SPEComp benchmarks where L1 prefetching has higher coverage and accuracy (Table 6-1).

6.5.3 Performance

Figure 6-7 shows normalized runtimes for our eight benchmarks for different combinations of compression and L1 prefetching, relative to the base case of no compression or prefetching. Table 6-3 shows the speedups and interaction coefficients for these combinations. L1 prefetching alone speeds up all benchmarks by up to 35%. Fma3d shows the least speedup at 0.2% since it is bandwidth-limited—at 20 GB/sec. available bandwidth—even with no prefetching, and prefetches have to compete with demand misses for pin bandwidth (with priority given to demand misses).

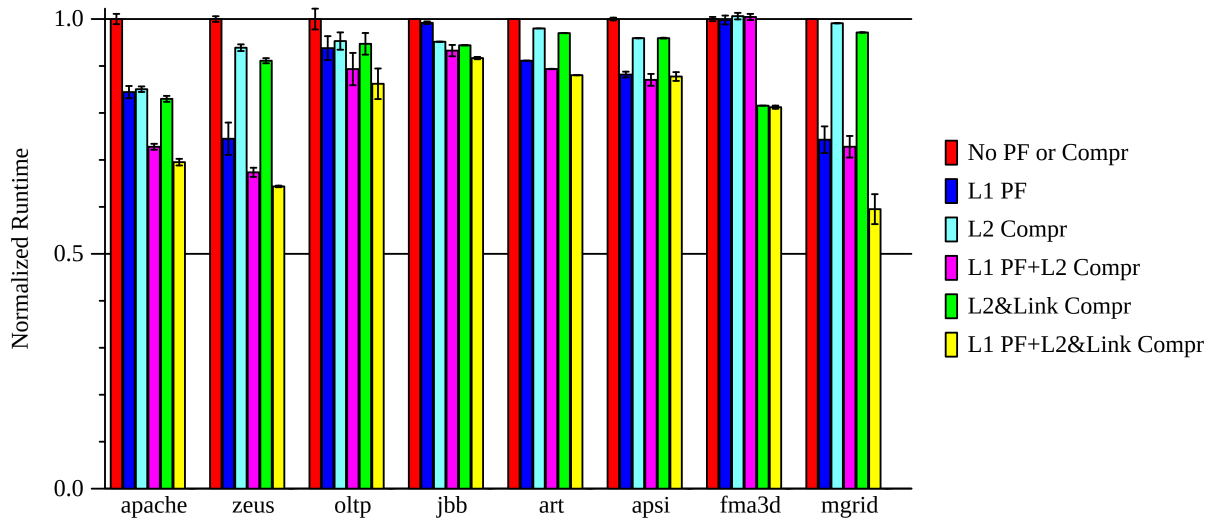


FIGURE 6-7. Performance of combinations of L1 prefetching and compression

Performance is normalized to the case of no prefetching or compression.

TABLE 6-3. Speedups and Interactions between L1 Prefetching and Compression

Benchmark	**apache**	**zeus**	**oltp**	**jbb**	**art**	**apsi**	**fma3d**	**mgrid**
Speedup (L1 PF)	18.4%	34.2%	6.6%	0.8%	9.7%	13.4%	0.2%	34.6%
Speedup (L2 C)	17.6%	6.5%	4.9%	5.1%	2.0%	4.2%	-0.6%	0.9%
Speedup (L1 PF, L2 C)	37.4%	48.5%	11.9%	7.2%	11.9%	14.9%	-0.4%	37.4%
Speedup (L2+Link C)	20.5%	9.7%	5.6%	5.9%	3.1%	4.2%	22.6%	2.9%
Speedup (L1 PF, L2+Link C)	43.8%	55.4%	16.0%	9.1%	13.6%	13.9%	23.1%	68.0%
Interaction(L1 PF, L2C)	-1.3%	3.9%	0.1%	1.2%	-0.1%	-2.8%	0.0%	1.2%
Interaction (L1 PF, L2+Link C)	0.8%	5.5%	3.1%	2.1%	0.4%	-3.6%	0.2%	**21.3%**

When combining compression and L1 prefetching (last row in Table 6-3), we also get an overall positive interaction coefficient for all benchmarks except apsi. This higher than expected speedup is because of the positive interactions between compression and prefetching.

6.5.4 Summary

L1 prefetching alone provides speedups across all benchmarks, while compression achieves speedups mostly for commercial benchmarks. The combination of L1 prefetching and compression can achieve speedups higher than the product of individual speedups due to two main positive interaction factors. First, cache and link compression reduce the increase in pin bandwidth demand due to L1 prefetching that triggers L2 fill requests. Second, L1 prefetching hides decompression overhead caused by hits to compressed lines. We showed that the effect of the first factor (reduction in pin bandwidth demand) outweighs the second.

6.6 Interactions Between Compression and Both L1 and L2 Prefetching

When combining both L1 and L2 prefetching, the same trends in the two previous sections hold. In this section, we show the combined impact of positive and negative interaction factors between compression

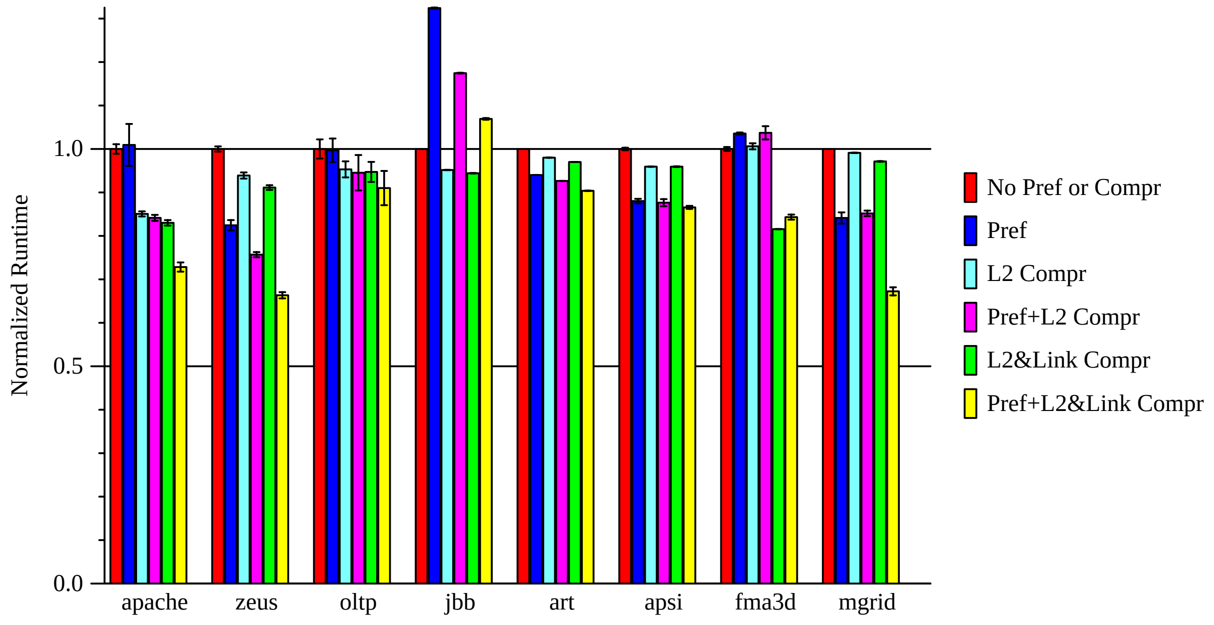


FIGURE 6-8. Performance of combinations of compression and both L1 and L2 prefetching

Performance is normalized to the case of no prefetching or compression.

TABLE 6-4. Speedups and Interactions between L1 and L2 Prefetching and Compression

Benchmark	**apache**	**zeus**	**oltp**	**jbb**	**art**	**apsi**	**fma3d**	**mgrid**
Speedup (L1+L2 PF)	-0.9%	21.3%	0.3%	-24.5%	6.4%	13.6%	-3.4%	18.9%
Speedup (L2 C)	17.6%	6.5%	4.9%	5.1%	2.0%	4.2%	-0.6%	0.9%
Speedup (L1+L2 PF, L2 C)	18.8%	32.1%	5.8%	-14.9%	7.9%	14.1%	-3.6%	17.4%
Speedup (L2+Link C)	20.5%	9.7%	5.6%	5.9%	3.1%	4.2%	22.6%	2.9%
Speedup (L1+L2 PF, L2+Link C)	37.3%	50.7%	9.9%	-6.5%	10.6%	15.5%	18.6%	48.7%
Interaction(L1+L2 PF, L2C)	2.0%	2.3%	0.5%	7.3%	-0.6%	-3.7%	0.5%	-2.1%
Interaction (L1+L2 PF, L2+Link C)	**15.0%**	**13.2%**	3.8%	**16.9%**	0.9%	-2.5%	0.2%	**21.5%**

and prefetching. Figure 6-8 shows normalized runtimes for different combinations of prefetching and compression. Table 6-4 shows the speedups and interaction coefficients.

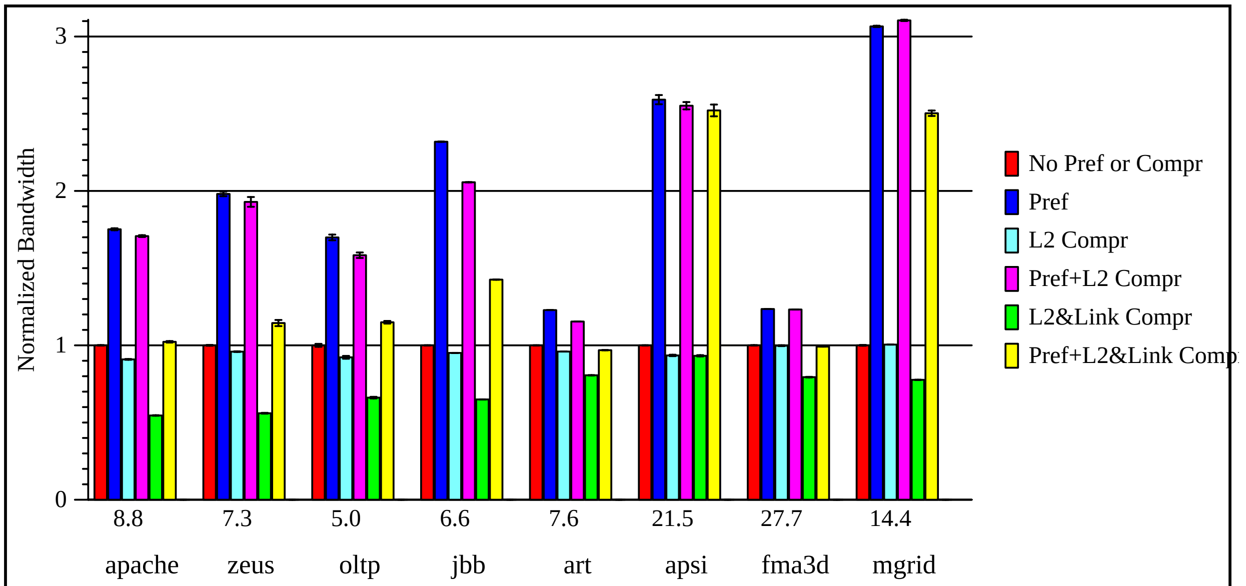


FIGURE 6-9. Off-chip bandwidth demand for L1&L2 prefetching and compression combinations

Bandwidth demand is obtained as the bandwidth utilized with an infinite available pin bandwidth. For each benchmark, bandwidth demand is normalized to the bandwidth demand with no prefetching or compression, shown at the bottom (in GB/sec.).

Combining L1 and L2 prefetching achieves a smaller speedup compared to the product of the two individual speedups. This is expected since L1 and L2 prefetching can redundantly avoid the same L2 misses, and since they combine to increase off-chip bandwidth demand (Figure 6-9). For three of the SPEComp benchmarks (apsi, fma3d and mgrid), the increase in bandwidth due to prefetching is enough to make these benchmarks bandwidth-limited, leading to smaller performance improvements. Overall, stride-based prefetching alone speeds up some benchmarks by 0-21%, while slowing down others by 1-25%. On the other hand, cache and link compression speed up all benchmarks by 3-23%.

When combining both L1 and L2 prefetching with cache and link compression, we achieve speedups of 10-51% for all benchmarks, except for jbb. The two schemes interact positively for all our benchmarks except apsi (last row of Table 6-4). The interaction coefficients can be as high as 22% (mgrid) or 17% (jbb). We attribute these large positive interaction coefficients the combination of positive interaction factors between both prefetching schemes and both compression schemes, as we discussed in more detail in the previous sections. We next study the sensitivity of these results to changes in pin bandwidth.

6.7 Sensitivity to Pin Bandwidth

In the previous sections, we showed that increasing off-chip bandwidth demand is one of the main negative effects of prefetching. We also demonstrated that compression interacts positively with prefetching by tolerating such demand increase. In this section, we analyze the impact of changing pin bandwidth on the performance of hardware prefetching and its interaction with compression. Our evaluation parameters remain the same except for available pin bandwidth. We show how the utilized bandwidth, performance and interaction coefficients change when pin bandwidth varies between 10 and 80 GB/sec.

6.7.1 Utilized Bandwidth

At 20 GB/sec. available bandwidth, many of our benchmarks (mainly SPEComp benchmarks) were bandwidth-limited (i.e., utilized bandwidth is close to available pin bandwidth and therefore limits performance). In this section, we show the impact of increasing the available bandwidth on pin bandwidth utilization of our benchmarks. We show the utilized bandwidth for commercial workloads (Figure 6-10) and SPEComp benchmarks (Figure 6-11) as pin bandwidth varies between 10 and 80 GB/sec. We note that the scale is not the same for different benchmarks.

For commercial benchmarks, our scaled-down versions of commercial benchmarks do not have significant utilized-bandwidth increases as the available pin bandwidth increases. Their bandwidth demand is at most 15.4 GB/sec., even with prefetching (Figure 6-9). These benchmarks are bandwidth-limited only for the extreme 10 GB/sec. pin bandwidth configuration.

SPEComp benchmarks, on the other hand, show significant increases in bandwidth demand as available pin bandwidth increases. With stride-based prefetching, apsi’s pin bandwidth demand increases to higher

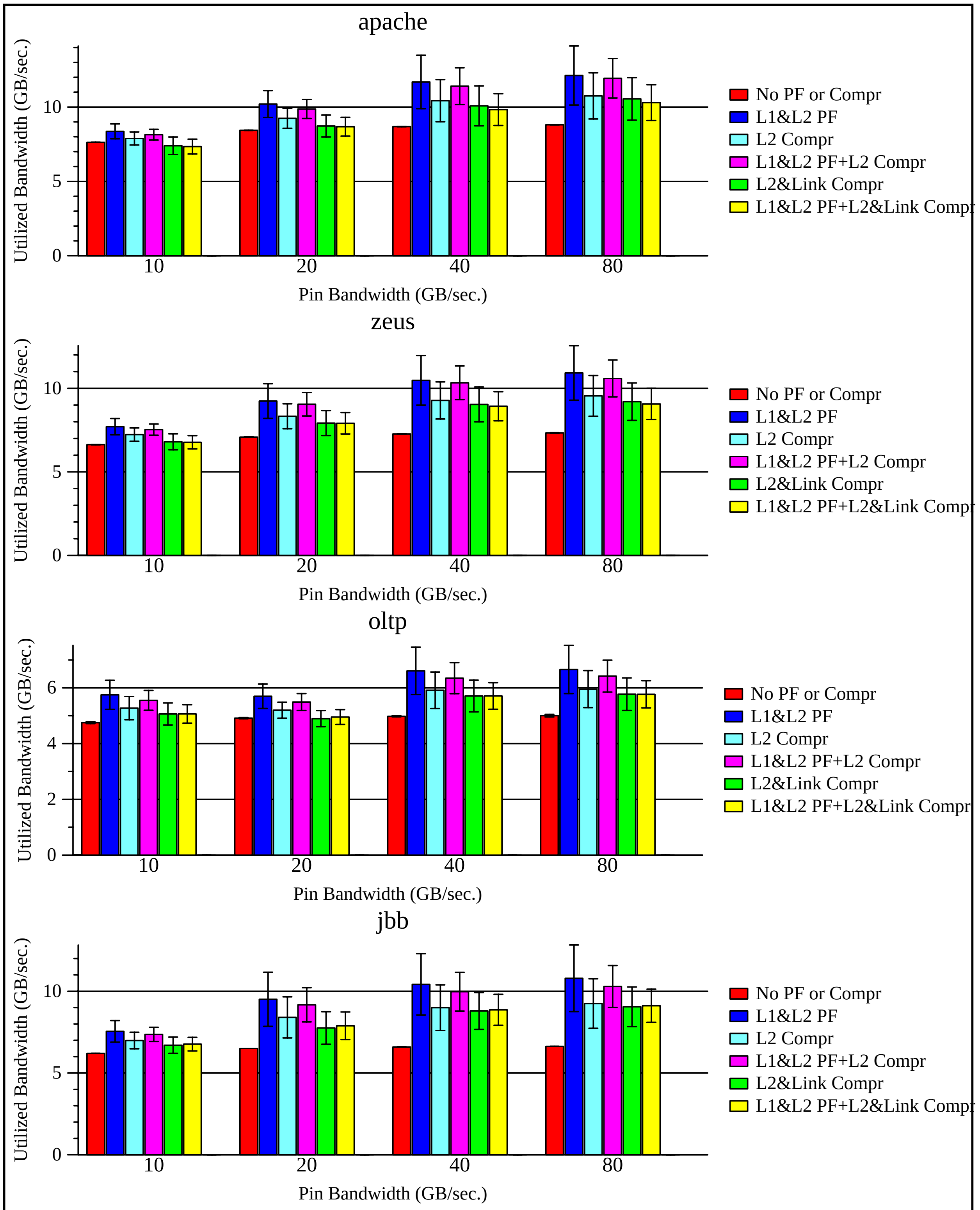


FIGURE 6-10. Pin bandwidth demand of different compression and prefetching combinations for commercial benchmarks when pin bandwidth varies from 10 to 80 GB/sec.

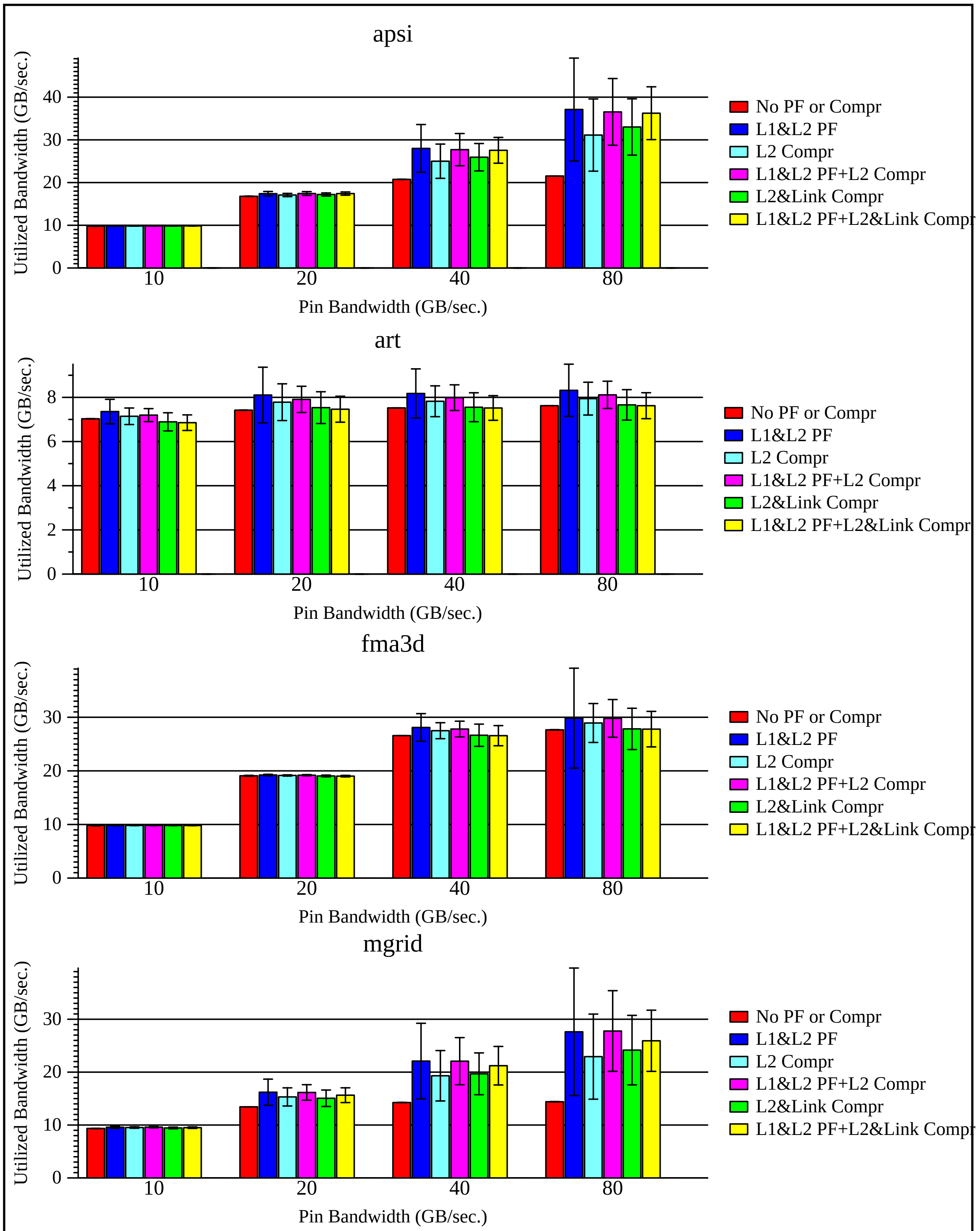


FIGURE 6-11. Pin bandwidth demand of different compression and prefetching combinations for SPEComp benchmarks when pin bandwidth varies from 10 to 80 GB/sec.

than 55.8 GB/sec. All SPEComp benchmarks are bandwidth-limited for the 10 GB/sec. configuration, and all except art are bandwidth-limited for the 20 GB/sec. configuration. Two of the benchmarks (apsi and mgrid) are even bandwidth-limited at 40 GB/sec. available pin bandwidth.

6.7.2 Performance

Since prefetching can significantly increase pin bandwidth demand, its impact on performance is expected to be negative for bandwidth-limited configurations. Cache and link compression can improve performance for systems that implement prefetching since they combine to reduce pin bandwidth demand. We show the impact of prefetching and compression on performance for commercial workloads (Figure 6-12) and SPEComp benchmarks (Figure 6-13) as pin bandwidth varies between 10 and 80 GB/sec.

For commercial workloads, the bandwidth-limited 10 GB/sec. pin bandwidth configuration shows a slowdown of 6-42% due to hardware prefetching. On the other hand, prefetching shows speedups for most benchmarks (except jbb) when bandwidth increases. When combined with cache and link compression, prefetching shows a speedup for all benchmarks (except jbb) for the 10 GB/sec. configuration that is close to the speedups achieved for bandwidth-abundant configurations. This is caused by the positive interaction of compression reducing pin bandwidth demand increase due to prefetching.

For SPEComp benchmarks, the performance improvement due to prefetching increases as available pin bandwidth increases. For example, apsi suffers a 2% slowdown for 10 GB/sec. pin bandwidth, but shows speedups of 14%, 75% and 157% for the 20, 40 and 80 GB/sec. configurations, respectively. For fma3d and mgrid, cache and link compression achieve significant speedups for bandwidth-limited configurations when combined with hardware prefetching. However, the impact of cache and link compression in most cases is limited due to the low compression ratios for these benchmarks.

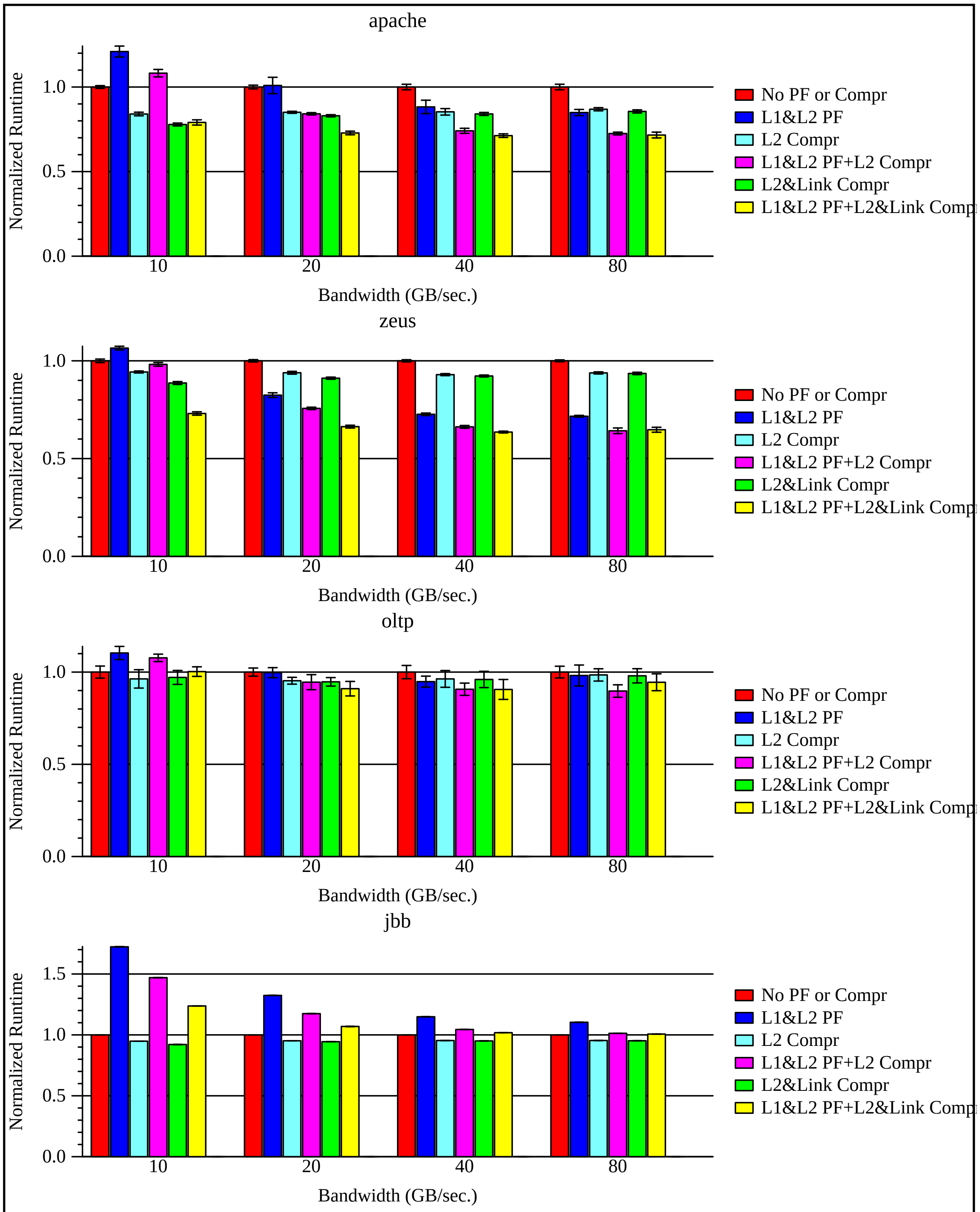


FIGURE 6-12. Performance of different compression and prefetching combinations for commercial benchmarks when pin bandwidth varies from 10 to 80 GB/sec.

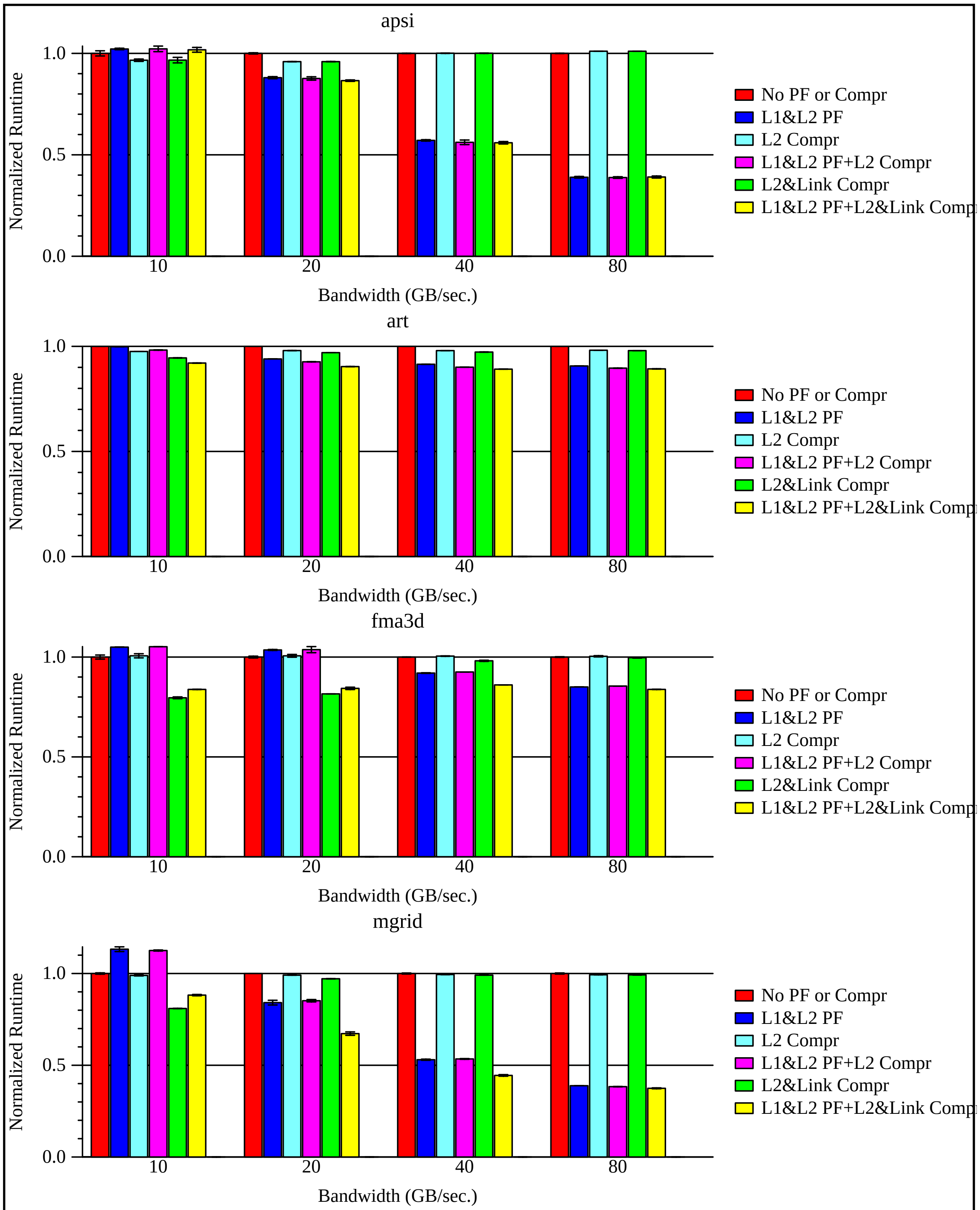


FIGURE 6-13. Performance of different compression and prefetching combinations for SPECcomp benchmarks when pin bandwidth varies from 10 to 80 GB/sec.

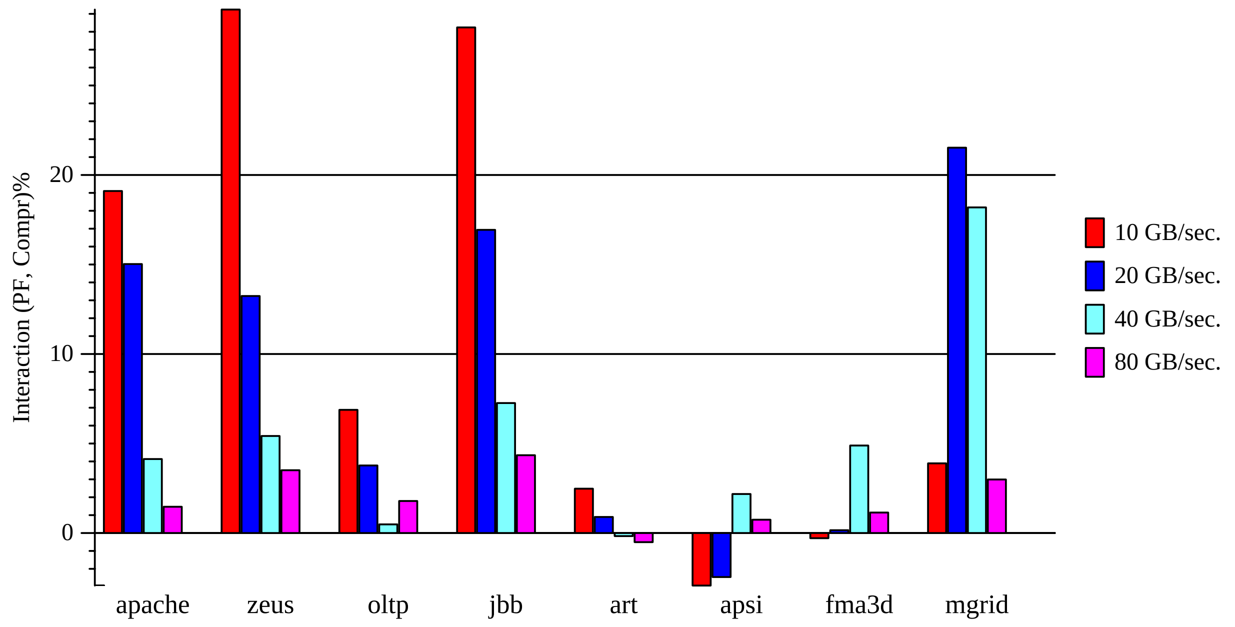


FIGURE 6-14. Interaction(%) between prefetching and compression as available pin bandwidth varies

6.7.3 Interaction Between Prefetching and Compression

The negative side effects of prefetching are most obvious for systems with limited bandwidth. Since compression tolerates some of these negative side effects, the positive interaction between compression and prefetching can be higher for bandwidth-limited configurations. Figure 6-14 presents the interaction terms between compression and prefetching when the available pin bandwidth changes from 10 to 80 GB/sec.

For commercial benchmarks, the interaction term is large (7-29%) for the 10 GB/sec. pin bandwidth configuration. The interaction is also significant (4-17%) for the 20 GB/sec. configuration. However, the interaction drops dramatically for the 40 and 80 GB/sec. configurations since the available bandwidth significantly exceeds demand, even with prefetching (Figure 6-9). Overall, the interaction is positive between compression and prefetching for all configurations.

For SPEComp benchmarks, the interaction is negative for some configurations since compression is less effective for these benchmarks. However, the negative interaction terms are limited to 3% or less. On the other hand, some configurations show significant positive interaction terms (as high as 22% for mgrid) due to the impact of link compression on reducing pin bandwidth demand caused by prefetching. Except for a few configurations, the interaction is mostly positive for SPEComp benchmarks.

6.8 Sensitivity to Number of CMP Cores

Prefetching schemes have been previously shown to be successful for uniprocessor systems. However, implementing prefetching on a CMP introduces contention for shared resources (caches and pin bandwidth) that reduce its performance improvement. In Figure 6-15, we show the performance improvement[3] due to stride-based prefetching, compression and other alternatives for our commercial benchmarks[4]. Performance improvements are shown relative to a base system with the same parameters as those in Chapter 5, except for the different number of processors. Stride-based prefetching alone improves uniprocessor performance by 61%, 73%, 11% and 2% for apache, zeus, oltp and jbb, respectively. However, such performance improvement decreases as we increase the number of processor cores on a CMP, eventually degrading performance. For a 16-processor CMP, stride-based prefetching shows no improvement for apache, and degrades the performance of zeus, oltp and jbb by 8%, 10% and 35%, respectively.

Cache and link compression with no prefetching achieve modest performance improvements for uniprocessors (20%, 7%, 2% and 6% for apache, zeus, oltp and jbb, respectively). However, compression and prefetching interact positively since compression mitigates some of the negative side effects due to

3. We use the term “performance improvement” instead of “speedup” in this section to avoid confusion with parallel speedup over a uniprocessor configuration. Performance improvement is defined as (S-1)*100%, where S is the speedup in performance relative to our base system of no prefetching or compression.

4. We did not have the necessary checkpoints for SPEComp benchmarks, except for the 8-processor checkpoints.

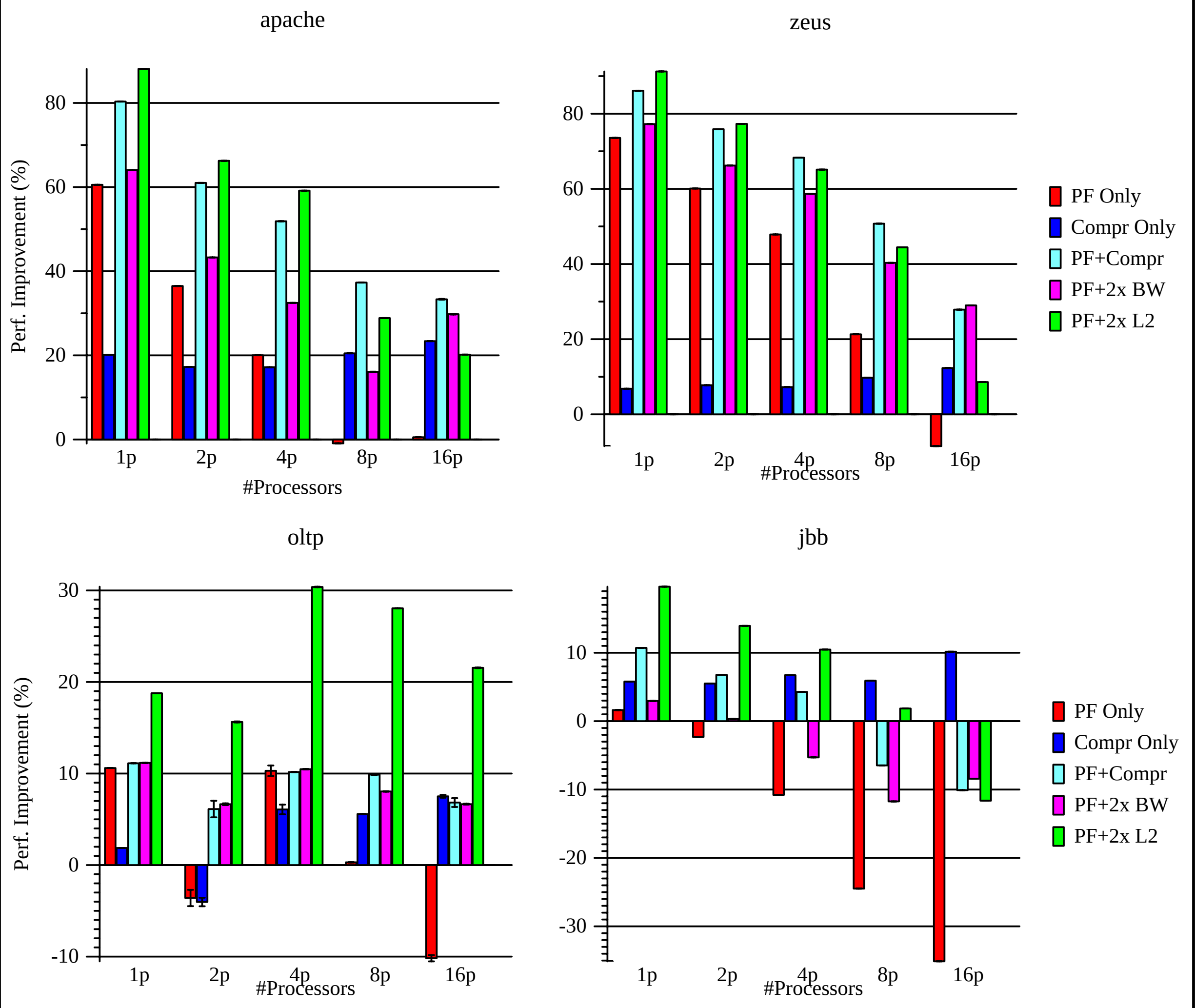


FIGURE 6-15. Performance improvement (%) for commercial benchmarks for different uniprocessor and CMP configurations

Performance improvement is shown relative to a configuration with no compression or prefetching. All configurations have a 20 GB/sec. available off-chip bandwidth except for the "PF+2x BW" which has a 40 GB/ sec. available bandwidth. All configurations have a 4 MB shared L2 cache except for the "PF+2x L2" that has an 8 MB cache.

prefetching. This leads to strong positive interactions between the two schemes for all CMP configurations. These interactions increase with the number of processor cores. On a 16-processor CMP, for example, performance improves for a combination of prefetching and compression by 33%, 28% and 7% for apache, zeus, and oltp, respectively. The performance degradation in jbb was limited to 10%.

The positive interactions between prefetching and compression result from compression mitigating the impact of prefetching on both pin bandwidth and cache size. Therefore, the performance improvement due to compression may exceed that of techniques that address pin bandwidth alone or cache size alone. On a CMP with hardware prefetching, implementing compression outperforms doubling pin bandwidth for all configurations (except for zeus and jbb on a 16-processor CMP). Furthermore, implementing compression outperforms doubling the cache size for many configurations (apache on 8- and 16-processors, zeus on 4-, 8- and 16-processors, and jbb on 16-processors). Compression is attractive since it is less expensive to implement than either of these alternatives.

6.9 Related Work

Hardware Prefetching. Hardware-directed prefetching has been proposed and explored by many researchers [26, 65, 94, 96, 102, 140], and is currently implemented in many existing systems [58, 59, 107, 117]. Jouppi [66] introduced stream buffers that trigger successive cache line prefetches on a miss. Chen and Baer proposed variations of stride-based hardware prefetching to reduce the cache-to-memory latency [26], and studied its positive and negative impacts on different benchmarks [25]. Dahlgren, et al., proposed an adaptive sequential (unit-stride) prefetching scheme that adapts to the effectiveness of prefetching [31]. Zhang and Torrellas proposed a scheme that targets irregular access patterns by binding together (in hardware) and prefetching groups of short data lines that are indicated by the compiler to be strongly related [140]. Tullsen and Eggers studied the negative side effects of software prefetching on bus utilization, cache miss rates and data sharing for a multiprocessor system, and proposed techniques to reduce some of these negative effects [123]. Lin, et al., mitigate the negative effects of prefetching on performance by prefetching only when the memory bus is idle (to reduce contention), and prefetching to lower replacement priorities than demand misses (to reduce cache pollution) [43].

CMP Prefetching. Prefetching has been proposed or implemented to improve CMP performance. IBM's Power4 [117] and Power5 [107] both support stride-based hardware prefetching to different levels of the cache hierarchy. Beckmann and Wood show that hardware stride-based prefetching can significantly improve performance for commercial and scientific benchmarks on an 8-processor CMP [17]. Huh, et. al., show that L1 prefetching can decrease L1 miss rates and improve performance for a 16-processor CMP with a NUCA L2 cache [63]. Ganusov and Burtscher propose dedicating one processor core of a CMP to prefetch data for a thread running on another core [50]. This dissertation differs from previous proposals since it studies some negative effects of prefetching on a CMP, and is the first to study the interactions between prefetching and hardware compression.

Prefetching and Compression. Zhang and Gupta [138] exploit their compressed cache design [139] to prefetch partial compressed lines from the next level in the memory hierarchy without increasing memory bandwidth, and with no need for prefetch buffers. They store compressed values in the cache, and use the freed up space to prefetch other compressed values (i.e., prefetch partial lines). Their proposal makes use of the positive interaction between compression and prefetching since compression frees up space and bandwidth that can be used by prefetching. Lee, et al., use a decompression buffer between their cache levels to buffer decompressed lines, which can be viewed as storing "prefetched" uncompressed lines to reduce decompression overhead [77, 78, 79]. This dissertation differs from these proposals as it introduces a general unified cache and link compression CMP design that interacts with hardware prefetching. We also explore in more detail different positive and negative interactions between compression and prefetching.

6.10 Summary

Many CMP designs implement hardware-based prefetching to hide L1 and L2 miss latencies. For such systems, compression can be less appealing if prefetching achieves most of the benefits due to compression.

However, prefetching schemes can greatly increase off-chip bandwidth demand. In addition, prefetching can increase demand on cache size for many benchmarks due to cache pollution. Since both cache size and pin bandwidth are shared resources in a CMP, the benefit of prefetching decreases dramatically. In this chapter, we show that hardware stride-based prefetching provides smaller performance improvements for CMPs than for uniprocessors, even hurting performance in some cases. We further show that cache and link compression partially compensate for the increased demand by effectively increasing cache size and pin bandwidth.

In the central result of this chapter, we showed that compression and prefetching have a strong positive interaction, improving performance by 10-51% for seven of our eight benchmarks for an 8-processor CMP. Compression and prefetching interact positively in three ways: link compression reduces prefetching's off-chip bandwidth demand; L1 prefetching hides part of the decompression penalty due to cache compression; and cache compression helps accommodate the increase in working set size due to prefetching. This implies that compression helps reduce the two main negative side effects of prefetching, and prefetching helps mitigate the main negative side effect due to compression. We also show a negative interaction between the two schemes, since a fraction of the misses avoided by compression can also be avoided by prefetching. Overall, compression and prefetching interact positively, and their combined speedup equals or exceeds the product of the two individual speedups for most benchmarks. Such positive interactions can lead to performance improvements for CMP configurations whose performance would have degraded with hardware stride-based prefetching alone.

Chapter 7

Balanced CMP Design: Cores, Caches, Communication and Compression

A fundamental question in chip design is how to best utilize the available on-chip transistor area. For CMPs, this translates to how to allocate chip area between cores and caches. Given a fixed transistor (i.e., area) budget, designers must choose an “optimal” breakdown between cores and caches. This choice is not obvious, since the 2004 ITRS Roadmap [45] predicts that transistor speed will continue to improve faster than DRAM latency and pin bandwidth (21%, 10%, and 11% per year, respectively). Should the design center on caches, to hide DRAM latency and conserve off-chip bandwidth, or on cores, to maximize thread-level parallelism?

Compression further complicates this trade-off. Cache compression increases the effective cache size for a given transistor budget, reducing both average memory latency and (possibly) contention for limited pin bandwidth. Link compression increases the effective pin bandwidth, potentially supporting more cores with smaller caches. Compression in any form adds latency overheads to compress and decompress data, possibly outstripping any improvements.

This chapter examines an important aspect of how to design balanced CMP systems. That is, given a fixed core design, how to allocate on-chip transistors to balance the demand on cores, caches, and communication. In particular, we study the role that compression plays in shifting this balance. To provide intuition about this trade-off, we develop a simple analytical model that estimates throughput for different CMP configurations with a fixed area budget (Section 7.1). We extend our model to include cache compression (Section 7.2) and link compression (Section 7.3). We discuss the many simplifying assumptions that affect our model's accuracy (Section 7.4). We use our model to estimate the "optimal" CMP configuration (in terms of the number of cores and cache sizes) for a set of parameters and the impact of compression on that optimal configuration (Section 7.5). We also study the sensitivity of CMP throughput to different model parameters (Section 7.6). We extend our simple model to include prefetching and study its interactions with compression (Section 7.7). We then use full-system simulation and commercial workloads to quantitatively evaluate the "optimal" design point given a fixed area assumption (Section 7.8). We compare simulation results to those of the analytical model to evaluate its relative error (Section 7.9). Although the analytical model makes too many simplifying assumptions to accurately predict absolute throughput, we show in Section 7.9 that it can provide insight by capturing the general trends. We discuss some related work (Section 7.10) and conclude (Section 7.11).

In this chapter, we make the following contributions:

- We introduce a simple analytical model that helps build intuition about the trade-off between cores, caches, and communication, and the role of compression in CMP design.
- We show, using our analytical model and simulation experiments, that compression can improve CMP throughput by nearly 30%.
- We use our analytical model to demonstrate that compression can lead to significant throughput improvements across a wide range of CMP system parameters.
- We show that both cache and interconnect compression can slightly shift the optimal CMP design towards having more cores, leading to more core-centric designs.
- We show that cache compression interacts positively with hardware prefetching across a wide range of workloads and CMP configurations. Positive interaction coefficients (as high as 33%) lead to throughput improvements that are significantly higher than the product of throughput improvement for either scheme alone.

7.1 Blocking Processor Model

A fundamental question in CMP design is how to best utilize the limited transistor area and split it between processor cores and caches. Ultimately, this comes down to a question of balance. A balanced CMP design allocates transistors to processor cores and on-chip caches such that neither cores, caches, nor communication is the only bottleneck. In order to gain some insight on the trade-off between cores and caches, we develop a simple model for a CMP with a fixed transistor budget. We use this model to roughly evaluate what the best CMP design configuration is and how it changes with different system parameters. We next describe an area model and a throughput model for a fixed-area CMP.

7.1.1 Cache Byte Equivalent (CBE) Area Model

Assume we have a CMP with a total fixed area A_{CMP} allocated to cores and caches that is a proper multiple of the area needed for a 1MB L2 cache:

$$A_{CMP} = m \cdot A_{1MB} \tag{7.1}$$

where A_{1MB} is the area required for 1 MB of L2 cache, and m is a design constant taking into account semiconductor process generation, cost objectives, etc. The area of one processor (including its private L1 caches) can be written also in terms of 1 MB L2 cache area:

$$A_p = k_p \cdot A_{1MB} \tag{7.2}$$

For a system of N processors, the area allocated to L2 caches ($A_{L2(N)}$) is simply the area not consumed by cores:

$$A_{L2(N)} = A_{CMP} - N \cdot A_p = m \cdot A_{1MB} - N \cdot k_p \cdot A_{1MB} = A_{1MB} \cdot (m - k_p \cdot N) \tag{7.3}$$

7.1.2 Throughput for a Fixed Chip Area

To estimate performance, we assume the cores are simple, in-order blocking processors. For a given workload, we can model the cycles per instruction for a single processor with an L2 cache of size S_{L2} as:

$$CPI(1) = CPI_{perfectL2} + missrate(S_{L2}) \cdot MissPenalty_{L2} \tag{7.4}$$

where $CPI_{perfectL2}$ is an estimate of the CPI of the processor core (including L1 caches) with a perfect L2 cache, *missrate*(S_{L2}) is the miss rate (per instruction) for a cache of size S_{L2}, and *L2_Miss_Penalty* is the average number of cycles needed for an L2 miss.

As we increase the number of cores, the area that remains available for cache decreases (Eq. 7.3). In addition, more non-idle cores lead to more threads competing for the L2 cache, therefore increasing L2 misses. To address this latter issue, we make the following two simplifying assumptions.

Assumption A (Sharing assumption). A pessimistic assumption is that cores do not actually share code or data in the L2 cache. In that case, for N processors, the working set is N times as large. We can approximate the effective cache size that each core sees as $1/N^{\text{th}}$ of the whole cache, or $S_{L2p} = S_{L2(N)}/N$. While this might accurately characterize a multi-tasking workload, it is pessimistic for our commercial workloads, which have been shown to heavily share both code and data [17]. Instead, we use the average number of sharers per block, $sharers_{av}(N)$, to adjust the working set size (essentially eliminating double counting of shared blocks). We note that the average number of sharers per block can vary with the number of processors. Using this sharing assumption, the size of the L2 cache used by a single processor is:

$$S_{L2p} = \frac{S_{L2(N)}}{N - sharers_{av}(N) + 1} \tag{7.5}$$

Assumption B (Square root assumption). To estimate the miss ratio for caches of different sizes, we use the well-known square root rule of thumb [57]. Thus for each core, its L2 miss rate can be computed in terms of a known miss rate for a cache of size S_{L2}:

$$missrate(S_{L2p}) = missrate(S_{L2}) \cdot \sqrt{\frac{S_{L2}}{S_{L2p}}} \tag{7.6}$$

From the previous two assumptions, the CPI of a single processor in a CMP with N processors sharing an L2 cache of size S_{L2_N} is (from Eq. 7.4):

$$CPI(N) = CPI_{perfectL2} + missrate(S_{L2p}) \cdot MissPenalty_{L2} \tag{7.7}$$

And from Eq. 7.6 substituting for $missrate(S_{L2p})$:

$$CPI(N) = CPI_{perfectL2} + MissPenalty_{L2} \cdot missrate(S_{L2}) \cdot \sqrt{\frac{S_{L2}}{S_{L2p}}} \tag{7.8}$$

Substituting assumption A’s sharing rule for $S_{L2p} = S_{L2(N)} / (N - sharers_{av}(N) + 1)$:

$$CPI(N) = CPI_{perfectL2} + MissPenalty_{L2} \cdot missrate(S_{L2}) \cdot \sqrt{\frac{S_{L2}}{S_{L2(N)}/(N - sharers_{av}(N) + 1)}} \tag{7.9}$$

Since the areas of caches are proportional to their sizes, we can substitute by the area of Eq. 7.3:

$$\begin{aligned} CPI(N) &= CPI_{perfectL2} + MissPenalty_{L2} \cdot missrate(S_{L2}) \cdot \sqrt{\frac{A_{L2} \cdot (N - sharers_{av}(N) + 1)}{(A_{1MB} \cdot (m - k_p \cdot N))}} \\ &= CPI_{perfectL2} + MissPenalty_{L2} \cdot \alpha \cdot \sqrt{\frac{(N - sharers_{av}(N) + 1)}{(m - k_p \cdot N)}} \end{aligned} \tag{7.10}$$

Where α aggregates all the invariant L2 factors other than *L2_Miss_Latency*:

$$\alpha = missrate(S_{L2}) \cdot \sqrt{\frac{A_{L2}}{A_{1MB}}} \tag{7.11}$$

To estimate aggregate system throughput, we make the further simplifying assumption that there is no other interference between threads (i.e., threads do not have to compete for cache or memory bandwidth). We can estimate throughput as the aggregate number of instructions per cycle for *N* processors:

$$IPC(N) = \frac{N}{CPI(N)} = \frac{N}{CPI_{perfectL2} + MissPenalty_{L2} \cdot \alpha \cdot \sqrt{\frac{(N - sharers_{av}(N) + 1)}{(m - k_p \cdot N)}}} \tag{7.12}$$

7.2 CMP Model with Cache Compression

Cache compression can improve CMP performance by increasing the effective density of the shared cache. To model the impact of cache compression on CMP design, we need to add two parameters to the model in the previous section:

- The cache compression ratio *c* is a workload property that measures the ratio between the effective cache size of a compressed cache and the original cache size. $S_{L2(N)}$ should be multiplied by *c* in our model.

- The decompression penalty *dp* is the average number of cycles per instruction required to decompress a block due to L1 misses to compressed L2 blocks. This should be added to $CPI_{perfectL2}$ in our model.

Our final model for *IPC*(*N*) now becomes:

$$IPC(N) = \frac{N}{CPI_{perfectL2} + dp + MissPenalty_{L2} \cdot \alpha \cdot \sqrt{\frac{(N - sharers_{av}(N) + 1)}{c \cdot (m - k_p \cdot N)}}} \tag{7.13}$$

7.3 CMP Model with Cache and Link Compression

The model in the previous sections assumed that there is no queuing delay due to pin bandwidth limitations. To model the effects of interconnect latency, we split the penalty due to an L2 miss into two components, memory latency and link latency:

$$MissPenalty_{L2} = MemoryLatency + LinkLatency \tag{7.14}$$

$$LinkLatency = ResponseTimePerMiss = ServiceTimePerMiss + QueuingDelayPerMiss \tag{7.15}$$

We use mean value analysis (MVA) to compute the *LinkLatency* and therefore the actual IPC. We first define the interconnect (link) throughput λ (in requests per cycle) as the product of the number of instructions per cycle and the number of misses per instruction:

$$\lambda = IPC(N) \cdot Missrate(S_{L2p}) \tag{7.16}$$

Using Little's law, the link utilization *U* is defined as:

$$U = \lambda \cdot \bar{X} \tag{7.17}$$

Where $\overline{X}$ is the average service time per miss. We assume that the service time is the physical link latency for data to get to/from memory. Assuming no transmission errors, this service time is deterministic. This is an M/D/1 queue, and its response time $\overline{R}$ can be computed from:

$$\overline{R} = LinkLatency = \overline{X} + \frac{U \cdot \overline{X}}{2 \cdot (1 - U)} \tag{7.18}$$

We then use this response time to compute a new value for *IPC(N)*, and iterate until the model converges.

Accounting for Memory-level Parallelism. We note that the model we discussed so far assumes a blocking memory model that doesn't handle parallel memory requests. Current systems can handle memory requests in parallel to exploit memory-level parallelism. To include such effect in our simple model, we assume that the average number of memory requests issued in parallel is mlp_{av}. The average memory latency that is used to compute $IPC(N)$ in this case can be divided by mlp_{av}, since each request on average will block the processor for a fraction of the total memory latency [70].

7.4 Model Limitations

The simple analytical model we described in this chapter is useful to qualitatively provide intuition about CMP throughput. However, our model makes many simplifying assumptions that affect its accuracy. Some of these simplifying assumptions are:

- We assume that missrates decrease linearly with the square root of the increase in cache size. While such trend has been demonstrated for smaller cache size, there is no evidence to support it holds for large caches (1 MB or larger).
- Many of the model assumptions are based on having a blocking in-order processor. For non-blocking and out-of-order processors, the impact of cache miss latency might be lower [70].

- We assume that many model parameters remain fixed even with changes in the number of cores or cache size. However, many parameters can vary with a change in CMP configuration (e.g., *dp* and *c*).
- We assume that pin bandwidth demand has a similar behavior to an M/D/1 queue. However, many off-chip requests tend to be clustered together in bursts, which implies that the Poisson incoming request distribution is not accurate. Furthermore, a deterministic service time is not accurate in the presence of transmission errors that require retransmission.
- IPC is not the best estimate of throughput for workloads that have abundant inter-thread interactions and operating system cooperation. For such benchmarks, a direct throughput measure (e.g., transactions per second) can be a better estimate. However, obtaining such estimate from IPC is not straightforward, since it requires estimating the number of instructions per transaction. Such number can vary between different CMP configurations due to a change in idle time or spin-lock waiting time.

In order to develop a model that can more accurately predict CMP throughput, future research can target developing a more complex model for CMP throughput that takes some of the above limitations into account.

7.5 Optimal CMP Configurations

In this section, we use our simple analytical model to estimate the optimal CMP configuration for a hypothetical benchmark. We show our base model parameters in Table 7-1, including both system and benchmark parameters. We chose benchmark parameters that approximate apache's behavior (from chapters 4 and 5). We assume that hardware prefetching is not implemented in the base system. We show the model results for four configurations (no compression, cache compression only, link compression only, both cache and link compression) in Figure 7-1. We make the following observations:

TABLE 7-1. Model Parameters

Area Assumptions	Total CMP area is equivalent to that of 8 MB of L2 cache area (m = 8). Each core (plus L1 caches) has the same area as 0.5 MB of L2 cache (k_p = 0.5). The two extreme configurations for the CMP are: 16 cores with no cache, or 8 MB of cache with no cores
$\mathrm{CPI}_{\mathrm{perfectL2}}$	1
Pin Bandwidth	20 GB/sec. pin bandwidth. Average service time per miss $\overline{X}$ = 72 bytes_per_miss / 4 bytes_per_second = 18 cycles.
Memory Latency	400 cycles
Missrate (8 MB)	10 misses / 1000 instructions
Compression Properties	Compression ratio c = 1.75, decompression penalty dp = 0.4 cycles per instruction
$\mathrm{sharers}_{\mathrm{av}}$(N)	1.0 sharers/line if N=1, 1.3 sharers/line otherwise

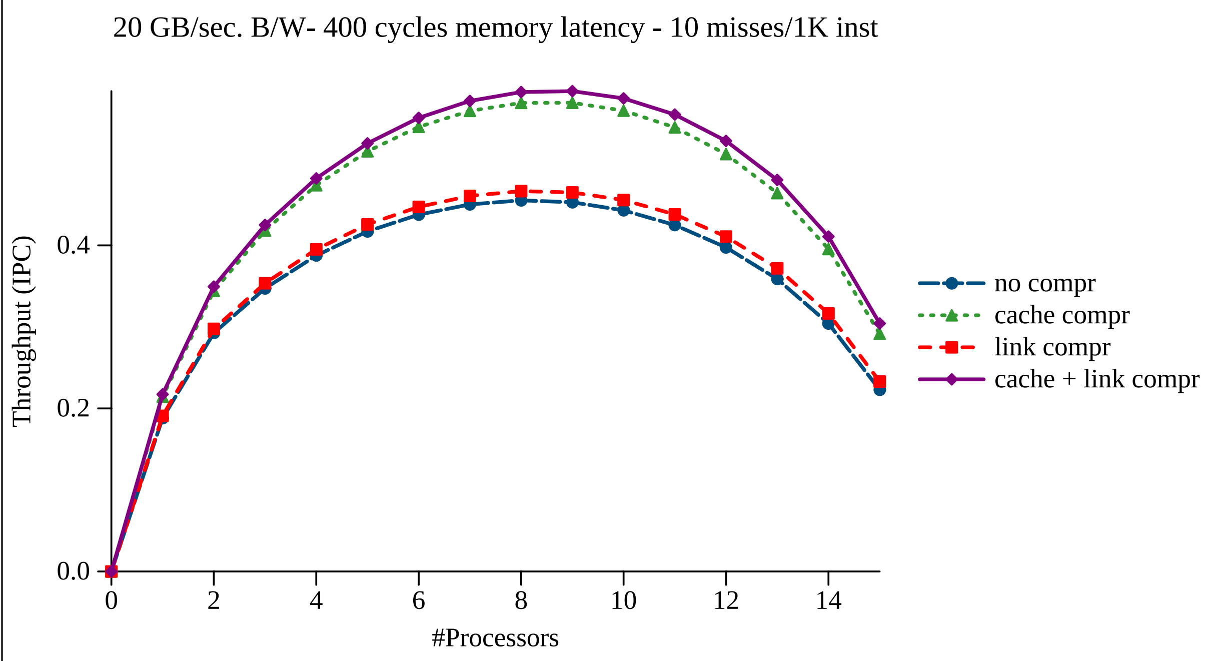


FIGURE 7-1. Analytical model throughput (IPC) for different processor configurations (x-axis) and different compression configurations

- All curves show increasing throughput that peaks at 8-9 processors after which throughput decreases. IPC increases as more processors are available up to a certain point (8 processors/4 MB L2). Beyond this point, the impact of memory latency on performance becomes more dominant, thus decreasing IPC.
- At 20 GB/sec. pin bandwidth and a base L2 missrate of 10 misses per 1000 instructions, pin bandwidth is not a critical resource. The impact of link compression on performance is limited (only a 2.5% throughput increase for the optimal 8 processor configuration).
- Since cache compression increases the effective cache size and reduces off-chip misses, its impact on throughput is significant for all configurations. Cache compression alone achieves up to a 26% improvement in throughput over the no compression case. Moreover, the increase in throughput is more significant for the optimal processor/cache configurations.
- The combination of cache and link compression improve throughput by up to 29%.

7.6 Sensitivity Analysis

Our analytical model allows us to easily evaluate and gain insight into the impact of different parameters on CMP throughput. In this section, we discuss the sensitivity of our results to different model parameters. In each of the next few subsections, we vary one parameter while leaving the remaining parameters constant.

7.6.1 Sensitivity to Pin Bandwidth

Both cache and link compression can reduce pin bandwidth utilization and therefore improve throughput for bandwidth-limited systems. As available pin bandwidth increases, the impact of compression on throughput decreases. Figure 7-2 shows the impact of pin bandwidth on throughput for our hypothetical benchmark. We make the following observations:

- When pin bandwidth is limited (10 GB/sec.), the peak of the throughput curve moves to the left. This is because pin bandwidth becomes a critical resource, and fewer processors with bigger caches are less limited by off-chip bandwidth. The optimal configuration for an uncompressed system with 10 GB/sec. pin bandwidth is at 7 processors, compared to 8 or 9 processors for most other configurations.
- Cache and link compression provide a significant increase in throughput across all bandwidth configurations. The throughput increase is bigger for low pin bandwidth (36% for the 10 GB/sec. system) due to the additional impact of link compression, but is still significant for high bandwidths. The throughput increase is at 26% for the 10 TB/sec. (i.e., 10000 GB/sec.) system.
- Compression tends to slightly shift the optimal configuration towards more cores. The optimal configuration for all systems (except the 10 GB/sec. system) is at 8 processors with no compression, and at 9 processors with compression. For the 10 GB/sec. system, the optimal configuration is at 7 processors with no compression, and at 8 processors with compression. This is because cache compression increases the effective cache size and reduces miss rates.

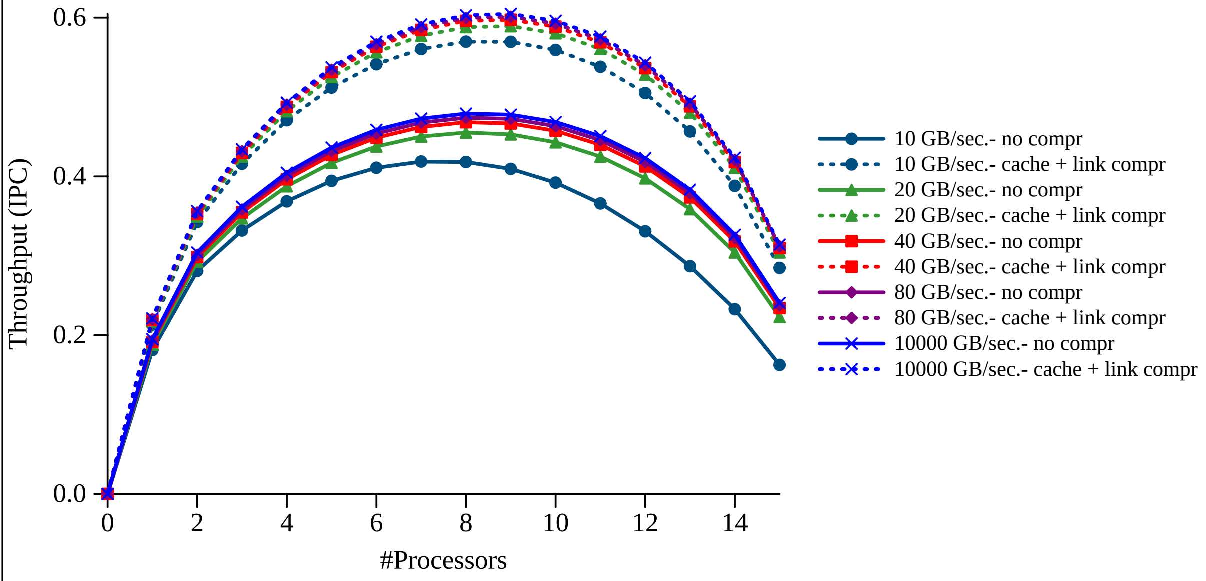


FIGURE 7-2. Analytical model sensitivity to pin bandwidth. Non-compressed configurations are represented by solid lines, and compressed configurations are represented by dotted lines

7.6.2 Sensitivity to Cache Miss Rate

Figure 7-3 shows the impact of the base L2 cache miss rate on throughput for our hypothetical benchmark. Since our model represents a simple blocking processor, cache miss rates have a significant impact on throughput. We make the following observations:

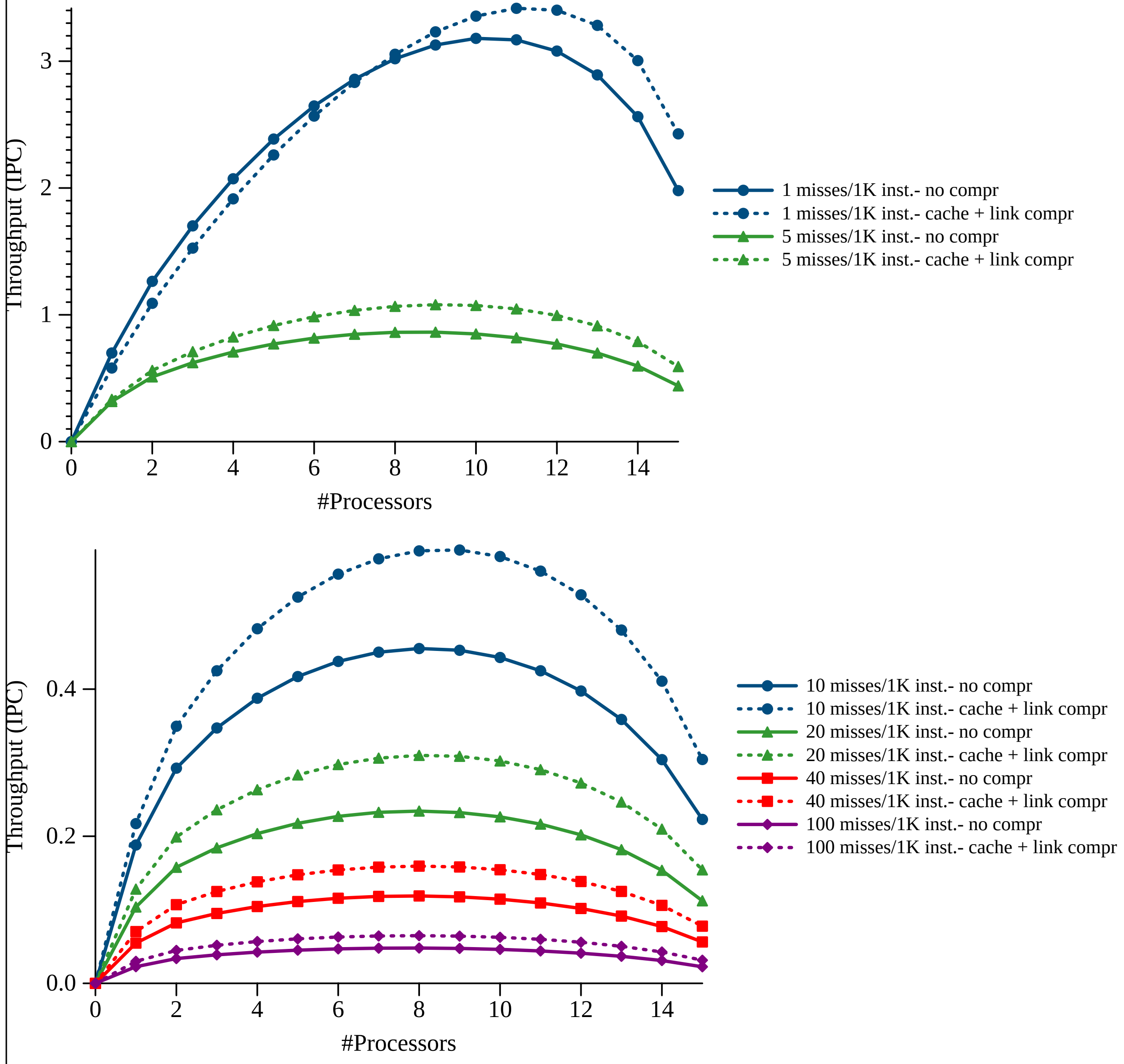


FIGURE 7-3. Analytical model sensitivity to L2 cache miss rates of an 8 MB cache

Since the scale widely varies, we show the low miss rates in the top graph and the high miss rates in the bottom graph. Non-compressed configurations are represented by solid lines, and compressed configurations are represented by dotted lines.

- When the miss rate is low (1-5 misses per 1000 instructions), compression has a small impact on performance for configurations with a small number of processors. Compression even slows down performance due to the decompression overhead for some configurations. However, compression still achieves non-trivial throughput improvements at the optimal design point (e.g., 7.5% for the 1 miss/1000 instructions configuration). Moreover, compression shifts the optimal configuration towards more cores (e.g., 11 cores vs. 10 for the 1 miss/1000 instructions configuration).
- When the miss rate is high, the impact of compression becomes more significant. Compression achieves large throughput improvements since it reduces both cache miss rate and pin bandwidth utilization. For the 20-100 misses/1000 instruction systems, compression achieves a 33-35% improvement in throughput compared to uncompressed systems. However, the optimal configuration does not change for these systems.

7.6.3 Sensitivity to Memory Latency

Figure 7-4 shows the impact of memory latency on throughput for our hypothetical benchmark. We vary the memory latency between 200 and 800 cycles. We make the following observations:

- The impact of compression on throughput is almost the same on both extremes (31% improvement for the 200-cycle system and 30% for the 800-cycle system). Two main factors contribute to the improvement in throughput. First, cache compression avoids cache misses, which reduces the impact of memory latency on throughput. Second, cache and link compression reduce pin bandwidth utilization, thus reducing link latency and improving IPC. For slower memory access latencies, the first factor is more significant. For faster memory access latencies, the increase in IPC increases pin bandwidth utilization so the second factor contributes more to throughput improvement.

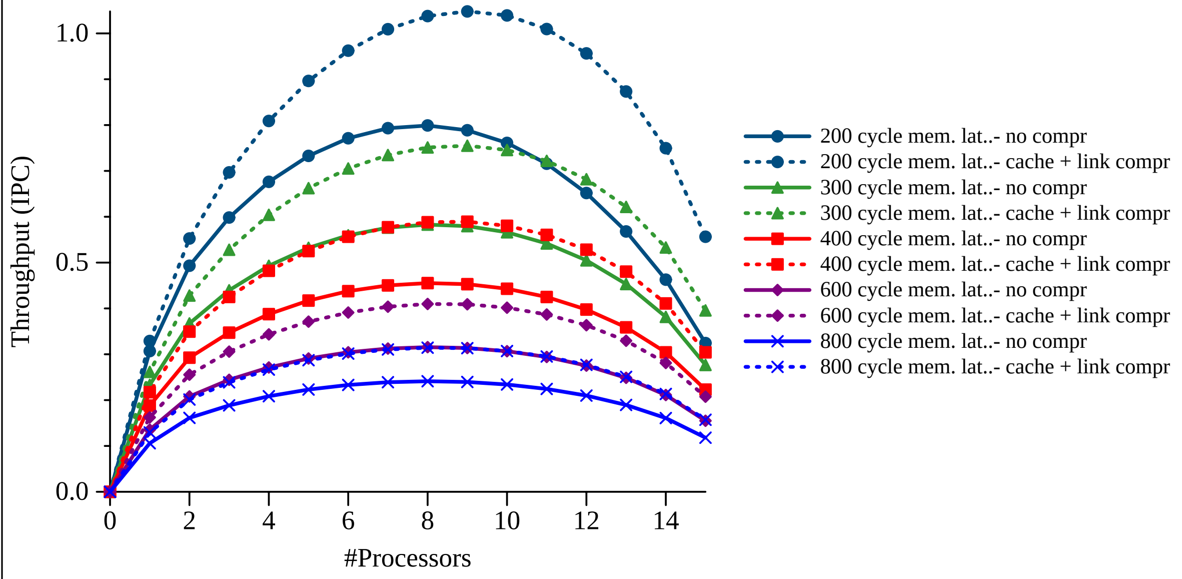


FIGURE 7-4. Analytical model sensitivity to memory latency. Non-compressed configurations are represented by solid lines, and compressed configurations are represented by dotted lines

- For all systems, cache and link compression shift the optimal design point from eight to nine processor cores. This is consistent with the observations in the previous sections.

7.6.4 Sensitivity to Compression Ratio

The success of cache and link compression is greatly dependent on compression ratio (i.e., the ratio between the compressed and uncompressed effective cache sizes). Figure 7-5 shows the impact of compression ratio on throughput with all other model parameters fixed. We compared an uncompressed system with systems of compression ratios between 1.1 and 2.0. As expected, throughput increases when compression ratios are higher. Throughput improvements range from 3% for a 1.1 compression ratio to 38% for a 2.0 compression ratio. In addition, the optimal design point shifts from eight to nine processors for the compression ratios of 1.5 or higher.

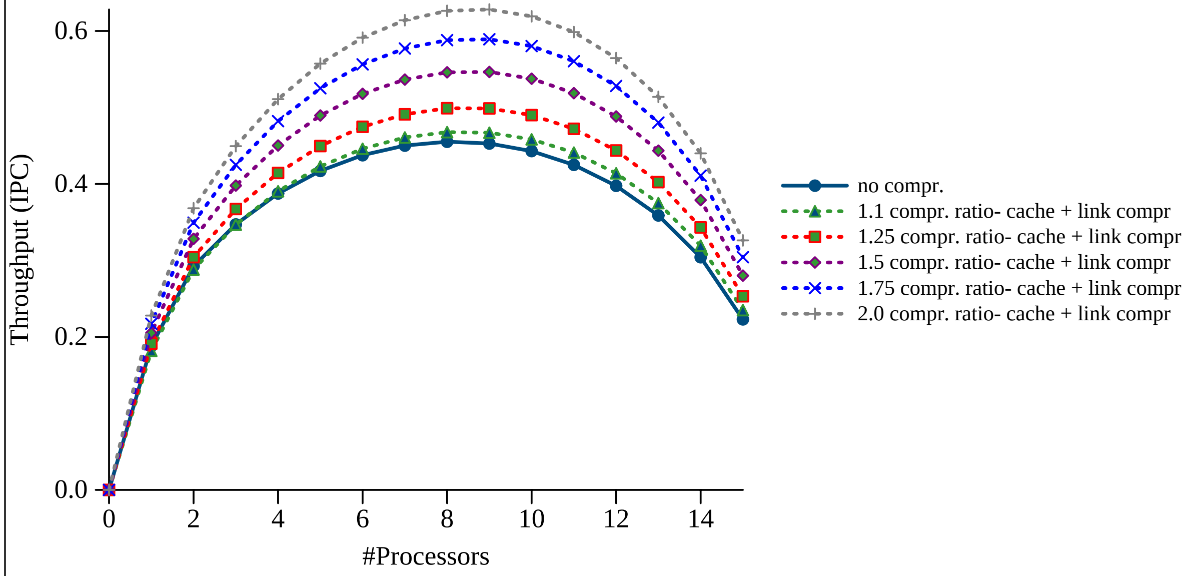


FIGURE 7-5. Analytical model sensitivity to compression ratio. No compression is compared to configurations of compression ratios 1.1, 1.25, 1.5, 1.75 and 2.0

7.6.5 Sensitivity to Decompression Penalty

As cache hit rates increase, the compression overhead per instruction due to accessing compressed lines (i.e., decompression penalty *dp* in our model) also increases. Therefore, we expect throughput improvements due to compression to decrease as decompression penalties increase. Figure 7-6 shows the impact of the decompression penalty on throughput with all other model parameters fixed. We vary the decompression penalty between 0.0 and 2.0. A decompression penalty of 0.0 represents an unrealistic best case where all decompression overheads are hidden from the L1 cache. A decompression penalty of 2.0 represents a pessimistic worst case where all load and store instructions (approximately 40% of all instructions) miss in the L1 cache and hit to compressed lines in the L2 cache, therefore incurring an additional penalty of 5 cycles for each load or store (or 5 * 40% = 2 cycles per instruction). We make the following observations:

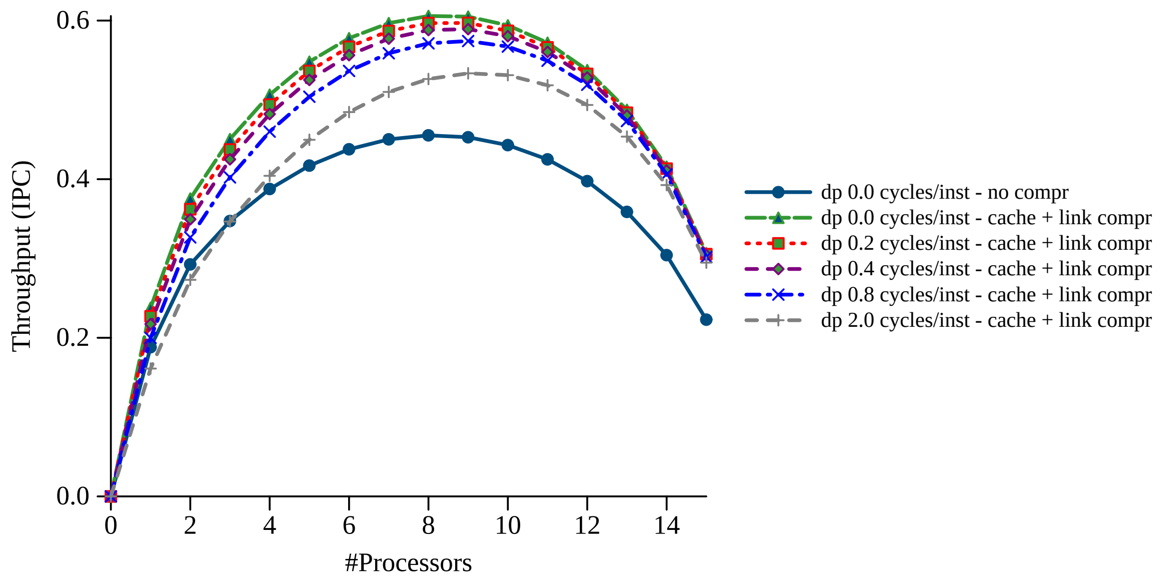


FIGURE 7-6. Analytical model sensitivity to decompression penalty. No compression is compared to compressed configurations of decompression penalties of 0.0, 0.2, 0.4, 0.8 and 2.0 cycles per instruction

- Small decompression penalties (0.4 cycles per instruction or less) only slightly increase throughput for our hypothetical benchmark. The maximum throughput for a perfect decompression overhead of zero cycles per instruction is only 3% higher than that of a system with 0.4 decompression cycles per instruction.
- For the worst case configuration (a decompression penalty of 2.0 cycles per instruction), the maximum throughput is reduced by 14% compared to that of a zero-cycle decompression penalty. However, even for this unrealistic worst case configuration, the maximum throughput is still 17% higher than that of an uncompressed system.
- As with many other configurations, compression with decompression penalties of 0.2-2.0 shift the optimal design point from eight to nine processor cores per chip.

7.6.6 Sensitivity to Perfect CPI

Wider issue queues and wider pipelines can decrease the number of cycles per instruction for a perfect L2 system ($CPI_{perfectL2}$ in our model). On the other hand, less effective L1 caches can increase the perfect CPI estimate. Figure 7-7 shows how compression improves throughput for different $CPI_{perfectL2}$ values between 0.25 and 3.0. The model does not take into account techniques that decrease the impact of cache misses on performance (e.g., runahead execution [37]). This figure shows that throughput improvement due to compression slightly increases when $CPI_{perfectL2}$ decreases. Throughput improves over an uncompressed system by 26% for a $CPI_{perfectL2}$ of 3.0, and by 31% for a $CPI_{perfectL2}$ of 0.25. This follows directly from the model because the relative impact of avoided cache misses on IPC increases (Eq. 7.13). In addition, the optimal design point shifts towards more cores for higher values of $CPI_{perfectL2}$.

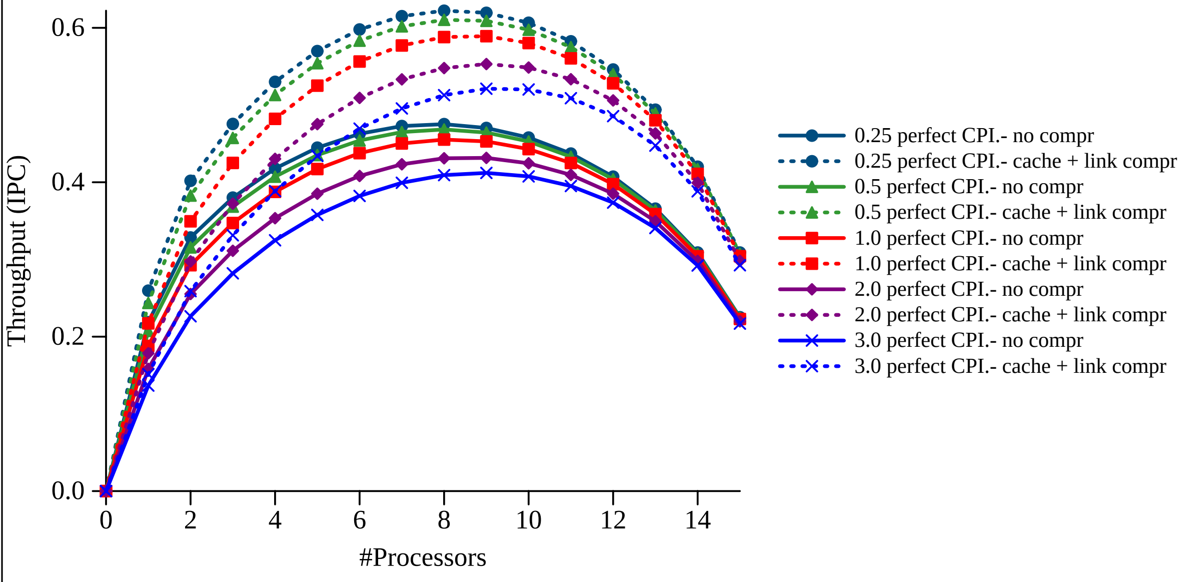


FIGURE 7-7. Analytical model sensitivity to perfect CPI. Non-compressed configurations are represented by solid lines, and compressed configurations are represented by dotted lines

7.7 CMP Model with Hardware Prefetching

Many current CMP systems implement hardware prefetching schemes to reduce cache misses or hide part of the memory latency. In order to model the impact of prefetching on throughput, we need to add two parameters to our simple model:

- Prefetching rate (*pfrate*) represents the number of prefetches issued per instruction. This directly affects the link latency, since it should be added to missrate in Eq. 7.16 to compute the link throughput:

$$\lambda = IPC(N) \cdot (Missrate(S_{L2p}) + pfrate) \tag{7.19}$$

- Prefetching avoided miss rate ($AvMissrate_{pf}(S_{L2})$) represents the difference between the number of misses per instruction when prefetching is implemented and the number of misses per instruction when prefetching is not implemented. This parameter can be positive or negative since prefetching can, in pathological cases, increase the cache miss rate. For simplicity, we assume that this missrate

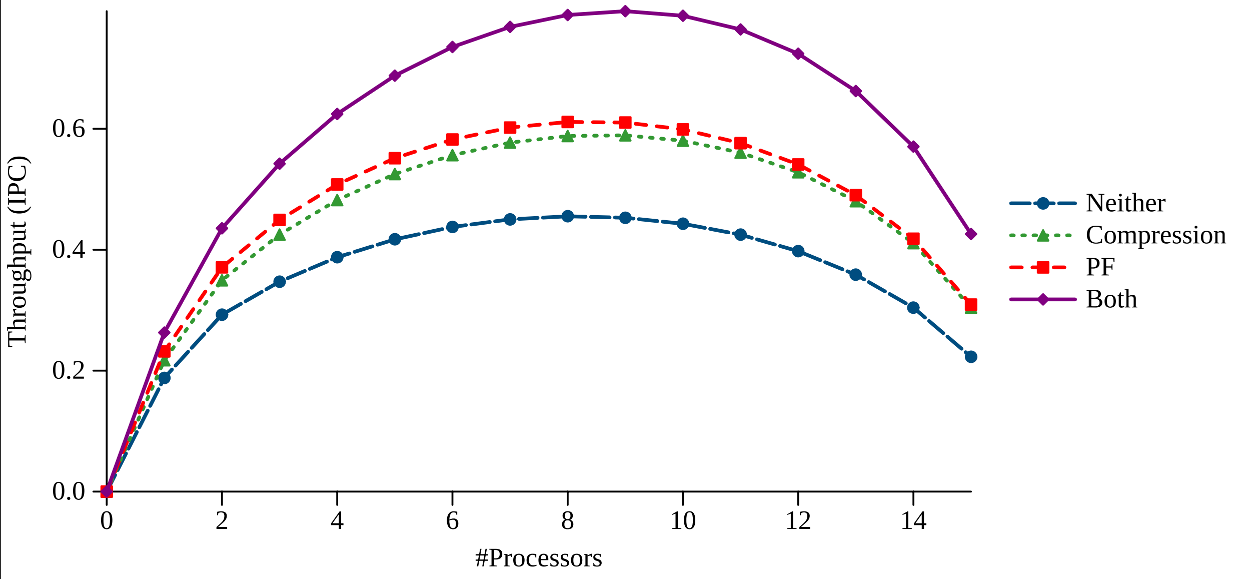


FIGURE 7-8. Analytical model results for four configurations: No compression or prefetching, compression only, prefetching only, and both

also follows the square root rule of thumb for regular misses. We also ignore the impact of partial prefetch hits. We assume that prefetching does not affect the average number of sharers per cache line ($sharers_{av}(N)$) or the CPI for a perfect L2 ($CPI_{perfectL2}$). This parameter should be subtracted from $missrate(S_{L2})$ in Eq. 7.11:

$$\alpha = (missrate(S_{L2}) - AvMissrate_{pf}(S_{L2})) \cdot \sqrt{\frac{A_{L2}}{A_{1MB}}} \tag{7.20}$$

To analyze the interaction between prefetching and compression, we show throughput (in terms of IPC) for systems with neither compression or prefetching, compression only, prefetching only, or both compression and prefetching in Figure 7-8. We assume our hypothetical benchmark has a *pfrate* of 15 prefetches per 1000 instructions and an avoided missrate of 3 per 1000 instructions. We observe from Figure 7-8 that prefetching alone increases the maximum throughput by 34%, and compression alone increases the maximum throughput by 29%. We also observe that the combination of prefetching and compression improves the maximum throughput by 75% with a positive interaction coefficient of 0.4% between prefetching and compression (as defined in Chapter 6). Prefetching alone does not shift the optimal design point. On the other hand, compression shifts the optimal point towards more cores compared to an uncompressed system, regardless of whether prefetching was implemented or not.

7.8 Evaluation for Commercial Workloads

Chip multiprocessor design involves achieving the right balance between cores, caches and communications to achieve the best possible system throughput. Our analytical model provided some qualitative insight into this design space. With few cores that cannot support enough threads, cores become a bottleneck and degrade system throughput. With too many cores and smaller caches, caches and/or pin bandwidth become a bottleneck and also degrade system throughput. The optimal CMP design lies somewhere between these two extremes. Our analytical model provides a simple and fast method to evaluate many high-level design choices. However, we use many simplifying assumptions in our model that do not hold for real systems. In order to quantitatively evaluate the CMP design space, we use simulation of commercial workloads.

In this section, we evaluate the performance of different core/cache configurations that have the same equivalent area. We show that compression can have a significant impact on all configurations, and that link compression also has a significant impact on bandwidth-limited configurations. We focus on our four commercial benchmarks in this section. We next discuss our simulation and workload setup.

7.8.1 Simulation Setup

The objective of our simulation experiments is to obtain more accurate estimates of throughput for commercial benchmarks compared to our analytical model. We measure throughput for commercial benchmarks using the number of transactions completed per billion cycles of runtime. We use the same base processor configuration of Chapter 6. However, our CMP design is different from that of Chapter 6 since we use a fixed-area processor model where each processor core and its associated L1 caches is area-equivalent to 0.5 MB of L2 cache area. The total chip area is the same as 8 MB of L2 cache.

Due to the time and space overhead of setting up simulation experiments, we only simulate configurations at two-processor increments. We simulate the following configurations represented as tuples of (number of processor cores, L2 cache size): (2, 7 MB), (4, 6 MB), (6, 5 MB), (8, 4 MB), (10, 3 MB), (12, 2 MB), (14, 1 MB).

In order to obtain a fair estimate of throughput for commercial workloads, we used the same number of users/threads for each workload across different processor configurations. We used the same commercial workloads described in Table 5-2, but we increased the number of users/threads we use for all processor configurations as follows:

OLTP. We used 256 total users for all processor configurations.

SPECjbb. We used 24 warehouses for all processor configurations.

Apache and Zeus. We used 6000 threads for all processor configurations.

We simulated our four benchmarks for all seven processor configurations and obtained estimates of throughput for each configuration[1]. Throughput for a real setup of these benchmarks will be dependent on the number of users/threads. However, we anticipate that the qualitative analysis (e.g., the shape of the throughput graphs) will remain the same.

7.8.2 Balanced CMP Design with Compression and Prefetching

With unlimited bandwidth, a CMP design has to divide the chip area between caches and cores to achieve the best throughput. As we showed in our analytical model, configurations with very few cores have fewer threads to run, while configurations with too many cores have higher miss rates. Both extremes can significantly limit throughput. In Figure 7-9, we show the impact of cores and caches on the throughput of commercial benchmarks for different compression and prefetching configurations (no compression or prefetching, prefetching only, compression only, and both compression and prefetching). We also show the impact on utilized bandwidth in Figure 7-10. The left most point in each line is an artificial point that corresponds to no cores and the whole chip composed of caches. We did not simulate the other extreme (all cores, no caches) because of the prohibitive simulation time, but its throughput would be very close to zero. We make the following observations:

- With no compression, adding more cores improves throughput up to a certain point for all benchmarks: 10p for zeus and oltp, and 12p for apache and jbb. Speedups for these optimal configurations over the 2-processor configurations were 4.3x, 3.7x, 1.8x, and 3.2x for apache, zeus, oltp and jbb, respectively.

1. We do not show results for apache on 10 processors since it showed some abnormal behavior that is not similar to the other configurations.

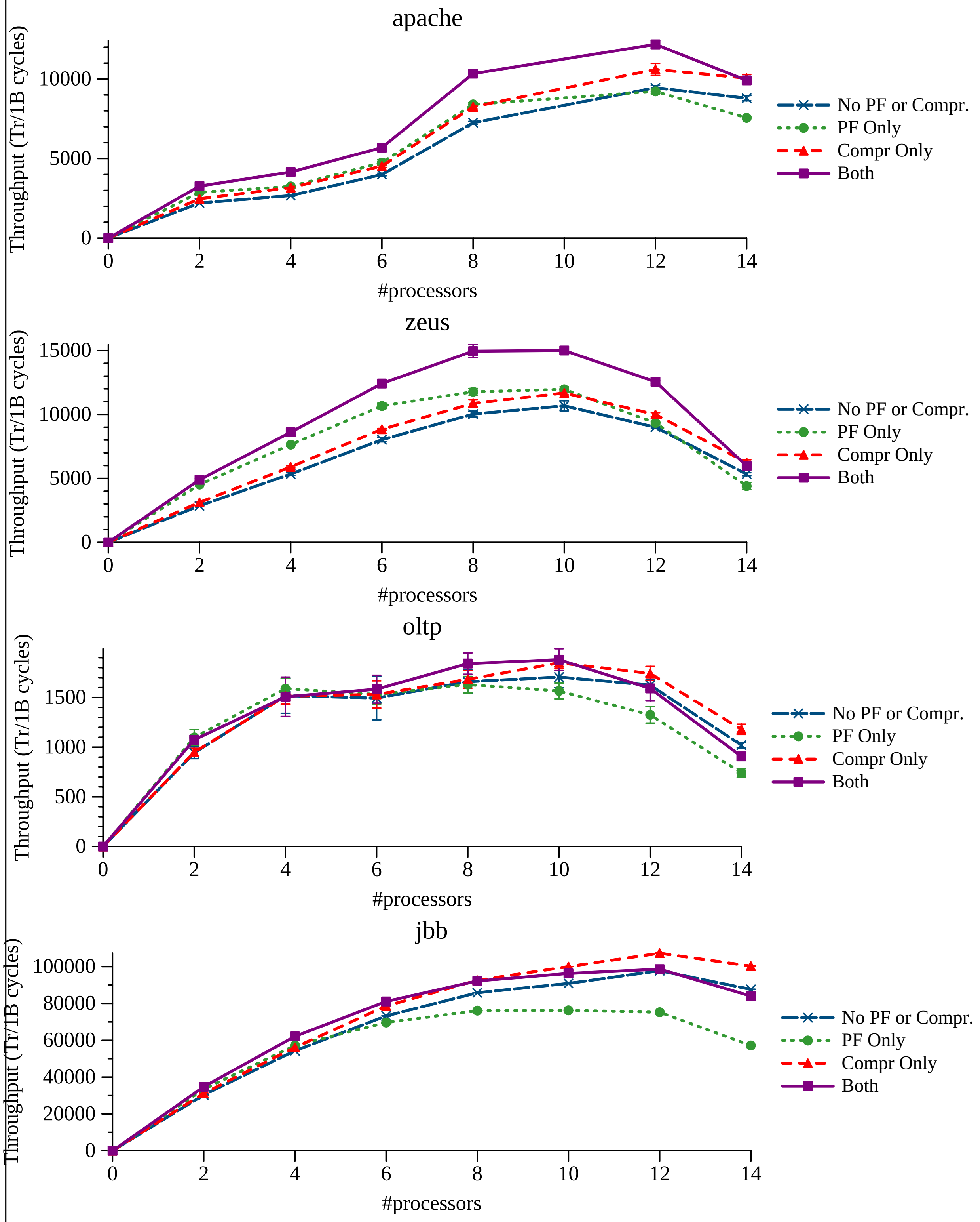


FIGURE 7-9. Commercial workload throughout for different compression and prefetching configurations. All Processor/cache configurations have a 20 GB/sec. pin bandwidth

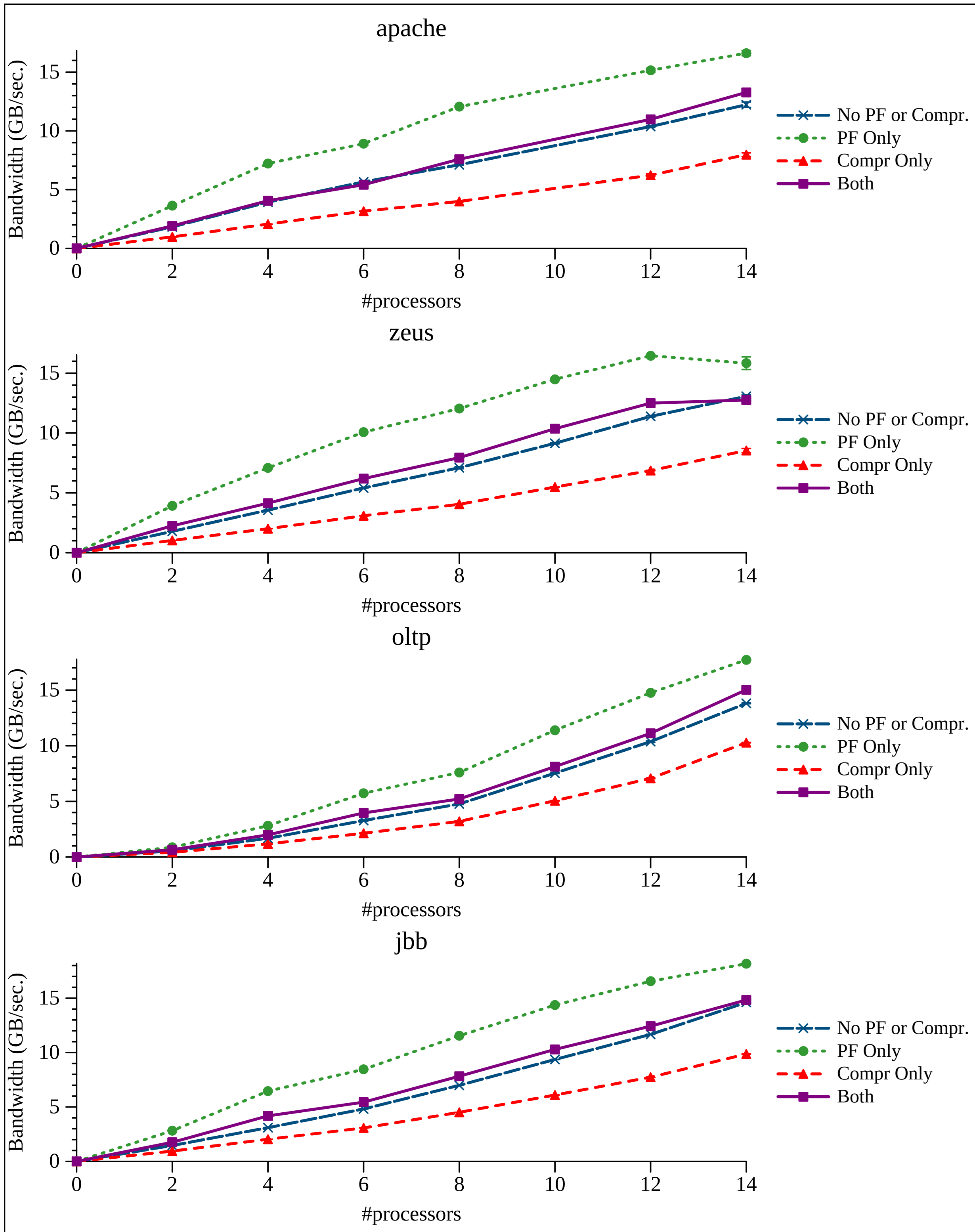


FIGURE 7-10. Utilized bandwidth for different compression and prefetching configurations of commercial workloads. All Processor/cache configurations have a 20 GB/sec. available pin bandwidth

- Compression alone helps all workloads and configurations achieve up to 13% improvement in throughput. This is mostly due to the impact of cache compression. At 20 GB/sec. pin bandwidth, link compression does not provide any significant impact on throughput except for the extreme configuration of 14 processors /1 MB L2 cache. Speedups for the optimal compression configurations over the 2-processor baseline with no compression or prefetching were 4.8x, 4.1x, 1.95x, and 3.5x for apache, zeus, oltp and jbb, respectively.
- Prefetching alone helps many workloads and configurations achieve up to 33% throughput improvement. However, for some benchmarks where prefetching is not effective (e.g., jbb), prefetching reduces throughput by up to 35%. In addition, prefetching reduces throughput for configurations with small cache sizes by up to 28% due to the increase in pin bandwidth demand. Speedups for the optimal compression configurations over the 2-processor baseline were 4.2x, 4.2x, 1.7x, and 2.5x for apache, zeus, oltp and jbb, respectively.
- Prefetching alone increases utilized pin bandwidth for all configurations by up to 44% (Figure 7-10). For some configurations with small cache sizes, the increase in pin bandwidth demand due to prefetching utilizes more than 90% of the available pin bandwidth. Compression alone decreases pin bandwidth utilization by up to 40%. When both compression and prefetching are implemented, utilized pin bandwidth is almost the same as the utilized pin bandwidth when neither is implemented.
- The combination of prefetching and compression provides significant throughput improvements compared to the two base cases (no compression or prefetching, and prefetching alone). Throughput improvement can be up to 49% compared to the base case of no compression or prefetching, and up to 27% compared to the base case of prefetching alone. Speedups for the optimal compression configurations over the 2-processor baseline with no compression or prefetching were 5.5x, 5.3x, 2.0x, and 3.3x for apache, zeus, oltp and jbb, respectively.

- The optimal design point for each benchmark does not shift towards more cores when compression is implemented. We attribute this to the fact that we are simulating throughput for different configurations at 2-processor increments, whereas our analytical model showed smaller differences. In many cases, compression only shifted the optimal design point by one processor.
- Prefetching and compression interact positivity for most configurations. We show the interaction coefficient of all benchmarks and configurations in Figure 7-11. The positive interaction coefficients can be up to 28% due to many of the same factors discussed in Chapter 6. The two schemes interact negatively only for a few configurations (2p zeus, 2p and 4p oltp). However, all negative interaction coefficients are less than 4%.

7.8.3 Impact of Limited Bandwidth

For bandwidth-limited systems, a CMP design has to balance the three C's—cores, caches, and communication—such that none is the sole bottleneck. For such a bandwidth-limited system, adding more cores

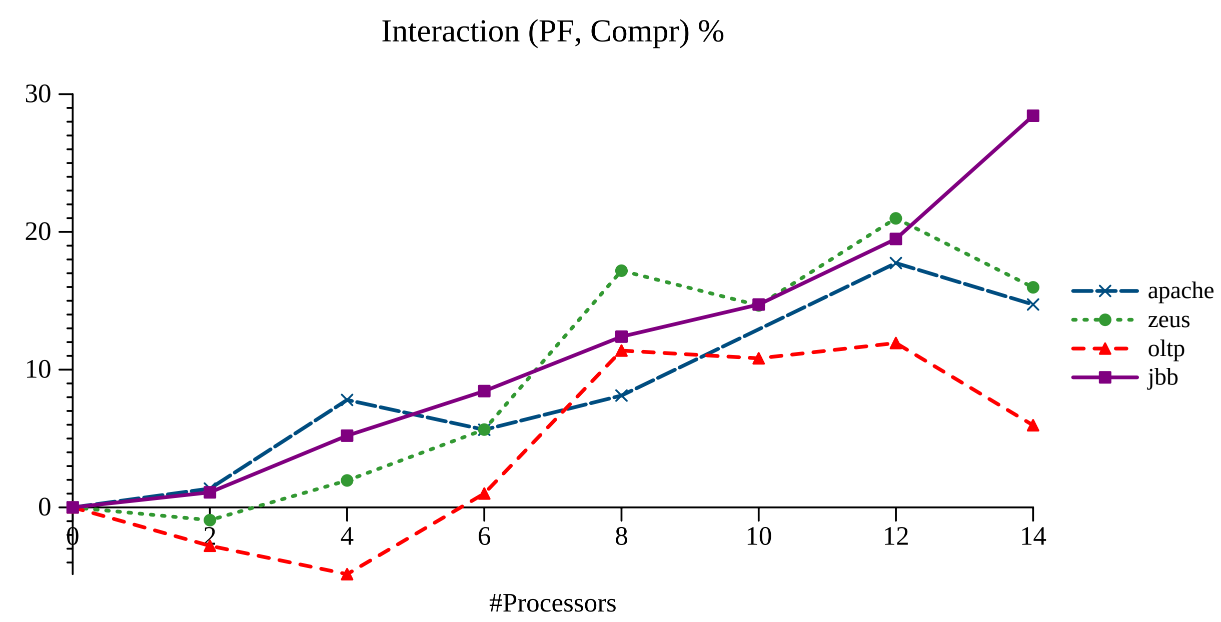


FIGURE 7-11. Interaction between compression and prefetching for all benchmarks and processor configurations

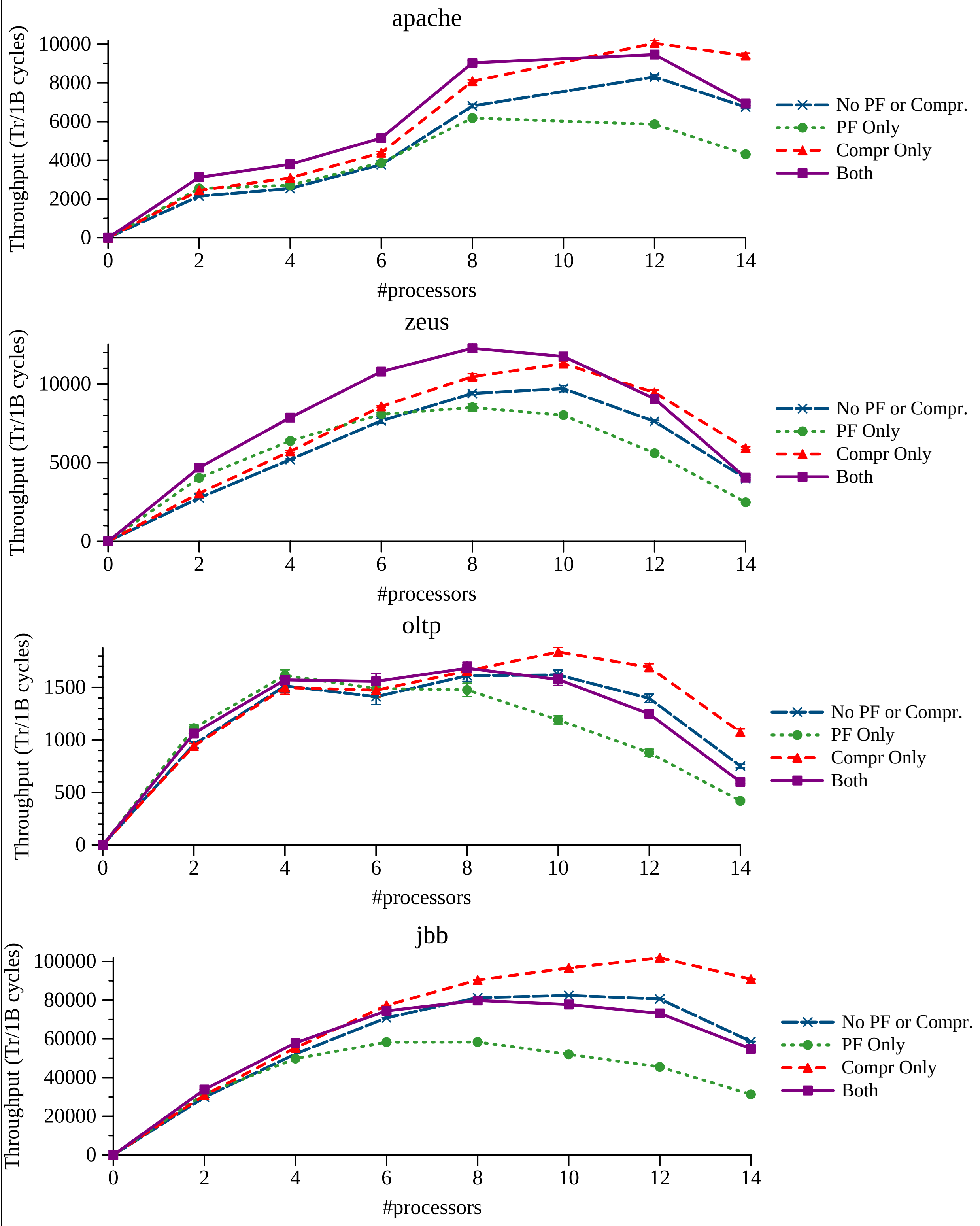


FIGURE 7-12. Commercial workload throughout for different compression schemes and processor configurations. All configurations have a 10 GB/sec. pin bandwidth

also improves throughput until bandwidth becomes a critical resource. For our four commercial benchmarks, the bandwidth demand increases with more cores and fewer caches.

Figure 7-12 demonstrates the trade-off between cores, caches, and communication for our commercial benchmarks assuming a 10 GB/sec. pin bandwidth. We make the following observations:

- When the utilized bandwidth is low, the lines follow the same trends of Figure 7-9. However, as pin bandwidth becomes a bottleneck, many uncompressed configurations (with or without prefetching) suffer from degraded performance as the number of cores increases and cache sizes decrease.
- Compression alone improves performance for all configurations by up to 25% compared to configurations without compression or prefetching that have the same number of cores. For this bandwidth-limited system, the optimal design point is also shifted towards more cores for jbb. When combined with prefetching, the two schemes interact positively. Such positive interaction is mostly caused by bandwidth savings due to compression that alleviate some of prefetching's extra bandwidth demand. The optimal configuration of both compression and prefetching has up to a 37% higher throughput compared to the optimal configuration of prefetching alone (since jbb has a slowdown due to prefetching).
- The optimal configuration with no compression or prefetching is at 10 cores for zeus, oltp and jbb; and 12 for apache cores. The optimal design point shifts towards more cores for some benchmarks with compression alone (e.g., jbb where it shifts to 12 processors). When compared with the base case of only prefetching, compression shifts the optimal design point towards more cores for all benchmarks.
- Prefetching and compression interact positivity for most configurations. We show the interaction coefficient of all benchmarks and configurations in Figure 7-13. The positive interaction coefficients can be up to 33%, and are higher for most configurations than the coefficients for 20 GB/sec. bandwidth. The two schemes interact negatively only for 2p and 4p oltp, with negative interaction coefficients below 3%.

- We note from Figure 7-13 that the interaction coefficients decrease for the 14-processor configuration for all benchmarks. Since such configuration has a small cache (1 MB), miss rates are high and bandwidth is constrained even in the absence of prefetching. With prefetching, link compression cannot reduce bandwidth demand to the point where demand is no longer a bottleneck. This decreases the positive interaction coefficients between compression and prefetching.

7.9 Model Validation

We used our simulation experiments in the previous section to validate results from our analytical model. We extracted model parameters from our simulation experiments. We used the model to estimate the number of transactions per billion cycles for each of our benchmarks using the model's IPC:

$$TransactionsPerBillionCycles = \frac{IPC \times 10^9}{InstructionsPerTransaction} \tag{7.21}$$

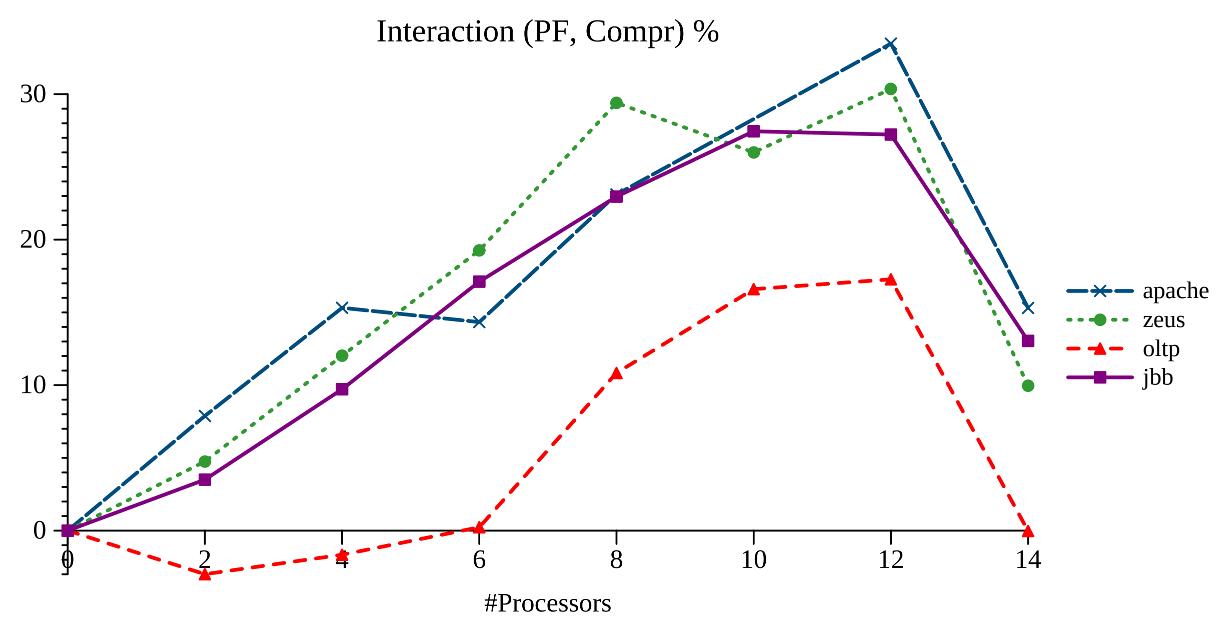


FIGURE 7-13. Interaction between compression and prefetching for 10 GB/sec. pin bandwidth

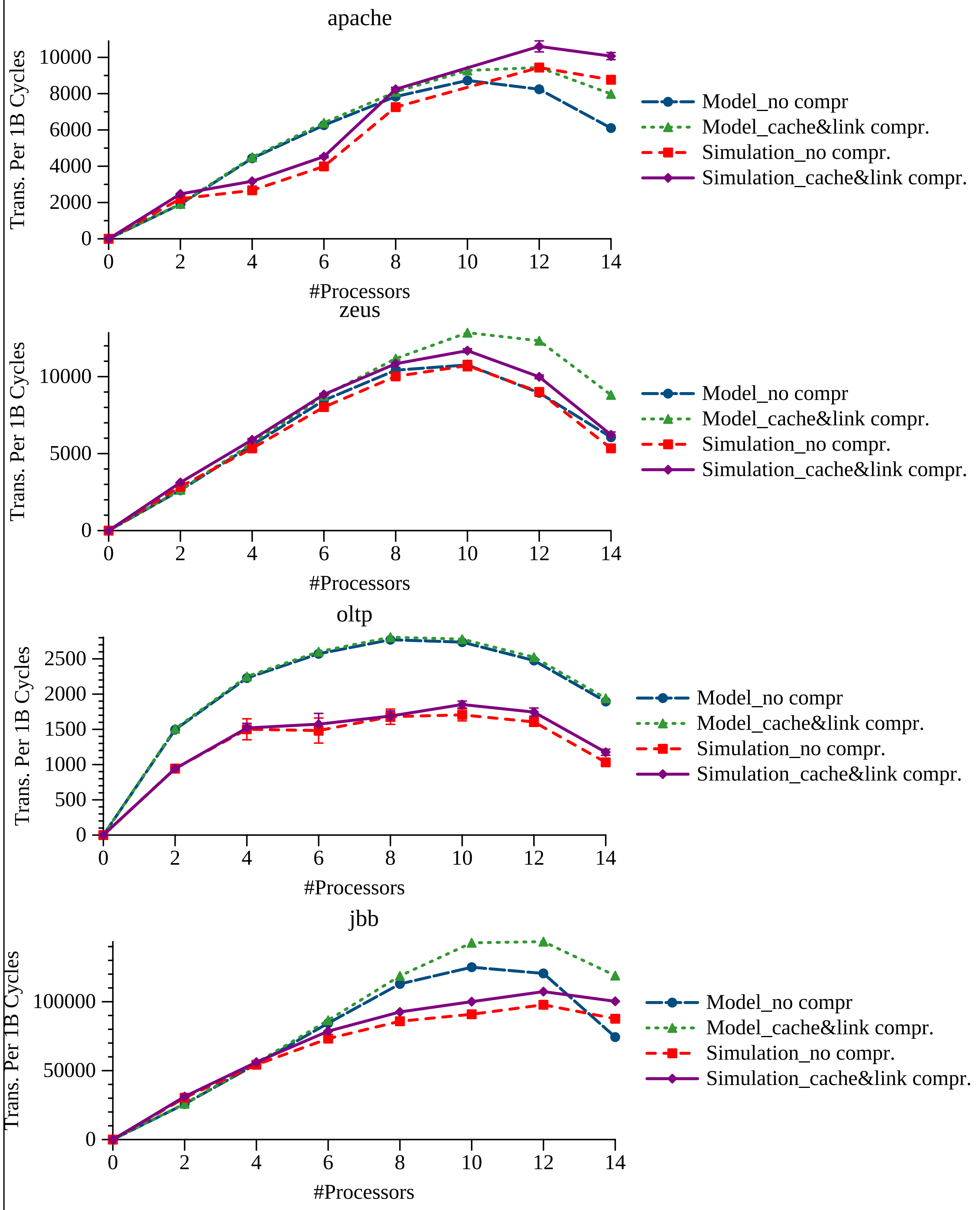


FIGURE 7-14. Comparing throughput estimates from analytical model and simulations

Throughput is shown in transactions per 1 billion cycles for all benchmarks.

We estimated performance for our four commercial benchmarks for the base case and for when cache and link compression are implemented. We compare model and simulation results in Figure 7-14. This figure demonstrates the following:

- The analytical model can successfully predict the general trend in throughput improvements for our workloads. The general shape of the throughput curve is similar for the analytical model and simulation.
- The relative error in the model's throughput estimates compared to simulation is quite high, as much as 43%, 29%, 46%, and 30% for apache, zeus, oltp, and jbb, respectively.
- The figure shows that except for apache, the model has errors in a single direction by over-estimating throughput. For apache, the model under-estimated throughput for some configurations as well. This is caused by the model's many simplifying assumptions (Section 7.4).
- The model successfully predicted the optimal configuration for zeus with and without compression. It also predicted the optimal configuration for apache and jbb with compression. For other benchmarks and configurations, we note that the model's predicted optimal configuration differs by 2-processors from simulation results. We note, however, that there is only a small difference in throughput (as measured by simulations) between the model's optimal and the simulation's optimal configuration.

In summary, our simple analytical model provided some qualitative estimates that can be used for initial exploration and to gain intuition into CMP design trade-offs. However, a more accurate model should be developed to predict throughput with more accuracy.

7.10 Related Work

There has been some work on chip design space that is related to our study. Farrens, et al., explored how to utilize a large number of transistors (by 1994 standards) and allocate them to processors, instruction caches and data caches. Using trace-based simulation, they found that a balance needs to be achieved between instruction and data caches, and that more processors can be placed on-chip to improve performance [41]. They defined the equivalent cache transistor (ECT) metric as a relative area metric between processor cores and caches.

Huh, et al., explored the CMP design space and tried to answer several questions: How many cores future CMPs will have, whether cores will be in-order or out-of-order, and how much cache area will there be in future CMPs [62]. They also pointed out pin bandwidth as being a potential limiting factor for CMP performance. They pointed out that the number of transistors per pin is increasing at an exponential rate, which they projected would force designers to increase the area allocated to on-chip caches at the expense of processor cores. In our design, this is not necessarily the case since compression shifts the balance point by increasing the effective cache size without significantly increasing its area.

Davis, et al., studied the throughput of area-equivalent multi-threaded chip multiprocessors (CMT) using simulation of throughput-oriented commercial applications [33]. They demonstrated that “mediocre” cores (i.e., small, simple cores with small caches and low performance) maximize the total number of CMT cores and outperform CMTs built from larger, higher performance cores. They also showed that throughput increases with the number of threads till it reaches a maximum and then degrades due to pipeline saturation or bandwidth saturation. Such behavior is similar to our throughput curves.

Li, et al., used uniprocessor traces of SPEC2000 benchmarks to estimate CMP throughput [85]. They assert that thermal constraints dominate other physical constraints in CMP design such as pin-bandwidth and power delivery. However, their work did not consider multi-threaded or commercial benchmarks.

7.11 Summary

In this chapter, we examined how to design CMP systems that balance cores, caches and communication resources. We discussed the role that compression plays in shifting this balance. We developed a simple analytical model that measures throughput for different CMP configurations with a fixed area budget. We used this model and simulation experiments to qualitatively and quantitatively show that compression can significantly improve CMP throughput (by 25% or more for some configurations). We showed that compression improves CMP throughput over a wide range of system parameters. We demonstrated that compression interacts positively with hardware prefetching, leading to improvements in throughput that are significantly higher than expected from the product of throughput improvements of either scheme alone. We show that compression can sometimes shift the optimal design point towards more processor cores, potentially leading to more core-centric CMP designs.

Chapter 8

Summary

In this chapter, we summarize the conclusions of this dissertation, and discuss some possible areas of future research.

8.1 Conclusions

Chip multiprocessors (CMPs) combine multiple processors on a single die. The increasing number of processor cores on a single chip increases the demand on the shared L2 cache capacity and the off-chip pin bandwidth. Demand on these critical resources can be further exacerbated by latency-hiding techniques such as hardware prefetching. In this dissertation, we explored using compression to effectively increase cache and pin bandwidth resources and ultimately CMP performance.

Cache compression stores compressed lines in the cache, potentially increasing the effective cache size, reducing off-chip misses, and improving performance. On the downside, decompression overhead can slow down cache hit latencies, possibly degrading performance. Link compression compresses communication messages before sending to or receiving from off-chip system components, thereby increasing the effective off-chip pin bandwidth, reducing contention and improving performance for bandwidth-limited configurations. While compression can have a positive impact on CMP performance, practical implementations of compression raise a few concerns (e.g., compression's overhead and its interaction with prefetching).

In this dissertation, we made several contributions that address concerns about different aspects of implementing compression. We proposed a compressed L2 cache design based on a simple compression scheme with a low decompression overhead. Such design can double the effective cache size for many benchmarks. We developed an adaptive compression algorithm that dynamically adapts to the costs and benefits of cache compression, and uses compression only when it helps performance.

We showed that cache and link compression both combine to improve CMP performance for commercial and scientific workloads by 3-20%. We demonstrated that compression interacts in a strong positive way with hardware prefetching, whereby a system that implements both compression and hardware prefetching outperforms systems that implement one scheme and not the other. Furthermore, the speedup due to both compression and prefetching (10-51% for all but one of our benchmarks) can be significantly higher than the product of the speedups of either scheme alone. We presented a simple analytical model that helps provide qualitative intuition into the trade-off between cores, caches, communication, and compression, and we used full-system simulation to quantify this trade-off for a set of commercial workloads.

8.2 Future Work

Opportunities for future research exist in cache and link compression, since this dissertation has not exhausted either of these areas. We next outline a few possible areas of future research.

Power. We did not study the full impact of cache and link compression on CMP power. On the one hand, we anticipate that compression can significantly reduce power by avoiding misses and reducing communication bandwidth. On the other hand, it can have an adverse effect on core power consumption due to compression and decompression of cache lines, as well as cache set repacking. We predict that compression can have a significant impact on power reduction, but such prediction merits further investigation.

Compression in multi-CMP systems. In this dissertation, we only studied the impact of compression on a single CMP. In a multi-CMP system, compression and decompression overheads will be added to inter-chip communication. Our adaptive compression scheme predicts whether compression is beneficial on a single chip, and is therefore not optimized for inter-chip communication. While we anticipate that more “optimal” predictors will not significantly change on-chip predictions regarding compression, such solutions need to be studied further.

Use of extra tags. In our decoupled variable-segment cache design, we support having more tags per cache set than existing lines. We use these extra tags to save information about compressed lines. We also use these tags to classify cache accesses and use this information to update our compression predictor. However, these extra tags can be used for many additional purposes. For example, they can be used to complement and improve cache replacement algorithms, classify useful and harmful prefetches, and trigger requests in speculative coherence protocols. Previous work has explored uses of additional tags, but the interactions of such uses with compression has not been studied.

Analytical CMP models. In this dissertation, we showed that a simple analytical model can provide intuition into CMP design. However, the many approximations we made in our model make it unsuitable to accurately predict throughput. Future research can explore analytical models with different levels of complexity to more accurately predict CMP throughput, a speed vs. accuracy trade-off.

The above areas do not present an exhaustive list of future related research. We hope that our work will enhance and motivate research in these and other areas.

References

[1] Bulent Abali, Hubertus Franke, Dan E. Poff, Jr. Robert A. Saccone, Charles O. Schulz, Lorraine M. Herger, and T. Basil Smith. Memory Expansion Technology (MXT): Software Support and Performance. *IBM Journal of Research and Development*, 45(2):287–301, March 2001.

[2] Bulent Abali, Hubertus Franke, Xiaowei Shen, Dan E. Poff, and T. Basil Smith. Performance of hardware compressed main memory. In *Proceedings of the Seventh IEEE Symposium on High-Performance Computer Architecture*, pages 73–81, January 2001.

[3] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104–114, May 1990.

[4] Edward Ahn, Seung-Moon Yoo, and Sung-Mo Steve Kang. Effective Algorithms for Cache-level Compression. In *Proceedings of the 2001 Conference on Great Lakes Symposium on VLSI*, pages 89–92, 2001.

[5] Haitham Akkary and Michael A. Driscoll. A Dynamic Multithreading Processor. In *Proceedings of the 31st Annual IEEE/ACM International Symposium on Microarchitecture*, pages 226–236, November 1998.

[6] Alaa R. Alameldeen, Milo M. K. Martin, Carl J. Mauer, Kevin E. Moore, Min Xu, Daniel J. Sorin, Mark D. Hill, and David A. Wood. Simulating a $2M Commercial Server on a $2K PC. *IEEE Computer*, 36(2):50–57, February 2003.

[7] Alaa R. Alameldeen, Carl J. Mauer, Min Xu, Pacia J. Harper, Milo M. K. Martin, Daniel J. Sorin, Mark D. Hill, and David A. Wood. Evaluating Non-deterministic Multi-threaded Commercial Workloads. In *Proceedings of the Fifth Workshop on Computer Architecture Evaluation Using Commercial Workloads*, pages 30–38, February 2002.

[8] Alaa R. Alameldeen and David A. Wood. Variability in Architectural Simulations of Multi-threaded Workloads. In *Proceedings of the Ninth IEEE Symposium on High-Performance Computer Architecture*, pages 7–18, February 2003.

[9] Alaa R. Alameldeen and David A. Wood. Adaptive Cache Compression for High-Performance Processors. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 212–223, June 2004.

[10] Alaa R. Alameldeen and David A. Wood. Frequent Pattern Compression: A Significance-Based Compression Scheme for L2 Caches. Technical Report 1500, Computer Sciences Department, University of Wisconsin–Madison, April 2004.

[11] Guido Araujo, Paulo Centoducatte, Mario Cartes, and Ricardo Pannain. Code Compression Based on Operand Factorization. In *Proceedings of the 31st Annual IEEE/ACM International Symposium on Microarchitecture*, pages 194–201, November 1998.

[12] Vishal Aslot, Max Domeika, Rudolf Eigenmann, Greg Gaertner, Wesley Jones, and Bodo Parady. SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. In *Workshop on OpenMP Applications and Tools*, pages 1–10, July 2001.

[13] Paul Barford and Mark Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *Proceedings of the 1998 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 151–160, June 1998.

[14] Luiz A. Barroso, Kourosh Gharachorloo, and Edouard Bugnion. Memory System Characterization of Commercial Workloads. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 3–14, June 1998.

[15] Luiz Andre Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, and Ben Verghese. Piranha: A Scalable Architec-

ture Based on Single-Chip Multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 282–293, June 2000.

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

[17] Bradford M. Beckmann and David A. Wood. Managing Wire Delay in Large Chip-Multiprocessor Caches. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, December 2004.

[18] Luca Benini, Davide Bruni, Alberto Macii, and Enrico Macii. Hardware-Assisted Data Compression for Energy Minimization in Systems with Embedded Processors. In *Proceedings of the IEEE 2002 Design Automation and Test in Europe*, pages 449–453, 2002.

[19] Luca Benini, Davide Bruni, Bruno Ricco, Alberto Macii, and Enrico Macii. An Adaptive Data Compression Scheme for Memory Traffic Minimization in Processor-Based Systems. In *Proceedings of the IEEE International Conference on Circuits and Systems, ICCAS-02*, pages 866–869, May 2002.

[20] A. F. Benner, M. Ignatowski, J. A. Kash, D. M. Kuchta, and M. B. Ritter. Exploitation of Optical Interconnects in Future Server Architectures. *IBM Journal of Research and Development*, 49(4):755–775, 2005.

[21] William Bryg and Jerome Alabado. The UltraSPARC T1 Processor - High Bandwidth For Throughput Computing, December 2005. http://www.sun.com/processors/whitepapers/UST1_bw_v1.0.pdf.

[22] Doug Burger, James R. Goodman, and Alain Kagi. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23th Annual International Symposium on Computer Architecture*, pages 78–89, May 1996.

[23] Ramon Canal, Antonio Gonzalez, and James E. Smith. Very Low Power Pipelines Using Significance Compression. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 181–190, December 2000.

[24] David Chen, Enoch Peserico, and Larry Rudolph. A Dynamically Partitionable Compressed Cache. In *Proceedings of the Singapore-MIT Alliance Symposium*, January 2003.

[25] Tien-Fu Chen and Jean-Loup Baer. A Performance Study of Software and Hardware Data Prefetching Schemes. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 223–232, April 1994.

[26] Tien-Fu Chen and Jean-Loup Baer. Effective Hardware-Based Data Prefetching for High Performance Processors. *IEEE Transactions on Computers*, 44(5):609–623, May 1995.

[27] Daniel Citron. Exploiting Low Entropy to Reduce Wire Delay. *IEEE TCCA Computer Architecture Letters*, 3, January 2004.

[28] Daniel Citron and Larry Rudolph. Creating a Wider Bus Using Caching Techniques. In *Proceedings of the First IEEE Symposium on High-Performance Computer Architecture*, pages 90–99, February 1995.

[29] Thomas M. Conte, Sanjeev Banerjia, Sergei Y. Larin, Kishore N. Menezes, and Sumedh W. Sathaye. Instruction Fetch Mechanisms for VLIW Architectures with Compressed Encodings. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 201–211, December 1996.

[30] Toni Cortes, Yolanda Becerra, and Raul Cervera. Swap Compression: Resurrecting Old Ideas. *Software - Practice and Experience Journal*, 46(15):567–587, December 2000.

[31] Fredrik Dahlgren, Michel Dubois, and Per Stenström. Sequential Hardware Prefetching in Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 6(7):733–746, July 1995.

[32] William J. Dally and John W. Poulton. *Digital Systems Engineering*. Cambridge University Press, 1998.

[33] John D. Davis, James Laudon, and Kunle Olukotun. Maximizing CMP Throughput with Mediocre Cores. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 51–62, September 2005.

[34] Fred Douglis. The Compression Cache: Using On-line Compression to Extend Physical Memory. In *Proceedings of 1993 Winter USENIX Conference*, pages 519–529, January 1993.

[35] Karel Driesen and Urs Holzle. Accurate Indirect Branch Prediction. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 167–178, June 1998.

[36] Pradeep Dubey. A Platform 2015 Workload Model: Recognition, Mining and Synthesis Moves Computers to the Era of Tera, June 2005. ftp://download.intel.com/technology/computing/archinnov/platform2015/download/RMS.pdf.

[37] Jim Dundas and Trevor Mudge. Improving Data Cache Performance by Pre-Executing Instructions Under a Cache Miss. In *Proceedings of the 1997 International Conference on Supercomputing*, pages 68–75, July 1997.

[38] Avinoam N. Eden and Trevor Mudge. The YAGS Branch Prediction Scheme. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 69–77, June 1998.

[39] Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, and Dean M. Tullsen. Simultaneous Multithreading: A Platform for Next-generation Processors. *IEEE Micro*, 17(5):12–18, September/October 1997.

[40] Magnus Ekman and Per Stenstrom. A Robust Main-Memory Compression Scheme. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 74–85, June 2005.

[41] M. Farrens, G. Tyson, and A. R. Pleszkun. A Study of Single-Chip Processor/Cache Organizations for Large Numbers of Transistors. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 338 – 347, April 1994.

[42] Matthew Farrens and Arvin Park. Dynamic Base Register Caching: A Technique for Reducing Address Bus Width. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 128–137, May 1991.

[43] Wi fen Lin, Steven K. Reinhardt, and Doug Burger. Reducing DRAM Latencies with an Integrated Memory Hierarchy Design. In *Proceedings of the Seventh IEEE Symposium on High-Performance Computer Architecture*, pages 301–312, January 2001.

[44] Brian A. Fields, Rastislav Bodik, Mark D. Hill, and Chris J. Newburn. Using Interaction Costs for Microarchitectural Bottleneck Analysis. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 228–241, December 2003.

[45] International Technology Roadmap for Semiconductors. ITRS 2004 Update. Semiconductor Industry Association, 2004. http://www.itrs.net/Common/2004Update/2004Update.htm.

[46] P.A. Franaszek, P. Heidelberger, D.E. Poff, R.A. Saccone, and J.T. Robinson. Algorithms and Data Structures for Compressed-Memory Machines. *IBM Journal of Research and Development*, 45(2):245–258, March 2001.

[47] P.A. Franaszek and J.T. Robinson. On Internal Organization in Compressed Random-Access Memories. *IBM Journal of Research and Development*, 45(2):259–270, March 2001.

[48] Peter Franaszek, John Robinson, and Joy Thomas. Parallel Compression with Cooperative Dictionary Construction. In *Proceedings of the Data Compression Conference, DCC'96*, pages 200–209, March 1996.

[49] Michael J. Freedman. The Compression Cache: Virtual Memory Compression for Handheld Computers. Technical report, Parallel and Distributed Operating Systems Group, MIT Lab for Computer Science, Cambridge, 2000.

[50] Ilya Ganusov and Martin Burtscher. Future Execution: A Hardware Prefetching Technique for Chip Multiprocessors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 350–360, September 2005.

[51] Kourosh Gharachorloo, Madhu Sharma, Simon Steely, and Stephen Von Doren. Architecture and Design of AlphaServer GS320. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 13–24, November 2000.

[52] Gregory F. Grohoski. Machine Organization of the IBM RISC System/6000 Processor. *IBM Journal of Research and Development*, 34(1):37–58, January 1990.

[53] Jon Haas and Pete Vogt. Fully-Buffered DIMM Technology Moves Enterprise Platforms to the Next Level. *Technology@Intel Magazine*, March 2005.

[54] Erik G. Hallnor and Steven K. Reinhardt. A Fully Associative Software-Managed Cache Design. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 107–116, June 2000.

[55] Erik G. Hallnor and Steven K. Reinhardt. A Compressed Memory Hierarchy using an Indirect Index Cache. Technical Report CSE-TR-488-04, University of Michigan, 2004.

[56] Lance Hammond, Basem A. Nayfeh, and Kunle Olukotun. A Single-Chip Multiprocessor. *IEEE Computer*, 30(9):79–85, September 1997.

[57] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, third edition, 2003.

[58] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, February 2001.

[59] Tim Horel and Gary Lauterbach. UltraSPARC-III: Designing Third Generation 64-Bit Performance. *IEEE Micro*, 19(3):73–85, May/June 1999.

[60] M. S. Hrishikesh, Norman P. Jouppi, Keith I. Farkas, Doug Burger, Stephen W. Keckler, and Premkishore Shivakumar. The Optimal Logic Depth Per Pipeline Stage is 6 to 8 Inverter Delays. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, May 2002.

[61] David A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proc. Inst. Radio Engineers*, 40(9):1098–1101, September 1952.

[62] Jaehyuk Huh, Stephen W. Keckler, and Doug Burger. Exploring the Design Space of Future CMPs. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 199–210, 2001.

[63] Jaehyuk Huh, Changkyu Kim, Hazim Shafi, Lixin Zhang, Doug Burger, and Stephen W. Keckler. A NUCA Substrate for Flexible CMP Cache Sharing. In *Proceedings of the 19th International Conference on Supercomputing*, June 2005.

[64] Martin Isenburg, Peter Lindstrom, and Jack Snoeyink. Lossless Compression of Predicted Floating-Point Geometry. *Computer Aided Design*, 37(8):869–877, July 2005.

[65] Doug Joseph and Dirk Grunwald. Prefetching Using Markov Predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 252–263, June 1997.

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

[67] Stephan Jourdan, Tse-Hao Hsing, Jared Stark, and Yale N. Patt. The Effects of Mispredicted-Path Execution on Branch Prediction Structures. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 58–67, October 1996.

[68] Ron Kalla, Balaram Sinharoy, and Joel M. Tendler. IBM Power5 Chip: A Dual Core Multithreaded Processor. *IEEE Micro*, 24(2):40–47, Mar/Apr 2004.

[69] Krishna Kant and Ravi Iyer. Compressibility Characteristics of Address/Data transfers in Commercial Workloads. In *Proceedings of the Fifth Workshop on Computer Architecture Evaluation Using Commercial Workloads*, pages 59–67, February 2002.

[70] Tejas Karkhanis and James E. Smith. A First-Order Superscalar Processor Model. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 338–349, June 2004.

[71] R. E. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[72] Nam Sung Kim, Todd Austin, and Trevor Mudge. Low-Energy Data Cache Using Sign Compression and Cache Line Bisection. In *Second Annual Workshop on Memory Performance Issues (WMPI), in conjunction with ISCA-29*, 2002.

[73] Morten Kjelso, Mark Gooch, and Simon Jones. Design and Performance of a Main Memory Hardware Data Compressor. In *Proceedings of the 22nd EUROMICRO Conference*, 1996.

[74] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro*, 25(2):21–29, Mar/Apr 2005.

[75] David J. Kuck. Platform 2015 Software: Enabling Innovation in Parallelism for the Next Decade, June 2005. ftp://download.intel.com/technology/computing/archinnov/platform2015/download/Parallelism.pdf.

[76] Rakesh Kumar, Victor Zyuban, and Dean Tullsen. Interconnections in multi-core architectures: Understanding Mechanisms, Overheads and Scaling. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, June 2005.

[77] Jang-Soo Lee, Won-Kee Hong, and Shin-Dug Kim. Design and Evaluation of a Selective Compressed Memory System. In *Proceedings of Internationl Conference on Computer Design (ICCD'99)*, pages 184–191, October 1999.

[78] Jang-Soo Lee, Won-Kee Hong, and Shin-Dug Kim. An On-chip Cache Compression Technique to Reduce Decompression Overhead and Design Complexity. *Journal of Systems Architecture:the EUROMICRO Journal*, 46(15):1365–1382, December 2000.

[79] Jang-Soo Lee, Won-Kee Hong, and Shin-Dug Kim. Adaptive Methods to Minimize Decompression Overhead for Compressed On-chip Cache. *International Journal of Computers and Application*, 25(2), January 2003.

[80] Jonghyun Lee, Marianne Winslett, Xiaosong Ma, and Shengke Yu. Enhancing Data Migration Performance via Parallel Data Compression. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 47–54, April 2002.

[81] Charles Lefurgy, Eva Piccininni, and Trevor Mudge. Evaluation of a high performance code compression method. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 93–102, November 1999.

[82] Haris Lekatsas, Jurg Henkel, and Wayne Wolf. Design and Simulation of a Pipelined Decompression Architecture for Embedded Systems. In *Proceedings of the International Symposium on Systems Synthesis*, pages 63–68, 2001.

[83] Haris Lekatsas and Wayne Wolf. Code compression for embedded systems. In *Proceedings of the 35th Annual Conference on Design Automation*, pages 516–521, 1998.

[84] Debra A. Lelewer and Daniel S. Hirschberg. Data Compression. *ACM Computing Surveys*, 19(3):261–296, September 1987.

[85] Yingmin Li, Benjamin Lee, David Brooks, Zhigang Hu, and Kevin Skadron. CMP Design Space Exploration Subject to Physical Constraints. In *Proceedings of the Twelfth IEEE Symposium on High-Performance Computer Architecture*, February 2006.

[86] Mikko H. Lipasti and John Paul Shen. Exceeding the Dataflow Limit via Value Prediction. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 226–237, December 1996.

[87] Jack L. Lo, Luiz Andre 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 Annual International Symposium on Computer Architecture*, pages 39–50, June 1998.

[88] Peter S. Magnusson et al. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, February 2002.

[89] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[90] Cameron McNairy and Rohit Bhatia. Montecito: A Dual-Core Dual-Thread Itanium Processor. *IEEE Micro*, 25(2):10–20, March/April 2005.

[91] Gordon E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, pages 114–117, April 1965.

[92] Todd Mowry and Anoop Gupta. Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, June 1991.

[93] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt. Runahead Execution: An Effective Alternative to Large Instruction Windows. *IEEE Micro*, 23(6):20–25, Nov/Dec 2003.

[94] Kyle J. Nesbit and James E. Smith. Data Cache Prefetching Using a Global History Buffer. In *Proceedings of the Tenth IEEE Symposium on High-Performance Computer Architecture*, pages 96–105, February 2004.

[95] Jose Luis Nunez and Simon Jones. Gbit/s Lossless Data Compression Hardware. *IEEE Transactions on VLSI Systems*, 11(3):499–510, June 2003.

[96] Alex Pajuelo, Antonio Gonz·lez, and Mateo Valero. Speculative Dynamic Vectorization. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, May 2002.

[97] J. Pomerene, T. Puzak, R. Rechtschaffen, and F. Sparacio. Prefetching System for a Cache Having a Second Directory for Sequentially Accessed Blocks, February 1989. U.S. Patent 4,807,110.

[98] Moinuddin K. Qureshi, David Thompson, and Yale N. Patt. The V-Way Cache: Demand Based Associativity via Global Replacement. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 544–555, June 2005.

[99] Majid Rabbani and Paul W. Jones. *Digital Image Compression Techniques*. SPIE Optical Engineering Press, first edition, 1991.

[100] Parthasarathy Ranganathan, Kourosh Gharachorloo, Sarita Adve, and Luis 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*, pages 307–318, October 1998.

[101] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–337, April 1994.

[102] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. Dependence Based Prefetching for Linked Data Structures. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115–126, October 1998.

[103] Gabriele Sartori. Fiber Will Displace Copper Sooner Than You Think, November 2005. http://www.luxtera.com/assets/Luxtera_WPFiberReplacesCopper.pdf.

[104] Falk Scholer, Hugh E. Williams, John Yiannis, and Justin Zobel. Compression of Inverted Indexes for Fast Query Evaluation. In *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 222–229, 2002.

[105] Andre Seznec. Decoupled Sectored Caches. *IEEE Transactions on Computers*, 46(2):210–215, February 1997.

[106] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, October 2002.

[107] B. Sinharoy, R.N. Kalla, J.M. Tendler, R.J. Eickemeyer, and J.B. Joyner. Power5 System Microarchitecture. *IBM Journal of Research and Development*, 49(4), 2005.

[108] G. S. Sohi and Amir Roth. Speculative Multithreaded Processors. *IEEE Computer*, 34(4), April 2001.

[109] G.S. Sohi, S. Breach, and T.N. Vijaykumar. Multiscalar Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

[110] Viji Srinvasan, Edward S. Davidson, and Gary S. Tyson. A Prefetch Taxonomy. *IEEE Transactions on Computers*, 53(2):126–140, February 2004.

[111] Lynn M. Stauffer and Daniel S. Hirschberg. Parallel Text Compression. Technical Report TR91-44, REVISED, University of California, Irvine, 1993.

[112] James A. Storer and Thomas G. Szymanski. Data Compression via Textural Substitution. *Journal of the ACM*, 29(4):928–951, October 1982.

[113] Ivan E. Sutherland and Robert F. Sproull. Logical effort: designing for speed on the back of an envelope. In *Proceedings of the 1991 University of California/Santa Cruz conference on Advanced research in VLSI*, pages 1–16, March 1991.

[114] Systems Performance Evaluation Cooperation. SPEC Benchmarks. http://www.spec.org.

[115] Robert Endre Tarjan and Andrew Chi-Chih Yao. Storing a Sparse Table. *Communications of the ACM*, 22(11):606–611, November 1979.

[116] Joel M. Tendler, Steve Dodson, Steve Fields, Hung Le, and Balaram Sinharoy. POWER4 System Microarchitecture. IBM Server Group Whitepaper, October 2001.

[117] Joel M. Tendler, Steve Dodson, Steve Fields, Hung Le, and Balaram Sinharoy. POWER4 System Microarchitecture. *IBM Journal of Research and Development*, 46(1), 2002.

[118] J. Torrellas, A. Tucker, and A. Gupta. Evaluating the Performance of Cache-Affinity Scheduling in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 24(2):139–151, 1995.

[119] Transaction Processing Performance Council. TPC-C. http://www.tpc.org/tpcc/.

[120] R. Brett Tremaine, T. Basil Smith, Mike Wazlowski, David Har, Kwok-Ken Mak, and Sujith Arramreddy. Pinnacle: IBM MXT in a Memory Controller Chip. *IEEE Micro*, 21(2):56–68, March/April 2001.

[121] R.B. Tremaine, P.A. Franaszek, J.T. Robinson, C.O. Schulz, T.B. Smith, M.E. Wazlowski, and P.M. Bland. IBM Memory Expansion Technology (MXT). *IBM Journal of Research and Development*, 45(2):271–285, March 2001.

[122] Aaron Trott, Robert Moorhead, and John McGinley. Wavelets Applied to Loseless Compression and Progressive Transmission of Floating Point Data in 3-D Curvilinear Grids. In *Proceedings of the 7th conference on Visualization*, pages 385–389, October 1996.

[123] Dean M. Tullsen and Susan J. Eggers. Limitations of Cache Prefetching on a Bus-Based Multiprocessor. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 278–288, May 1993.

[124] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proceedings of the 23th Annual International Symposium on Computer Architecture*, pages 191–202, May 1996.

[125] Bryan Usevitch. JPEG2000 compliant lossless coding of floating point data. In *Proceedings of the 2005 Data Compression Conference*, page 484, March 2005.

[126] Raj Vaswani and John Zahorjan. The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 26–40, 1991.

[127] Jeffrey Scott Vitter. Design and Analysis of Dynamic Huffman Codes. *Journal of the ACM*, 34(4):825–845, October 1987.

[128] J. D. Warnock and et al. The Circuit and Physical Design of the POWER4 Microprocessor. *IBM Journal of Research and Development*, 46(1):27–51, January 2002.

[129] Paul R. Wilson, Scott F. Kaplan, and Yannis Smaragdakis. The Case for Compressed Caching in Virtual Memory Systems. In *Proceedings of the USENIX Annual Technical Conference*, pages 101–116, June 1999.

[130] Wisconsin Multifacet GEMS Simulator. http://www.cs.wisc.edu/gems/.

[131] Ian H. Witten, Radford M. Neal, and John G. Cleary. Arithmetic Coding for Data Compression. *Communications of the ACM*, 30(6):520–540, June 1987.

[132] Wm. A. Wulf and Sally A. McKee. Hitting the Memory Wall: Implications of the Obvious. *Computer Architecture News*, 23(1):20–24, 1995.

[133] Jun Yang and Rajiv Gupta. Energy Efficient Frequent Value Data Cache Design. In *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 197–207, November 2002.

[134] Jun Yang and Rajiv Gupta. Frequent Value Locality and its Applications. *ACM Transactions on Embedded Computing Systems*, 1(1):79–105, November 2002.

[135] Jun Yang, Youtao Zhang, and Rajiv Gupta. Frequent Value Compression in Data Caches. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 258–265, December 2000.

[136] Y. Yoshida, B.Y. Song, H. Okuhata, T. Onoye, and I. Shirakawa. An Object Code Compression Approach to Embedded Processors. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 265–268, August 1997.

[137] Youtao Zhang and Rajiv Gupta. Data Compression Transformations for Dynamically Allocated Data Structures. In *Proceedings of the International Conference on Compiler Construction (CC)*, pages 24–28, April 2002.

[138] Youtao Zhang and Rajiv Gupta. Enabling Partial Cache Line Prefetching Through Data Compression. In *Proceedings of the 2003 International Conference on Parallel Processing*, pages 277–285, October 2003.

[139] Youtao Zhang, Jun Yang, and Rajiv Gupta. Frequent Value Locality and Value-centric Data Cache Design. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 150–159, November 2000.

[140] Zheng Zhang and Josep Torrellas. Speeding up Irregular Applications in Shared-Memory Multiprocessors: Memory Binding and Group Prefetching. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 188–199, June 1995.

[141] Jacob Ziv and Abraham Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.

[142] Jacob Ziv and Abraham Lempel. Compression of Individual Sequences Via Variable-Rate Coding. *IEEE Transactions on Information Theory*, 24(5):530 –536, September 1978.

[143] Nivio Ziviani, Edleno Silva de Moura, Gonzalo Navarro, and Ricardo Baeza-Yates. Compression: A Key for Next-Generation Text Retrieval Systems. *IEEE Computer*, 33(11):37–44, November 2000.