Using Compression for Energy-Optimized Memory Hierarchies

By

Somayeh Sardashti

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN - MADISON

2015

Date of final oral examination: 2/25/2015

The dissertation is approved by the following members of the Final Oral Committee:

Prof. David A. Wood, Professor, Computer Sciences

Prof. Mark D. Hill, Professor, Computer Sciences

Prof. Nam Sung Kim, Associate Professor, Electrical and Computer Engineering

Prof. André Seznec, Professor, Computer Architecture

Prof. Gurindar S. Sohi, Professor, Computer Sciences

Prof. Michael M. Swift, Associate Professor, Computer Sciences

Abstract

In multicore processor systems, last-level caches (LLCs) play a crucial role in reducing system energy by i) filtering out expensive accesses to main memory and ii) reducing the time spent executing in high-power states. Increasing the LLC size can improve system performance and energy by reducing memory accesses, but at the cost of high area and power overheads. In this dissertation, I explored using compression to effectively improve the LLC capacity and ultimately system performance and energy consumption.

Cache compression is a promising technique for expanding effective cache capacity with little area overhead. Compressed caches can achieve the benefits of larger caches using the area and power of smaller caches by fitting more cache blocks in the same cache space. However, previous compressed cache designs have demonstrated only limited benefits due to internal fragmentation, limited tags, and metadata overhead. In addition, most previous proposals targeted improving system performance even at high power and energy overheads.

In this dissertation, I propose two novel compressed cache designs that are optimized for energy: Decoupled Compressed Cache (DCC) [21][22] and Skewed Compressed Cache (SCC) [23]. DCC and SCC tightly pack variable size compressed blocks to reduce internal fragmentation. They exploit spatial locality to track compressed blocks while reducing tag overheads by tracking super-blocks. Compared to conventional uncompressed caches, DCC and

SCC improve the cache miss rate by increasing the effective capacity and reducing conflicts. Compared to DCC, SCC further lowers area overhead and design complexity.

In addition to proposing efficient compressed cache designs, I take another step forward to study compression benefits for real applications running on real machines. Since most proposals on compressed caching are on non-existing hardware, architects evaluate those using detailed simulators with small benchmarks. So, whether cache compression would benefit real applications running on real machines is not clear. In this dissertation, I address this question by analyzing the compressibility of several real applications, including production servers of the Computer Sciences Department of UW-Madison. I show that compression could in fact be beneficial to many real applications.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I have received an incredible amount of help, support, and encouragement from many people over the past years. I would not be able to make it this far without such support.

It has been truly an honor for me to work with my adviser, Professor David A. Wood. He is an outstanding researcher and a brilliant mentor. Throughout these years, David gave me directions on how to find the right problem to work on, how to become a critical thinker, and how to present my ideas. David has very high standards in his work. He gave me freedom and flexibility to explore and enjoy research while he was making sure that I am on the right path and I pay attention to insights and intuitions. On both professional and personal levels, he provided me with the mentorship and support whenever I needed it the most. I am so thankful to him.

Many other professors have also profoundly affected me. I would like to thank Professor Mark D. Hill. Mark has always been a mentor for me. I feel lucky to interact with Mark closely. I have learned a great deal from Mark not only about computer architecture, but also on time management, communicating with others, and presenting ideas. I also express my deep gratitude to Professor André Seznec. André is a brilliant scientist. I have had the great pleasure to collaborate with him. In addition, I've benefited greatly from taking classes and getting feedback on my research from many other faculty members of UW-Madison. I would like to thank Professor Guri Sohi, Professor Mike Swift, Professor Mikko Lipasti, Professor Mary Vernon, Professor Remzi Arpaci-Dusseau, Professor Karu Sankaralingam, Professor Jeff Nauthghon, and Professor Nam Sung Kim.

It has been an honor for me to be part of Wisconsin Architecture community. Main advantage of being in such a great group is the constant mentorship I have received from our former and current students. I would like to thank Dan Gibson, Alaa Alameldeen, Kevin Moore, Dan Sorin, Milo Martin, Babak Falsafi, Jichuan Chen, Yasuko Eckert, Luke Yen, Haris Volos, Arka Basu, Srinath Sridharan, Gagan Gupta, Mike Marty, Philip Wells, Emily Blem, Derek Hower, Marc De Kruijf, Jayaram Boba, Jason Power, Marc Orr, Joel Hestness, Rathijit Sen, Lena Olson, Hongil Yoon, Tony Nowatzki, Mohammad Shoaib Bin Altaf, Nilay Vaish, Newsha Ardalani, Vijay Thiruvengadam, Swapnil Haria, Jayneel Gandhi, Chris Feilbach, and Sankaralingam Panneerselvam. I would like to give special thanks to Jayneel Gandhi, Sankaralingam Panneerselvam, Rathijit Sen, and Dan Gibson for helping me developing and setting up the infrastructures in my projects.

I have also benefited greatly from interacting with many members of larger computer community. I would like to thank Dr. Srilatha Manne, Professor Margaret Martonosi, Professor Sarita Adve, Professor Natalie Enright Jerger, Professor Kathryn S McKinley, Professor Cindy Rubio González and Leng Leng Tan. These wonderful smart women have inspired me and have been my role models. I am so thankful for the great mentorship and support that I have received from them. In conferences, I have had the great chance to interact with many brilliant computer architects, including Professor Joel Emer, Dr. Dileep Bhandarkar, Dr. John Davis, Professor Tom Wenisch, and Professor Babak Falsafi. In Micro 2013, I had a profound discussion with Professor Joel Emer on my research, which has led to better understanding of problems and better solutions in my work. In summer 2011, I did an internship at AMD under supervision of Dr. John Kalamatianos. I would like to thank John. Working with John and his team was a great opportunity for me to learn about the state of the art in computer architecture.

Both the UW’s Computer Systems Lab (CSL) staff and Condor Project staff have been invaluable in providing hardware and software resources I have required. Especially, I would like to thank Tim, the member of CSL. His support enabled me to extend my compression studies to real servers.

My research has been financially supported in part by different sources for which I am grateful: the National Science Foundation (CNS-0916725, CCF-1017650, CNS- 1117280, and CCF-1218323) and a University of Wisconsin Vilas award.

Last but not least, I could not achieve any of these without tremendous support and encouragement from my family. First and foremost I thank my husband, Dr. Hamid Reza Ghasemi, for all his love, support, encouragement, understanding, and patience throughout all of the ups and down of the graduate school. He is my inspiration and joy in my life. I would not be here without his support. I would like to thank my parents for their encouragement, constant support, and believing in me. During this time that I have been far from home, they have always been there for me, encouraged me to progress, advised me to be patient in hard times, and cheered for me for every single one of my successes. My brothers have played an important role in the person I am today. I am so grateful to have them. My eldest brother, Babak, has always encouraged and supported me to get good education. He was the one making sure that I get into great schools, and sign up for English classes every semester from early in my life. My second brother, Behrouz, is the one who inspired me to become an engineer. While I was a teenager, he was taking me to electronics and computer technical fairs and conferences in Tehran to get to know the new technology. Finally, my youngest brother, Mohammad, has always been the kindest and the most supportive brother. He is the one I can always count on. My lovely nieces, Maryam and Fatemeh, and my nephew, Amir Ali, have been the joy in my life, and a good reason not to give up in hard times. I also would like to thank my parents-in-laws, brothers-in-laws, and my sisters-in-laws for their constant love, support and encouragements. I feel so lucky and fortunate to have these many supportive people around myself.

Finally, I would like to dedicate this dissertation to my husband, Dr. Hamid Reza Ghasemi, and my parents, Aghdas Zeinali and Khosro Sardashti.

Chapter 1
Introduction

One of the major challenges for computer architects is today's high power consumption of computer systems. According to the Department of Energy, data centers can consume up to 100 to 200 times more energy than a standard office building, at a cost of billions of dollars per year [78]. In mobile and desktop technologies, lower power consumption is also critical to obtaining longer battery life and lower electricity costs. Future computer systems, however, face continuing power and energy challenges as the power-per-transistor is no longer scaling down as rapidly as the density [71]. Thus, computer architects must consider power and energy as the main design constraints, rather than only focusing on performance.

Although processors are supposed to burn energy when computing, they burn a lot more energy when communicating data. In particular, a large fraction of energy is consumed by the memory hierarchy. However, memory systems have not been classically designed to minimize energy. In the new era of power-constrained computer designs, caches, which are long used to reduce effective memory latency and increase effective bandwidth, play an increasingly important role in reducing system energy. Keckler [72] showed that last-level caches (LLCs) are especially important, since obtaining operands of a double-precision multiply-add from off-chip memory requires approximately 200x the energy of the operation. Sodani [76] showed that caches represent 45% and 12% of core power for non-compute-heavy and compute-heavy floating point applications, respectively, on Intel's Knight's Corner processor. Yet, off-chip accesses consume so much energy that the LLC can miss on 89% of accesses and still break even [76]. Thus, improving cache effectiveness is important, not only for system performance, but also for system energy.

Increasing cache size can improve performance for most workloads, but comes at significant area cost. For example, the well-known "square root" power law [73] predicts that doubling LLC size will reduce misses by ~30%, on average. But it obviously doubles LLC area, which already accounts for 15–30% of the die area of most processors [74]. In this dissertation, I explore using compression to effectively increase cache capacity and ultimately reduce overall system energy.

Cache compression seeks to increase effective cache size by compressing and compacting cache blocks while incurring small overheads [20]. For example, previously proposed techniques have the potential to double effective LLC capacity, while increasing LLC area by only ~8%. Unfortunately, previous compressed cache designs fail to achieve high potentials of compression mainly due to internal fragmentation and extra tags overheads. In addition, they mostly focus on performance benefits of compression, and are not optimized for energy.

In this dissertation, I propose using cache compression to improve system energy. Through extensive analysis, I determine sources of inefficiencies in previously proposed compressed caches. I propose two novel compressed caches that significantly improve cache utilization and so system energy with small area overheads. I also do holistic analysis on compression both in the main memory and caches, for a wide range of real applications and production servers, as well as standard benchmarks running on real machines.

In this chapter, I first discuss the main reasons behind current power and energy problems in Section 1.1. I then present the sources of energy inefficiencies in multicore systems in Section 1.2, emphasizing the role of memory hierarchy. In Section 1.3, I motivate using caches as energy filters to improve system energy, and advocate using compression. In Section 1.4, I identify the main contributions of this dissertation, and provide a roadmap for the remainder of this document in Section 1.5.

1.1 The End of Dennard Scaling

Over the past several decades, computer architects discovered innovative techniques to scale processor performance. They took advantage of more available transistors (Moore's law) at roughly constant power and cost per chip [77]. The semiconductor technology is now facing serious challenges in further scaling transistors and integrating them into chips at an exponential rate. Even with smaller transistors, the fundamental driver of Moore's law was Dennard scaling [71]. The key effect of Dennard scaling was that as transistors got smaller, the power density was constant. For example, if there was a reduction in a transistor's linear size by two, the power used to fall by a factor of four (with voltage and current both halving). It is still possible to etch smaller transistors. However, it is challenging to further drop the voltage in order for the processors to run faster. Further decreasing voltage increases leakage power more rapidly, heating the chip, and possibly threatening complete breakdown. With the end of Dennard scaling, 21st century computer architects can no longer focus solely on performance, and must confront power and energy as the main design constraints.

1.2 Where does energy go?

In current processors, storing and communicating data are more energy expensive than computation. In multicore systems, a large fraction of energy is consumed by the memory hierarchy. Memory systems, however, have not been classically designed to minimize energy.

Memory systems play a critical role in the new era of power-constrained computer designs. Figure 1-1 is borrowed from Keckler's Micro keynote talk [72]. It shows the energy burnt for a simple operation versus the energy consumed to obtain its operands from across the chip at 40nm. It estimates a double-precision multiply-add at 50pJ but obtaining its operands locally

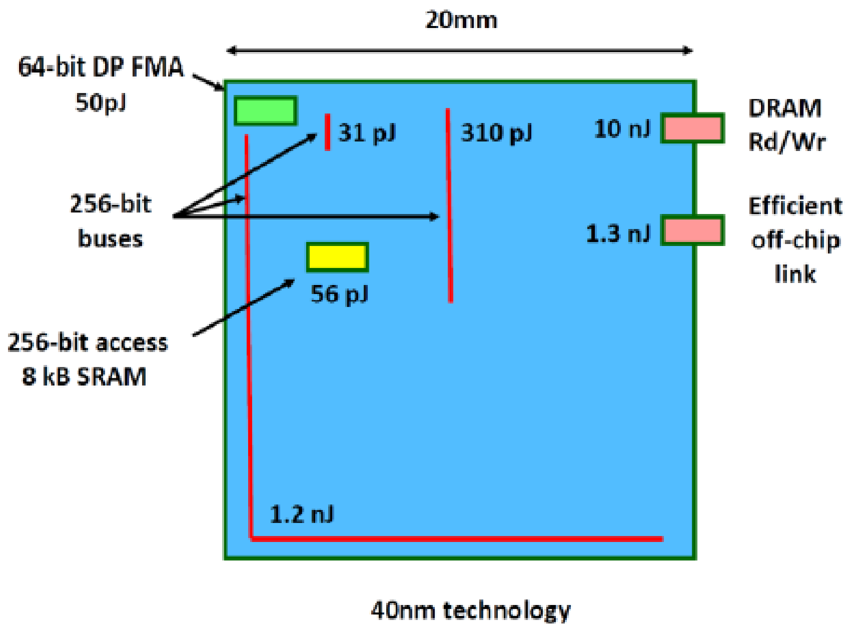


Figure 1-1: Communication vs. computation energy [72].

(1mm) uses 1.7x more energy (31pJ for the bus plus 56pJ to access on-chip SRAM storage). Alternatively, accessing non-local operands is much more expensive: cross-chip 25x (1200pJ+56pJ), or off-chip DRAM 200x (1300pJ+10000pJ). This emphasizes the importance of on-chip cache memories in reducing system energy. Thus, improving effective cache utilization and exploiting locality is crucial, not just for performance, but also for energy efficiency.

1.3 Caches as Energy Filters

Caches have long been used in processor systems. Early work focused on using caches to reduce effective access time or latency. Later, caches were also used to reduce required memory bandwidth, partly, to enable snooping symmetric multiprocessors. In modern multicore processor systems, where the memory hierarchy accounts for a large fraction of total system energy, caches play a critical role in reducing energy.

Caches filter out expensive off-chip memory accesses, and replace them with much cheaper cache accesses. Although caches have been originally designed to hide the latency gap between processors and main memory, the energy gap is an order of magnitude higher. Thus, caches can save system energy, even with extremely high miss rates [76].

Energy-optimized memory hierarchies can also afford to spend energy and time to improve efficiency. An architectural technique that eliminates a miss at a cache level is energy efficient as long as it dissipates less energy than the avoided miss. Since misses are very expensive at the larger levels of the hierarchy, new cache designs can afford to spend energy to reduce misses. A technique can also afford to spend time at caches to improve energy efficiency. Modern processors use a variety of latency-tolerance techniques including data dependence speculation, out-of-order execution, and (simultaneous) multithreading to mitigate the effect of long-latency operations such as cache misses. A processor with a specific combination of techniques has a sphere of latency tolerance, within which performance is relatively insensitive to small changes in latency. Thus, improving the cache hit rate is effective, even at the cost of (relatively small) extra cache access latency and energy.

In this dissertation, I advocate using compression in caches to address the energy challenges in multicore systems. Cache compression seeks to significantly improve cache utilization by fitting more blocks into the same space while incurring small latency, energy, and area overheads due to compressing and decompressing cache blocks. Compression has been studied for every level of the cache hierarchy to increase effective capacity, reduce miss rates, improve performance [20], reduce energy [45] or reduce cache power and area [53]. Cache compression is harder than other levels of the memory hierarchy, since performance is sensitive to cache latency, especially for L1 and L2 caches. But as multicore systems move to having three or more levels of cache, the sensitivity to LLC latency decreases, allowing systems to consider more effective, longer latency compression algorithms. Thus, in this dissertation, I revisit compressed caching at LLC and propose energy-optimized compressed LLCs to improve system energy.

1.4 Thesis Contributions

In this section, I briefly explain the most important contributions of this dissertation.

1.4.1 Understanding Potentials and Limits of Compressed Caching

Designing a compressed cache typically has two main parts: a compression algorithm to represent the same data blocks with fewer bits, and a compaction mechanism to fit compressed blocks in the cache. There are several compression algorithms that exploit regular patterns and the redundancy to compress data. For cache compression, in particular, the compression and decompression latency, area and power overheads of an algorithm matter, in addition to its ability to achieve a good compression ratio (i.e., original size over compressed size). In this dissertation, I use practical hardware-based compression algorithms with fairly good compressibility and low overheads. In Chapter 2, I show that compression can more than triple the effective capacity of a cache while increasing accessing latency by few cycles, and access energy, negligibly.

To achieve the potentials of a given compression algorithm, the compaction mechanism—how to store and track more compressed blocks in the same space—plays a critical role. In order to track more blocks in the cache, a compressed cache needs extra tags and metadata. An ideal design would fit variable size compressed blocks tightly to reduce internal fragmentation, while keeping tag and metadata overheads low. Since I am proposing to use compressed caching for energy-efficiency as well as performance, an ideal compaction mechanism would also avoid incurring energy and latency overheads. In Chapter 3, I categorize previous proposals based on three main design factors: (1) how to provide the additional tags, (2) allocation granularity of compressed blocks, and (3) how to find the corresponding block given a matching tag. Depending on these design factors, most previous proposals have demonstrated only limited benefits from compression, mainly due to internal fragmentation and limited tags. In addition, previously proposed techniques mostly focused on exploiting compression for performance, even at high energy costs.

1.4.2 Decoupled Compressed Cache: Exploiting Spatial Locality for Energy Optimization

In designing a Decoupled Compressed Cache (DCC), I have four main goals: (1) keeping tag and other metadata overheads low, (2) increasing cache utilization by tightly packing variable size compressed blocks, (3) optimizing for energy by eliminating sources of energy overheads present in previous designs, and (4) providing a practical design.

In order to increase the number of compressed blocks while keeping tag overheads low, DCC proposes managing cache tags at multiple granularities. Although current multicore caches typically support a single block size, most workloads exhibit spatial locality at multiple granularities. For most applications, many neighboring blocks can exist in the cache at the same time. Instead of tracking these blocks separately, DCC exploits spatial locality, and uses super-block (also known as sectors [75][1]) tags. A super-block tag tracks four aligned contiguous cache blocks. Since these neighboring blocks share a single address tag, using super-block tags slightly increases cache area overhead, while it allows DCC to track up to four times more blocks in the cache.

[1] I use the unambiguous terms super-block, block, and sub-block, rather than the original, but sometimes confusing terms sectors, blocks, and segments.

DCC increases cache utilization by reducing internal fragmentation. It compresses 64-byte blocks into variable numbers of sub-blocks (e.g., 0 to 4 16-byte sub-blocks). DCC decouples the address tags, allowing any data sub-block in a set to map to any tag in that set, to reduce fragmentation within a super-block. In other words, DCC allows sub-blocks of a block to be non-contiguous within a set. In this way, it eliminates the re-compaction overheads of previous variable size compressed caches [57], while reducing internal fragmentation with sub-blocking. Since each sub-block in the data array could belong to any block, DCC keeps additional metadata (i.e., back pointers) to find the owner of each data sub-block. An optimized Co-DCC design further reduces internal fragmentation (and increases effective capacity) by compacting the compressed blocks from a super-block into the same set of data sub-blocks.

In this work, I also demonstrate that DCC is practical. I present a concrete design for DCC and show how it can be integrated into a recent commercial LLC design (AMD Bulldozer LLC) with little additional complexity.

1.4.3 Skewed Compressed Caches

Given a compression algorithm, an ideal compressed cache tightly packs variable size compressed blocks to increase effective capacity, has a simple design, low tag area overheads and fast lookups. These goals are at odds with each other. Previously proposed compressed caches either do not support variable compressed block sizes [39][47][48][45] or need to keep extra metadata to find a compressed block [20][46][50], which increases overheads and complexity. Even our proposal DCC requires per sub-block back pointers to locate a compressed block. In addition, since DCC manages super-blocks and blocks independently, that complicates its replacement policy.

I propose a Skewed Compressed Cache (SCC) to fill this gap. SCC allocates variable size compressed block while eliminating the need for extra metadata to track blocks. Our goal is to improve LLC effective capacity by compacting variable size compressed blocks in such a way that we can fit and track them with no extra storage overhead and with low design complexity.

Similar to DCC, SCC exploits spatial locality and uses super-block tags to track more compressed blocks with low tag area overhead. It also allows variable size compressed blocks to reduce internal fragmentation. SCC retains direct tag-data mapping to eliminate extra metadata (i.e., no back pointers). SCC does this using novel sparse super-block tags and a skewed associative mapping that takes compressed size into account. SCC also simplifies cache replacement. On a conflict, SCC always replaces one sparse super-block tag and all of the one to eight adjacent blocks packed into the corresponding data entry. This is much simpler than DCC, which may need to replace blocks that correspond to multiple super-blocks, as DCC tracks all blocks of a super-block with only one tag. Like DCC, SCC achieves performance comparable to that of a conventional cache with twice the capacity and associativity. But SCC does this with less than half the area overhead (2.6% vs. 6.8%) of DCC.

1.4.4 MythBusters: on Compression Effectiveness in the Memory Hierarchy

Most compressed cache proposals rely on workload properties that have only been demonstrated to hold for small, CPU-centric benchmarks and very short (simulated) runtimes. Thus, it is largely a matter of faith that these properties hold for large, real-world workloads running for long periods of time. Here, I treat these workload properties as myths—stories that may not be true—that must be tested. I explore 8 myths—common assertions and conventional wisdoms about compression effectiveness—that span a broad range of design options, including compression algorithms, granularity, and compression locality. Rather than limiting ourselves to standard CPU benchmarks and detailed architectural simulation, this study includes up to 24 hour measurements of live workloads, including production servers (e.g., web, file and database servers), memory-intensive desktop applications (e.g., Google Chrome), mobile benchmarks, and emerging big data applications. Through extensive analysis, I show that two of the eight myths are “Busted!,” two are “Plausible,” and the rest are “Confirmed!”.

1.5 Thesis Organization

I begin this dissertation by discussing an overview of the background and related work on compression (Chapter 2). In Chapter 3, I discuss previously proposed compressed caches, and analyze their limitations. In Chapter 4, I present the Decoupled Compressed Cache (DCC), its hardware implementation, and an evaluation of its main properties. In Chapter 5, I present the Skewed Compressed Cache (SCC). In Chapter 6, I present MythBuster, which studies the compressibility of real applications running on real machines. Finally, Chapter 7 concludes this dissertation and outlines some potential areas of future research.

Chapter 2
Compression Algorithms: Background and Related Work

In information theory, entropy of a source input is the amount of information contained in that data. Entropy denotes the number of distinct values in the source. Low entropy suggests that data can be represented with fewer bits. Although computer designers try to use efficient coding for different data types, the memory footprints of many applications still have low entropy. This is mainly due to repeated bit patterns such as repeated characters in a string, zeros, and small values. Zeros are in particular common due to uninitialized variables, null pointers, or zero padding in memory pages. Small values (e.g., small integers) can also be represented with few

Table 2-1: Compression Algorithms Taxonomy.

		Frequent-Value Based	**Significance Based**
General-Purpose	**Static**	**Static Huffman Coding [24]** **FVC [38]**	**Significance-Based Address Compression [43][44]**
	Dynamic	**Lempel-Ziv [27][28]** **Dynamic Huffman Coding [25]** **BDI [19]** **C-PACK [18]**	**FPC [20]**
Special-Purpose	**Static**	**Instruction Compression [1]-[10]**	**-**
	Dynamic	**Floating-point Compression [12]-[16]**	**-**

bits, while we often allocate them similar to maximum values (e.g., 64 bits).

Compression algorithms compress a data message by exploiting the low entropy in the data. They map a data message to compressed code words by operating on a sequence of input bytes. In this section, we first describe and classify different compression algorithms. We then explain the main metrics to evaluate the success of a given compression algorithm. Finally, we evaluate the potentials of some popular compression algorithms for a wide range of applications.

2.1 Compression Algorithm Classifications

Several compression algorithms have been proposed. There are, in general, two types of compression algorithms: lossless and lossy. In lossless algorithms, decompression can exactly recover the original data, while with lossy algorithms only an approximation of the original data can be recovered. Lossy algorithms are mostly used in voice and image compression where lost data do not affect their usefulness. On the other hand, memory content compression techniques are lossless since any single memory bit loss or change can affect the validity of the results in most computer programs. Therefore, in this thesis, we focus only on lossless compression algorithms.

Table 2-1 presents a taxonomy of well-known compression algorithms. For each algorithm, based on its techniques and applications, we classify it into:

- **General-Purpose versus Special-Purpose:** General-purpose algorithms target compressing various data types independent of their semantics. Several existing algorithms fall in this category, including BZIP2, UNIX gzip, and most algorithms used in compressed caches or memory. Unlike general-purpose algorithms, specialized compression algorithms are optimized for specific data types, exploiting the semantic knowledge of the data being compressed. Image/video compression and texture compression in GPUs [17] are good examples of specialized compression. General-purpose techniques tend to achieve low compressibility for certain data types, including instructions and floating-point data. Thus, many have proposed specialized algorithms to improve instruction and floating-point compressibility. However, in this thesis, I focus on low-overhead general-purpose compression algorithms suitable for compression in the memory hierarchy.

- **Static versus Dynamic:** Static compression algorithms provide a fixed mapping from the input data message to output code words [29]. They represent a sequence of input bits with the same code words every time that sequence appears in the input data message. They require two passes on the input data: one pass to determine the mapping, and a second pass for compression. On the other hand, dynamic algorithms do not require any previous knowledge of the data input. They do compression on the fly and might change the mapping over time. In general, dynamic compression techniques are more widely adopted in hardware as they do not require any pre-processing of input data. Some techniques have a hybrid approach, they are neither completely static nor completely dynamic [29].

- **Frequent-Value-Based versus Significance-Based:** Frequent-value-based algorithms exploit a small number of distinct values that tend to repeat very frequently in the memory footprint of an application. For example, zero blocks are common in many applications. Thus, a simple form of a frequent-value-based compression technique is to detect zero blocks (ZERO [37]). Significance-based compression algorithms, on the other hand, are based on the observation that many values are small, and do not require the full space allocated for them. For example, small integer values are sign-bit extended into 32-bit or 64-bit blocks, while all the information can be retrieved from the least-significant bits. There are several variations of frequent-value-based and significance-based compression algorithms proposed for hardware-based cache/memory compression techniques [18][20].

Below, we present some popular compression algorithms that are frequently used for compressing data in the memory hierarchy. We also explain how they fit in our taxonomy.

2.1.1 Lempel-Ziv (LZ) Coding

Lempel-Ziv (LZ) coding and its derivatives [27][28] are the most popular lossless dynamic compression algorithms, which form the basis for many hardware implementations. LZ methods parse data input on the fly using a dictionary in LZ78 [28] and a sliding window in LZ77 [27], which is the equivalent of an explicit dictionary. LZ78 compresses data by exploiting redundancy. It builds a dictionary on the fly. It replaces a repeated symbol (e.g., a string) with a reference to the dictionary. If a match does not exist, it adds the new symbol to the dictionary. LZ algorithms are, in general, effective at exploiting redundancy due to symbol frequency, character repetition, and highly used patterns.

2.1.2 Huffman Coding

Huffman algorithms represent more frequent symbols using shorter code words (i.e., fewer bits). Huffman coding derives a variable-length code table for each source symbol. It derives this table based on the probability or frequency of occurrence of each symbol in the data input. To build this table, static Huffman coding [24] needs an extra pass on the input data to compute the probabilities. Vitter [25] proposed dynamic Huffman coding that only requires one pass of data to compress.

2.1.3 Frequent Value Compression (FVC)

Yang and Gupta [38] introduced a frequent value locality phenomenon: for a running application at any execution point, a small number of distinct values occupy a large fraction of its memory footprint. They also found that the identity of the frequent values, which are fairly uniformly scattered across the memory, remains quite stable over the execution of a program. They exploited frequent value locality to compress data in the memory hierarchy. To find frequent values, they proposed three different profiling approaches: one profiling run for each application before any main run, an initial profiling phase per application execution, and contiguous profiling of a program during its execution.

2.1.4 Frequent Pattern Compression (FPC)

FPC [57] is a general-purpose compression algorithm, mainly optimized for compressing small cache/memory data blocks. It exploits the fact that many values are small (e.g., small integers) and can be stored to a fewer number of bits (e.g., 4 bits), but are normally stored in full 32-bit or 64-bit words. FPC compresses data blocks on a word-by-word basis by storing common word patterns in a compressed format accompanied with an appropriate prefix. It applies significance-based compression at word granularity (4 bytes), detecting and compressing a word to: 4 bits if 4-bit sign-extended, 8 bits if one-byte sign-extended, 16 bits if half-word sign-extended or half-word padded with a zero half-word, or two half-words each a byte sign-extended.

2.1.5 Significance-Based Address Compression

Farrens and Park [43] exploited the redundant information in high-order bits of addresses transferred between processor and memory to improve memory bandwidth. They basically used a variation of significance-based compression, cached high-order bits of addresses and only transferred the low-order address bits as well as small indices (in place of the high-order address bits). Citron and Rudolph [44] applied a similar approach to addresses. They stored common high-order bits in address words in a table and transferred only the low order bits plus and index between the processor and memory.

2.1.6 Base-Delta-Immediate (BDI)

BDI compression algorithm [19] is a low-overhead general-purpose algorithm for compressing data in on-chip caches and the main memory. It is based on the observation that the values within a small cache/memory data block have a low dynamic range (i.e., small differences in their values). BDI represents a block using one or more base values and an array of differences from the base values. Finding the optimum base value is complicated. Thus, to avoid compression latency increase and to reduce hardware complexity, for each block, BDI uses the first value and zero as the base values. In this way, BDI can compress/decompress all values in parallel.

2.1.7 Cache Packer (C-PACK)

C-PACK [18] is designed specifically for hardware-based cache compression. C-PACK compresses a data-block at a 4-byte word granularity. It detects and compresses frequently appearing words (such as zero words) to fewer bits. In addition, it also uses a small dictionary to compress other frequently appearing patterns. The dictionary has 16 entries, each storing a 4-byte word. The dictionary is built and updated on the fly per data block. C-PACK checks whether each word of the input block would match a dictionary entry (even partially). If so, C-PACK then stores the index to that entry in the output compressed code. Otherwise, C-PACK inserts the word in the dictionary. Due to the dictionary, processing multiple words in parallel while permitting an accurate dictionary match is challenging. Thus, C-PACK compresses/decompresses only two words at each cycle.

2.1.8 Instruction Compression

General-purpose compression algorithms usually perform poorly for instruction blocks as instructions have complicated bit patterns compared to regular data blocks. Several specialized compression techniques have been proposed to improve compressibility of instructions. Instruction compression is in particular important in embedded systems, where instruction storage is expensive. Instruction compression can also improve performance by effectively increasing instruction fetch bandwidth.

Most instruction compression techniques find frequently used instruction sequences in the instruction stream, replacing those with small code words to reduce instruction size [1][2][3][4][5][6][7][8][9][10]. For example, Lefurgy et al. [1] proposed a post-compilation analyzer that examines a program, and replaces common instruction sequences with small code words. The processor fetches these code words and expands them to the original sequence of instructions in the decode stage. Their technique benefits programs in embedded processors where instruction memory size is expensive. Benini et al. [5] similarly compressed the most commonly executed instructions to reduce energy in embedded systems. They decompressed instructions on the fly by a hardware module located between the processor and memory.

Cooper et al. [6] explored compiler techniques for reducing memory needed to load and run program executables for a RISC-like architecture. They reduced instruction size using pattern-matching techniques to identify and coalesce together repeated instruction sequences. Similarly, Wolfe and Chanin [7] targeted reducing the instruction size of RISC architectures using compression. They designed a new RISC system that can directly execute compressed programs. They used an instruction cache to manage compressed programs. The processor executes instructions from the cache, so the compression is transparent to the processor.

Thuresson and Stenström [10] evaluated the effectiveness of different dictionary-based instruction compression techniques in reducing instruction size. Dictionary-based instruction compression techniques statically identify identical instruction sequences in the instruction stream and replace them by a code word. Later, at runtime, they replace the code word by the corresponding instruction sequence (i.e., the dictionary entry). The authors showed that this technique can reduce instruction size significantly.

Thuresson et al. [11] addressed increased instruction-fetch bandwidth and larger instruction footprint in VLIW systems using compression. They compressed at compile time by analyzing what subset of a wide instruction set is used in each basic block based on profiling. They also proposed a decompression engine that comprises a set of tables that dynamically convert a narrow instruction into a wide instruction.

2.1.9 Floating-Point Compression

Similar to instructions, floating-point data is generally not compressible with general-purpose compression algorithms. There are several proposals to improve compression for floating-point data. Isenburg et al. [12][13] proposed a compression technique to reduce storage size for floating-point geometric coordinates in scientific and industrial applications. They proposed a lossless compression technique using predictive coding. For each coordinate, they predicted values in floating-point and compressed the corrections from the actual value using context-based arithmetic coding. Lindstrom and Isenburg [14] also presented an online lossless compression of floating-point data to accelerate I/O throughput in real simulation runs. They also used prediction, and for each data value, they predicted it from previously encoded data. They then compressed the difference between the actual and predicted value.

Ratanaworabhan et al. [15] proposed an algorithm to compress sequences of IEEE double-precision floating-point values. They used value prediction, predicted each value in the sequence, and XORed it with the true value. They then encoded and compressed the residual simply by dropping the high-order zero bits (leading-zero compress). In another work [16], the authors further extended compression for fast double-precision floating-point data.

2.2 Metrics to Evaluate the Success of a Compression Algorithm

There are several parameters to evaluate the success of a compression algorithm, including compression ratio, compression and decompression latency, and area and power overheads of compression and decompression units. Compression ratio is defined as the size of the original uncompressed data divided by the size of the compressed data. The higher the compression ratio is, the higher the compression benefits (e.g., saved space) would be. However, there are usually tradeoffs between compression ratio and compression/decompression overheads. The higher compressibility of more complex algorithms usually comes with higher overheads, including compression and decompression latency, area, and power. Thus, many approaches have favored simple algorithms with lower overheads, but at the same time, low compressibility [19][20].

Existing tradeoffs change per design point. For example, in compressed caches and memory, decompression latency is particularly important as it lies on the critical path and can degrade performance. Cache compression is harder than other levels of the memory hierarchy, since performance is sensitive to cache latency, especially for L1 and L2 caches. But as multicore systems move to having three or more levels of cache, the sensitivity to LLC latency decreases, allowing systems to consider more effective, longer latency compression algorithms. In addition, many systems use different mechanisms to hide memory latency, such as OOO cores or multi-threading. Those systems can tolerate the extra decompression latency better, so they could possibly benefit from more complicated algorithms. Similarly, more complex algorithms are better suited at the main memory than caches, where cache hierarchy effectively hides the extra latency. For example, IBM MXT uses a complex algorithm to improve memory capacity [30]. Recently, Arelakis and Stenstrom [46] showed how an aggressive compression algorithm like Huffman coding can be suitable for caches.

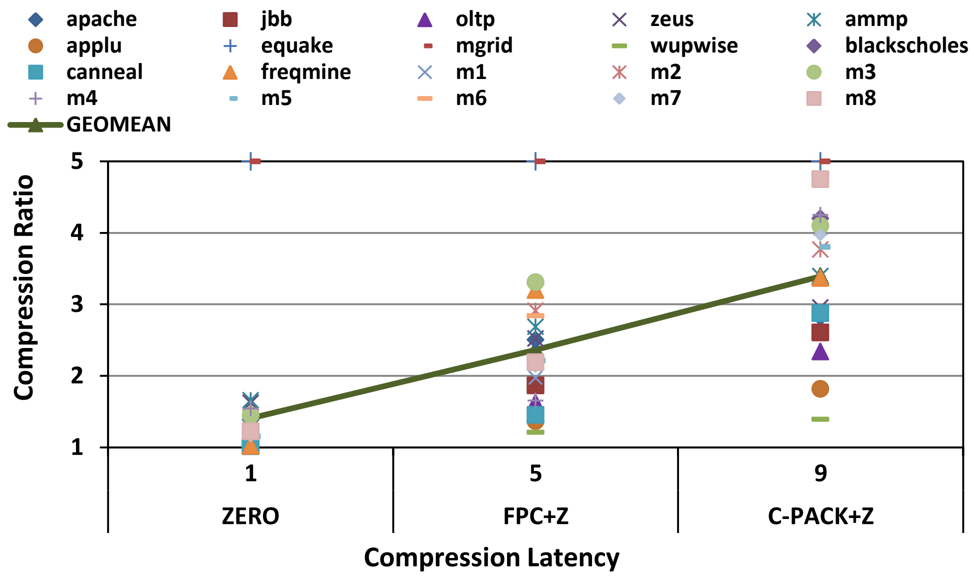


Figure 2-1: Compression ratio of different compression algorithms.

2.3 Compression Potentials

Although some data and most instructions are difficult to compress, most workloads are highly compressible using general-purpose algorithms. Figure 2-1 illustrates the trade-off between decompression latency and compression ratio (i.e., original size over compressed size) for three hardware-based compression algorithms. A simple zero-block detection algorithm (denoted ZERO [37]) has single-cycle decompression latency, but only achieves an average compression ratio of 1.4 and only really benefits a few workloads. Adding a more complex significance-based compression algorithm, FPC+Z (FPC [20] augmented with ZERO) works for a broader range of workloads and improves the average compression ratio to 2.4, but increases decompression latency to five cycles, assuming 12 FO4 gate delays per cycle [57]. Finally, adding dictionary-based compression, C-PACK+Z (C-PACK [18] augmented with ZERO) increases the average compression ratio to 3.4. However, C-PACK takes 9 cycles to decompress a 64-byte data block at 3.2GHz [21][22]. Such a high compression ratio suggests the potential for a similarly large normalized effective cache capacity, that is, the number of compressed blocks stored divided by the maximum number of uncompressed blocks that could be stored. Because multi-megabyte LLCs already have relatively long access times (e.g., 30 cycles) and very high miss penalties (e.g., greater than 150 cycles and ~60 nJ), the benefit of higher compression ratio with C-PACK+Z has the potential to outweigh the longer decompression pipeline.

Chapter 3
Managing Compressed Data in the Memory Hierarchy: Background and Related Work

Compression has been studied for every level of the cache hierarchy to increase effective capacity, reduce miss rates, improve performance, reduce energy or reduce cache power and area. Designing a compressed cache or memory typically involves: a *compression algorithm* to compress blocks, a *compaction mechanism* to fit the compressed blocks in the cache or memory, and a set of policies for managing compressed data. In the previous chapter, I explained several compression algorithms that exploit repeated patterns and redundancy within data blocks to achieve a good compression ratio.

In this section, I focus on compaction mechanisms, and policies to manage compressed caches and memory. I first present the background and related work on cache compression. I describe the fundamentals of compacting compressed data in caches, and present how previous designs limit compression benefits. Finally, I will briefly explain related work on memory compression.

3.1 Compressed Caches: Background and Related Work

3.1.1 Cache Compaction Mechanisms: Fundamentals and Limits

3.1.1.1 Design Fundamentals

While a compressor produces variable size codes at bit granularity, conventional caches operate on fixed size blocks (e.g., 64B). Thus, to achieve the potentials of a given compression algorithm, the compaction mechanism plays a critical role to manage compressed blocks in the cache. Table 3-1 shows a taxonomy of the current state of the art. We can classify previous work using three main design factors: (1) *how to provide the additional tags*, (2) *allocation granularity of compressed blocks, and* (3) *how to find the corresponding block given a matching tag*.

Number of Tags: To track more blocks, compressed caches require additional tags. Traditionally, compressed caches double the number of tags (i.e., 2x Block Tags) to track up to

Table 3-1: Compressed Caches Taxonomy.

Tags		**Data**		
		Half-Block	**Sub-Block**	**Byte**
Per Block	**Direct One-to-One Tag Mapping**	CC[39] Lee et al. [47][48] Significance-compression [45]	-	-
	Decoupled Forward Pointers	-	VSC [20]	SC2[46]
	Decoupled Back Pointers	-	IIC-C [50]	-
Per Super Block	**Direct One-to-One Tag Mapping**	-	SCC [new]	-
	Decoupled Forward Pointers	-	-	-
	Decoupled Back Pointers	-	DCC[new]	-

2x cache blocks in the cache [20]. At the LLC, further increasing the number of tags is costly as the LLC is already one of the largest on-chip components. In this dissertation, I propose an alternative approach to increase the number of tags with low area overhead. My proposals, Decoupled Compressed Cache (DCC) and Skewed Compressed Cache (SCC), exploit spatial locality and use super-block tags to effectively track more blocks while keeping the overheads low. DCC, for example, uses the same number of tags as a regular cache, but each tag tracks a 4-block super-block (i.e., <u>1x Super-Block Tags</u>), and can map up to four cache blocks. Tracking super-blocks only slightly increases tag area compared to the same size regular cache.

Allocation Granularity and Tag-Data Mapping: The subsequent issues are at what granularity to allocate compressed blocks, and how to find a compressed block given a matched tag (i.e., tag-data mapping). Traditional caches store small blocks (e.g., 64B) and maintain a <u>direct one-to-one relationship</u> between tags and data, so a matching tag implicitly identifies the corresponding data. In compressed caches, there is usually a trade-off between allocation granularity and required metadata to track compressed blocks. On the one hand, like many memory allocators, it is generally beneficial to tolerate some internal fragmentation than to deal with arbitrary variability. On the other hand, by lowering allocation granularity, we could lower internal fragmentation and fit more blocks in the cache, but at higher metadata costs. Previous proposals differ on how they balance this trade-off.

The earliest compressed caches maintain such a direct relationship by allowing only one compressed size (i.e., half the block size). Yang et al. [39] exploited the value locality phenomenon to design a first-level compressed cache (Compression Cache). Each cache line of the Compression Cache (CC) stores either one uncompressed line or two lines compressed to at least half their original sizes [39]. Lee et al. [47][48] proposed a compressed cache that selectively compresses cache blocks if and only if they can be compressed to less than half their original size. They proposed several techniques to reduce the decompression overhead [47] including selectively compressing blocks only if their compression ratio is less than a certain threshold, parallel decompression, and buffering recently accessed blocks at the L2 cache in an uncompressed format. In another work [48], they compressed block pairs and stored them in a single line if both lines compressed by 50% or more. In this way, they free a cache block in an adjacent set; however, they need to check two sets for a potential hit on every access, which increases power overheads. Kim et al. [45] also compressed cache blocks into half using a significance-based compression scheme to improve cache utilization. They compressed and stored a cache block as a half block if the block's upper half was either all zeros or all ones. Otherwise, they stored the whole block as uncompressed. Overall, these techniques, which I refer to as Fixed Compression (FixedC), limit compressibility by failing to take advantage of blocks that compress by less than 50%, and so introducing internal fragmentation.

More recent designs reduce internal fragmentation by decoupling tag-data mapping [20] [46][50]. Alameldeen and Wood [20] presented a compressed cache that compacts compressed blocks into a variable number of sub-blocks (also called segments), using the FPC compression algorithm [20] . Their proposal, which I refer to as Variable Size Compression (VSC), reduces internal fragmentation, since all blocks in a set share the same pool of sub-blocks. It stores a compressed block into contiguous sub-blocks in its corresponding data set. VSC decouples tag-data mapping and keeps a block's compressed size in its tag to locate the block. On a lookup, VSC finds the block by adding up the size of all its previous blocks in its corresponding set.

Using this technique, VSC keeps metadata small. However, on a block update, when the block's compressibility and size might change, VSC requires moving other blocks in the corresponding set in order to make enough contiguous space for the accessing block (i.e., re-compaction). As updates happen frequently, re-compactions incur high dynamic energy overheads on the cache.

To increase cache utilization, SC^2 [46] also decouples tag-data mapping. It compresses blocks into a variable number of bytes, storing a byte index field in each tag to locate the starting byte of a compressed block in a set. For example, in a 16-way associate cache, it stores 10 extra bits per tag to locate the block in its corresponding data set. By allocating a block into contiguous sub-blocks, SC^2 also requires evicting adjacent blocks on updates when the block size has increased.

Hallnor et al. [50] extended their earlier indirect index cache [49] to support compression (IIC-C). IIC-C compresses blocks into a variable number of sub-blocks using the LZSS algorithm [32]. Unlike VSC and SC^2, IIC-C eliminates re-compaction overhead by allocating the sub-blocks of a block anywhere in the data array. However, to locate a block, it incurs huge metadata overhead by storing the corresponding set index of each sub-block in the tag (i.e., forward pointers). For example, for an 8MB LLC with 64-byte blocks, 16-byte sub-blocks, and doubled number of tags, IIC-C incurs about 24% area overhead, while it at most doubles the effective capacity. Further increasing the number of tags will make its area overhead even worse.

In this dissertation, I propose two different approaches to eliminate internal fragmentation through sub-blocking, while I eliminate the overheads involved with decoupling tag-data mapping. My proposal DCC supports variable size compression by decoupling tag-data mapping. Unlike previous proposals, DCC provides a low-overhead decoupling mechanism. DCC decouples sub-blocks from the address tag to eliminate expensive re-compaction when a block's size changes. DCC allocates sub-blocks of a block in one set of the data array, not necessarily in contiguous space (unlike VSC), but in order. DCC still needs extra metadata to find a block in the data array. To keep the metadata overhead small, instead of storing where each sub-block of a block locates (i.e., forward pointers [50]), I use a few bits per data sub-block in a data set to represent its owner block (i.e., back pointers).

My proposal SCC similarly compacts blocks into a variable number of sub-blocks to reduce internal fragmentation, but retains direct tag-data mapping to find blocks quickly and eliminate extra metadata (i.e., no forward or back pointers). SCC does this using novel sparse super-block tags and skewed associative mapping that takes compressed size into account.

3.1.1.2 Limits of Previously Proposed Compressed Caches

As I showed in Figure 2-1 of Chapter 2, C-PACK+Z compression algorithm achieves an average compression ratio of 3.4 for a wide range of application. Ideally, a compressed cache could fit 3.4 times more compressed blocks in the same space using this algorithm. However, previous compressed cache designs fail to achieve this potential for three main reasons. First, all hardware caches map blocks into sets, introducing an internal fragmentation problem since a compressed block must (at least in current designs) be stored entirely within one set. In Figure 3-1, the BytePack column represents an idealized compressed cache with infinite tags that compacts compressed blocks on arbitrary byte boundaries. BytePack achieves an average normalized effective capacity of 3.1. Note that some low memory intensive workloads, such as ammp, have small working sets, which fit in a small cache even though they have highly

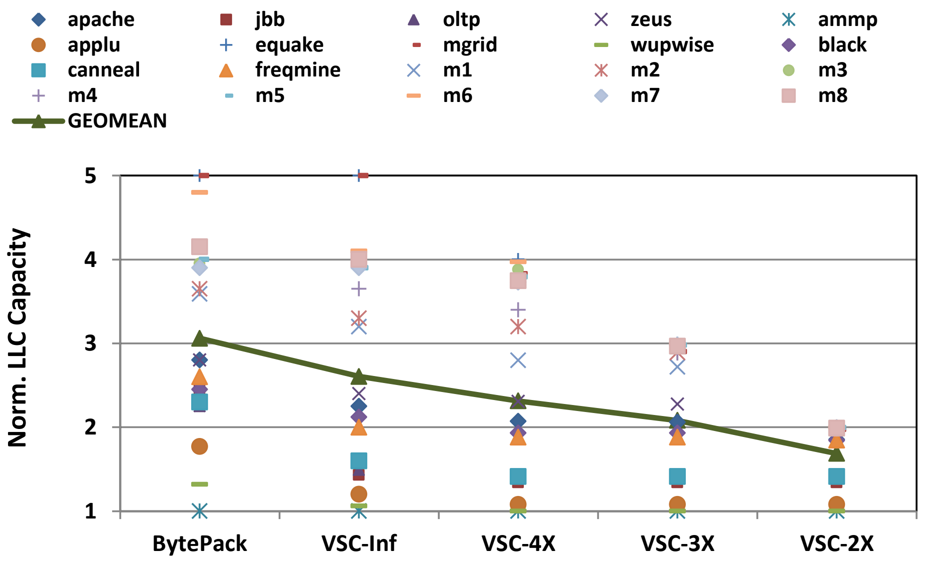


Figure 3-1: Limits of previous compressed caches.

compressible data. Second, practical compressed caches introduce a second internal fragmentation problem by compacting compressed blocks into one or more sub-blocks, rather than storing compressed data on arbitrary byte boundaries. The column labeled VSC-Inf in Figure 3-1 illustrates that compacting compressed blocks into 0–4 16-byte sub-blocks (but with infinite tags per set) degrades normalized effective capacity from 3.1 to 2.6, on average. Finally, compressed caches have a fixed number of tags per set. The remaining columns in Figure 3-1 illustrate that reducing the number of tags from infinite to a more practical twice Baseline, degrades the normalized effective capacity from 2.6 to 1.7, on average.

Figure 3-1 results suggest two approaches to unlocking the potential of cache compression. First, reduce the internal fragmentation within a set. However, this must be done with care in today’s energy constrained environment. VSC-2X relaxes the mapping constraint between tags and data and compacts compressed blocks into a variable number of contiguous sub-blocks [20]. VSC-2X can compact more blocks in the cache than a simple FixedC compressed cache, which only compacts compressed blocks into half-blocks (i.e., 32-byte sub-blocks). However, VSC needs to repack the sub-blocks in a set whenever a block's size changes, to make contiguous free space. This can significantly increase the cache bank occupancy and dynamic energy. Figure 3-2 shows the average number of accessed bytes at LLC normalized to Baseline. FixedC decreases the average number of accessed bytes by 36% compared to Baseline due to accessing shorter compressed blocks. On the other hand, VSC-2X increases the number of accessed bytes at LLC by nearly a factor of three since it needs to repack sets (copying almost half a set, on average). This significantly increases LLC dynamic energy.

The second approach to improving cache compression is to increase the number of tags per set. Figure 3-1 shows that increasing the tags from twice the Baseline to four times the Baseline increases the normalized effective capacity from 1.7 to 2.3, on average. However, done naively, this significantly increases the area overhead. Figure 3-3 shows the area overhead (compared to Baseline) versus normalized effective capacity. A variable size compression cache (VSC-2X) with twice as many tags as Baseline [20] increases LLC area by 8%. However, quadrupling the

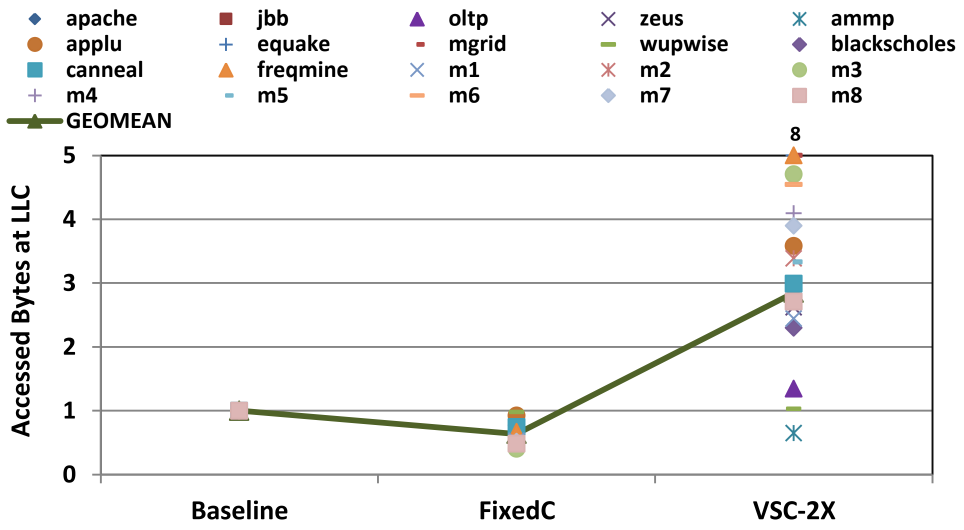


Figure 3-2: VSC overhead on the number of LLC accessed bytes.

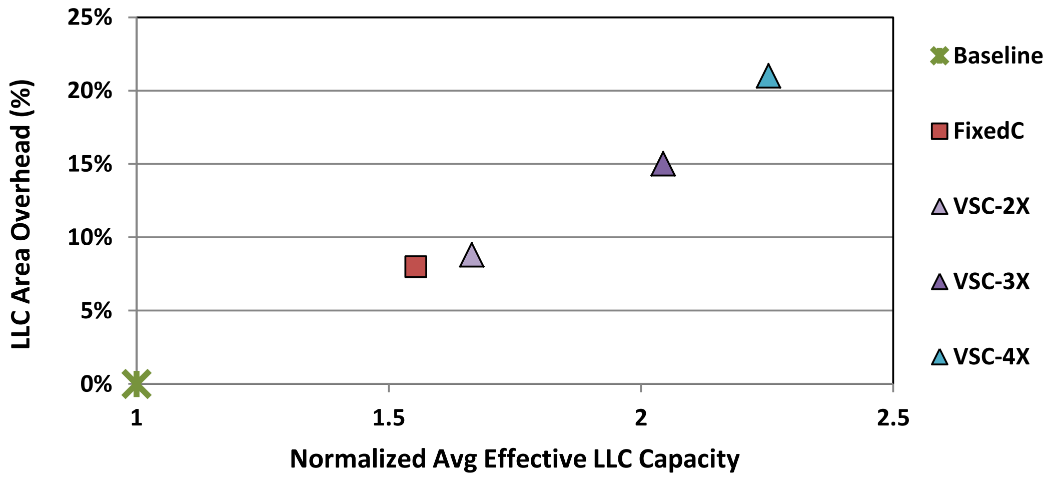


Figure 3-3: Area overhead of different cache designs.

number of tags (VSC-4X) increases LLC area by ~21%.

3.1.2 Policies to Manage Compressed Caches

Adaptive compression: Compressed caches introduce a trade-off between cache capacity and cache access latency. On the one hand, they can improve effective cache capacity by storing more cache blocks, resulting in a possibly lower cache miss rate. On the other hand, they incur higher access latency as they decompress compressed blocks. These trade-offs change depending on different parameters, including the sensitivity of the applications to cache latency and capacity, cache level and decompression latency. For capacity-sensitive workloads, compression can improve performance by reducing costly misses to the next level of hierarchy, while for cache insensitive workloads or latency-sensitive workloads, the latency overhead of decompression can impact performance. The overhead is higher with longer latency decompression techniques and at lower levels of cache hierarchy (L1 or L2).

To balance this trade-off, Alameldeen and Wood [20] proposed an adaptive policy that dynamically adapts to the costs and benefits of cache compression. In a two-level cache hierarchy, they employed compression only at the L2 cache. They kept uncompressed blocks at the L1 cache as decompression overhead on the L1 cache hit latency can significantly impact performance. On a cache allocate, they compressed a new block if the entire cache appeared to be benefiting from compression. To determine whether compression was beneficial, they used the cache replacement algorithm's stack depth. They kept a global saturating counter to keep track of whether compression (could have) eliminated a miss or incurred an unnecessary decompression. On each cache access, they incremented this counter by the L2 miss penalty if compression could elide a miss, and decremented the counter by the decompression latency if the access would have been a hit even without compression. Using this counter, they predicted whether to allocate future cache lines in compressed or uncompressed form. They showed that by dynamically monitoring workloads' behavior, their adaptive compressed cache achieved the benefits of compression for cache sensitive workloads, while avoiding performance degradation for others. Their adaptive mechanism can be used in other compressed caches and at other levels of cache hierarchy as long as we are using an LRU replacement policy.

Tailored replacement policy: Compressed caches typically use the same replacement policy as traditional caches that treat all blocks similarly, while the sizes of the cache blocks vary depending on their compressibility. Baek et al. [55] proposed a size-aware compressed cache management, Effective Capacity Maximizer (ECM), to improve the performance of compressed caches. They used cache block size as a hint to select a victim to improve cache performance. In a compressed cache, the eviction overhead varies based on the size of the evicted and evictee cache blocks. If the size of the new block is larger than the victim block, the compressed cache needs to evict more blocks. Thus, they considered block size in the cache management policies to increase effective capacity. They classified blocks as big-size or small-size based on their compressed size in comparison with a threshold. They dynamically adjusted this threshold on every block insertion. Using a DRRIP [56] framework, they proposed a size-aware insertion policy that gives the big-size cache blocks a higher chance of eviction. On evictions, they also chose the biggest-size cache block in case there were multiple possible victims. Employing these policies, they showed that ECM has the potential to improve effective capacity, cache miss rate and overall system performance. Pekhimenko et al. [54] similarly proposed tailored replacement and insertion policies for compressed caches.

Interactions with prefetching: Alameldeen and Wood [57] showed that compression and prefetching can interact in strong positive ways. Prefetching, in general, suffers from bandwidth pressure and cache pollution, while compression can alleviate both of these. Similarly, prefetching can help compression by hiding the decompression latency. Alameldeen and Wood [57] proposed an adaptive prefetching mechanism that enables prefetching whenever beneficial. They used cache compression's extra tags to detect useless and harmful prefetches. In their compressed cache, they doubled the number of tags to potentially track twice compressed blocks. However, in many cases, not all the blocks are compressible, so there are extra tags not being used. They leveraged these tags to track recently evicted blocks and to find whether prefetched blocks were evicting useful ones. They used a saturating counter that they incremented on useful prefetches, and decremented on useless or harmful prefetches. Using this counter, they disabled prefetching when it did not help. Overall, they showed by leveraging the interaction between compression and prefetching, they can significantly improve performance.

3.1.3 Cache Compression to Improve Cache Power and Area

In addition to adopting compression to improve cache effective capacity, some techniques aim at reducing cache power and area using compression. In general, in compressed caches, we can reduce cache power on a block access if the power burnt to compress/decompress the block is lower than the power burnt to access the compressed block. Thus, in all these techniques, they use simple compression algorithms (such as significance-based) with small power overheads at the cost of lower compressibility compared to LZ-based compression algorithms.

Yang et al. [40] exploited frequent value locality to improve cache dynamic power. They compressed a cache line into half, if possible, otherwise, stored it as uncompressed. They partitioned the cache data array into two sub-arrays such that on an access to a compressed block (i.e., a frequent value), they would only activate the first data sub-array. Otherwise, it would require an additional cycle to access the second data sub-array. In this way, they could reduce cache dynamic energy consumption for frequent value accesses, which are dominant, at the cost of higher access time for non-frequent value accesses.

Significance-compression [45] similarly improves cache power by accessing half of a cache block if compressed, and packing more blocks in the cache. Dynamic zero compression (DZC) [52] also reduces the L1 cache dynamic power by only storing and accessing non-zero bytes of a block in the data array.

In a recent work, Kim et al. [53] aimed at reducing the L2 cache area and power in single processor embedded systems. They halved the L2 cache size, and compressed cache blocks and stored them in half size in the L2 cache. If a block was not compressible, they stored its first half in the L2 cache, and its second half in a small cache, called the residue cache. By reducing the size of the L2 cache and accessing half-sized blocks, they reduced both area and power.

3.2 Compressed Memory: Background and Related Work

Compression also has applications at other levels of memory hierarchy. In this section, I describe some hardware-based and software-based memory compression schemes that target increasing effective memory capacity, improving memory bandwidth, and reducing memory energy.

3.2.1 Hardware-Based Memory Compression

IBM Pinnacle [29] was the first commercially available memory controller that employs real-time main-memory compression. It employed IBM's Memory Expansion Technology (MXT) in a single-chip memory controller to effectively double the main memory capacity without significant overheads. MXT uses a parallelized variation of the Lempel-Ziv algorithm known as LZI as the compression algorithm [27]. It compresses 1-KB cache blocks (same size as in the L3) into 0 to 4 256-byte sub-blocks depending on the block compressibility. As blocks have variable size when compressed, the memory controller needs to find where the block is by translating the block address on the system bus to the physical address in the physical memory. To do so, MXT keeps mapping in a Compression Translation Table (CTT) with one entry per block. CTT is stored at a reserved location in the physical memory. Each entry includes four physical sub-block addresses, each pointing to a 256-byte sub-block in the physical memory. On an access, the memory controller performs real to physical address translation by a lookup in the

CTT to find where the sub-blocks of a block reside. MXT has shown to be effective for many applications and servers. Compared to a standard uncompressed memory, it has negligible penalty due to decompression latency. However, to deal with the variable memory size, this scheme requires support from the operating system.

Kjelso et al. [62] also explored compression potentials at the main memory. They presented an X-Match hardware compression algorithm. X-Match maintains a dictionary, processes 4-byte sub-blocks replacing them with a shorter code in case of a match or a partial match with a dictionary element. They analyzed the compressibility of some Unix real applications using the X-Match algorithm, and demonstrated an average double increase in effective memory capacity with compression. Nunez and Jones [63] further proposed XMatchPRO, a high-throughput hardware FPGA-based X-Match implementation.

Ekman and Stenstrom [59] used a frequent pattern compression scheme [20] to compress memory contents. They addressed some of the drawbacks of MXT. First of all, they used a simple compression algorithm with small decompression latency (5 cycles) as opposed to MXT's complicated LZ algorithm with 64 cycles of decompression latency. Second, to find a block in the main memory, the operating system maps the uncompressed virtual address space directly to a compressed physical address space by storing the size of each block in a page in its corresponding page table entry and using a small and fast TLB-like structure.

In a recent study, Pekhimenko et al. [61] proposed Linearly Compressed Pages (LCP) for compression at the main memory. They used a simple compression algorithm, Base-Delta-Immediate Compression (BDI) [19]. To simplify block access in physical memory, LCP uses one fixed size for compressed cache blocks within a given page. In this way, the location of a compressed cache block within a page is simply the product of the index of the cache block within the page and the compressed size.

Zhang and Gupta [64] introduced a class of transformations that modify the representation of dynamically allocated data structures commonly used in pointer intensive programs. They compressed the fields of a node in a dynamic structure by compressing 32-bit address pointers and integer words into 15-bit entities, and packing two compressed fields in the space of one. To access data in the compressed format, they added six instructions, data compression extension (DCX), to the MISP instruction set.

Dusser et al. [60] proposed decoupled zero-compressed memory (DZC). They exploited null blocks that represent a significant fraction of the working set of many applications. DZC is a hardware compressed memory that only targets null data blocks. DZC represents null blocks with a bit. To store non-null blocks, DZC manages the main memory as a decoupled sectored set-associative cache with each page treated as a sector. Compared to other compression mechanisms, DZC limits the benefit by only focusing on null blocks.

Shafiee et al. [65] presented MemZip that exploits compression for improving memory bandwidth, energy, and reliability. Most techniques use compression at the main memory to improve memory capacity. MemZip, however, compresses blocks in the main memory, but does not pack them to make more space. By storing compressed blocks, on a memory access, MemZip would access fewer bytes. On a read, it first reads out metadata that tell the exact number of bursts required to fetch the compressed cache block. It next transfers the block over exactly that burst length. In this way, it saves memory bandwidth and energy. It further uses the space freed by compression to improve reliability using better ECC coding.

Sathish et al. [114] exploits compression for data transferred between a GPU and its off-chip memory to provide higher effective bandwidth. They use a combination of both lossy and lossless compression applying compression to floating-point numbers after truncating their least-significant bits. In this way, they can improve bandwidth with little impact on overall computational accuracy.

3.2.2 Software-Based Memory Compression

In general, previously proposed software-based compressed memories store actively accessed data as uncompressed while storing others in a dedicated section of the main memory in a compressed format [66][67][68][69]. On a page fault in the uncompressed section, they search the compressed section. They then decompress and move the compressed page. In this way, the compressed section basically acts as a cache to hide the latency to the disk.

Apple OS X Mavericks employs compression to increase performance [70]. With OS X Mavericks, compressed memory allows Mac to free up memory space, when needed. As Mac approaches maximum memory capacity, OS X automatically compresses data from inactive apps, making more memory available for active processes. Linux zcache similarly compresses file pages that are in the process of being reclaimed storing them in memory.

Chapter 4
Decoupled Compressed Caches

4.1 Overview

Cache compression can increase effective cache capacity, reduce misses, improve performance, and potentially reduce system energy. However, as I discussed in Chapter 3, previous compressed cache designs have demonstrated only limited benefits mainly due to internal fragmentation and limited tags. In addition, most previous proposals targeted improving system performance even at high power and energy overheads. For example, VSC [20] allows variable compressed block sizes, but requires high-overhead re-compaction, which involves moving on average half of the blocks in a set, to make enough contiguous space on updates. Since updates happen frequently, re-compactions incur high dynamic energy overheads on the cache.

In this chapter, I present Decoupled Compressed Cache (DCC) [21]. DCC has five main goals: (1) increasing the number of tags to track more compressed blocks without incurring high area overheads, (2) eliminating energy inefficient re-compactions while allowing variable compressed block sizes, (3) reducing internal fragmentation to further improve effective capacity by packing compressed blocks tightly, (4) incurring low area and power overheads, and (5) providing a practical design.

As I discussed in Chapter 3, the first fundamental factor in designing a compressed cache is to provide extra tags to track more blocks in the cache. Simply increasing the number of tags (e.g., 4x tags) would increase area and power overheads by an unacceptable amount. In order to increase the number of tags while keeping the overheads low, DCC proposes managing cache tags at multiple granularities. Although caches typically support a single block size, most workloads exhibit spatial locality at multiple granularities, and thus, many neighboring blocks may exist in the cache at the same time. Instead of tracking these blocks separately, DCC exploits spatial locality, and uses super-block tags. A super-block tag tracks a group of aligned contiguous cache blocks (e.g., 4 blocks). While previous cache designs used super-blocks tags to reduce the number of tags, DCC keeps the same number of tags and use super-block tags to track more blocks. Since these neighboring blocks share a single address tag, using super-block tags slightly increases cache area overhead. Compared to a regular cache, DCC basically replaces each tag entry with a super-block tag. DCC compresses each 64-byte block independent of its neighbors. However, it tracks up to four neighbors with one super-block tag. In this way, it can track up to four times more blocks in the cache with low area and power overheads.

To reduce internal fragmentation, DCC compresses a 64-byte block into variable number of sub-blocks (e.g., 0 to 4 16-byte sub-blocks), and allocates these sub-blocks in the data array. Previous proposals [57][20][50] supported variable size compression at high metadata and power overheads. DCC, however, decouples the address tags—allowing any data sub-block in a set to map to any tag in that set—to reduce fragmentation within a super-block [75]. In another word, DCC allows sub-blocks of a block to be non-contiguous within a set. In this way, it eliminates the re-compaction overheads of previous variable size compressed caches [20], while reducing internal fragmentation with sub-blocking. Since each sub-block in the data array could belong to any block, DCC keeps a few bits per data sub-block in a data set to represent its owner block (i.e., back pointers).

Although sub-blocking reduces internal fragmentation, it reduces compression effectiveness because of wasted space within sub-block boundaries. For example, even if a block can fit in 10 bytes, DCC allocates a 16-byte sub-block for that. Reducing sub-block size would help, but could increase area and power overheads significantly. An optimized Co-DCC design further reduces internal fragmentation (and increases effective capacity) by compacting the compressed blocks from a super-block into the same set of data sub-blocks, but uses more metadata.

Although many researchers have studied the potential of cache compression, the computer industry has shown lower interest in integrating these ideas in new processors due to possible design complexities. In this work, I take an additional step to demonstrate that DCC is practical. I present a concrete design for DCC and show how it can be integrated into a recent commercial LLC design (AMD Bulldozer LLC) with little additional complexity.

I evaluate DCC using the GEMS full-system simulator [81]. I show that DCC can improve average performance and system energy by 10% and 8%, respectively. Importantly, this is better than a conventional LLC of twice the capacity, and uses only 8% more area than a same-size uncompressed Baseline. In comparison with previous proposals, FixedC and VSC-2X, DCC nearly doubles the performance and energy benefits for comparable area overheads. Co-DCC further reduces runtimes and system energy, but at the cost of some additional complexity.

In the rest of this chapter, I show the potential in exploiting spatial locality for improving compression effectiveness in Section 4.2, present DCC design in Section 4.3, describe a practical design in Section 4.4, describe the experimental methodology and results in Sections 4.5 and 4.6, and conclude the chapter in Section 4.7.

4.2 Spatial Locality at LLC

Although current multicore caches typically support a single block size, most workloads exhibit spatial locality at multiple granularities. Figure 4-1 shows the distribution of neighboring blocks in a conventional LLC with a tag per 64-byte block (workloads and simulation parameters described in Section 4.5). Neighboring blocks are defined as those in a 4-block aligned super-block (i.e., aligned 256-byte region). The graph shows the fraction of blocks that are part of a Quad (all four blocks in a super-block co-reside in the cache), Trios (three blocks out of four co-reside), Pairs (two blocks out of four co-reside), and Singletons (only one block out of four resides in the cache). Pairs and Trios are not necessarily contiguous blocks, but represent two or

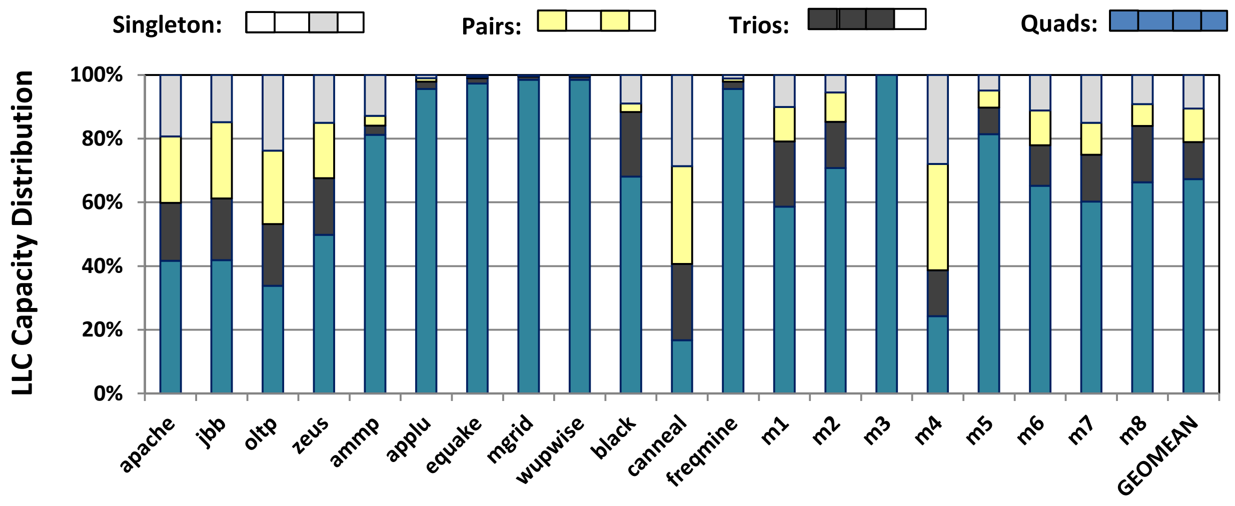


Figure 4-1: Distribution of LLC cache blocks.

three blocks, respectively, that could share a super-block tag. Although access patterns differ, the majority of cache blocks reside as part of a Quad, Trio, or Pair. For applications with streaming access patterns (e.g. mgrid) Quads account for essentially all the blocks. Other workloads exhibit up to 29% singletons (canneal), but Quads or Trios account for over 50% of blocks for all but two of our workloads (canneal and gcc).

Super-blocks (also known as sectors [75]) have long exploited coarser-grain spatial locality to reduce tag overhead [82][83][75]. Super-blocks associate one address tag with multiple cache blocks, replicating only the per-block metadata such as coherence state. Figure 4-2(a) shows one set of a 4-way-associative super-block cache (SC), with 4-block super-blocks. Using 4-block super-blocks reduces tag area by 70% compared to a conventional cache. However, Figure 4-2(a) illustrates that Singletons, Pairs, and Trios—such as, super-blocks D, C, and A, respectively—result in internal fragmentation, which can lead to significantly higher miss rates [75].

Seznec showed that decoupling super-block tags from data blocks helps reduce internal fragmentation [75]. Decoupled super-block caches (DSC) increase the number of super-block

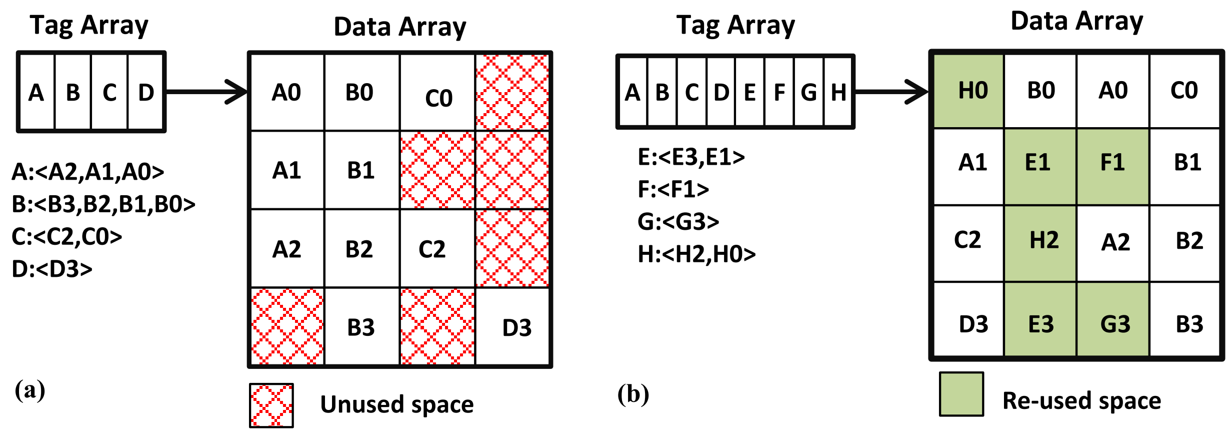


Figure 4-2: (a) Sectored Cache (b) Decoupled Sectored Cache.

tags per set and use per sub-block back pointers to identify the corresponding tag. Figure 4-2(b) illustrates how decoupling can reduce fragmentation by allowing two Singletons (i.e., blocks F1 and G3) to share the same super-block. DSC uses more tag space than SC, but less than a conventional cache since back pointers are small.

In this work, I use decoupled super-block tags to improve cache compression in two ways. First, super-blocks reduce tag overhead, permitting more tags per set for comparable overhead. Second, decoupling tags and data reduces internal fragmentation and, importantly, eliminates re-compaction when the size of a compressed block changes.

4.3 Decoupled Compressed Cache: Architecture and Functionality

In this section, I describe Decoupled Compressed Cache (DCC) and Co-Compacted DCC (Co-DCC) designs in detail. While these designs may be applicable to other levels of the cache hierarchy, I target the LLC in this work.

4.3.1 DCC Architecture

To improve compression effectiveness at the LLC, DCC exploits super-blocks and manages the cache at three granularities: coarse-grain super-blocks, single cache blocks, and fine-grain sub-blocks. DCC tracks super-blocks, which are groups of aligned, contiguous cache blocks (Figure 4-3(d)), while it compresses and stores single cache blocks as variable number of sub-blocks.

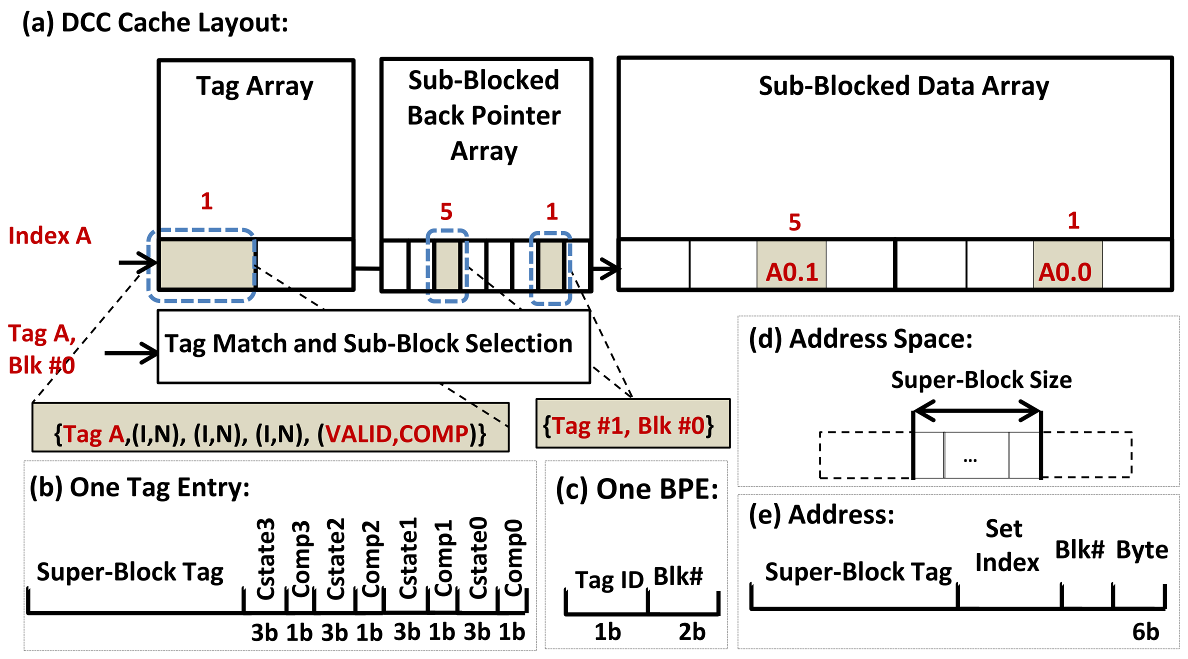


Figure 4-3: DCC cache design.

Figure 4-3(a) shows the key components of DCC architecture for a small 2-way-set associative cache with 4-block super-blocks, 64-byte blocks, and 16-byte sub-blocks. DCC consists of Tag Array, Sub-Blocked Back Pointer Array, and Sub-Blocked Data Array. DCC is indexed using the super-block address bits (Set Index in Figure 4-3 (e)). Note that like all super-block caches, this index uses higher order bits. In this way, all blocks of the same super-block are mapped to the same data set.

DCC explicitly tracks super-blocks through the tag array. The tag array is a largely conventional super-block tag array. Figure 4-3 (b) shows one tag entry that consists of one tag per super-block (Super-block tag) and coherence state (CState) and compression status (Comp) for each block of the super-block. Since all four blocks of a super-block share a tag address, the tag array can map four times as many blocks as the same size conventional cache with minimal area overhead. DCC holds as many super-block tags as the maximum number of uncompressed blocks that could be stored. For example, in Figure 4-3, for a 2-way-associative cache, it holds two super-block tags in each set of the tag array. In this way, each set in the tag array can map eight blocks (i.e., 2 super-blocks * 4 blocks/super-block), while a maximum of two uncompressed blocks can fit in each set. In the worst case scenario, when there is no spatial locality (i.e., all singletons) or cached data is uncompressible, DCC can still utilize all the cache data space, for example, by tracking two singletons per set in Figure 4-3 (a).

DCC compresses and compacts cache blocks into a variable number of data sub-blocks. It dynamically allocates these sub-blocks in the sub-blocked data array. The data array is a mostly conventional cache data array, organized in sub-blocks. In Figure 4-3 (a), it provides eight 16-byte sub-blocks per set, for a total of 128 bytes. This is only one quarter of the data space mapped by each set in the tag array (i.e., 2 super-blocks * 4 blocks/super-block * 64 bytes/block = 512). Thus using this configuration the tag array has the potential to map four times as many blocks as can fit in the same size uncompressed data array.

DCC decouples sub-blocks from the address tag to eliminate the expensive re-compaction when a block's size changes. DCC allocates sub-blocks of a block in the data array not necessarily in contiguous space (unlike VSC [20]) but in order. For example, in Figure 4-3 (a), block A0 is compressed into two sub-blocks (A0.1 and A0.0) that are stored in the sub-block #5 and the sub-block #1 in the data array.

For a low-overhead decoupled tag-data mapping, DCC uses small back pointers as one level of indirection to locate sub-blocks of a compressed block. For each sub-block in the data

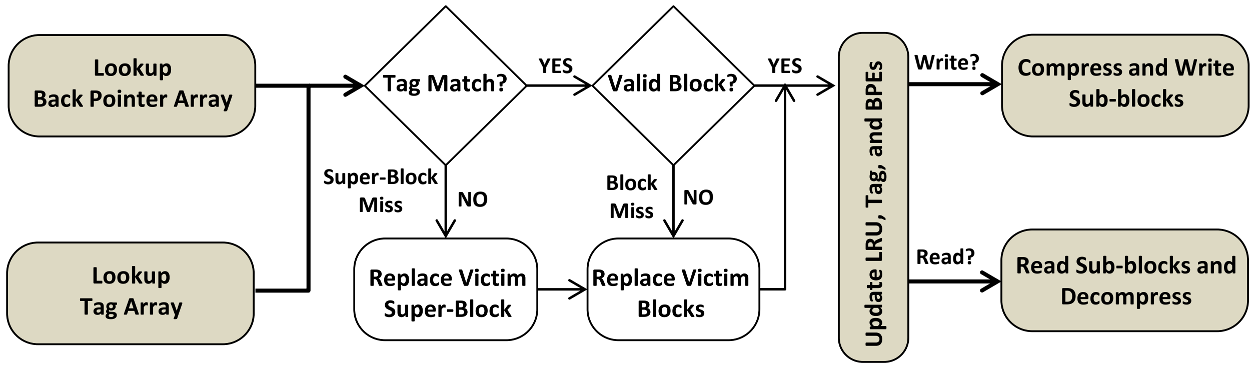

Lookup Back Pointer Array
Lookup Tag Array
Tag Match?
YES
Valid Block?
YES
Super-Block Miss
NO
Block Miss
NO
Replace Victim Super-Block
Replace Victim Blocks
Update LRU, Tag, and BPEs
Write?
Compress and Write Sub-blocks
Read?
Read Sub-blocks and Decompress

Figure 4-4: DCC cache lookup.

array, the back pointer array keeps one back pointer entry (BPE) identifying the owner block of that sub-block. To do so, a BPE stores a corresponding block's tag ID and block ID (Figure 4-3 (c)). Tag ID (e.g., 1 bit for a 2-way-associative cache) refers to the super-block tag entry matching this block in the same set of the tag array (e.g., 1 in Figure 4-3 (a)). Block ID refers to the position of a block within its encompassing super-block. DCC derives Block ID (e.g., 2 bits for a 4-block super-block) from the block address (Blk# in Figure 4-3 (e)). Using the tag ID and block ID, a BPE encodes the owner block of a sub-block with minimum metadata. In this way, the back pointer array enables low-overhead variable size compression, while it slightly increases the LLC area (discussed in Section 4.5).

4.3.2 DCC Lookup Process

Figure 4-4 shows the DCC lookup procedure for different scenarios. On a cache lookup, both the tag array and the back pointer array are accessed in parallel. In the common case of a cache hit, both the block and its corresponding super-blocks are found available (i.e., tag matched and block is valid). In the event of a cache hit, the result of the tag array and the back pointer array lookup determines which sub-blocks of the data array belong to the accessing block. On a read, those sub-blocks are read out next, and the corresponding tag entry and BPEs are updated. In Figure 4-3, for example, on a read access to block A0, Tag A, Index A, and block ID (e.g., #0) are derived from the address (Figure 4-3 (e)). The corresponding set of the tag array and the back pointer array (indexed by Index A) are read out. The tag match and sub-block selection logic then identify whether the block is available and where its sub-blocks locate in the data array. For instance, the tag entry #1 in the tag array matches super-block A, and its block #0 is available (CState #0 is valid). The sub-block selection logic finds the matched BPEs (BPEs #5 and #1 for A0) using the matched tag ID (e.g., 1 for A) and the block ID (e.g., 0 for A0). Since there is a one-to-one correspondence between BPEs and data sub-blocks, the corresponding sub-blocks are then read out of the data array (e.g., the sub-blocks #5 and #1 for A0) and decompressed.

On the other hand, in case of a cache miss, DCC must allocate the compressed block in the data array. A cache miss occurs when the block is not available in the cache even if its super-block is available. If its super-block is available (Block Miss in Figure 4-4), the accessing block will be allocated in the data array, and its corresponding tag entry and BPEs will be updated. This might require replacing one or more cache blocks to make enough space for this block. If its super-block is not available (Super-Block Miss in Figure 4-4), I might need to replace another super-block (e.g., the least recently used one). In this case, the blocks belonging to the victim super-block are evicted from the LLC as well. I handle the eviction process in the background by storing the victim super-blocks in a small buffer until all of their blocks are evicted from the cache. In this way, their tag entries can be released to allocate the new super-blocks.

Unlike conventional caches, on a write (or update) to a compressed cache, the block compressed size might change. To fit a larger block, DCC needs more sub-blocks, which may force a block (or a super-block) eviction. On the other hand, if the new compressed size is smaller, DCC would deallocate the unused sub-blocks, and update the corresponding tag entry and BPEs.

When DCC allocates a block or updates an existing block, it allocates from a set of free sub-blocks in the corresponding set. In the presented design, free sub-blocks are those pointing to invalid blocks. Since both the tag array and the back pointer array are accessed in parallel on a cache lookup, the cache controller gets the set of free data sub-blocks by finding those whose corresponding BPEs are pointing to invalid blocks. Thus, the cache controller always makes sure free sub-blocks are pointing to invalid blocks. An alternative design is to use an extra bit per BPE representing its validity, which would slightly increase area, but might simplify the logic.

4.3.3 Co-DCC: Reducing Internal Fragmentation by Co-Compacting Super-Blocks

DCC uses sub-blocks to reduce internal fragmentation, but it still limits the benefits due to internal fragmentation within sub-blocks. For example, DCC would allocate a 16-byte sub-block for a 10-byte compressed block. Compressing cache blocks and compacting them to byte granularity eliminates internal fragmentation but at high hardware overheads (discussed in Section 4.6). Using larger block sizes can also help reducing internal fragmentation by packing more data in the same space. However, increasing cache block size can lead to cache pollution and higher energy overheads [84].

Co-DCC exploits spatial locality to further optimize DCC to reduce internal fragmentation. As I show in earlier sections, for many applications, neighboring blocks co-reside in the LLC. Co-DCC exploit spatial locality, and treats super-blocks (e.g., a quad) as one large block. It

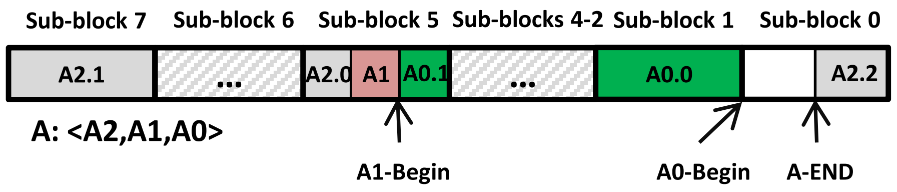


Figure 4-5: Co-DCC co-compaction

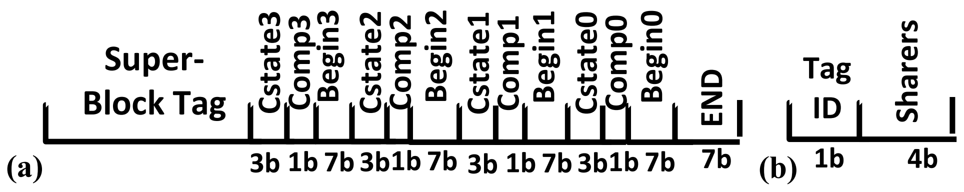


Figure 4-6: (a) One Co-DCC tag entry (b) One Co-DCC BPE.

dynamically compacts compressed blocks of one super block into the same set of sub-blocks. By co-compacting super-blocks, Co-DCC can get some of the benefits of BytePack (packing the compressed blocks at byte granularity) with much lower overheads, as shown in Section 4.6.1. In the next sub-section, I show how Co-DCC works, and how it can be integrated to DCC design with small changes.

4.3.3.1 Co-DCC Design

Co-DCC operates mostly similar to DCC, except for co-compacting super-blocks. When allocating a block to an existing super-block, Co-DCC compacts and stores the compressed block with the existing blocks of the same super-block. Figure 4-5 shows an example of how Co-DCC works for the same configuration used in Figure 4-3. In this example, Co-DCC stores and co-compacts A0, A1 and A2, which all belong to the super-block A, in chronological order in a 2-way set associative data set. When allocating block A1, since it fits in less than a sub-block, it shares a sub-block with A0 (in the sub-block #5). When A2 is allocated, A2 can also share some space (A2.0) with A0 and A1 in the sub-block #5. Its remaining sub-blocks (A2.1 and A2.2) need to be allocated in free sub-blocks of the set. In this example, there is not enough in-order space available for A2 if we want to share sub-block #5 among these blocks. Therefore, unlike DCC that stores the sub-blocks of a block in order, Co-DCC stores them not necessarily in order but in a round-robin fashion (e.g., block A2 in Figure 4-5). In this way, it will not need to move blocks if there is not enough in-order space available when co-compacting them. However, this design can slightly increase access latency (described in Section 4.4).

Co-DCC can be integrated with DCC design with some small changes in the tag array and the back pointer array. Figure 4-6 shows one Co-DCC tag entry and one BPE for the same configuration used in Figure 4-3. Since the first byte of a compressed block can be stored anywhere in a data sub-block (e.g., A2.0 in Figure 4-5), Co-DCC tracks each block's starting byte separately in its corresponding tag entry (e.g., 7-bit Begin0 in Figure 4-6(a)). Co-DCC also tracks the last occupied byte of each super-block in its corresponding tag entry (7-bit End in Figure 4-6(a)). When allocating a new block to an existing super-block, Co-DCC stores it next to this last byte if there is free space in that sub-block, and updates this pointer.

Unlike DCC, where each data sub-block belongs to only one block, Co-DCC can share one sub-block among multiple blocks of the same super-block. For example, A0.1, A1, and A2.0 share the sub-block #5 in Figure 4-5. Therefore, each Co-DCC BPE tracks its sharers by storing a small bit-vector (e.g., 4-bit Sharers in Figure 4-6(b)). Each bit of the sharers bit-vector shows if its corresponding block shares that sub-block. This information will slightly increase the LLC area (Section 4.5), but allows Co-DCC to fit more blocks in the cache by reducing internal fragmentation.

4.4 A Practical Design for DCC

(Co-)DCC can be integrated into the LLC of a recent commercial design with relatively little additional complexity and, more importantly, no need for an alignment network. The AMD Bulldozer implements an 8MB LLC that is broken into four 2MB sub-caches, each sub-cache consists of four banks that can independently service cache accesses [85]. Figure 4-7 illustrates the data array of one bank in the LLC and shows how it is divided into 4 sequential regions (SR). Each sequential region runs one phase (i.e., half a cycle) behind the previous region and contains a quarter of a cache block (i.e., 16 bytes). Figure 4-7 shows how block A0's four 16-byte sub-blocks (e.g., A0.0–A0.3) are distributed to the same row in each sequential region. Each subsequent sequential region receives the address a half cycle later and takes a half cycle longer to return the data. Thus, a 64-byte block is returned in a burst of four cycles on the same data bus. For example, A0.1 is returned one cycle after A0.0 in Figure 4-8(a).

DCC requires only a small change to the data array to allow non-contiguous sub-blocks. In Figure 4-7, block B1 is compressed into 2 sub-blocks (B1.0 and B1.1), stored in sequential regions #1 and #2, but not in the same row. To select the correct sub-block, DCC must send additional address lines (i.e., 4 bits for a 16-way-associative cache) to each sequential region (illustrated by the dotted lines in Figure 4-7). DCC must also enforce the constraint that a compressed block's sub-blocks are allocated to different sequential regions to prevent sequential region conflicts.

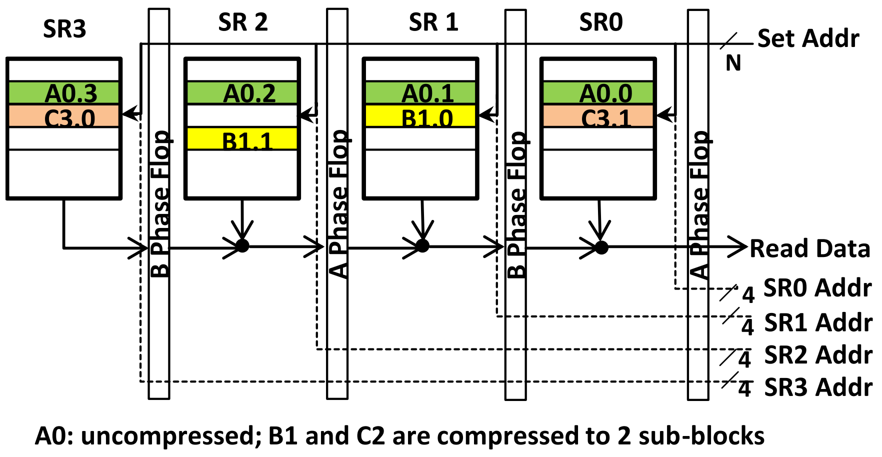


Figure 4-7: (Co-)DCC Data Array Organization.

Figure 4-8(b) illustrates DCC timing when reading block B1. As described in Section 4.3, the back pointer array is accessed in parallel with the tag array. The sub-block selection logic finds the BPEs corresponding to this block using its block ID (derived from its address) and the matched tag ID, which is found by the tag match logic. The sub-block selection logic can only be partially overlapped with the tag match logic since it needs the matched tag ID. To calculate the latency overhead of the sub-block selection, I implemented the tag match and the sub-selection logic in Verilog, synthesized in 45nm and scaled to 32nm [86]. The sub-block selection logic adds less than half a cycle to the critical path, which I conservatively assume increases the access latency by one cycle. Figure 4-8(b) shows how the matching sub-blocks are returned and fed directly into the decompression logic, which accepts 16-byte per cycle and has a small FIFO buffer to rate match. Decompression starts as soon as the first sub-block arrives (e.g., B1.0), which depends upon which sequential region it resides in. Since sub-block B1.0 resides in sequential region 1, there is one extra cycle (worst case is 3 cycles). Note that because the decompression latency is deterministic (9 cycles), DCC can determine at the end of sub-block selection when the data will be ready and whether the decompression hardware can be bypassed.

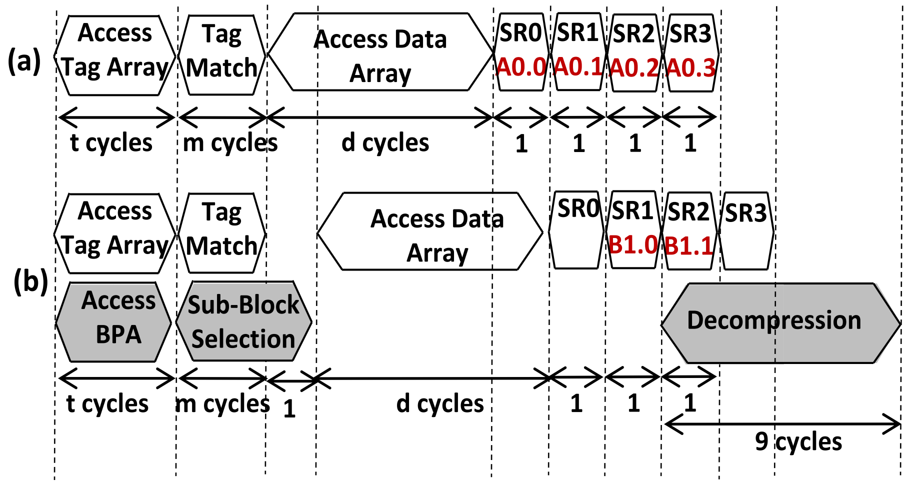


Figure 4-8: (a) Timing of a conventional cache and (b) DCC.

Thus, even though completion times vary, DCC has ample time to arbitrate for the response network.

Figure 4-7 also shows how block C3 is allocated by Co-DCC. Co-DCC also stores sub-blocks of a block in different regions, but allocates them in round-robin fashion and not necessarily in order. Therefore, Co-DCC cannot necessarily start decompression as soon as it reads the first sub-block (e.g., C3.1 will be read out first before C3.0). To handle these cases, Co-DCC must buffer the sub-blocks and pass them to the decompression logic in order. The decompression logic must also pre-align the first sub-block, since the compressed block doesn't necessarily start in the first byte. The reordering and pre-alignment add up to 3 additional cycles compared to DCC.

Table 4-1: Simulation parameters.

Cores	OOO, 3.2 GHz, 4-wide issue, 128-entry Instruction Window.
L1I$/L1D$	Private, 32-KB, 8-way, 2 cycles, HP transistors.
L2 $	Private, 256-KB, 8-way, 10 cycles, HP transistors.
L3 $	Shared, 8-MB, 16-way, 8 banks, 30 cycles, LSTP transistors.
Main Memory	4GB, 16 Banks, 800 MHz bus frequency DDR3, 60.35 nJ per Read, 66.5 nJ per Write, and 4.25W static power.

4.5 Experimental Methodology

4.5.1 Simulation Configurations

I evaluate (Co-)DCC using a full-system simulator based on GEMS [81]. I model a multicore system with three levels of cache hierarchy (Table 4-1) [74]. I use an 8MB LLC that is broken into 8 banks, each divided into 4 sequential regions. Note that although I use a different cache configuration than AMD Bulldozer LLC, I model the timing and allocation constraints of sequential regions at the LLC in detail, as discussed in Section 4.4. I use CACTI [87] to model power at 32nm. I also use a detailed DRAM power model developed based on the Micron Corporation power model [88] with energy per operation listed in Table 4-1. In this section, I report total system energy that include energy consumption of processors (cores + caches), on-chip network, and off-chip memory.

Table 4-2: Configurations.

Baseline	Conventional **16-way-associative 8MB LLC.**
2X Baseline	Conventional **32-way-associative 16MB LLC.**
FixedC	**2x tags** per set (i.e., **32** tags per set). Each cache block is compressed to half if compressible.
VSC-2X	**2x tags** per set (i.e., **32** tags per set). A block is compressed into 0-4 **16-byte sub-blocks**.
DCC	**Same number of tags** per set (i.e., **16** tags per set). Each tag tracks up to 4 blocks (**4-block Super-Blocks**). Blocks are compressed individually to 0-4 **16-byte sub-blocks**.
Co-DCC	Similar to DCC, except it dynamically co-compacts blocks of the same super-blocks.

Table 4-2 shows the configurations I use. For (Co-)DCC, I use 4-block super-blocks, 64-byte blocks, and 16-byte sub-blocks. With these parameters, DCC has similar area overhead as FixedC and VSC-2X (Section 4.6). Alternative super-block and sub-block sizes can be used. I use 4-block super-blocks, since not all workloads would benefit from larger super-blocks due to their limited spatial locality. Using smaller sub-blocks also potentially improves compression effectiveness by reducing internal fragmentation, but at the cost of higher hardware complexities and overheads (discussed in Section 4.6).

4.5.2 Workloads

Our evaluations use representative multi-threaded and multi-programmed workloads from Commercial workloads [89], SPEC-OMP [92], PARSEC [91], and mixes of SPEC CPU2006 benchmarks, summarized in Table 4-3. I evaluate eight multi-programmed workloads with different mixes of compute-bound and memory intensive benchmarks. Each workload consists of 8 threads evenly divided among the named Spec2006 benchmarks. For example, cactus-mcf-milc-bwaves runs two copies of each of the four benchmarks.

Table 4-3: Workloads.

Suite	Workloads
Commercial	apache, jbb, oltp, zeus
SPEC-OMP	ammp, applu, equake, mgrid, wupwise
PARSEC	blackscholes, canneal, freqmine
Spec2006 (denoted as m1-m8)	bzip2, libquantum-bzip2, libquantum, gcc, astar-bwaves, cactus-mcf-milc-bwaves, gcc-omnetpp-mcf-bwaves-lbm-milc-cactus-bzip, omnetpp-lbm

Figure 4-9 shows the sensitivity of our workloads to the LLC capacity and the LLC access latency. Compressed caches in general benefit cache capacity sensitive workloads by providing higher effective cache capacity. On the other hand, they might hurt cache latency sensitive workloads due to the decompression latency. I categorize our workloads as cache latency sensitive if they observe more than 1% runtime slowdown compared to Baseline when I use the same size cache with 9 cycles extra LLC access latency, which represents the decompression latency. Many of our workloads (e.g., freqmine and oltp) are sensitive to cache latency and observe up to 6% (for oltp) slow down with the slower cache. I also categorize our workloads

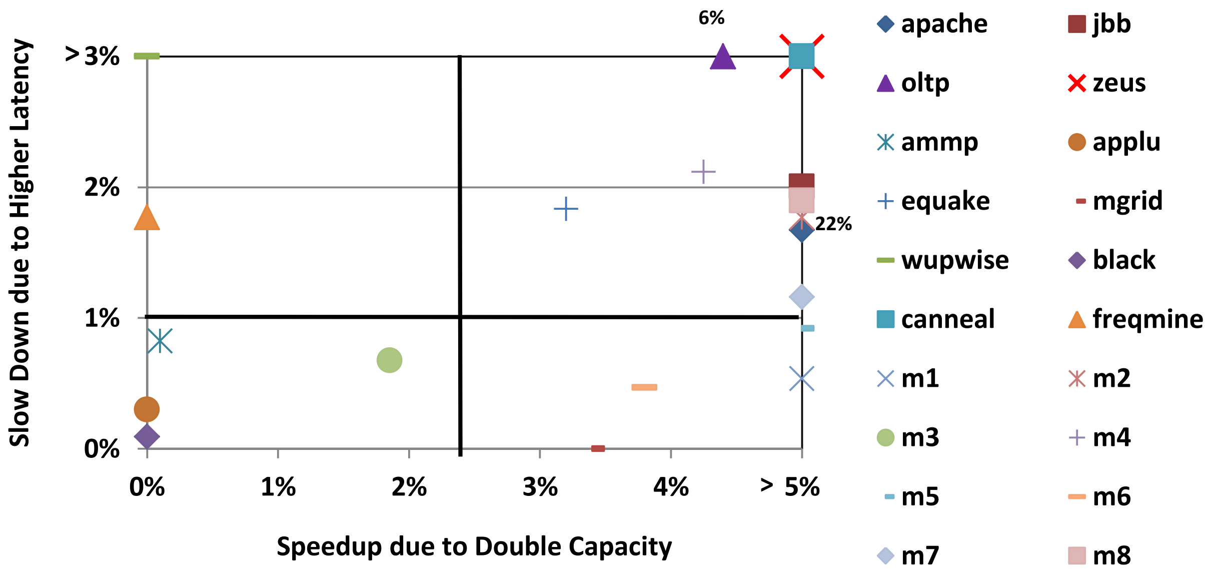


Figure 4-9: Cache sensitivity of our workloads.

that observe more than 2% speedup with double LLC capacity (with the same access latency as Baseline) as cache capacity sensitive. Our workloads have a wide range of sensitivity to cache capacity (maximum 22% speedup for apache). Among our workloads, ammp, applu, blackscholes, and libquantum are cache insensitive. I run each workload for approximately 500M instructions with warmed up caches. I use a work-related metric, run each workload for a fixed number of transactions/iterations and report the average over multiple runs to address workload variability [90].

4.6 Evaluation

4.6.1 Area and Power

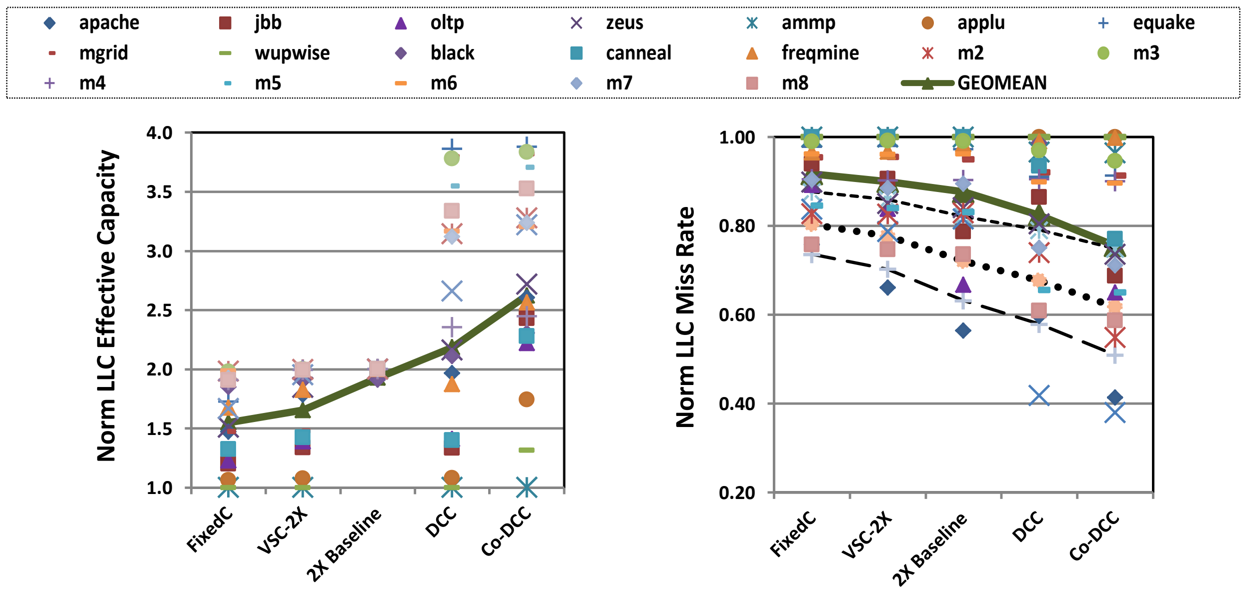
Compressed caches can increase cache area due to their extra metadata. Table 4-4 shows the quantitative area overheads of DCC, Co-DCC, FixedC and VSC-2X over the same size conventional cache (16-way-associative 8MB LLC) with the parameters in Table 4-1 and Table 4-2. DCC uses the same number of tags as Baseline, but almost doubles the per-block metadata largely due to the back pointers. However, since the data array is much larger than the tag array,

Table 4-4: LLC area overheads of different compressed caches over the conventional cache.

Components	**DCC**	**Co-DCC**	**FixedC/VSC-2X**	**VSC-3X**	**VSC-4X**	**DCC-BytePack**
Tag Array	2.1%	11.3%	6.3%	12.7%	18.8%	2.1%
Back Pointer Array	4.4%	5.4%	0%	0%	0%	70.6%
Compressors	0.6%	0.6%	0.6%	0.6%	0.6%	0.6%
Decompressors	1.2%	1.2%	1.2%	1.2%	1.2%	1.2%
Total Area Overhead	**8.3%**	**18.5%**	**8.1%**	**14.5%**	**20.6%**	**74.5%**

Cacti calculates the overall LLC area overhead as about ~6% [87]. DCC's area overhead is similar to FixedC and VSC-2X, which track twice as many tags per set (e.g., 32 tags per 16 blocks). Co-DCC increases metadata stored per block, as discussed in Section 4.3.3, resulting in 16% area overhead compared to Baseline. Co-DCC still has less area overhead than naively quadrupling the number of tags (VSC-4X). It also incurs much lower overhead compared to a DCC configuration with no packing constraint. DCC-BytePack (i.e., packing compressed blocks at byte granularity) can increase compression effectiveness by reducing internal fragmentation. However, using 1-byte sub-blocks requires 16 times more BPEs per set than (Co-)DCC with 16-byte sub-blocks. BytePack would also require a complex alignment network to compact the bytes into 16-byte sub-blocks before passing them to the decompression hardware. Table 4-4 also includes the area overhead of (de-)compression units. Since C-PACK+Z's decompressors produce 8 bytes per cycle, I match the cache bandwidth by considering two decompressors per cache bank. Since compression is not on the critical path, I consider one compressor per bank. For the LLC configuration in Table 4-1, we need 8 compressors and 16 decompressors resulting to an extra 1.8% area overhead.

Compressed caches can also increase the LLC per-access dynamic power and the LLC static power due to their extra metadata. DCC, similar to FixedC and VSC-2X, increases the LLC per-access dynamic power by 2% and the LLC static power by 6%. Co-DCC also incurs 6% overhead on the LLC per-access dynamic power and 16% LLC static power overhead [87]. I model these overheads as well as the power overheads of (de-)compression in detail.



(a) Normalized LLC effective capacity **(b) Normalized LLC miss rate**

Figure 4-10: The LLC effective capacity and the LLC miss rate normalized to Baseline.

4.6.2 Effective Cache Capacity

Result 1: By exploiting spatial locality, DCC achieves on average 2.2 times (up to 4 times) higher LLC effective capacity compared to Baseline, resulting in 18% lower LLC miss rate on average and up to 38% lower LLC miss rate.

Result 2: Co-DCC further improves the effective cache capacity by co-compacting the blocks in a super-block. It achieves on average 2.6 times and up to 4 times higher effective capacity and on average 24% and up to 42% lower LLC miss rate.

Result 3: (Co)-DCC provides significantly higher effective cache capacity and lower miss rate than FixedC and VSC-2X. (Co-)DCC also performs on average better than 2X Baseline with much lower area overhead.

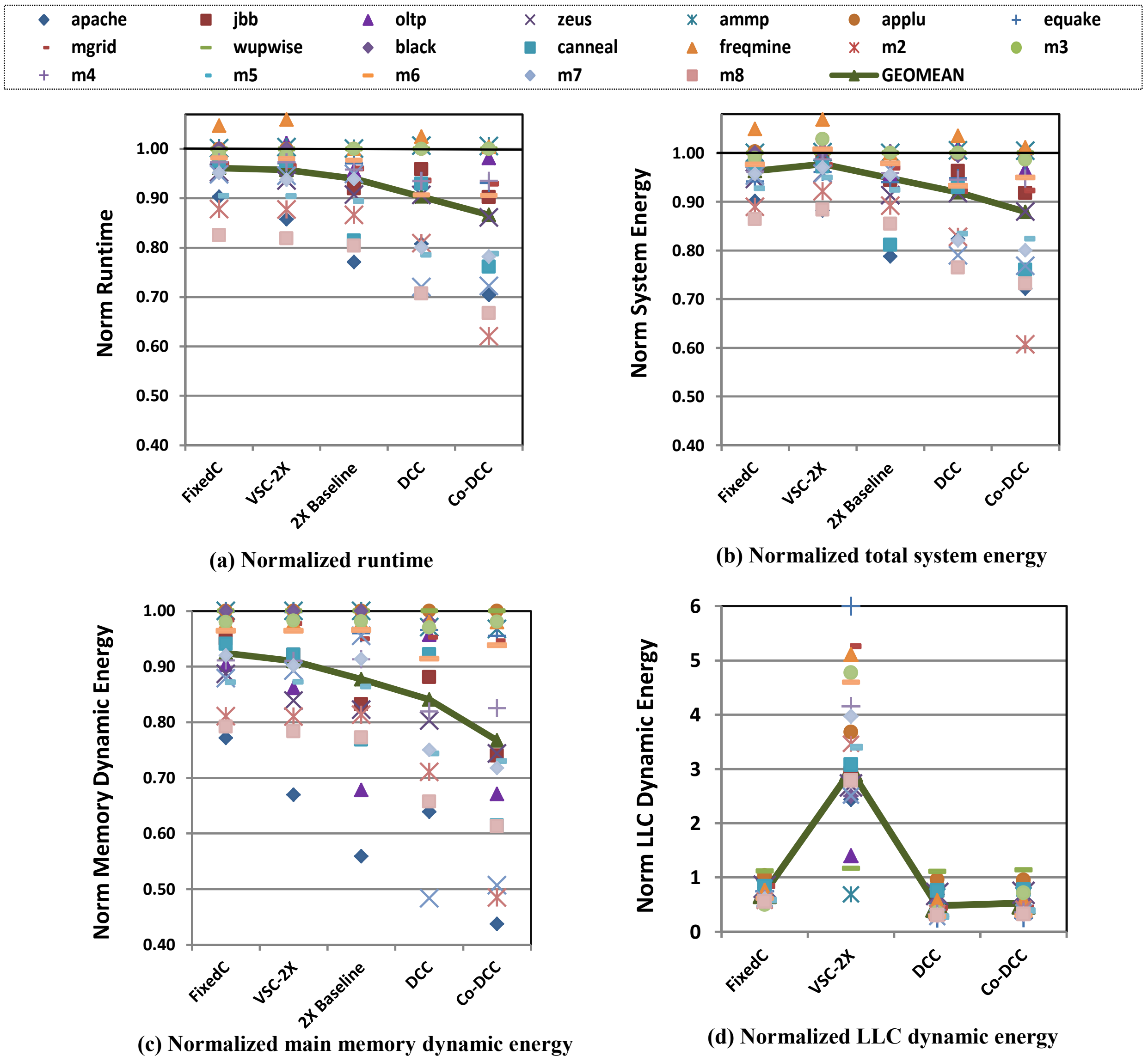
Compressed caches improve effective capacity by fitting more blocks in the same space. They can achieve the benefits of larger cache sizes with lower area and power overheads. Figure 4-10(a) and Figure 4-10(b) plot the LLC effective capacity and the LLC miss rate of different techniques normalized to Baseline. I calculate the effective cache capacity by periodically counting valid LLC cache blocks. I measure the LLC miss rate as the total number of misses per thousand executed instructions (MPKI). Figure 4-10(b) also plots the average LLC miss rate reduction predicted using the well-known power law for miss rate [73] in dashed lines. This model predicts the cache miss rate will be inversely proportional to the increased capacity with an scaling factor typically set to 0.5 (i.e., "square root" power law), 0.3, or 0.7 (the higher the scaling factor, the lower the predicted miss rate). The average improvement I found for our workloads is less than what these models predict. I hypothesis this is because our workloads represent a wide range of cache sensitivities and I am not picking only highly cache sensitive ones.

DCC can significantly improve the LLC effective capacity and the LLC miss rate for many applications by fitting more compressed blocks. On average, DCC provides 2.2x (i.e., 17.6MB) higher effective capacity and 18% lower LLC miss rate compared to Baseline. DCC benefits differ per workload, depending on the workload's sensitivity to cache capacity, compression ratio, and spatial locality. It achieves highest benefits for cache sensitive workloads with good compressibility and spatial locality (e.g., apache and omnetpp-lbm/m8). Workloads with low spatial locality (e.g., canneal) or low compression ratio (e.g., wupwise) observe lower improvements. Cache insensitive workloads (e.g., blackscholes) also do not benefit from compression.

Co-DCC further improves compression effectiveness by reducing internal fragmentation within data sets. Co-DCC achieves, on average, 2.6x higher effective capacity (i.e., 20.8MB) and 24% lower miss rate than Baseline. By fitting more compressed blocks in the cache, compared to DCC, Co-DCC can further reduce the LLC miss rate for almost half of our workloads, including commercial workloads (e.g., 18% lower miss rate for jbb), canneal, and some of our Spec2006 mixes (e.g., 19% lower miss rate for libquantum-bzip2/m2). By co-compacting super-blocks, Co-DCC gets some of the benefits of the idealized BytePack with much lower hardware overheads, as discussed in Section 4.6.

Compared to FixedC and VSC-2X, (Co-)DCC provides higher LLC effective capacity and lower miss rate. Both FixedC and VSC-2X can at most double effective cache capacity compared to Baseline (i.e., 16MB). FixedC achieves on average 1.5x higher effective capacity and 8% lower miss rate than Baseline. VSC-2X provides slightly higher benefits (1.7x effective capacity, and 10% lower miss rate). Increasing VSC tag space can improve its benefits. For example, VSC-4X has similar miss rate reduction as DCC, but with 2.6x higher area overhead.

Compared to 2X Baseline, (Co-)DCC effectively more than doubles cache capacity with lower overheads. DCC achieves higher LLC effective capacity than 2X-Baseline for majority of our workloads. It provides lower LLC miss rate reduction than 2X-Baseline (within 27%) for apache, jbb, oltp and gcc, which have lower compression ratio and spatial locality compared to other workloads. For these workloads, Co-DCC provides similar or better LLC miss rate reduction than 2X-Baseline by reducing internal fragmentation.



(a) Normalized runtime

(b) Normalized total system energy

(c) Normalized main memory dynamic energy

(d) Normalized LLC dynamic energy

Figure 4-11: performance and energy normalized to Baseline.

4.6.3 Overall Performance and Energy

Result 4: DCC and Co-DCC improve the LLC efficiency and boost system performance by 10% (up to 29%) and 14% (up to 38%) on average, respectively.

Result 5: DCC and Co-DCC save on average 8% (up to 24%) and 12% (up to 39%) of system energy, respectively, due to shorter runtime and fewer accesses to the main memory.

Result 6: DCC and Co-DCC achieve respectively 2.5x and 3.5x higher performance improvements, and 2.2x and 3.3x higher system energy improvements compared to FixedC and VSC-2X.

Result 7: (Co-)DCC also improves the LLC dynamic energy by about 50% on average due to accessing fewer bytes. On the other hand, VSC-2X hurts the LLC dynamic energy for majority of our workloads due to its need for energy-expensive re-compactions.

By improving the LLC utilization and reducing accesses to the main memory (i.e., the lower LLC miss rate), (Co-)DCC significantly improves system performance over Baseline. Figure 4-11(a) plots runtime of different techniques normalized to Baseline. DCC and Co-DCC improve performance by 10% (up to 29% for omnetpp-lbm/m8) and 14% (up to 38% for libquantum-bzip2/m3) on average, respectively. For cache sensitive applications with medium-to-high compressibility and medium-to-high spatial locality (e.g., apache and zeus), (Co-)DCC achieves significant performance improvements by fitting more blocks in the cache. They provide lower improvements for applications with low spatial locality and low compression ratio (e.g., canneal and gcc). On the other hand, compressed caches, including (Co-)DCC, can hurt performance of workloads sensitive to the LLC access latency (e.g., freqmine) due to the decompression latency. (Co-)DCC hurts performance by less than 3% (for freqmine). Cache insensitive workloads also do not benefit from compressed caches. An adaptive technique can be employed to further reduce these overheads [20], which is orthogonal to our proposals.

(Co-)DCC significantly outperforms FixedC, VSC-2X and 2X-Baseline by effectively more than doubling the cache capacity. FixedC and VSC-2X limit compression effectiveness in improving system performance, achieving on average 4% and 5% performance improvements, respectively. (Co-)DCC outperforms 2X-Basline for majority of our workloads. 2X-Baseline performs better than DCC for six of our workloads (within 11% for canneal). These workloads have lower spatial locality (e.g. canneal), lower compression ratio (e.g., jbb), or higher sensitivity to cache latency (e.g., freqmine) than the rest of our workloads. Co-DCC improves performance for more workloads, providing slightly lower performance than 2X-Baseline only for three workloads (within 3% for freqmine).

(Co-)DCC improves system energy both due to shorter runtime and fewer accesses to the main memory. Figure 4-11(b) shows the total system energy of different techniques. DCC and Co-DCC reduce the total system energy by 8% (up to 24% for omnetpp-lbm/m8) and 12% (up to 39% for libquantum-bzip/m2) on average, respectively. Figure 4-11(c) plots the main memory dynamic energy for these techniques. (Co-)DCC significantly reduces the main memory dynamic energy by reducing the number of cache misses. Compared to FixedC and VSC-2X, (Co-)DCC achieves higher energy savings. Although VSC-2X provides slightly higher performance and lower main memory dynamic energy consumption than FixedC, its system energy saving is less due to its high overheads on the LLC dynamic energy. Figure 4-11(d) shows the dynamic energy of different compressed caches normalized to Baseline. FixedC, DCC and Co-DCC improve the LLC dynamic energy by 27%, 52% and 46% on average over Baseline, respectively. On the other hand, VSC-2X significantly increases the LLC dynamic energy (about 3x) by increasing the number of cache accesses.

I also measured the sensitivity of (Co-)DCC to different design parameters including the decompression latency and the LLC access latency. Our simulations (not shown here) show that reducing decompression latency (for the same C-PACK+Z algorithm) from 9 cycles to 3 cycles only slightly increases (Co-)DCC performance. It achieves on average 1% and up to 3% higher performance than the results shown in Figure 4-11(a). I also studied the sensitivity of (Co-)DCC to the LLC cache access latency. Our simulation results (not shown here) show that even reducing the LLC access latency to 20 cycles (33% faster LLC) does not significantly impact (Co-)DCC results.

4.7 Conclusions

In this work, I propose Decoupled Compressed Cache, which exploits spatial locality to improve both the performance and energy-efficiency of cache compression. DCC manages the cache at three granularities, tracking super-blocks while dynamically compressing and allocating single blocks as variable number of sub-blocks. It addresses the issues with conventional compressed caches, and achieves significantly higher LLC effective cache capacity while incurring low area overheads. It also decouples sub-blocks from the address tag to eliminate energy-expensive re-compaction when a block's size changes. A further optimized design (Co-DCC) reduces internal fragmentation in the cache by co-compacting super-blocks. I show that on average, DCC and Co-DCC reduce system energy by 8% and 12%, respectively, and improve performance by 10% and 14%, respectively, compared to the same size conventional cache. (Co-)DCC nearly doubles compression benefits compared to previous proposals with comparable overheads.

Chapter 5
Skewed Compressed Caches

5.1 Overview

A compressed cache design must balance three frequently-conflicting goals: i) tightly compacting variable-size compressed blocks to reduce internal fragmentation, ii) keeping tag overheads low, and iii) allowing fast lookups by eliminating the need for extra metadata to locate compressed blocks. Previous compressed cache designs, including our proposal DCC, achieved at most two of these three goals. As we showed in Table 3-1, the earliest compressed caches do not support variable compressed block sizes [39][47][48][45], allowing fast lookups with relatively low area overheads, but achieve lower compression effectiveness due to internal fragmentation. More recent designs [20][46][50] improve compression effectiveness using variable-size compressed blocks, but at the cost of extra metadata and indirection latency to locate a compressed block. For example, DCC requires per-block back pointers to locate a block. DCC also complicates cache management, specifically replacements, due to managing blocks and super-blocks separately on evictions.

In this chapter, we propose Skewed Compressed Cache (SCC), which achieves all three goals. SCC exploits the fact that most workloads exhibit both (1) spatial locality (i.e., neighboring blocks tend to reside in the cache at the same time), and (2) compression locality (i.e., neighboring blocks tend to compress similarly) [61]. Like DCC, SCC exploits spatial locality by tracking super-blocks, e.g., an aligned, adjacent group of blocks (e.g., eight 64-byte blocks). Using super-blocks allows SCC to track up to eight times as many compressed blocks with little additional metadata. Unlike DCC, SCC also exploits compression locality by compacting neighboring blocks with similar compression ratio into the same physical data entry, tracking them with one tag.

SCC does this using a novel *sparse super-block tag*, which tracks anywhere from one block to all blocks in a super-block, depending upon their compressibility. SCC compacts neighboring blocks to the same data block and tracks them with one tag, if they are similarly compressible. For example, a single sparse super-block tag can track: all eight blocks in a super-block, if each block is compressible to 8 bytes; four adjacent blocks, if each is compressible to 16 bytes; two adjacent blocks, if each is compressible to 32 bytes; and only one block, if it is not compressible. By allowing variable compressed block sizes—8, 16, 32, and (uncompressed) 64 bytes—SCC is able to tightly compact blocks and achieve high compression effectiveness.

Using sparse super-block tags allows SCC to retain a direct, one-to-one tag-data mapping, but also means that more than one tag may be needed to map blocks from the same super-block. SCC minimizes conflicts between blocks using two forms of skewing. First, it maps blocks to different cache ways based on their compressibility, using different index hash functions for each cache way [93]. To spread all the different compressed sizes across all the cache ways, the hash function used to index a given way is a function of the block address. Second, SCC skews compressed blocks across sets within a cache way to decrease conflicts [94][95] and increase effective cache capacity.

Compared to DCC, SCC eliminates the extra metadata needed to locate a block (i.e., the back pointers), reducing tag and metadata overhead. SCC's direct tag-data mapping allows a simpler data access path with no extra latency for a tag-data indirection. SCC also simplifies cache replacement. On a conflict, SCC always replaces one sparse super-block tag and all of the one to eight adjacent blocks packed into the corresponding data entry. This is much simpler than DCC, which may need to replace blocks that correspond to multiple super-blocks as DCC tracks all blocks of a super-block with only one tag.

Compared to conventional uncompressed caches, SCC improves cache miss rate by increasing effective capacity and reducing conflicts. In our experiments, SCC improves system performance and energy by on average 8% and 6% respectively, and up to 22% and 20% respectively. Compared to DCC, SCC achieves comparable or better performance, with a factor of four lower area overhead, a simpler data access path, and a simpler replacement policy.

This chapter is organized as follows. We discuss background on skewed associative caching in Section 2. Then, Section 3 presents our proposal: the Skewed Compressed Cache. Section 4 explains our simulation infrastructure and workloads. In Section 5, we discuss the overheads of compressed caches. We present our evaluations in Section 6. Finally, Section 7 concludes the chapter.

5.2 Skewed Associative Caching

SCC builds on ideas first introduced for skewed-associative caches. In a conventional N-way set-associative cache, each way is indexed using the same index hash function. Thus

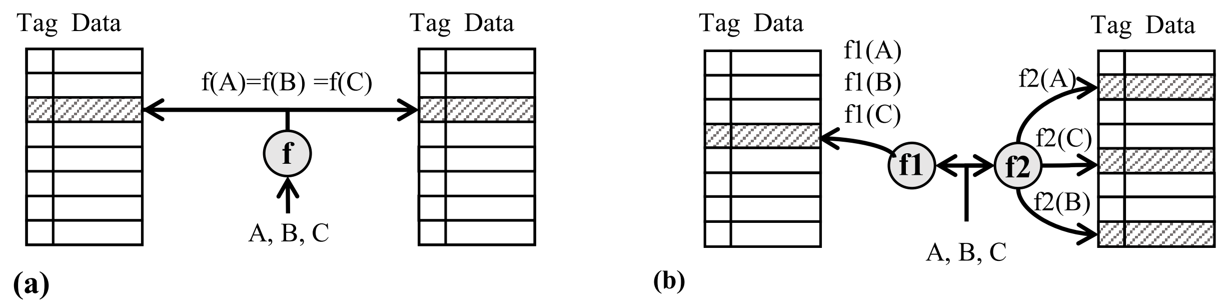


Figure 5-1: (a) two-way set associative cache (b) skewed associative cache.

conflict misses arise when more than N cache blocks compete for space in a given set. Increasing associativity reduces conflict misses, but typically causes an increase in cache access latency and energy cost. Skewed associative caches [94][95] index each way with a different hash function, spreading out accesses and reducing conflict misses.

Figure 5-1 (a) shows a simple 2-way associative cache, which indexes all cache ways with the same function. In this example, blocks A, B, and C all map to the same set. Thus, only two of these blocks can stay in the cache at any time. Figure 5-1 (b) illustrates a skewed associative cache, which indexes each cache way with a different hash function. In this example, even though blocks A, B, and C map to the same set using function f1, they map to different sets using function f2 in the second cache way. In this way, all three of these blocks can reside in the cache at the same time. By distributing blocks across the sets, skewed associative caches typically exhibit miss ratios comparable to a conventional set-associative cache with twice the ways [94][95].

Skewed associativity has also been used to support multiple page sizes in the same TLB [94][115], at the cost of reduced associativity for each page size. Using different, page-size specific hash functions for each way, such a TLB can look for different size page table entries in parallel. In this work, we use a similar skewing technique but use compressed size, rather than page size, to select the appropriate way and hash-function combinations.

5.3 Skewed Compressed Cache

Previously proposed compressed caches either do not support variable-size compressed blocks [39][47][48][45] or need extra metadata to find a compressed block, increasing overhead and complexity [20][46][50]. SCC stores neighboring compressed blocks in a power-of-two number of sub-blocks (e.g., 1, 2, 4, or 8 8-byte sub-blocks), using sparse super-block tags and a skewed associative mapping that preserves a one-to-one direct mapping between tags and data.

SCC builds on the observation that most workloads exhibit (1) spatial locality, i.e., neighboring blocks tend to simultaneously reside in the cache, and (2) compression locality, i.e., neighboring blocks often have similar compressibility [61]. SCC exploits both types of locality to compact neighboring blocks with similar compressibility in one physical data entry (i.e., 64 bytes) if possible. Otherwise, it stores neighbors separately.

SCC differs from a conventional cache by storing a sparse super-block tag per data entry. Like a conventional super-block (aka sector) cache, SCC's tags provide additional metadata that can track the state of a group of neighboring blocks (e.g., up to eight aligned, adjacent blocks). However, SCC's tags are sparse because—based on the compressibility of the blocks—they may map only 1 (uncompressed), 2, or 4 compressed blocks. This allows SCC to maintain a conventional one-to-one relationship between a tag and its corresponding data entry (e.g., 64 bytes).

SCC only maps neighboring blocks with similar compressibility to the same data entry. For example, if two aligned, adjacent blocks are each compressible to half their original size, SCC will allocate them in one data entry. This allows a block's offset within a data entry to be directly determined using the appropriate address bits. This eliminates the need for additional metadata (e.g., back pointers in DCC [21]) to locate a block.

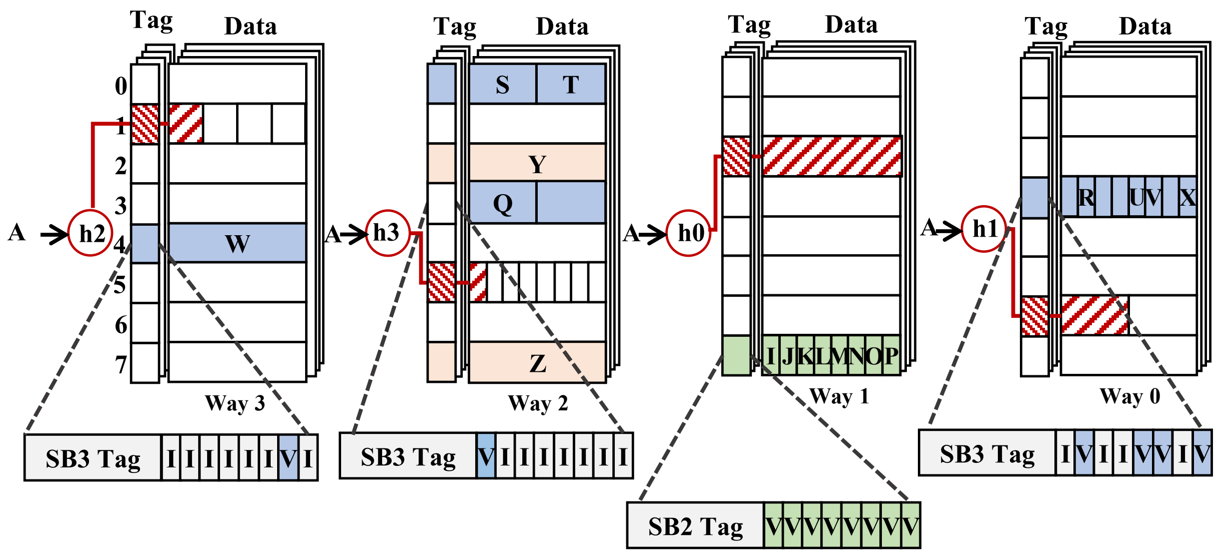
SCC's cache lookup function is made more complicated because the amount of data mapped by a sparse super-block tag depends upon the blocks' compressibility. SCC handles this by using a block's compressibility and a few address bits to determine in which cache way(s) to place the block. For example, for a given super-block, uncompressed blocks might map to cache way #0, blocks compressed to half size might map to cache way #2, etc. Using address bits in the placement decision allows different super-blocks to map blocks with different compressibility to different cache ways. This is important, as it permits the entire cache to be utilized even if all blocks compress to the same size.

To prevent conflicts between blocks in the same super-block, SCC uses different hash functions to access ways holding different size compressed blocks. On a cache lookup, the same address bits determine which hash function should be used for each cache way. Like all skewed associative caches, SCC tends to have fewer conflicts than a conventional set-associative cache with the same number of ways.

5.3.1 SCC Functionality

Figure 5-2 illustrates SCC functionality using some examples. This figure shows a 16-way cache with 8 cache sets. The 16 cache ways are divided into four way groups, each including four cache ways. For the sake of clarity, Figure 5-2 only illustrates super-blocks that are stored in the first way of each way group. This example assumes 64-byte cache blocks, 8-block super-blocks, and 8-byte sub-blocks, but other configurations are possible. A 64-byte cache block can compress to any power-of-two number of 8-byte sub-blocks (i.e., 1, 2, 4, or 8 sub-blocks). Eight aligned neighbors form an 8-block super-block. For example, blocks I—P belongs to SB2.

SCC associates one sparse super-block tag with each data entry in the data array. Each tag can map (1) a single uncompressed cache block, (2) two adjacent compressed blocks, each compressed to 32 bytes, (3) four adjacent compressed blocks, each compressed to 16 bytes, or (4) eight adjacent compressed blocks, each compressed to 8 bytes. A tag keeps appropriate per-block metadata (e.g., valid and coherence) bits, so it may not be fully populated. If all eight neighbors exist and are compressible to one 8-byte sub-block each, SCC will compact them in one data entry, tracking them with one tag. For example, all blocks of SB2 are compacted in one data entry in set #7 of way #1. SCC tracks them with the corresponding tag entry with the states of all blocks set as valid (V in Figure 5-2). If all cache blocks were similarly compressible, SCC would be able to fit eight times more blocks in the cache compared to a conventional uncompressed cache. On the other hand, in the worst-case scenario when there is no spatial locality (i.e., only one out of eight neighbors exists in the cache) or blocks are not compressible, SCC can still utilize all cache space by allocating each block separately. For example, there are only blocks Y and Z from SB4 present in the cache, and neither are compressible. Thus, SCC stores them separately in two different sets in the same way group, tracking them separately with their corresponding tags.



SB1: A_ _ _ _ _ _ _; SB2: IJKLMNOP; SB3: QRSTUVWX; SB4: YZ_ _ _ _ _ _

Figure 5-2: Skewed Compressed Cache.

SCC uses a block's compressibility or compression factor (CF) and a few address bits to determine in which way group to place the block. A block's compression factor is zero if the block is not compressible, one if compressible to 32 bytes, two if compressible to 16 bytes, and three if compressible to 8 bytes. For instance, in Figure 5-2, block A maps to a different set in each cache way depending on its compressibility, shown in hatched (red) entries. SCC allocates A in way group #0, #1, #2, or #3 if A is compressible to 32 bytes (4 sub-blocks), 64 bytes (8 sub-blocks), 8 bytes (1 sub-block), or 16 bytes (2 sub-blocks), respectively. These mappings would change for a different address, so that each cache way would have a mix of blocks with different compression ratios. For instance, SCC allocates block A and block I in cache way #1, if A is uncompressible and I is compressed to 8 bytes (1 sub-block). Using this mapping technique, for a given block, its location determines its compression ratio. This eliminates the need for extra metadata to record block compressibility.

Although SCC separately compresses blocks, it maps and packs neighbors with similar compressibility into one physical data entry. For example, SCC compacts blocks I to P (SB2) into a single physical data entry (set #7 of way #1) as each block is compressed to 8 bytes. However, when neighboring blocks have different compressibility, SCC packs them separately into different physical data entries. For instance, blocks of SB3 (blocks Q to X) have three different compression ratios. SCC allocates blocks R, U, V, and X, which are compressible to one sub-block each, in one physical data entry (set #3 of way #0). It tracks them with the corresponding tag entry (also shown in Figure 5-2) with valid states for these blocks. It stores adjacent blocks S and T in a different physical entry since each one is compressed to four sub-blocks. It also stores block Q in way #2 as it is compressible to 32B. Finally, it allocates block W separately as it is not compressible, tracking it with a separate sparse super-block tag shown in Figure 5-2.

Within a physical data entry, a block offset directly corresponds to the block position in its encompassing super-block. In Figure 5-2, for example block X is the first block of SB3, similarly its position in the physical data entry in cache way #0 is fixed in the first sub-block. In this way, unlike previous work, SCC does not require any extra metadata (e.g., back pointers [21] or forward pointers [50]) to locate a block in the data array. By eliminating the need for extra pointers, SCC simplifies data paths, provides fast lookups, lowers area overhead and design complexity, while still allowing variable compressed sizes.

While eliminating extra metadata simplifies SCC's design, it has the potential to hurt cache performance by increasing conflict misses and lowering effective cache associativity. A conventional 16-way set-associative cache can allocate a block in any cache way, but SCC

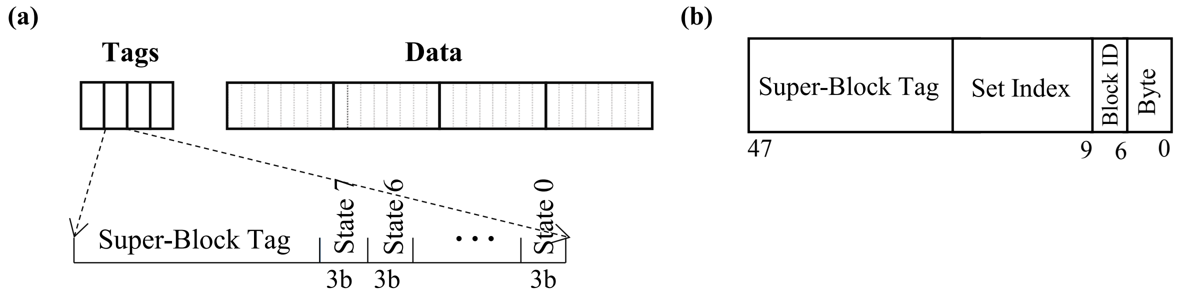


Figure 5-3: (a) One set of SCC (b) Address.

restricts a block to a 4-way way group based on the block's compression factor. For example, when storing block A with compressed size of 16B, SCC can store it only in one of the four cache ways (including way #3) grouped together in Figure 5-2. To mitigate the effect of this restriction, SCC employs skewing inside way groups, indexing each cache way with a different hash function to spread out accesses. This helps to reduce conflict misses and increases effective associativity.

5.3.2 SCC Structure

Structurally, SCC shares many common elements with previously proposed compressed caches [21] and the multi-page size skewed-associative TLB [94]. Figure 5-3 (a) shows one set of SCC tag array and its corresponding data set for a 4-way associative cache. Similar to a regular cache, SCC keeps the same number of tags as physical data entries in a cache set (e.g., 4 tags and 4 data entries per set in Figure 5-3). However, unlike a regular cache, which tracks exactly one single block per tag entry, SCC tag entries track a super-block containing 8 adjacent blocks. Figure 5-3 illustrates that each tag entry includes the super-block tag address and per-block coherency/valid states (e.g., eight states for 8-block super-blocks). Figure 5-2 also shows some examples of tag entries for block W in set #4 of way #3, blocks I—P in set #7 of way #1, and blocks R,U,V,X in set #3 of way #0. The data array is largely similar to a conventional cache data array, except it is organized at sub-blocks (e.g., 8 bytes).

$$W_1W_0 = A_{10}A_9 \wedge CF_1CF_0 \qquad (1)$$

Unlike a regular cache that can allocate a block in any cache way, SCC takes into account block compressibility. Equation (1) shows the way selection logic that SCC uses when allocating a cache block. It uses the block compression factor (CF_1CF_0) and two address bits ($A_{10}A_9$) to select the appropriate way group (W_1W_0). The block compression factor (CF_1CF_0) is zero if the block is not compressible, one if compressible to 32 bytes, two if compressible to 16 bytes, and three if compressible to 8 bytes. SCC maps neighboring blocks with similar compressibility to the same data entry. Thus, the way selection logic uses address bits $A_{10}A_9$, which are above the super-block offset. Note that since SCC uses address bits in way selection, even if all cache blocks are uncompressible (CF == 0), they will spread out among all cache ways.

$$\text{Set Index} = \begin{cases} h_0(\{A_{47}\text{—}A_{11}, A_8A_7A_6\}) & \text{if CF==0} \\ h_1(\{A_{47}\text{—}A_{11}, A_8A_7\}) & \text{if CF==1} \\ h_2(\{A_{47}\text{—}A_{11}, A_8\}) & \text{if CF==2} \\ h_3(A_{47}\text{—}A_{11}) & \text{if CF==3} \end{cases} \qquad (2)$$

SCC uses different set index functions to prevent conflicts between blocks in the same super-block. Just using bit selection, e.g., the consecutive bits beginning with A_{11}, would result in all blocks in the same super-block mapping to the same set in a way group, resulting in unnecessary conflicts. For example, if none of the blocks were compressible, then all eight uncompressed blocks would compete for the four entries in the selected way group (in Figure 5-2). To prevent this, SCC uses the index hash functions shown in (2), which draw address bits from the Block ID for the less compressible blocks. These functions map neighboring blocks to the same set only if they can share a data entry (based on their compression factor). SCC also uses different hash functions [95] for different ways in the same way group, to further reduce the possibility of conflicts.

Within a 64-byte data entry, a compressed blocks location depends only on its compression factor and address, eliminating the need for extra metadata. Equation 3 shows the function to compute the byte offset for a compressed block within a data entry.

$$\text{Byte Offset} = \begin{cases} \text{none} & \text{if CF==0} \\ A_6 \ll 5 & \text{if CF==1} \\ A_7A_6 \ll 4 & \text{if CF==2} \\ A_8A_7A_6 \ll 3 & \text{if CF==3} \end{cases} \qquad (3)$$

5.3.3 SCC Cache Operations

Figure 5-4 illustrates how SCC operates for the main cache operations. On a cache lookup, since the accessing block's compressibility is not known, SCC must check the block's corresponding positions in all cache ways. To determine which index hash function to use for each way, SCC uses (4), the inverse of (1).

$$CF_1CF_0 = A_{10}A_9 \wedge W_1W_0 \qquad (4)$$

For example, in Figure 5-2, when accessing block A, the tag entries in set #1 of way #3, set #5 of way #2, set #2 of way #1, and set #6 of way #0 (i.e., all hatched red tag entries) are checked for a possible match. A cache hit occurs if its encompassing super-block is present (i.e., a sparse super-block tag match), and the block state is valid. On a read hit, SCC uses the compression factor and appropriate address bits (using (3)) to determine which of the corresponding sub-blocks should be read from the data array.

On a write hit (e.g., a write-back to an inclusive last-level cache), the block's compressibility might change. If the block can still fit in the same place as before (i.e., its new size is less than or equal to the old one), SCC will update the block in place. Otherwise, SCC invalidates the current version of the block first by setting its corresponding state to invalid. Note that neighboring blocks that share the data entry are not affected. SCC then allocates a new entry as described below for a cache miss. Fortunately, this case does not arise very frequently; simulation results show that on average 97% of updated blocks fit in their previously allocated space.

SCC handles cache misses and write hits that do not fit in their previous space the same way. SCC first uses the block's (new) compression factor and address to search whether an existing sparse super-block of the right size has already been allocated for a neighboring block. For example, consider a write to block R in Figure 5-2 that changes the compression factor from 3 (8 bytes) to 1 (32 bytes). SCC would invalidate the old copy of R in set #7 of way #0 and write the new data in set #3 in way #2.

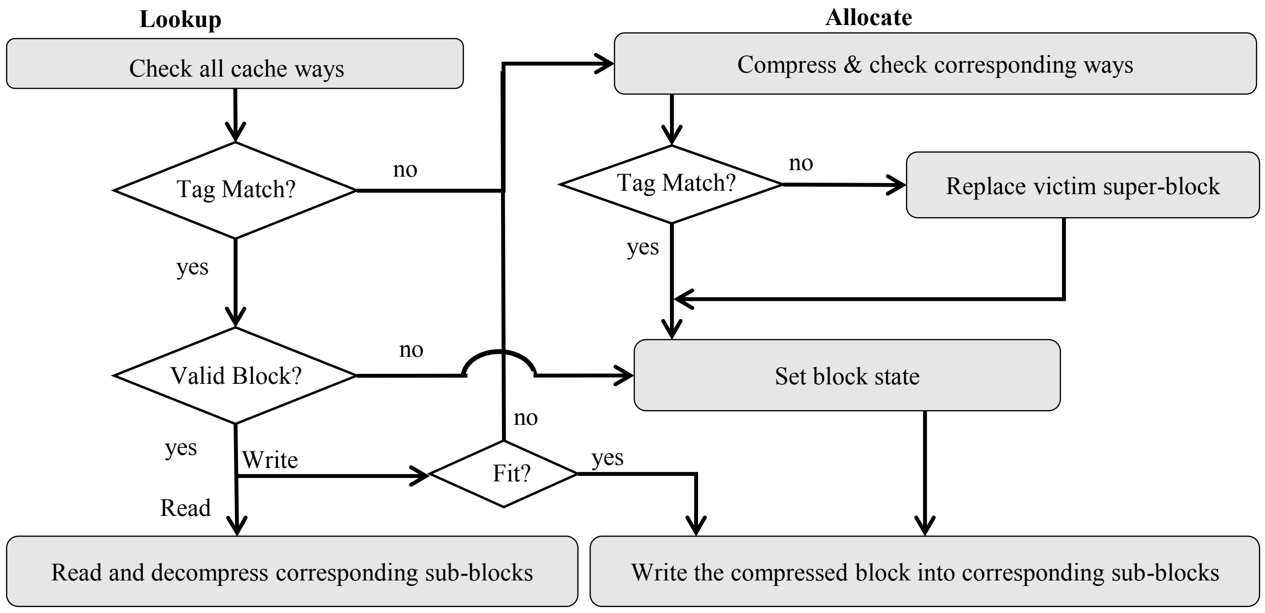

Lookup
Check all cache ways
Tag Match?
no
yes
Valid Block?
no
yes
Write
Read
Fit?
no
yes
Read and decompress corresponding sub-blocks
Allocate
Compress & check corresponding ways
Tag Match?
no
Replace victim super-block
yes
Set block state
Write the compressed block into corresponding sub-blocks

Figure 5-4: SCC Operations.

Detecting a sparse super-block hit is more complex than a normal tag match for two reasons. First, the size of the sparse super-block—and hence the number of tag bits that must be checked—depends upon the compression factor. For example, to detect that block R can be reallocated to the sparse super-block in set #3 of way #2, SCC must make sure that not only the super-block tag bits match, but that bits A8 and A7 also match, since the compression factor is 1 (32 bytes). Second, since SCC does not store bits A8A7A6 in the tag entry, it must infer them from the coherence states. For example, SCC can infer that both A8 and A7 are one in set #3 of way #2 by testing if either State7 or State6 are valid (in this example State7 is valid because block Q is valid).

If no matching sparse super-block tag with the right compression factor exists, SCC needs to select and evict a victim to make room. SCC selects the least-recently-used super-block tag within the way group (e.g., one of 4 ways in Figure 5-2). It then evicts all blocks that map to that tag's corresponding data entry. For example, if SCC needs to allocate a new block in set #0 of way #2, it would free that data entry by evicting blocks S and T (i.e., both cache lines in that data entry). Note that the rest of blocks from SB3 will stay in the cache in set #4 of way #3 (block W) and set #3 of way #0 (blocks R,U,V,X). For victim blocks, SCC can determine their compression factor based on the cache way and tag address using (4). After evicting the victim blocks, SCC updates the sparse super-block tag and inserts the new compressed block into the appropriate sub-blocks of the data entry.

SCC's replacement mechanism is much simpler than that needed by DCC. In DCC, allocating space for a block can trigger the eviction of several blocks, sometimes belonging to different super-blocks. In case of a super-block miss, all blocks associated with the victim super-block tag must be evicted, unlike SCC that evicts only blocks belonging to a particular data entry. In addition, in DCC, blocks belonging to other super-blocks may need to be evicted too. Thus, determining which block or super-block is best to replace in DCC is very complex.

SCC also never needs to evict a block on a super-block hit, while DCC may. SCC will allocate the missing block in its corresponding data entry, which is guaranteed to have enough space since the compression factor is used as part of the search criteria. In DCC, a super-block hit does not guarantee that there is any free space in the data array.

Table 5-1: Simulation Parameters.

Processors	8, 3.2 GHz, 4-wide issue, out-of-order
L1 Caches	32 KB 8-way split, 2 cycles
L2 Caches	256 KB 8-way, 10 cycles
L3 Cache	8 MB 16-way, 8 banks, 27 cycles
Memory	4GB, 16 Banks, 800 MHz DDR3.

5.4 Methodology

Our target machine is an 8-core multicore system (Table 5-1) with OOO cores, per-core private L1 and L2 caches, and one shared last level cache (L3) [74]. We implement SCC and other compressed caches at the L3. We evaluate SCC using full-system cycle-accurate GEMS simulator [81]. We use CACTI 6.5 [87] to model area and power at 32nm. We report total energy of cores, caches, on-chip network, and main memory.

We simulate different applications from SPEC OMP [92], PARSEC [91], commercial workloads [89], and SPEC CPU 2006. Table 5-2 shows the list of our applications. We run mixes of multi-programmed workloads from memory-bound and compute-bound SPEC CPU 2006 benchmarks. For example, for astar-bwaves, we run four copies of each benchmark. In Table 5-2, we show our applications in increasing LLC MPKI (Misses per Kilo executed Instructions) order for the Baseline configuration. We classify these workloads into: low memory intensive (L), medium memory intensive (M), and high memory intensive (H) if their LLC MPKI is lower than one, between one and five, and over five respectively. We run each workload for approximately 500M instructions with warmed up caches. To address workload

Table 5-2: Applications.

	Application	**LLC MPKI**
Low Mem Intensive	ammp	0.01
	blackscholes	0.13
	canneal	0.51
	freqmine	0.65
Medium Mem Intensive	bzip2 (mix1)	1.7
	equake	2.2
	oltp	2.3
	jbb	2.7
	wupwise	4.3
High Mem Intensive	gcc-omnetpp-mcf-bwaves-lbm-milc-cactus-bzip (mix7)	8.4
	libquantum-bzip2 (mix2)	9.3
	astar-bwaves (mix5)	9.3
	zeus	9.3
	gcc-166 (mix4)	10.1
	apache	10.6
	omnetpp-4-lbm-4(mix8)	11.2
	cactus-mcf-milc-bwaves (mix6)	13.4
	applu	25.9
	libquantum(mix3)	43.9

variability, we simulate each workload for a fixed number of work units (e.g., transactions) and report the average over multiple runs [90].

We study the following configurations at LLC:

- **Baseline** is a conventional 16-way 8MB LLC.
- **2X Baseline** is a conventional 32-way 16MB LLC.
- **FixedC** doubles the number of tags (i.e., 32 tags per set) compared to Baseline. Each cache block is compressed to half if compressible, otherwise stored as uncompressed.
- **VSC** doubles the number of tags compared to Baseline. A block is compressed and compacted into 0-4 contiguous 16-byte sub-blocks.
- **DCC_4_16** has same number of tags per set (i.e., 16 tags per set) as the Baseline, but each tracks up to 4 neighboring blocks (4-block super-blocks). In DCC, one tag tracks all

blocks belonging to a super-block. A block is compressed to 0-4 16-byte sub-blocks, compacted in order but not necessarily in contiguous space in a set. This is the main configuration we used in evaluating DCC in Chapter 4.

- **DCC_8_8** is similar to DCC_4_16, but it tracks up to 8 neighboring blocks (8-block super-blocks). A block is compressed to 0-8 8-byte sub-blocks.
- **SCC_8_8** has same number of tags per set (i.e., 16 tags per set) as the Baseline, but each tracks up to 8 neighboring blocks (8-block super-blocks). Unlike DCC, SCC might use multiple sparse super-block tags to track blocks of a super-block in case all cannot fit in one data entry. A block is compressed to 1-8 8-byte sub-blocks. A given block can be mapped to a group of four cache ways (out of 16 ways) based on block address and compressibility.
- **SCC_4_16** is similar to SCC_8_8, but it tracks 4-block super-blocks. A block is compressed to 1-4 16-byte sub-blocks. For a given address, we divide the cache into three way groups containing 4 ways, 4 ways, and 8 ways, respectively. We map a block to these groups if the block is uncompressed, compressed to 32-bytes, or compressed to 16-bytes, respectively.
- **Skewed Base** models a 4-way skewed associative cache with conventional tags (no super-blocks) and no compression.

5.5 Design Complexities

Compressed caches effectively increase cache capacity at the cost of more metadata. Table 5-3 shows the area breakdown of different compressed caches compared to Baseline. We assume

Table 5-3: Compressed Caches Area Overhead relative to Baseline.

	Tags	Coherence Metadata	Compression Metadata	Total LLC Overhead
FixedC	5.3%	0.6%	0.3%	6.2%
VSC	5.3%	0.6%	1.1%	7.0%
DCC 4_16	-0.1%	1.7%	5.2%	6.8%
DCC 8_8	-0.3%	3.8%	11.8%	15.3%
SCC 4_16	-0.2%	1.7%	0	1.5%
SCC 8_8	-0.4%	3.9%	0	3.5%

a 48-bit physical address space. These compressed caches differ in the way they provide needed tags to track compressed blocks, and their tag-data mapping. In Table 5-3, we separate their area overhead caused by more tags (including tag addresses and LRU information), extra metadata for coherence information, and extra metadata for compression (including any compression flag, compressed block size, etc.).

The earlier FixedC and VSC designs double the number of tags, which increases the LLC area by about 6%. FixedC requires no additional metadata for tag-data mapping, since it retains a one-to-one tag-data relationship. It only stores a 1-bit flag per block to represent if a block is compressed or not. VSC allows variable-size compressed blocks, requiring three bits of additional metadata per block to store its compressed size. VSC uses this modest additional metadata to determine the location of a compressed block.

DCC uses the same number of tags as Baseline, but each tag tracks a 4- or 8-block super-block. The tags use fewer bits for the matching address, thus compared to a regular cache tags are smaller. On the other hand, DCC needs additional coherence state for each block. DCC_4_16, with 4-block super-blocks, increases LLC area by 1.7% due to more coherence states. By doubling the super-block size, DCC_8_8 can track twice as many blocks but increases the additional area overhead to 3.8%. DCC decouples tag-data mapping, requiring extra metadata to hold the back pointers that identify a block's location. DCC keeps one back pointer entry (BPE) per sub-block in a set. In DCC_4_16, back pointer entries incur 5.2% area overhead. Smaller sub-block sizes can reduce internal fragmentation, and so improve cache utilization, but at the cost of more BPEs. DCC_8_8 uses 8-block sub-blocks, has 16*8 BPEs per set, resulting in 11.8% extra area overhead for the metadata.

SCC also tracks super-blocks, and thus has tag overhead lower than a conventional cache, but differs from DCC in two ways. First, SCC only needs (pseudo-)LRU state for the tags, while DCC maintains additional state for the decoupled sub-blocks. Second, SCC does not require extra metadata to track a block's location because of its direct tag-data mapping. SCC keeps only the tag address, LRU state and per-block coherence states. SCC_4_16 incurs 1.5% area overhead, more than a factor of 4 lower overhead than DCC_4_16. Similarly, SCC_8_8 incurs 3.5% area overhead, 78% less area overhead than DCC_8_8.

5.6 Evaluation

5.6.1 Cache Utilization

Figure 5-5 shows the effective capacity of the alternative cache designs normalized to Baseline for our workloads. We calculate the effective capacity of a cache by periodically counting the number of valid blocks. An ideal compressed cache would have a normalized effective capacity that is the same as the application's compression ratio. Practical compressed

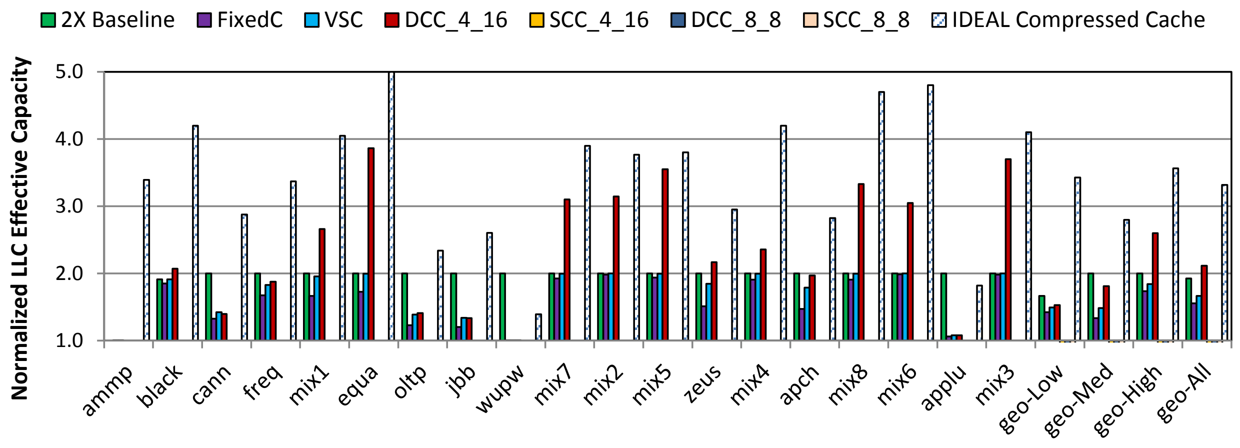


Figure 5-5: Normalized LLC effective capacity.

caches trade off effective capacity for lower overheads and lower complexity. In addition, some low memory intensive workloads, such as ammp, have small working sets, which fit in a small cache even though they have highly compressible data.

Figure 5-5 also shows that compressed caches can achieve much of the benefit of doubling the cache size, despite their low area overheads. 2X Baseline, which doubles the area used by the LLC, can hold on average 1.9 times more blocks (at most 2x and only 1.6x for the low memory intensive applications). FixedC and VSC provide, on average, 1.5x and 1.6x the normalized effective capacity, respectively. Like 2X Baseline, they can hold at most twice as many blocks since they have exactly twice as many (regular) tags.

SCC and DCC can further increase effective capacity because tracking super-blocks allow a maximum effective capacity equal to the super-block size (e.g., 4x and 8x). SCC_4_16 and SCC_8_8 provide, on average, normalized effective capacities of 1.7 and 1.8. SCC achieves the highest effective capacity for memory intensive workloads (on average ~2.3), outperforming 2X Baseline.

DCC achieves a greater normalized effective capacity than SCC because its decoupled tag-data mapping reduces internal fragmentation and eliminates the need to ever store more than one tag for the same super-block. In DCC, non-neighboring blocks can share adjacent sub-blocks, while in SCC only neighboring blocks can share a data entry and only if they are similarly compressible. In addition, DCC does not store zero blocks in the data array, while SCC must allocate a block with compression factor of 3 (i.e., 8 bytes). DCC_4_16 and DCC_8_8 achieve, on average, normalized effective capacities of 2.1 and 2.4, respectively. Of course, this comes at more than four times the area overhead and higher design complexity compared to SCC.

5.6.2 Cache Miss Rate

Figure 5-6 shows the LLC MPKI (Misses per Kilo executed Instructions) for different cache designs. Doubling cache size (2X Baseline) improves LLC MPKI by 15%, on average, but at significant area and power costs. Compressed caches, on the other hand, increase effective capacity and reduce cache miss rate with smaller overheads.

SCC improves LLC miss rate, achieving most of the benefits of 2x Baseline. On average,

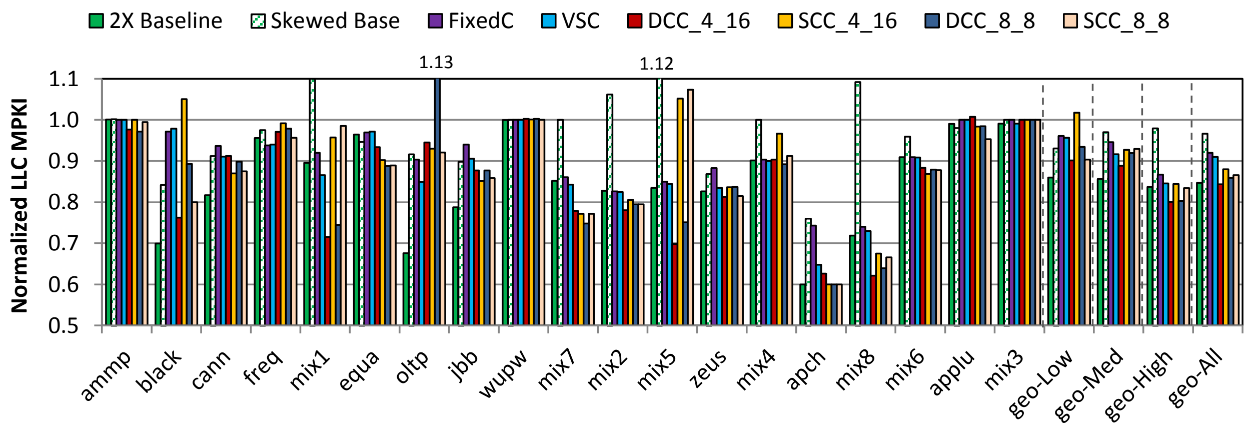


Figure 5-6: Normalized LLC MPKI.

SCC provides about 13% lower LLC MPKI than Baseline. It achieves the greatest improvements for memory intensive workloads (on average %16). SCC's improvements come from two sources: reduced capacity misses and reduced conflict misses. By increasing effective capacity using compression, SCC obviously tends to reduce capacity misses. But SCC also reduces conflict misses as a result of its skewed-associative tag mapping. SCC primarily uses skewing to map different size compressed blocks to one of four way groups, while preserving a direct, one-to-one tag-data mapping. SCC further uses skewing to reduce conflicts between blocks within a 4-way way group.

To show the impacts of skewing on miss rate, Skewed Base in Figure 5-6 models a 4-way skewed cache. On average, Skewed Base performs in the same range as the 16-way Baseline (about 4% lower MPKI). For some workloads, such as Apache and Zeus, skewing reduces conflict misses significantly by spreading out the accesses. In SCC, this results to even lower miss rate of these workloads due to compression. On the other hand, for few workloads (mix2, mix5, and mix8), skewing cannot compensate the negative impacts of lowering the associativity in Skewed Base. For those workloads, SCC shows lower miss rate improvements, and even 7% LLC miss rate increase for mix5.

Compared to DCC, SCC provides similar improvements with a factor of 4 lower area overheads. By tracking super-blocks, both DCC and SCC perform better than FixedC and VSC. Although DCC_8_8 achieves higher effective capacity than DCC_4_16 and SCC, it performs on average similar to DCC_4_16. For oltp, DCC_8_8 even increases LLC MPKI by about 13% as mapping 8 neighbors to the same set can increase conflict misses.

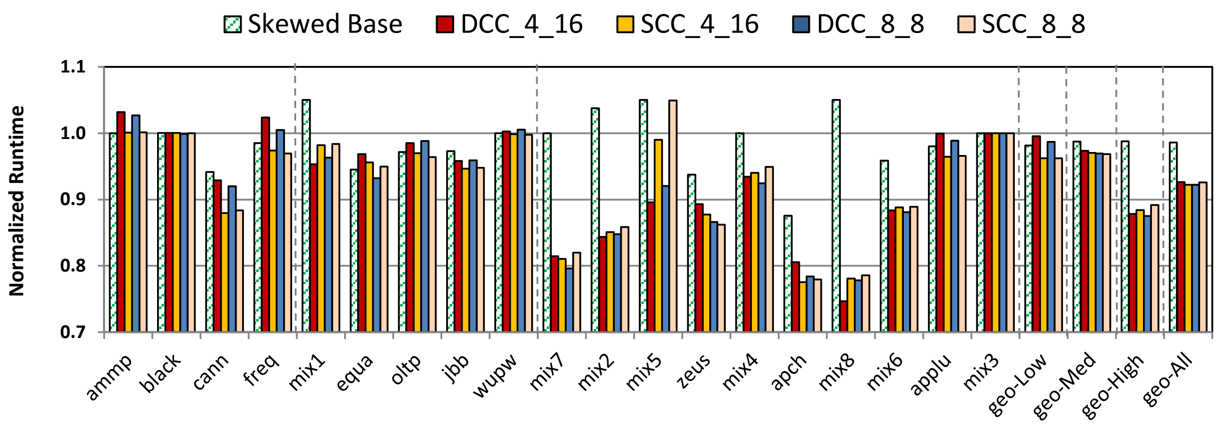


Figure 5-7: Normalized performance of different SCC and DCC configurations.

For completeness we also analyzed a design that combines skewed associativity with DCC. Due to the decoupled tag-data mapping of DCC, adding skewing to DCC results in a more complicated design and replacement policy that we do not consider practical to implement. DCC stores sub-blocks of a block anywhere in a cache set. When applying skewing, this means the sub-blocks of a block can be indexed to different sets. Thus, a BPE needs to store set index as well resulting to high area overheads (~15% area overhead for a configuration similar to DCC_4_16). In addition, skewing can significantly complicate replacement policy in DCC. A block allocation can trigger multiple block evictions as a block can be allocated across different sets. Our results (not shown here) show that adding skewing to DCC improves it marginally.

5.6.3 System Performance and Energy

Figure 5-8, Figure 5-7, and Figure 5-9 show system performance and energy of different cache designs. Our reported system energy includes both leakage and dynamic energy of cores, caches, on-chip network, and off-chip main memory.

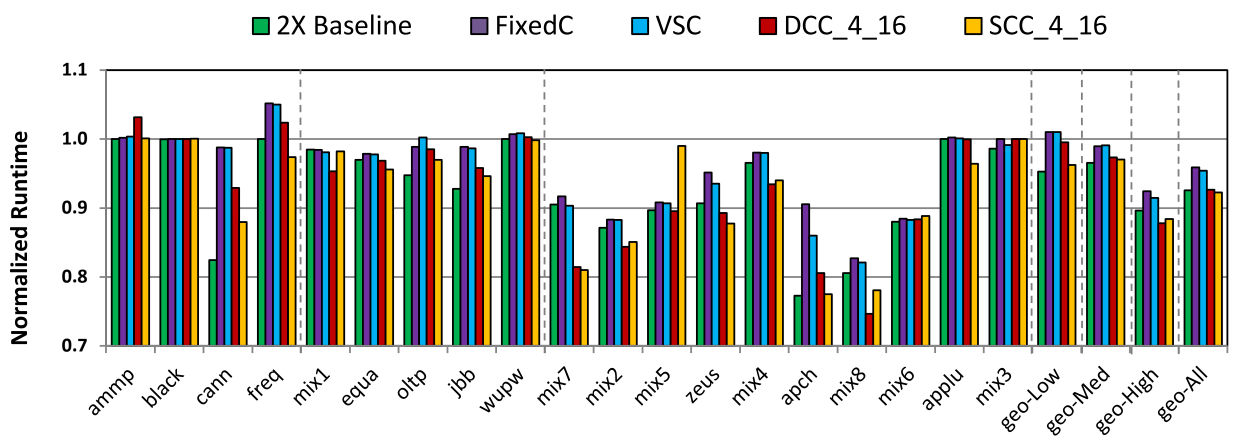


Figure 5-8: Normalized performance of different cache designs.

By increasing cache efficiency and reducing accesses to the main memory, compressed caches can improve system performance and energy, achieving the benefits of larger caches. SCC improves system performance and energy by up to 22% and 20% respectively, and on average 8% and 6% respectively. SCC achieves comparable benefits as previous work DCC with a factor of four lower area overheads.

SCC benefits differ per application. It provides the highest improvements for memory intensive workloads (on average 11% and maximum of 23% faster runtime for apache). On the other hand, it has the smallest gains for low memory intensive workloads (on average 4%). For cache insensitive workloads, such as ammp, blackscholes and libquantum (mix3), SCC does not impact their performance and energy.

Figure 5-7 also shows the performance of Skewed Base, which basically separates skewing impacts on SCC performance. In Skewed Base, a block can be mapped to a group of 4 ways based on its address. Each of those ways is hashed differently. For some workloads, such as apache, Skewed Base improves their performance and energy by spreading out accesses. For

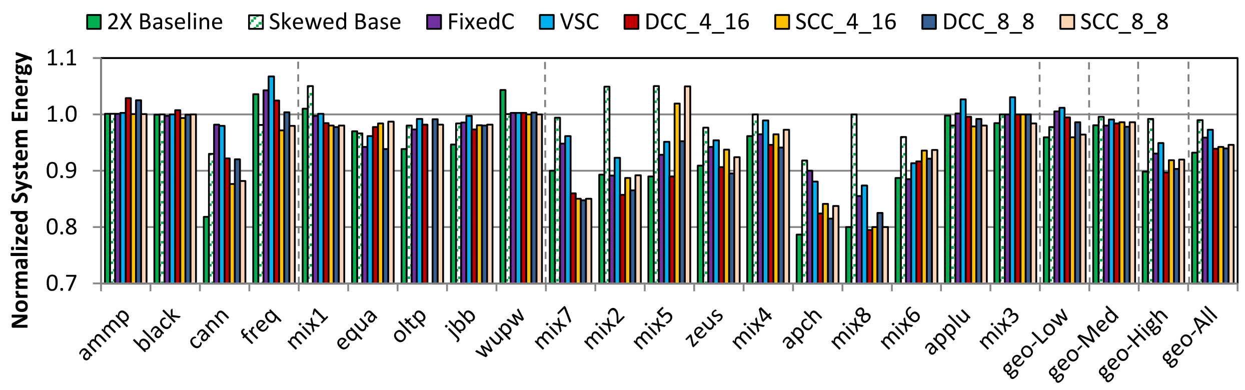


Figure 5-9: Normalized system energy.

these workloads, although SCC has smaller effective capacity than previous work DCC, SCC overall miss rate improvement is comparable to DCC and 2X Baseline. On the other hand, for few workloads (mix1, mix5, and mix8), skewing cannot compensate the effect of lower effective associativity in Skewed Base. For these workloads, SCC achieves lower performance and energy improvements. For mix5 (astar-bwave), SCC has about 5% increase in runtime and energy.

5.7 Conclusions

In this chapter, we propose Skewed Compressed Cache, a new low-overhead hardware compressed cache. SCC compacts compressed blocks in the last-level cache in such a way that it can find them quickly, and minimize the storage overhead and design complexity. To do so, SCC uses sparse super-block tags to track more compressed blocks, compact blocks into a variable number of sub-blocks to reduce internal fragmentation, but retain a direct tag-data mapping to find blocks quickly and eliminate the extra metadata.

SCC proposes a direct tag-data mapping by exploiting compression locality. It compresses blocks to variable sizes, and at the same time eliminates the need for extra metadata (e.g., back pointers). It dynamically packs neighboring blocks with similar compressibility in the same space tracking them with one sparse super-block tag. SCC further uses skewing to spread out blocks for lower conflicts. Like previous work DCC, SCC achieves performance comparable to that of a conventional cache with twice the capacity and associativity. But SCC does this with less area overhead (1.5% vs. 6.8%).

Chapter 6
On Compression Effectiveness in the Memory Hierarchy

6.1 Overview

There have been several recent proposals on exploiting compression in the memory hierarchy. Software-based approaches focus on compressing pages in main memory. For example, Apple OS X compresses the least recently used pages to free up memory for active processes [70]. Hardware-based techniques span the memory hierarchy, using simpler algorithms to keep design complexity and overheads low. For example, several recent proposals, including DCC and SCC, seek to double (or more) the effective capacity of the last level cache in a multicore processor [21][22][23].

Since most of these proposals are on non-existing hardware, architects evaluate those using detailed simulators. Due to the complexity of existing simulators, simulations are slow. For example, the gem5 simulator [96] with an OOO processor configuration runs a benchmark approximately 10,000 times slower compared to running it directly on a real machine. Simulating a one-hour desktop application (e.g., watching a movie on YouTube), if at all possible, would take over one year of simulation time. Consequently, most researchers end up evaluating their proposals with small benchmarks for short runtimes. While previous studies, such as SimPoint [97], have sought to identify representative points within a workload, they have

Table 6-1: Myths on compression.

M1	Most workloads are compressible: 50% of workloads have compression ratio ≥ 2.
M2	Cache data are more compressible than memory data.
M3	Floating point data is mostly uncompressible.
M4	Instructions are mostly uncompressible.
M5	Compression locality: neighboring blocks have similar compressibility.
M6	Bigger blocks are more compressible.
M7	More complex compression algorithms improve compressibility.
M8	Sub-blocking eliminates most internal fragmentation.

focused on microarchitecture or workload behavior, not memory compression. Therefore, many of studies on compression rely on workload properties that have only been demonstrated to hold for small benchmarks and short runtimes. Thus, we cannot clearly infer how they would work for large, real-world workloads running for long periods of time.

In this chapter, I take a holistic approach toward compression focusing on compression both in main memory and in caches. I explore 8 myths (i.e., commonly made assertions and conventional wisdoms) spanning a wide range of different design options consisting of compression algorithms, granularity of compression, compression locality, etc. These myths, listed in Table 6-1, arise from previous proposals mostly through simulation of standard benchmarks. I evaluate the strength of each myth under several design parameters, resulting in 12 overall findings. For each myth, I rate them as "Busted!" if I cannot replicate them, "Plausible" if I can only replicate them for few applications or with certain parameters, and "Confirmed!"[2] if the myth holds.

[2] This terminology is inspired by MythBusters: http://en.wikipedia.org/wiki/MythBusters

Unlike most previous work, rather than focusing on small CPU benchmarks, I quantitatively evaluate compressibility of workloads for up to 24 hours. In addition to standard CPU benchmarks (e.g., SPEC CPU2006), I study the production servers of the Computer Sciences Department of UW-Madison (e.g., web, file and database servers), memory-intensive desktop applications (e.g., Google Chrome), mobile benchmarks, and emerging big data applications. To evaluate compressibility on real machines, I have developed a toolset that enables measuring compressibility of cache and memory contents for any running application. Through my extensive analysis, I show that two of the eight myths are "Busted!," two are "Plausible," and the rest are "Confirmed!".

This chapter is organized as follows. I discuss the eight popular myths on compression in Section 2. I explain our infrastructure, tools, and workloads in Section 3. I then test and analyze the myths in Section 4. Finally, Section 5 concludes the chapter.

6.2 Myths about Compression

Several proposals have been exploiting compression to improve cache and memory utilization. Many of these studies rely on workload properties that have only been demonstrated to hold for small, CPU-centric benchmarks and very short runtimes using simulators. In this work, we treat these workload properties as myths that must be tested. Table 6-1 lists these myths, and we describe them in detail below:

Myth 1: Several proposals show that many workloads are compressible [20][21][23]. Many applications must benefit from compression to justify hardware-based compression mechanisms. To quantify this myth (**M1**) at least half the workloads are compressible to half (i.e. compression ratio ≥ 2). This is a statement of overall compression effectiveness. If this turns out to be false, it would mean that compression might not be of general use.

Myth 2: Compression effectiveness could also change depending on where in the memory hierarchy it is applied. Most previous work only focuses on one level of the memory hierarchy, and ignores the possible drawbacks or benefits at other levels. Mahapatra et al. [98][99] states that (**M2**) caches are more compressible than memory. They show that compressibility is lower at higher levels of the memory hierarchy, achieving the best compressibility at L1 caches. We evaluate the credentials of this myth under different design options. This would give designers insights on where they can get the highest benefits from compression: caches or main memory?

Myths 3 and 4: The conventional belief is that compressibility depends on quality of data being compressed. For a given compression algorithm, data type is usually an indicator of compressibility. (**M3**) Floating-point data and (**M4**) instructions are mostly uncompressible using general-purpose compression mechanisms [42]. We use different compression algorithms to test these myths.

Myth 5: Although compressibility changes per memory region or page, (**M5**) neighboring blocks have similar compressibility [61]. We do extensive analysis to find out if this myth holds, and if so, under what circumstances.

Myth 6: In addition to quality of data, quantity or granularity of data matters too. Conventional belief is that (**M6**) compressibility improves when compressing larger chunks of memory [98][99][21]. Usually a lot of effort is needed to compress larger granularity of data (e.g., compressing a 4096B page versus compressing a 64B block). Thus, we examine whether these efforts and overheads worth the possible higher compressibility.

Myth 7: Compression algorithm is a key player in compressed caches and memory. Myth **M7** arises from conventional wisdoms that more complex algorithms improve compressibility [19][20][21]. Using our framework, we analyze if this myth holds, and if so to what extent.

Myth 8: Finally, for a given compression algorithm, compaction mechanism (i.e., how to pack compressed blocks) plays an important role to get the benefits of compression [20][21][23]. As storing blocks at byte granularity is not practical, several previous works use different sub-blocking mechanisms. They store compressed blocks as multiple small sub-blocks to reduce internal fragmentation. In **M8**, we check whether sub-blocking could reduce internal fragmentation especially for real applications.

6.3 Infrastructure

In this section, we describe the infrastructure, tools, and workloads that we use to test these myths. We have two tools, one to study compressibility of data in main memory and one to study compressibility of data placed in the caches. Both our tools measure compressibility in real systems for any running application. We use representative real applications and benchmarks, and use rigorous methodology for measurements.

6.3.1 Platforms

Servers: In this work, we evaluate compressibility of three servers in production use in the Computer Sciences Department of UW-Madison: a webserver, a fileserver, and a Postgres database server. All these servers run RedHat Enterprise Linux 6.5. The Fileserver and the Postgres server run on a machine with 2 Intel Xeon cores, while the webserver run on a 4-core Intel Xeon machine.

Desktop machine: In addition to servers, we evaluate several real desktop applications and benchmarks. We run those on a desktop machine with Ubuntu 13.10. The machine has 4 Intel Corei5-2500K cores.

6.3.2 Tools

In this section, we present our toolsets to study compressibility in main memory and caches on real machines. Unlike simulators, our tools are fast. They enable us to measure compressibility of actively used production servers for long period of time (e.g., 24 hours). Such study would take over a year on a simulator like gem5. In addition, using these tools, we can study any running application, eliminating hassles of benchmarking.

6.3.2.1 Memory Compression Tool

We have developed a tool to study the compressibility of blocks in main memory for a running application. The basic idea is to take snapshots from the physical memory of a running application periodically. To do so, our tool uses ptrace to connect to the process(es) of a running application. Through the ptrace interface, our tool reads the pages of each process present in the physical memory and measures their compressibility.

Our tool takes the following steps to calculate memory compressibility of an application. A running application on Linux relates to one parent process and maybe multiple spawned processes. In case there is more than one process (e.g., for servers and multi-threaded applications), we repeat the next steps for each process. Our tool also handles synonyms. It does not re-count compressibility of a physical page in case more than one virtual page maps to it. The page size is 4096B, and the block size is 64B (i.e., typical cache block size), unless otherwise stated.

Step 1: For a running process with a given process id (i.e., pid), our tool finds its virtual memory regions by accessing "/proc/pid/maps”. For each virtual memory region, this file includes the start and the end addresses along with descriptions of the region (e.g., heap or stack).

Step 2: For each page in a given virtual memory region (found in step 1), we then access "/proc/pid/pagemap" to find whether the page is present in the physical memory, and if so, to get its physical address.

Step 3: For the physical pages found in step 2, we then read their contents through ptrace interface, and calculate their compressibility. We repeat step 2 and step 3 for all pages in each region.

For an application, we repeat this procedure (step1-3) periodically. For short running benchmarks, we take samples every few seconds. For long running servers, we take a snapshot every half an hour. For real desktop applications, we repeat the experiment every few minutes.

6.3.2.2 Cache Compression Tool

Unlike memory, there is no direct way to scan cache contents in a real machine for a specific running application. To find the compressibility of cached data in a real system, we build on BadgerTrap [100], a tool that enables instrumentation of x86-64 TLB misses. BadgerTrap allows tracing data TLB misses in a Linux machine for a running application. It does so by converting hardware-assisted page walks to page faults handled by a special software-assisted TLB handler. Our basic approach is to use this tool to get a random sample of data memory blocks accessed by CPUs for a running application.

To analyze the compression ratio of data blocks randomly accessed by CPUs, we periodically flush TLBs in the Linux timer interrupt handler. We then analyze the access that causes the first TLB miss. Although data blocks could bypass the caches, modern processors store most accessed blocks in on-chip caches. Thus, measuring compressibility of randomly accessed data blocks could represent the compressibility of data blocks randomly accessed at L1 data caches.

Using BadgerTrap, we can get samples of accesses to data blocks but not instruction blocks. As instructions are read-only, their compressibility in main memory is the same as their compressibility in caches. Also, as this tool involves modifying Linux kernel, we use it on our desktop machine to measure cache compressibility of real desktop applications and benchmarks. It is not feasible for us to use it on our production servers.

6.3.3 Applications

Since compression is a candidate in many platforms including servers, desktops, and mobiles, we consider a suite of workloads that span these. Table 6-2 summarizes our applications, and we describe them in detail below:

Servers: We analyze three servers providing production service: a webserver, a fileserver, and a Postgres database server. These servers are running all the time to service our department. For these servers, we analyze them for 24 hours. These servers usually have multiple processes running simultaneously.

Desktop Applications: We use five representative desktop applications running on our desktop machine for about an hour. We use Google Chrome while streaming a one-hour long movie on YouTube, Firefox while browsing a Wikipedia page with text and pictures (http://en.wikipedia.org/wiki/United_States), gedit text editor while editing a large text file, Open Office Writer while editing a version of this chapter, and Open Office Calc while editing an excel file.

Gaming: We run three games supported on Ubuntu: Battle for Wesnoth (Wesnoth), Extreme Tux Racer (Tracer), and Pingus. We play each game for about an hour on the desktop platform explained in Section 3.1.

Desktop Benchmarks: We use SPEC CPU2006 suite with reference input sets. We use benchmarks from both floating-point (SPEC-CFP) and integer (SPEC-CINT) categories.

Table 6-2: Applications summary.

Domain	Application
Servers (24 hours)	An AFS Fileserver.
	A Webserver. It is in charge of "www.cs.wisc.edu" webpage.
	A Postgres database server for Linux backup metadata.
Desktop Applications (1 hour)	youtube: Google Chrome while streaming a video on Youtube.
	wiki: FireFox while browsing a Wikipedia page.
	gedit while editing a large text file.
	openWrt: Open Office Write while editing a version of this chapter.
	openCalc: Open Office Calc while editing an excel file.
Gaming (1 hour)	Extreme tux racer (Tracer)
	Pingus
	Battle for Wesnoth (Wesnoth)
Desktop Benchmarks (to completion)	SPEC CINT: astar, bzip2, gcc, gobmk, h264, hmmer, libquantum, mcf, omnetpp, perlbench, sjeng.
	SPEC CFP: lbm, milc, namd, povray, soplex, sphinx.
Mobile Client (to completion)	Coremark [103] for ten million iterations.
	BBench [105].
Big Data (to completion)	Graph500 [101].
	Memcached [102].
	Graph-analytics [102].

Mobile Client: We use CoreMark [103] that is a widely used benchmark for evaluating mobile systems. We also use BBench [105], "a web-page rendering benchmark comprising 11 of the most popular sites on the internet today".

Big Data: We use graph-analytics and memcached from Cloudsuite [102], and Graph500 [101] as a representative of emerging big data applications. Memcached (or Data Serving) simulate the behavior of a Twitter caching server. Graph analytics runs a machine learning and data mining software. We run them with default parameters [104]. Graph500 generates a large graph, compresses it into sparse structures, and then does parallel breadth-first search. We use it with scaling factor of 24 (8GB memory footprint).

6.4 Compression Algorithms

In this work, we study four representative compression algorithms. The first three (C-PACK+Z [18], FPC+Z [20], and BDI [19]) have practical hardware implementations, and are suitable for hardware-based cache/memory compression. We also study a more ideal case using the gzip UNIX utility. For all these algorithms, we compress 64-byte blocks unless otherwise mentioned. We use C-PACK+Z in most experiments as a representative of hardware-based algorithms, as it is shown to have a good compressibility with low overheads compared to other algorithms [21].

C-PACK+Z: Cache Packer (C-PACK) [18] is a lossless compression algorithm that is designed specifically for hardware-based cache compression. C-Pack compresses a data-block at a 4-byte word granularity. It detects and compresses frequently appearing words (such as sign-extended words or zero words) to fewer bits. In addition, it also uses a small dictionary to compress other frequently appearing patterns. The dictionary has 16 entries, each storing a 4-byte word. The dictionary is built and updated per data block. C-PACK checks whether each word of the input block would match a dictionary entry (even partially). If so, C-PACK then stores the index to that entry in the output compressed code. Otherwise, C-PACK inserts the word in the dictionary. C-PACK takes 16 cycles to compress a 64-byte data block, and 9 cycles to decompress at 3.2GHz [21]. In this work, we use a modified version of C-PACK (C-PACK+Z) that also detects zero blocks [21].

FPC+Z: Frequent Pattern Compression (FPC) is a significance-based compression algorithm [20]. It exploits the fact that many values are small (e.g., small integers) and do not require the full space allocated for them. FPC compresses data blocks on a word-by-word basis by storing common word patterns (such as sign-extended words or repeated bytes) in a compressed format accompanied with an appropriate prefix. Compared to dictionary-based approaches, FPC has lower decompression latency. FPC decompresses a 64-byte line in five cycles, assuming 12 FO4 gate delays per cycle. We also augment FPC to detect zero blocks (FPC+Z).

BDI: Base-Delta-Immediate (BDI) compression algorithm [19] is another low-overhead algorithm optimized for cache/memory compression. It is based on the observation that in a cache/memory block, many words have small differences in their values. BDI encodes a block as one or more base-values and an array of differences from the base-values or simply zero. In this work, we use a version of BDI that is optimized for capacity (i.e., high compression ratio). It uses two base values, and takes 2-3 cycles to decompress a block.

gzip: To estimate the potential of compression with a complex algorithm, we use gzip. gzip is based on the DEFLATE algorithm, which is a combination of LZ77 and Huffman coding. We run gzip with its highest compressibility level (i.e., gzip -9). This algorithm is too complex for hardware-based compressed caches or memory. However, gzip can give us an approximate bound on compressibility of our applications when compressing large chunks of memory (e.g., a page). However, gzip is not a real bound for all data types including floating point as it is not optimized for those.

6.5 MythBusters: Testing Myths on Compression

In this section, we evaluate the strength of the myths on compression. Our basic strategy is to address each myth in the context of our applications. In several cases we present additional analysis in which we vary the base configuration (presented in Section 3) to determine separate findings for the myth in question.

6.5.1 Myth 1: Many Workloads Are Highly Compressible

Several studies have shown that many workloads (mostly benchmarks) are highly compressible using basic general-purpose algorithms [19][21]. We quantify this myth as at least 50% of workloads have compression ratio ≥ 2. In this section, we evaluate this myth for our applications with a hardware-based data-independent compression algorithm.

Figure 6-1 shows the compression ratio of our applications in main memory. We evaluate compression ratio (i.e., original block size / compressed block size) of our applications using C-PACK+Z algorithm. As discussed in section 3.2, we use our memory compression tool and periodically calculate the compression ratio of 64-byte blocks of a running application present in main memory. We report the average compression ratio over all the snapshots taken throughout the application runtime.

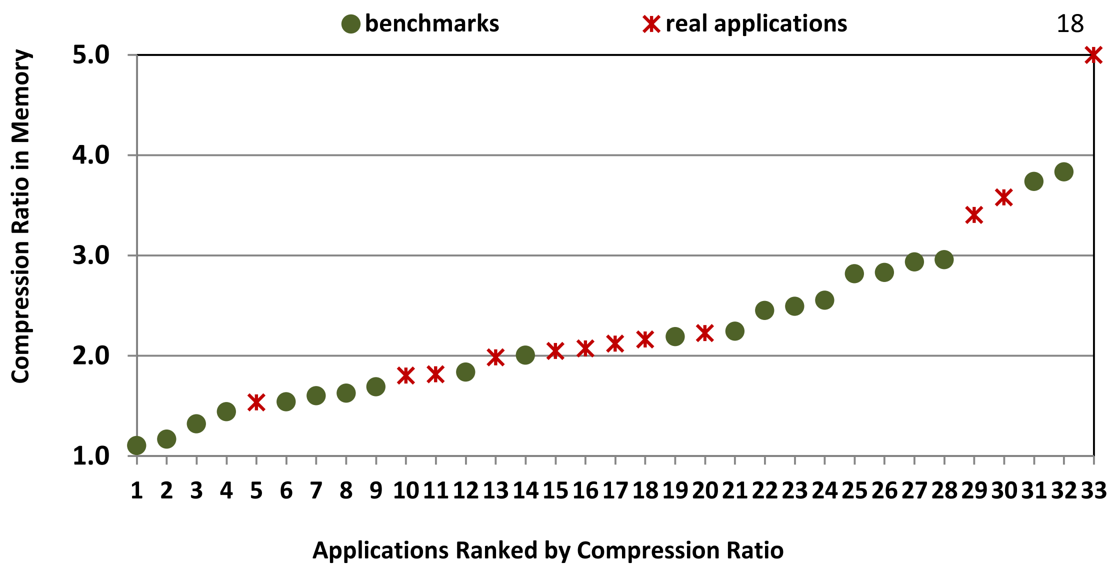


Figure 6-1: Compression ratio of our applications in memory with C-PACK+Z.

On average we observe the compression ratio of 2.3 across all applications. Our real applications and servers have the average compression ratio of 2.6 with the highest compression ratio of 18 for the file server (due to a large number of zero blocks) and the lowest compression ratio of 1.5 for gedit text editor. The average compression ratio of our benchmarks is 2.1. Among our benchmarks, milc has the lowest compression ratio (1.1), and bzip2 has the highest compression ratio (3.8).

To better understand these results we show the distribution of compressed block sizes in Figure 6-2. We show the cumulative distribution for all applications in skinny gray lines, while we highlight some representative applications using wider colored lines. Overall, on average 18% of blocks are zero, 16% of blocks are uncompressible (64B), and the rest of the blocks are compressible to 1B to 63B. In some real applications, like Tracer and Fileserver, and benchmarks, like bzip2, zero blocks and highly compressed blocks are dominant in memory resulting in high compression ratios. On the other hand, for low compressible applications, such

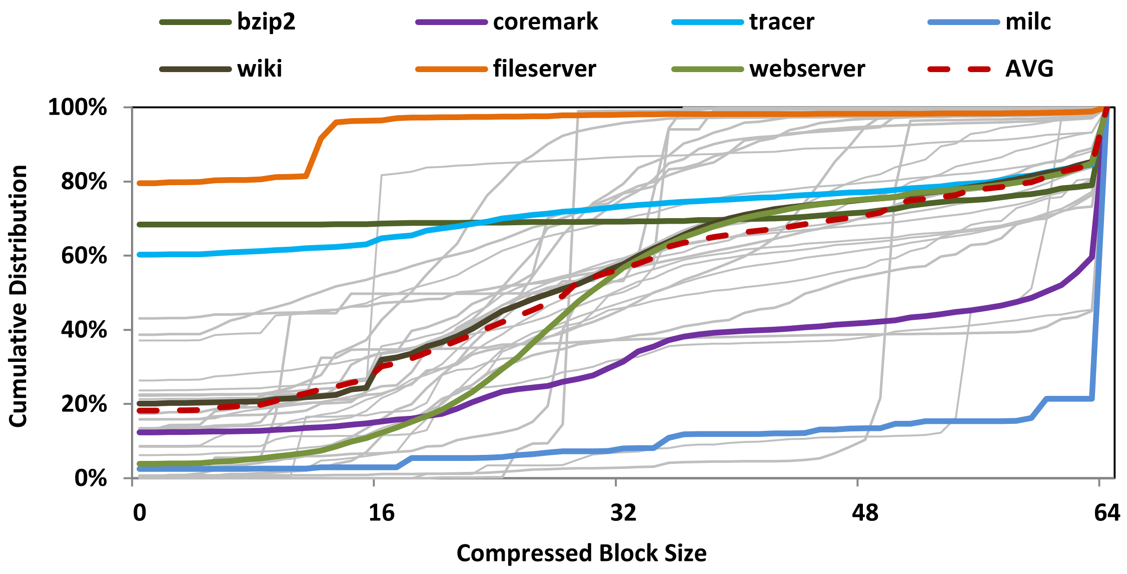


Figure 6-2: Cumulative distribution of compressed block sizes in main memory.

as coremark and milc, many blocks are uncompressible or poorly compressible (i.e., they have large compressed block sizes). The rest of applications, such as our webserver and wiki (browsing a wikipedia page on FireFox), have more uniform distributions.

Although our results suggest that many applications can benefit from general-purpose hardware compression, we have to be aware of the worst cases too. The bad news is that 12 of our applications have compressibility < 2 and even six have compression ratio ≤ 1.5. These applications, including coremark, SPEC CFP benchmarks, and gedit, might still benefit from special-purpose compression algorithms, such as those optimized for floating-point data [15][16][106].

Myth 1: [Confirmed!] *64% of our applications have compression ratio ≥ 2. This holds for both the benchmarks and real applications.*

6.5.2 Myth 2: Cache Data Are More Compressible than Memory Data

Several proposals have exploited compression at different levels of the memory hierarchy. Most of them do not differentiate the impacts of compression in main memory versus caches. Mahapatra et al. [98][99] show that data in L1 caches are more compressible than data in main memory using the same compression algorithms for their studied applications.

To test this myth, we use our memory compression and cache compression tools with the C-PACK+Z algorithm. We measure the average compression ratio of 64B blocks in main memory and compare it with the average compressibility of 64B data blocks accessed by CPUs. As we discussed in Section 3.2, the compressibility of data blocks randomly accessed by CPUs is an indicator of the compressibility of data blocks randomly accessed at L1 data caches. Since our

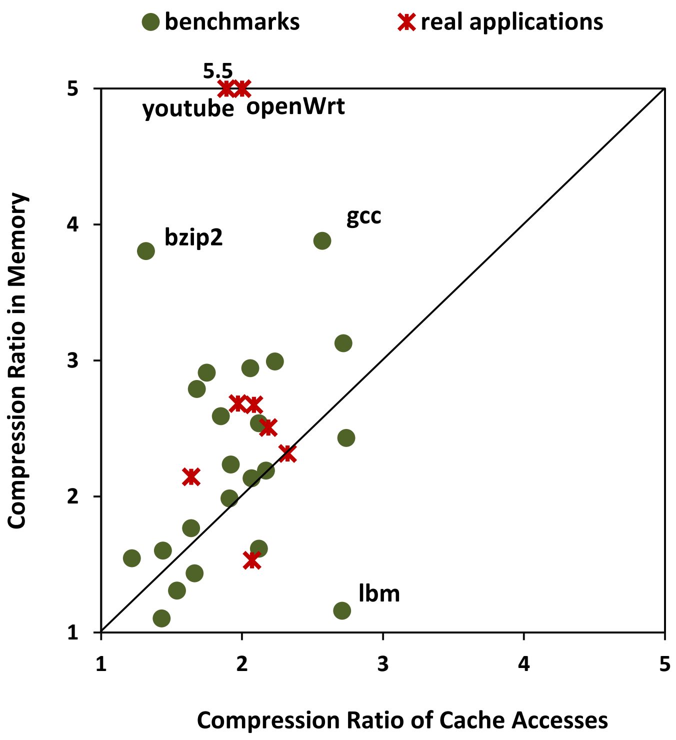


Figure 6-3: Data block compression: cache vs.

cache compression tool could only analyze data blocks (not instructions), we evaluate compressibility of only data blocks in main memory (i.e., blocks in stack and heap regions). For our benchmarks and real applications except servers, Figure 6-3 shows the average compression ratio of data blocks accessed by CPUs (on the X axis) versus the compression ratio of data blocks in main memory (on the Y axis).

Unlike Mahapatra et al. [98][99], we found that for all but one application (lbm) cache data are not more compressible than memory data. Majority of our real applications and benchmarks have similar compressibility in main memory and L1 data caches. Out of 30 applications, for 18 applications, the compression ratio of memory data blocks is similar to the compression ratio of L1 data blocks (i.e., less than 0.5 different). For the rest of applications (except lbm), compression ratio is considerably higher in memory than L1 caches. For example, bzip2 has the compression ratio of 3.8 in memory, while the compression ratio of its cached blocks is 1.3. Similarly, youtube has the compression ratios of 5.5 in memory and 1.9 in L1 caches.

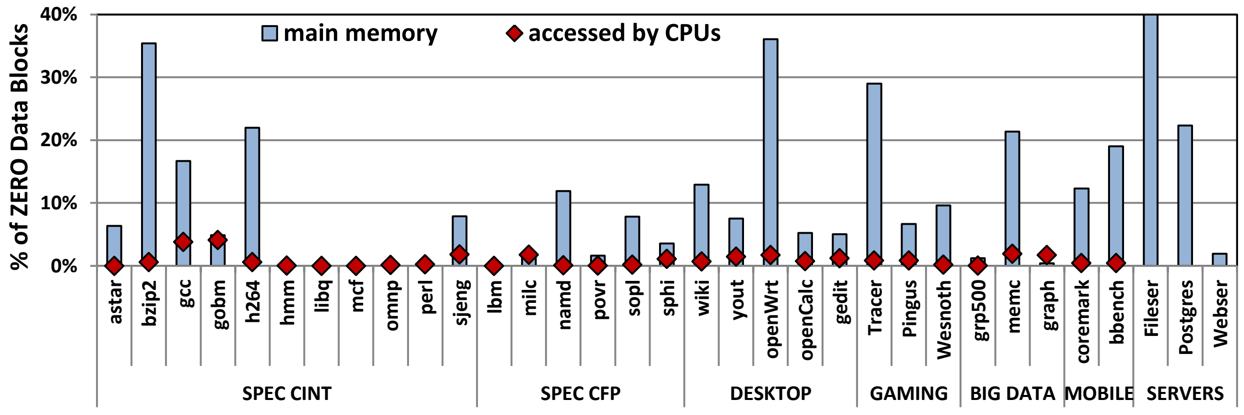


Figure 6-4: Percent of zero data blocks: memory vs. caches.

In general, for many applications the memory content is more compressible than cache content mainly due to larger percent of zero blocks in memory. Figure 6-4 shows the percent of data blocks that are zero in memory versus those accessed by CPUs. For some of our real applications, a significant number of data blocks are zero in memory. For example, for openWrt, Fileserver, Postgres server, and Tracer, 36%, 40%, 22% and 29% of blocks in stack and heap regions are zero respectively. This results in high compression ratios for these applications in memory. Similarly for some of our benchmarks, such as bzip2, gcc, and h264, the number of zero blocks is significant in memory as oppose to the caches.

Most zero blocks in memory are due to zero padding at the end of pages and zero pages. Thus, most of these zero blocks are never read by the CPUs or placed in caches, resulting in a

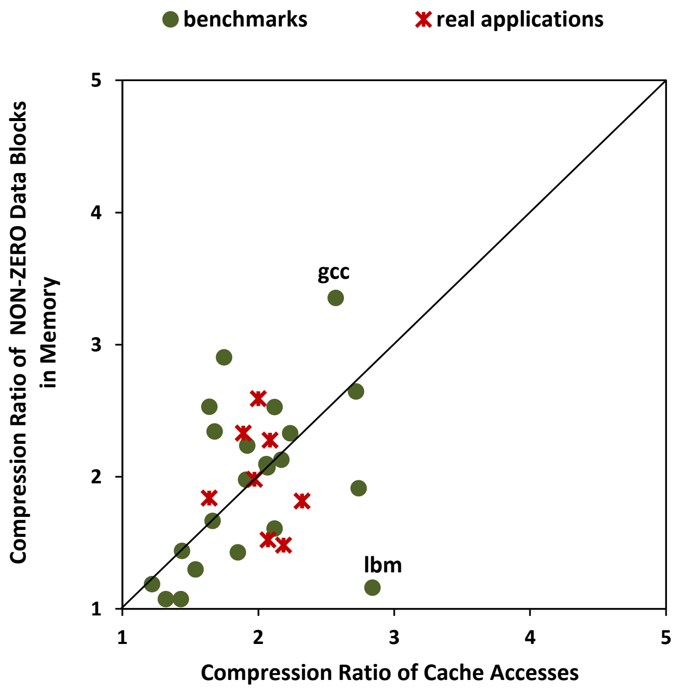


Figure 6-5 NON-ZERO block compression: cache vs. memory.

low number of zero blocks (less than 5%) and lower compression ratio at caches. To check this, in Figure 6-5, we compare the compression ratio of non-zero data blocks in main memory against the compression ratio of all blocks (including zeros) in caches. When we exclude zero data blocks, we observe more similar compression ratio in main memory and L1 caches.

Here, we compare compressibility of memory data versus cache data using the same configurations (i.e., C-PACK+Z algorithm at 64B block granularity). However, as we will discuss later, we can use even more complex compression techniques to compress larger chunks of data in main memory to further improve its compressibility.

Myth 2: [Busted!] *For almost all of our applications, the compression ratio in memory is either similar or better than the compression ratio in the L1 data caches.*

Finding 2.1: *Higher compressibility in main memory is mainly due to the abundance of zero blocks in memory.*

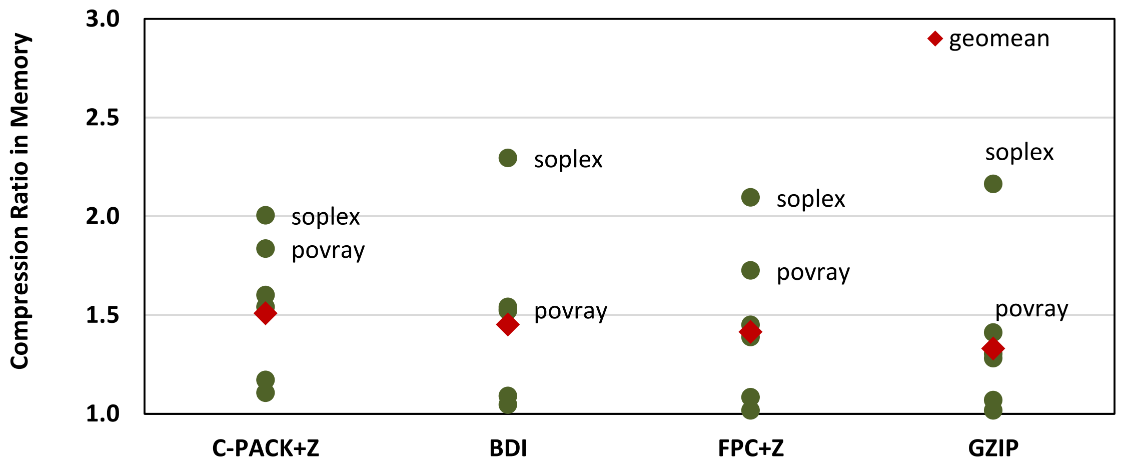


Figure 6-6: Compression ratio of SPEC CFP benchmark.

6.5.3 Myth 3: Floating-Point Data Is Mostly Uncompressible

Although there are several compression algorithms proposed for floating point applications, floating point data is considered mostly uncompressible using general-purpose compression algorithms [42]. In this section, we analyze compressibility of the selected benchmarks from SPEC CFP (listed in Table 6-2).

Figure 6-6 shows the compression ratio of SPEC CFP benchmarks in memory using different algorithms. For all these algorithms, on average the compression ratio is less than 1.5 with the highest compression ratio for soplex and povray. To better understand these results, Figure 6-7 shows the percent of integer blocks in data regions (i.e., stack and heap) in memory on the X axis, and overall memory compression ratio using C-PACK+Z on the Y axis. Similar to Kant et al. [42], we classify blocks as integer if the 8MSBs are 0x00 or 0xFF in each 32-bit word. In this way, we detect small integer values, which are most common. C-PACK+Z, similar

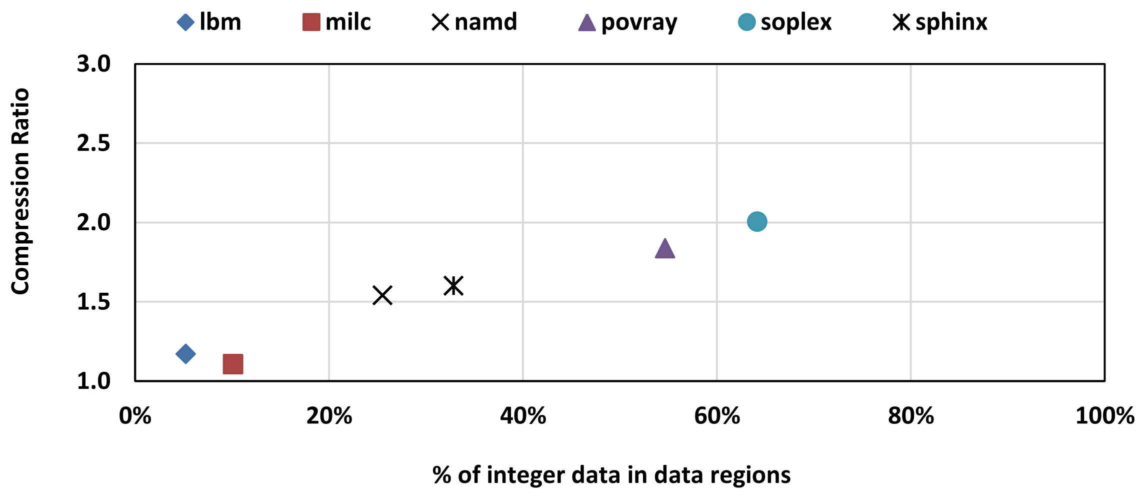


Figure 6-7: Percent of integer blocks in data regions of memory vs. compression ratio of SPEC CFP using C-PACK+Z.

to other basic algorithms, is mostly optimized for small integer values. Thus, in Figure 6-7, the higher the percentage of integer blocks, the higher the compression ratio is. In other words, integer data is more compressible than non-integer data. Among these benchmarks, povray and soplex have the highest percent of integer values, resulting in their higher compression ratio. On the other hand, lbm and milc have mostly non-integer values, and so the lowest compression ratios. In general, the higher the percentage of non-integer blocks (including floating-point data), the lower the compression ratio is.

There are several proposals to improve compression for floating-point data. Mahapatra et al. [98][99] state that floating-point benchmarks of SPEC CPU2000 have lower entropy, and so potentially higher compressibility than integer benchmarks. Burtscher et al. [16][106] also propose specialized compression algorithms that exploit the similarities among a sequence of floating-point values. Although these techniques achieve high compressibility for floating-point data, they might not be viable for on-line hardware cache/memory compression due to their complexity and overheads. Since our focus is on more general compression algorithms, we do not evaluate these techniques here.

Myth 3: [Confirmed!] *floating-point data have low compression ratio with general-purpose algorithms.*

6.5.4 Myth 4: Instructions Are Mostly Uncompressible

Existing general-purpose compression algorithms usually achieve good compressibility for data blocks with repeated bit patterns; however, instruction blocks have more complicated coding. In addition, one 64-byte instruction block can include multiple different instructions

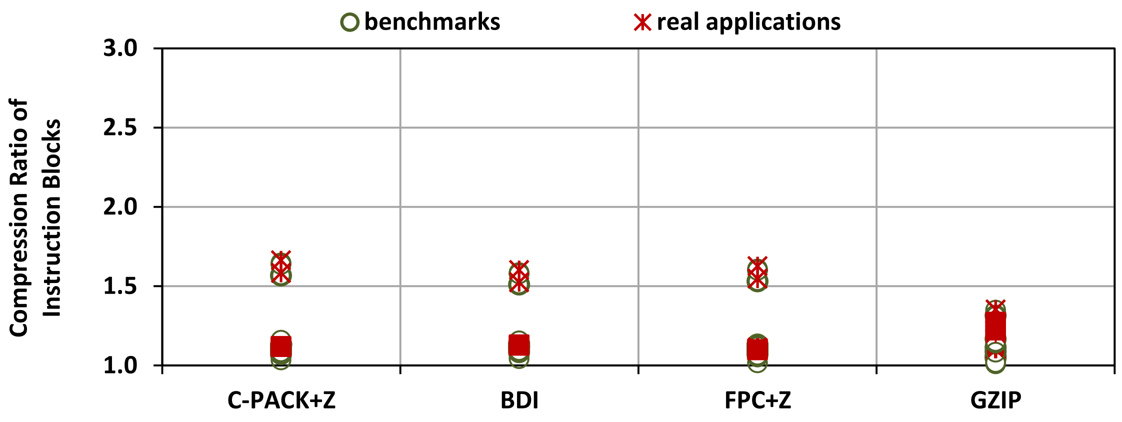


Figure 6-8: Compression ratio of instruction blocks.

(e.g., X86 instructions can have variable length) with low resemblance. Thus, instruction blocks are mostly considered as uncompressible using general-purpose compression algorithms [42]. To evaluate this myth for our applications, we analyze compressibility of instruction blocks (i.e., blocks in instruction/binary region) in main memory.

Figure 6-8 shows the average compression ratio of 64-byte instruction blocks in memory using different compression algorithms. In general, all our studied algorithms perform similarly and poorly for instruction blocks of our applications. Overall, the compression ratio of instruction blocks is on average 1.13, and up to 1.6 for gedit, Wiki, povray and soplex using C-PACK+Z.

Figure 6-9 shows the cumulative distribution of compressed block sizes in instruction/binary regions. Most applications have similar distributions, shown as skinny gray lines. We highlight a few representative applications. Zero blocks exist in instruction regions due to zero padding. There are on average 5% (up to 50% for openCalc) zero blocks in these regions. Note that, we do not count zero blocks in Figure 6-8 when we calculate the average compression

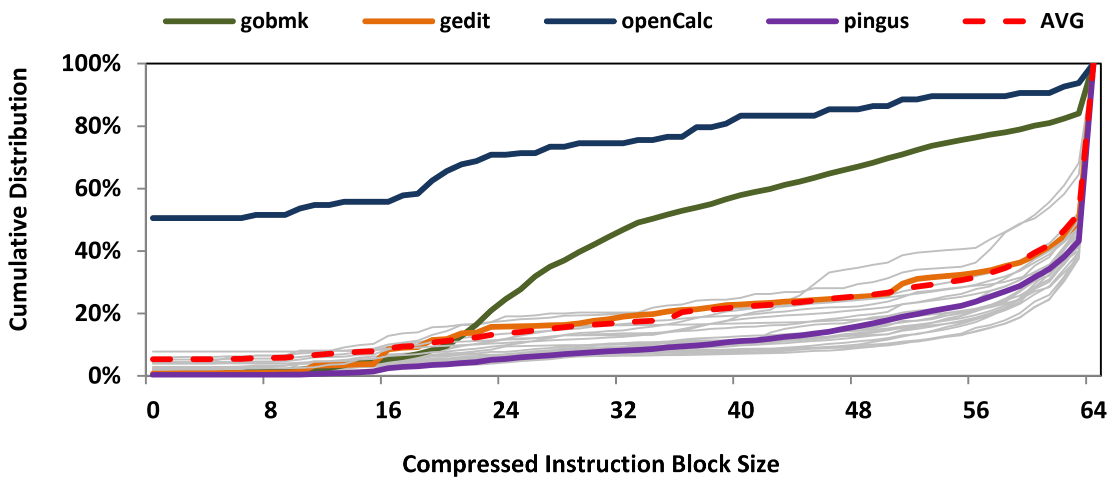


Figure 6-9: Cumulative distribution of compressed block sizes in instruction memory regions.

ratio of instruction blocks. Ignoring zero blocks, on average over 70% of instruction blocks are uncompressible (i.e., compressed block size of 64B) or poorly compressible (i.e., compressed block size ≥ 48B).

Although instruction blocks are mostly uncompressible using general algorithms, there are several specialized techniques to improve compressibility of instructions. These techniques usually find frequently used instruction sequences in instruction stream, replacing those with small code-words [1][4][107]. These techniques are usually applied after compilation and code generation. Mahapatra et al. [98][99] also show that instruction caches have low entropy and so high potentials for compression. Compressing instruction caches could be useful to reduce cache miss rate in case of large instruction footprints. However, since instructions account for a small fraction (i.e., few pages) of memory footprint for most applications, compressing instructions have low impacts on overall memory utilization of many applications.

Myth 4: [Confirmed!] *On average, over 70% of instruction blocks are uncompressible or poorly compressible.*

6.5.5 Myth 5: Neighboring Blocks Have Similar Compressibility

Neighboring blocks tend to have similar characteristics including access rate, hit or miss patterns [108]. Previous work [61] also shows that blocks of a page have similar compressibility (i.e., compression locality). Pekhimenko et al. [61] assume that "all cache lines within a page should be compressed to the same size". They propose Linearly Compressed Pages (LCP) that uses a fixed size for compressed blocks within a given page of main memory to simplify lookups. In SCC [23], we also exploit compression locality within small regions (4 or 8 neighboring blocks) in the L3 cache. In this way, as we discussed in the previous section, SCC simplifies cache lookups by compressing and fitting the neighboring blocks in one 64-byte data block if possible. These techniques [61][23] would benefit applications that have high compression locality. Otherwise, they might lower compression benefits due to internal fragmentation. In this section, we explore whether compression locality holds for our applications, and if so to what extent.

Figure 6-10 shows the cumulative distribution of unique compressed block sizes within a 4KB page. We use C-PACK+Z to compress each block to 0 to 8 8-byte sub-blocks. For each page, we then find the distribution of blocks ranked from the most common size to the least common size. The most common size (1-MCS) changes per page. For example, for Wiki, the most common size is zero in one page, and 32B in another. For each application, we then report the overall distribution in Figure 6-10.

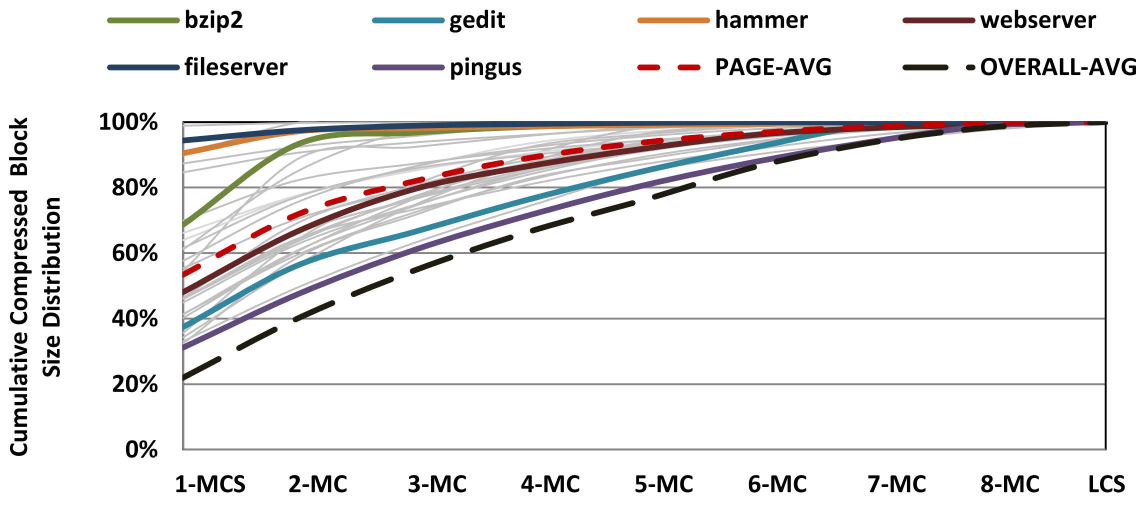


Figure 6-10: Cumulative distribution of compressed block sizes within a page in main memory. Sizes are ordered from the most common size (1-MCS) to the least common size (LCS) for each individual application on the X axis.

Figure 6-10 also shows the OVERALL-AVG, the cumulative distribution (most common to least common) of compressed block sizes across all pages of all workloads. This curve shows that the most common compressed block size over all workloads, 64 bytes, accounts for 22% of all blocks and the top three most common sizes, 64, 0, and 32 bytes, account for 55% of all blocks. In contrast, PAGE-AVG averages the frequency of the most common size within a page across all pages and all workloads (where different pages may have different most common sizes). If compressed block sizes were randomly distributed across pages, we would expect the per-page distribution be similar to the overall average distribution. Instead, PAGE-AVG shows that the most common block size within a page accounts for 53% of the blocks, while the top three most common sizes account for 82% of the blocks. These results clearly show that pages exhibit compression locality, with the most common block size within a page occurring more than twice as often as the most common block size overall. We observed similar results using other compression algorithms, e.g., BDI, and other sub-block sizes.

The most common compressed block size changes per application. For many of our applications, either zero or 64 bytes is the most common block size. 64B blocks (i.e., uncompressible blocks) are most common for gedit, hammer and milc, while zero blocks are most common in bzip2, Postgres server, Fileserver, and Tracer.

Applications exhibit different levels of compression locality. Figure 6-11 shows the overall and per-page distributions of compressed block sizes for two representative workloads. Webserver exhibits compression locality similar to the overall average: the most common block size (32B) accounts for 27% of blocks overall (Webserver-overall), while the most common size within a page accounts for 48% of blocks (Webserver-per-page). This locality holds even for the second and the third most common block sizes, similar to the average across all workloads. In contrast, gedit exhibits no compression locality, with the per-page distribution (gedit-per-page) essentially identical to its overall distribution of block sizes (gedit-overall).

Even though some applications have low compression locality within pages, they can have better locality in smaller memory regions. Figure 6-12 shows the cumulative distribution of

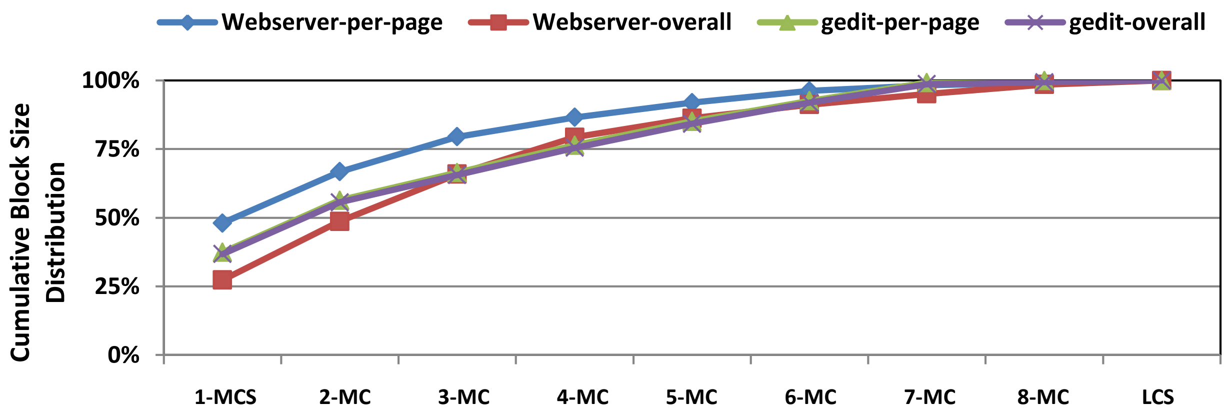


Figure 6-11: Cumulative distribution of compressed block sizes within a page in main memory (per-page) versus overall memory footprint for two representative applications.

compressed block sizes for gedit at different memory region sizes. Although gedit has low compression locality at page granularity, it has high locality at small regions. For example, in a small 8-block region, the most common size would account for 80% of the blocks. This means on average 6 blocks of 8 blocks have the same compressed block size. This could be due to similar compressibility of elements in data structures.

Myth 5: [Plausible!] *On average, the most common block size accounts for 53% of blocks within a page. However, the distribution has a heavy tail and even 3 different block sizes only account for 82% of the blocks in a page.*

Finding 5.1: *Compression locality is higher in smaller memory regions.*

6.5.6 Myth 6: Bigger Blocks Are More Compressible

Conventional belief is that larger blocks are more compressible [98][99][21]. In general more redundancy is present within larger blocks, but more effort is also needed to compress them. In this section, we study the impact of block granularity on compression ratio. We analyze the trade-offs to see whether the efforts worth the potential benefits of compressing larger

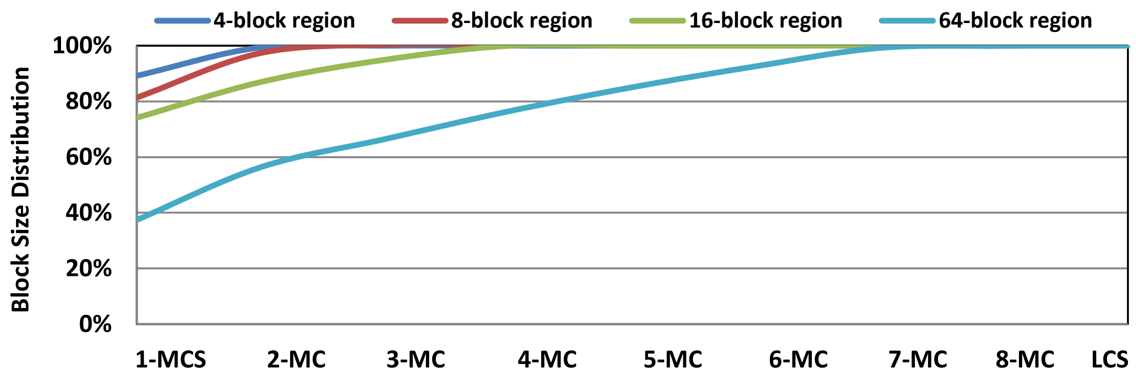


Figure 6-12: Cumulative distribution of compressed block sizes for gedit within regions with different sizes.

blocks.

Figure 6-13 shows the compression ratio of our applications in main memory using C-PACK+Z algorithm. We change block size from the small regular size (64B) to the page size (4096B). For C-PACK+Z, we use the default parameters described in Section 3.4. Compression ratio increases as we increase block size. Among our applications, servers, desktop applications, and SPEC CINT gain higher compressibility than SPEC CFP and mobile benchmarks at larger block granularities.

Figure 6-14 shows the average compression ratio of our servers with different compression algorithms at different block granularities. Compression sensitivity to block size differs per algorithm. While compressibility improves at larger block sizes with C-PACK+Z, larger block sizes do not impact compressibility with FPC+Z, and even can hurt efficiency of BDI. FPC+Z compresses each word in a block separately, so it is not sensitive to block size. The efficiency of BDI reduces at page granularity as it would be hard to find a small number of base values that

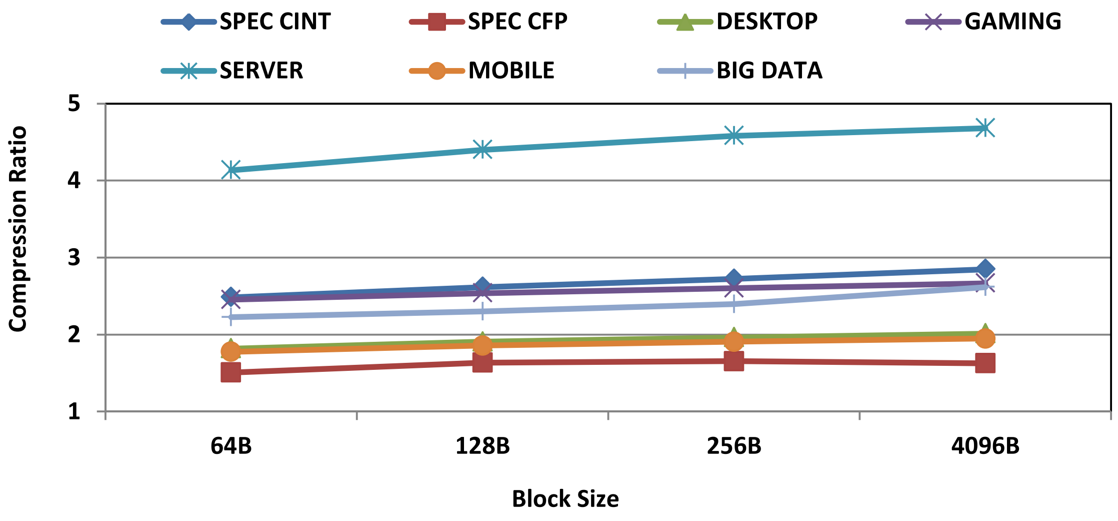


Figure 6-13: Compression with different block

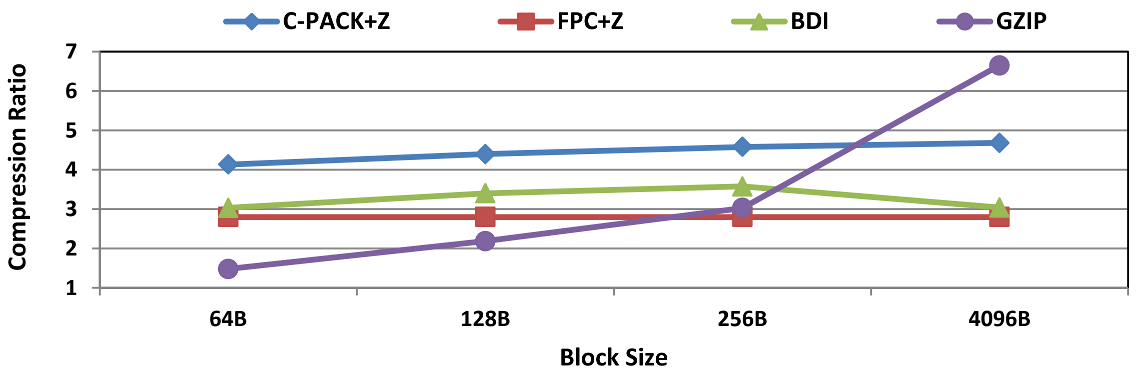


Figure 6-14: Compression ratio of servers in main memory.

are good for all words in a page. On the other hand, gzip is highly sensitive to block size. When applied at page granularity, gzip can provide on average 4.5x, and up to 12x (for Fileserver) higher compression ratio for servers.

Figure 6-13 and Figure 6-14 emphasize on the importance of picking the right compression algorithm at specific design points. While using larger blocks improve compression ratio, the benefits come at higher costs. Using larger blocks might negate the benefits from compression by increasing cache pollution, reducing cache efficiency, and incurring energy overheads [109]. Alternatively, as we showed in Chapter 4, DCC [21] dynamically detect neighboring blocks and co-compact them to get some of the compression benefits of larger blocks at lower overheads.

Myth 6: [Plausible!] *On average, compression ratio improves for block sizes larger than 64B. Overall, compressibility is not a direct function of block size. Compression sensitivity to block size depends on many factors including compression algorithm and data being compressed.*

Finding 6.1: *Compression sensitivity to block size depends more on the compression algorithm than the applications.*

6.5.7 Myth 7: More Complex Compression Algorithms Improve Compressibility

Compression algorithm is a key player in compressed caches and memory. In general, the conventional wisdom says more complex algorithms improve compressibility [19][20][21]. In this section, we study if this myth holds using different algorithms.

Figure 6-15 summarizes the compression ratio of different algorithms in main memory. We use 64B blocks with BDI, FPC+Z, and C-PACK+Z. Regarding complexity, we can order these algorithms as: BDI (the simplest), FPC+Z, C-PACK+Z, and gzip (the most complex). BDI has the simplest design, and so the lowest decompression latency. gzip uses the most complex technique, useful for software mechanisms. Despite the high complexity of gzip, it does not perform nearly as well as other algorithms for small block sizes. C-PACK+Z gains the highest compression ratio since it is especially designed to exploit replications in small cache/memory blocks. FPC+Z and BDI perform similarly on average despite more complexity of FPC+Z. At page granularity, gzip performs the best as it is optimized for compressing large amount of data.

Complexity is not necessarily an indicator of effectiveness of an algorithm. The key is to

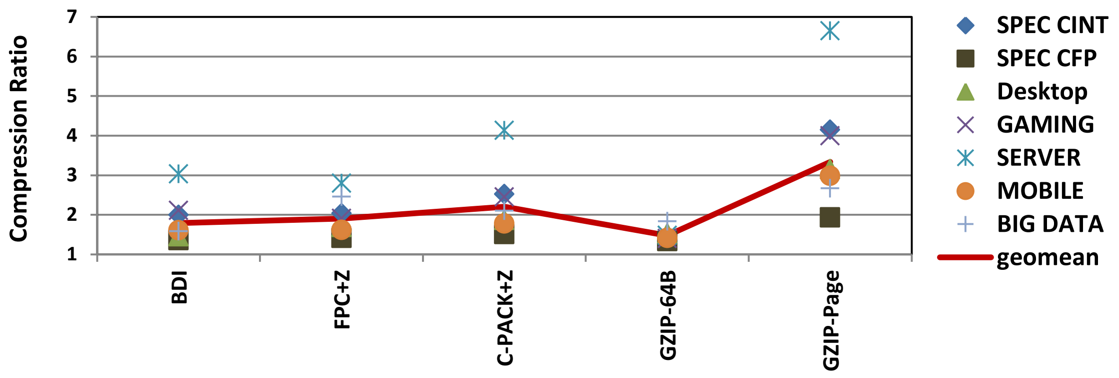


Figure 6-15: Memory compression with different algorithms.

find the algorithm that is tailored for the data being compressed at a particular level of the memory hierarchy. Besides compression ratio, there are different parameters to evaluate the success of a compression algorithm in the memory hierarchy, including compression/decompression latency, design complexity, and area and power overheads. For example, although C-PACK+Z gains higher compression ratio than BDI, its higher decompression latency might negate its benefits if applied at L1 caches. However, at higher levels of the memory hierarchy (i.e., L3 or memory), most applications can tolerate its decompression latency [21][22].

For a given compression algorithm, the next question is whether increasing its complexity would improve its effectiveness. For example, increasing dictionary size in a dictionary-based algorithm might improve its effectiveness [18]. Here, we study this myth for C-PACK+Z.

Figure 6-16 shows the effect of dictionary size on the compression ratio of our applications

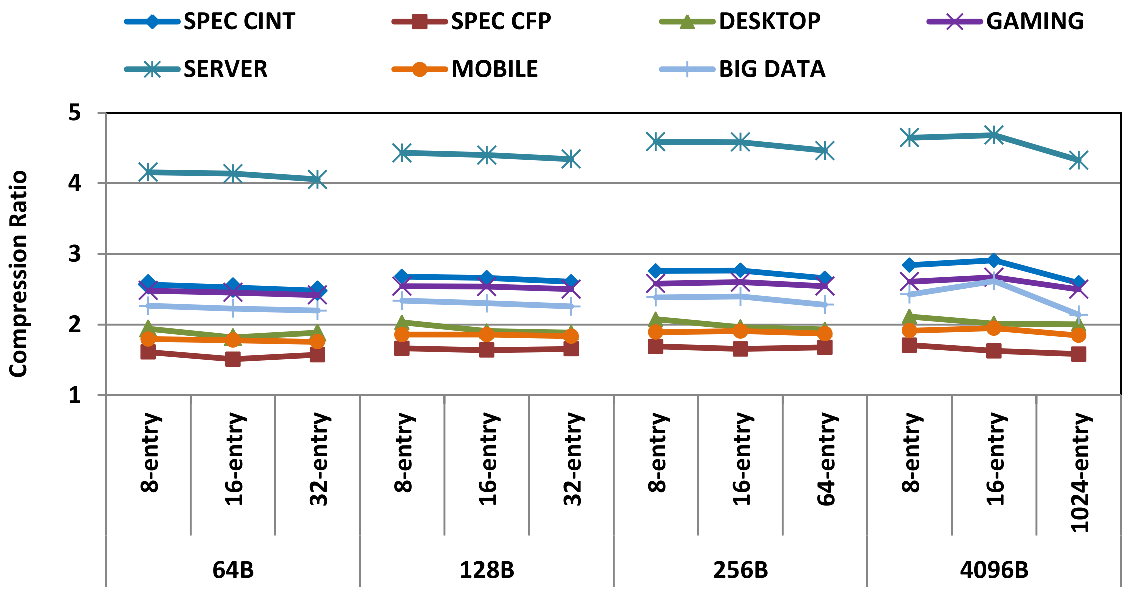


Figure 6-16: Sensitivity of compression ratio to dictionary size.

using C-PACK+Z at different block granularities. At each block granularity, we measure compression ratio with the default-size dictionary (16-entry), the half-size dictionary (8-entry), and the full-size dictionary. For example, when compressing a 128B block, which has 32 4-byte words, we measure the compression ratio with 8-entry, 16-entry and 32-entry dictionary sizes. A 32-entry dictionary can hold all words of a 128-byte block in case no word is repeated.

There are trade-offs in changing the dictionary size. While a larger dictionary can detect more repeated values, it would increase overheads. For example, with a 64-entry dictionary, C-PACK+Z stores 6 bits (the index to the matched entry) per matched word in the output code. While with an 8-entry dictionary it stores only 3 bits per word. This increase of metadata in the output code would negate the benefits of larger dictionary for majority of our applications even with large block sizes.

Myth 7: [Busted!] *Algorithm complexity is not always an indicator of better compressibility.*

Finding 7.1: *Overall, increasing the dictionary size does not improve compressibility. It can even hurt effectiveness of the algorithm.*

6.5.8 Myth 8: Sub-blocking Eliminates Most Internal Fragmentation

Managing variable-size compressed blocks complicates compressed cache/memory designs. Storing compressed blocks at byte granularity has huge overheads on cache metadata [21]. An alternative is to compress blocks into variable number of small sub-blocks to eliminate internal fragmentation while allowing variable-size compressed blocks [20][21][22][23].

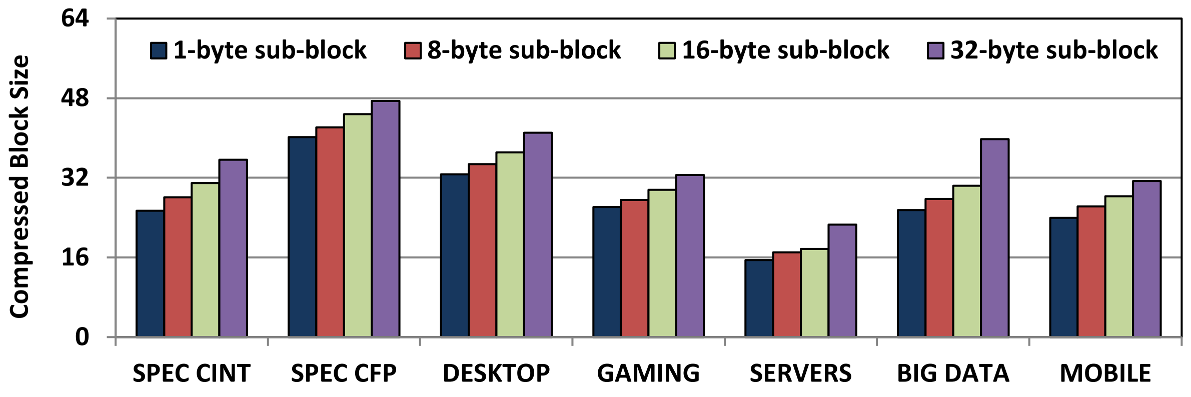


Figure 6-17: Average compressed block size using C-PACK+Z with different sub-block sizes.

Figure 6-17 shows the average compressed block size of our applications when using different sub-block sizes. The higher the compressed block size is, the lower the effective capacity would be. Using 32-byte sub-blocks (i.e., compressing a block to half if possible) would increase the average compressed size by 33% and up to 2 times (for libquantum). This means roughly on average 33% increase in internal fragmentation. Using small sub-block sizes, we can gain some of the benefits of byte-granularity with significantly lower overheads. For example, in Figure 6-17, 8-byte or 16-byte sub-blocks perform better than 32-byte sub-blocks (i.e., lower compressed block sizes), and get most benefits of 1-byte sub-blocks. Using 8-block sub-blocks would increase the average compressed size by only 7% compared to 1-byte sub-blocks, significantly lowering internal fragmentation compared to 32-byte sub-blocks.

Myth 8: [Confirmed!] *Compressing blocks to small sub-blocks reduce internal fragmentation.*

6.6 Conclusions

In this chapter, I take a holistic approach toward compression in the memory hierarchy. I explore 8 popular myths arise from conventional wisdoms and past experiences on compression, including a broad range of different design options (e.g., compression algorithm). I evaluate them using real-world workloads (such as production servers) and refine them into overall 12 findings about compression effectiveness. Through extensive analysis, I show that two of the eight myths are “Busted!,” two are “Plausible,” and the rest are “Confirmed!”. The analysis provides insights into compression in the memory hierarchy.

Chapter 7
Conclusions

In modern processors, last level caches (LLCs) mitigate the limited bandwidth and high latency of off-chip main memory. LLCs also play an increasingly important role in reducing memory system energy as they can filter out energy-expensive memory accesses. Increasing the LLC size can improve system performance and energy by reducing memory accesses, but at the cost of high area and power overheads. In this dissertation, I explored using compression to effectively improve the LLC capacity and ultimately system performance and energy consumption.

Cache compression is a promising technique for expanding effective cache capacity with little area overhead. Compressed caches can achieve the benefits of larger caches using the area and power of smaller caches by fitting more cache blocks in the same cache space. Ideally, a compressed cache design must balance three frequently-conflicting goals: i) tightly compacting variable-size compressed blocks, ii) keeping tag and other metadata overheads low, and iii) allowing fast lookups. Previous compressed cache designs achieved at most two of these three goals, limiting the potential benefits of compression. In addition, most previous proposals targeted improving system performance even at high power and energy overheads.

In this dissertation, I made several contributions that address concerns on different aspects of cache compression. I presented two novel compressed cache designs: Decoupled Compressed

Cache (DCC) [21][22] and Skewed Compressed Cache (SCC) [23]. DCC and SCC are both optimized for energy, eliminating the sources of energy inefficiencies in previous designs, while tightly packing variable size compressed blocks. They exploit spatial locality to reduce tag overheads by tracking super-blocks. Compared to conventional uncompressed caches, DCC and SCC improve the cache miss rate by increasing the effective capacity and reducing conflicts. Compared to DCC, SCC further lowers area overhead and design complexity.

Despite years of research on compressed caches, the industry has not yet adopted the use of cache compression. In this dissertation, in addition to presenting novel techniques to improve compression effectiveness, I showed that our designs can be implemented with limited changes to existing designs. Another main concern is that since most proposals on compressed caching are on non-existing hardware, architects evaluate those using detailed simulators with small benchmarks. So, whether cache compression would benefit real applications running on real machines was an open question. In this dissertation, I addressed this question by analyzing the compressibility of several real applications, including production servers of the Computer Sciences Department of UW-Madison. I showed that compression could in fact be beneficial to many real applications.

7.1 Directions for Future Work

While in this dissertation, I re-visited compressed caching for improving system performance and energy, I believe there are several opportunities for future research on compression in the memory hierarchy. Here, I outline few possible areas of research:

7.1.1 Adaptive Selective Compression

Although compression improves the performance and system energy of some applications, it might not help or it might even hurt the performance or energy of some others. In this thesis, I focused on compressing the LLC. At lower levels of the cache hierarchy (e.g., the L2 cache), at which sensitivity to cache latency is higher, more applications might get hurt from compression. Even for a specific application, different phases of one application may also show different levels of compressibility and cache sensitivity. Therefore, dynamically balancing the benefits of compression with its overheads is important. Here, I briefly explain how we can achieve this balance.

Adaptive Compression: Several techniques can be used to adaptively control compression. Alameldeen and Wood [57] used stack depth information of the cache replacement algorithm to determine whether or not to compress a block. By dynamically monitoring workloads' behavior and disabling compression when not effective, they balanced the benefits of compression for cache sensitive workloads, while avoiding performance degradation for others. Another possible technique for determining whether to compress a block or not is to use sampling mechanisms [79]. We can reserve a group of cache sets to compress their blocks all the time, and have another group that we never compress. By tracking and comparing the miss rates of these two groups, we can enable compression when it results in a lower number of misses. These techniques can be used with SCC and DCC. In addition, since SCC and DCC track blocks at multiple granularities, we could also leverage spatial locality, and the fact that many contiguous blocks have similar compressibility allows for better prediction of which blocks to compress or not.

Selective Compression: In addition to dynamically controlling compression, we can be selective with regard to which specific blocks to compress or not. Not all blocks brought to the cache are similarly sensitive to latency. Instruction blocks are usually on the critical path, and any extra delay on their access time might hurt performance. Some loads are also critical and need to complete early to prevent processor stalls (i.e., critical), while others could tolerate a longer latency (i.e., non-critical). The criticality of a load mostly depends on the chain of instructions dependent on that load. If most instructions following that load depend on it, the processor may stall if that load takes a long time. Previous work presented hardware-based techniques to classify the criticality of loads [80]. My hypothesis is that by considering the criticality of data stored in the caches, and compressing non-critical ones, we could balance the benefits of compression (i.e., lower miss rate) versus its overheads (i.e., decompression latency), especially at lower levels of the cache hierarchy (e.g., L2).

7.1.2 Exploiting High Tag-Reach and Coarse-Granularity

In large caches, accessing the data array incurs much higher energy and latency overheads than the tag array. Thus, using the available data space efficiently is critical for improving energy costs. I define **Tag-Reach** as the ratio of the maximum number of cache blocks mapped by the tag array over the maximum number of uncompressed blocks held in the data array. Regular caches have tag-reach of one. A compressed cache must have a tag-reach greater than one, so that it can fit more blocks in the cache. Both DCC and SCC provide high tag-reach by tracking super-blocks. They can track up to 4x (using 4-block super-blocks), and 8x (using 8-block super-blocks) blocks. With compression, DCC and SCC hold on average about twice as many blocks as a conventional cache, using on average 50% and 25% of available tags, respectively.

However, on average 50% of available tags in DCC and 75% of tags in SCC are not employed. Therefore, there is plenty of space left for improvement. Here, I explain some potential techniques.

- **Exploiting extra tags for tracking non-reusable blocks in the LLC:** Inclusion has been widely used in commercial processors to simplify the LLC coherence and reduce on-chip traffic. On the other hand, inclusion can reduce locality due to replication. For instance, some cache blocks are being accessed once at the LLC and possibly multiple times at lower cache levels (i.e., streaming access patterns) [110][113]. Storing these blocks at the LLC reduces the LLC effective associativity, and thus locality. DCC and SCC can be extended to leverage these properties. Using their high tag-reach, I can store these blocks as data-less at the LLC, storing only their tags, and so releasing data space to fit more data blocks at the LLC.

- **Global cache space management:** Another possible way to leverage the high tag-reach of SCC and DCC is for better cache hierarchy space management. The memory requirements of threads or applications running on different cores of a multicore processor usually vary [111][112]. For example, some threads are usually more critical and affect system performance and energy the most. I can leverage this variation for better cache management. In addition to sharing the LLC, SCC and DCC can enable better aggregate cache space management across the cache hierarchy. They can enable cores to even share their private cache space if necessary. For larger private caches, which are common in commercial systems, some threads or applications can trade a portion of their private caches with more critical threads running on other cores. I can leverage extra tags at the LLC for implementing this feature. For these blocks, I can store their tags at the LLC while keeping the whole cache blocks (tag+data) in another core's private cache space. Later, on a new request, the block can be transferred to the requester's private cache. In this way, I potentially expand the LLC shared capacity by stealing space from different cores' private caches. Thus, I can replace some of the off-chip accesses with cheaper on-chip accesses.

- **Interaction with prefetching:** The extra tags provided by our designs could also be used to further improve cache utilization through techniques orthogonal to compression, such as prefetching. In this thesis, I did not study the interaction between prefetching and compression in our designs. Similar to previous work [57], I can use the extra tags to better predict useful and harmful prefetches. In addition, I predict that prefetching, especially stream prefetching, could further improve the benefits of DCC and SCC, as there would be more neighboring blocks in the cache, and so more populated super-blocks.

7.1.3 Parallel Lookup with SCC

In this dissertation, I focused on compressing the LLC. To reduce power consumption, current LLC designs use sequential tag-data access, checking the tag array first before accessing the data array. Typical compressed caches build on this technique, checking whether a block is compressed and where it is located in the data array first. Unlike LLCs, the lower-level caches (L1 or L2) use a parallel tag-data access model reading both the tag array and the data array simultaneously. In this way, they keep access latency low at the cost of higher power consumption as both the tag array and the data array would be accessed even on cache misses. Unlike VSC [57] and even our proposal DCC, which cannot support parallel tag-data access, SCC can also support parallel tag/data array access. Not only does SCC allow fast lookups, which are suitable for lower-level caches, but it also can save power on cache accesses, thanks to its special direct tag-data mapping.

Since SCC eliminates any extra metadata or level of indirection to locate a compressed block, it supports fast low-power parallel access. Given a block address, in each cache way, SCC knows its compressed size and exact location within a set. Thus, when reading the tag array and the data array in parallel, instead of reading out the whole 64B blocks, SCC will read out less (e.g., 32B if compressed to half). In this way, compared to a regular parallel cache, SCC can significantly reduce cache dynamic energy, which is considerable at L1/L2. For example, in a 4-way cache associative cache, SCC reads 120 bytes (64B from way #1 (uncompressed) + 32B from way #2 + 16B from way #3 + 8B from way #4), reducing cache dynamic energy by 53%. This gain is not possible by typical compressed caches or DCC as they do not know the block size before checking the tag array.

References

[1] C. Lefurgy, P. Bird, I. Chen, T. Mudge. Improving code density using compression techniques. *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, p.194-203, December 01-03, 1997, Research Triangle Park, North Carolina, USA.

[2] Yuan-Long Jeang, Jen-Wei Hsieh, Yong-Zong Lin. An Efficient Instruction Compression/Decompression System Based on Field Partitioning. *2005 IEEE International Midwest Symposium on Circuits and Systems*, Aug. 7-10, 2005.

[3] Subash Chandar, Mahesh Mehendale, R. Govindarajan. Area and Power Reduction of Embedded DSP Systems using Instruction Compression and Re-configurable Encoding. *Journal of VLSI Signal Processing Systems*, v.44 n.3, p.245-267, September 2006.

[4] Chen, P. Bird, and T. Mudge. The impact of instruction compression on I-cache performance. *Tech. Rep. CSE-TR-330-97*, EECS Department, University of Michigan, 1997.

[5] L. Benini, A. Macii, E. Macii, and M. Poncino. Selective instruction compression for memory energy reduction in embedded systems. *Proceedings IEEE International Symposium Low-Power Electronics and Design*, pp.206 -211 1999.

[6] Keith D. Cooper, Nathaniel McIntosh. Enhanced code compression for embedded RISC processors. *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, p.139-149, May 01-04, 1999, Atlanta, Georgia, USA.

[7] Andrew Wolfe and Alex Chanin. Executing compressed programs on an embedded RISC architecture. *In Proceedings of the 25th annual international symposium on Microarchitecture (MICRO 25)*. 1992.

[8] S.Y. Larin, and T.M. Conte. Compiler-driven cached code compression schemes for embedded ILP processors. *Proceedings. 32nd Annual International Symposium on Microarchitecture*, vol., no., pp.82,92, 1999.

[9] H. Lekatsas and W. Wolf. SAMC: a code compression algorithm for embedded processors. *Trans. Comp.-Aided Des. Integ. Cir. Sys.* 18, 12 (November 2006), 1689-1701.

[10] Martin Thuresson, Per Stenström. Evaluation of extended dictionary-based static code compression schemes. *Conf. Computing Frontiers* 2005: 77-86.

[11] Martin Thuresson, Magnus Själander, Per Stenstrom. A Flexible Code Compression Scheme Using Partitioned Look-Up Tables. *Proceeding HiPEAC '09 Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, 2009.

[12] Martin Isenburg, Peter Lindstrom, Jack Snoeyink. Lossless Compression of Floating-Point Geometry. *Proceedings of CAD'3D*, May 2004.

[13] Martin Isenburg, Peter Lindstrom, Jack Snoeyink. Lossless Compression of Predicted Floating-Point Geometry. *Computer-Aided Design*, Volume 37, Issue 8, pages 869-877, July 2005.

[14] Peter Lindstrom, Martin Isenburg. Fast and Efficient Compression of Floating-Point Data. *IEEE Transactions on Visualization and Computer Graphics, Proceedings of Visualization* 2006, 12(5), pages 1245-1250, September-October 2006.

[15] P. Ratanaworabhan, J. Ke and M. Burtscher. Fast Lossless Compression of Scientific Floating-Point Data. *Proc. Data Compression Conf. (DCC ',06)*, pp. 133-142, Mar. 2006.

[16] M. Burtscher, P. Ratanaworabhan. FPC: A High-Speed Compressor for Double-Precision Floating-Point Data. *IEEE Transactions on Computers*, vol.58, no.1, pp.18, 31, Jan. 2009.

[17] Andrew Beers, Maneesh Agrawala, Navin Chaddha. Rendering from Compressed Textures. *Computer Graphics, Proc. SIGGRAPH*: 373–37, 1996.

[18] Xi Chen, Lei Yang, Robert P. Dick, Li Shang, Haris Lekatsas. C-pack: a high-performance microprocessor cache compression algorithm, *IEEE Transactions on VLSI Systems,* 2010.

[19] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2012. Base-delta-immediate compression: practical data compression for on-chip caches. *In Proceedings of the 21st international conference on Parallel architectures and compilation techniques (PACT '12)*. ACM, New York, NY, USA, 377-388.

[20] A. Alameldeen and D. Wood. Adaptive Cache Compression for High-Performance Processors. *In Proceedings of the 31st Annual International Symposium on Computer Architecture*, 2004.

[21] S. Sardashti and D. Wood. Decoupled Compressed Cache: Exploiting Spatial Locality for Energy-Optimized Compressed Caching. *Annual IEEE/ACM International Symposium on Microarchitecture*, 2013.

[22] S. Sardashti and D. Wood. Decoupled Compressed Cache: Exploiting Spatial Locality for Energy Optimization. *IEEE Micro Top Picks from the 2013 Computer Architecture Conferences*.

[23] S. Sardashti, A. Seznec, and D. Wood. Skewed Compressed Caches. *In the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*, 2014.

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

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

[26] 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.

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

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

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

[30] 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.

[31] 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.

[32] 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.

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

[34] Jonghyun Lee, MarianneWinslett, 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.

[35] 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.

[36] 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.

[37] J. Dusser, T. Piquet, and A. Seznec. Zero-content augmented cache. *In Proceedings of the 23rd international conference on Supercomputing*, 2009.

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

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] Angelos Arelakis, Per Stenstrom. SC2: A statistical compression cache scheme. *In Proceedings of the 41st Annual International Symposium on Computer Architecture*, 2014.

[47] 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.

[48] 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.

[49] 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*, 2000.

[50] E. Hallnor, S. Reinhardt. A Unified Compressed Memory Hierarchy. *In Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, 2005.

[51] Julien Dusser, Thomas Piquet, André Seznec. Zero-content augmented caches. *In Proceedings of the 23rd international conference on Supercomputing*, 2009.

[52] Luis Villa, Michael Zhang, and Krste Asanovic. Dynamic zero compression for cache energy reduction. *In Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, 2000.

[53] Soontae Kim, Jesung Kim, Jongmin Lee, Seokin Hong. Residue Cache: A Low-Energy Low-Area L2 Cache Architecture via Compression and Partial Hits. *In Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011.

[54] Gennady Pekhimenko, Tyler Huberty, Rui Cai, Onur Mutlu, Phillip P. Gibbons, Michael A. Kozuch, and Todd C. Mowry. Exploiting Compressed Block Size as an Indicator of Future Reuse. *Proceedings of the 21st International Symposium on High-Performance Computer Architecture (HPCA)*, Bay Area, CA, February 2015.

[55] Seungcheol Baek, Hyung Gyu Lee, Chrysostomos Nicopoulos, Junghee Lee, and Jongman Kim. ECM: Effective Capacity Maximizer for High-Performance Compressed Caching. *In Proceedings of IEEE Symposium on High-Performance Computer Architecture*, 2013.

[56] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer. High performance cache replacement using re-reference interval prediction (rrip). *In Proceedings of the 37th annual international symposium on Computer architecture*, ISCA ’10, pages 60–71, New York, NY, USA, 2010. ACM.

[57] A. R. Alameldeen and D. A. Wood. Interactions between compression and prefetching in chip multiprocessors, *Proc. Int. Symp. High-Performance Computer Architecture*, pp.228 - 239 2007.

[58] Bulent Abali, Hubertus Franke, Xiaowei Shen, Dan E. Poff, and T. Basil Smith. Performance of Hardware Compressed Main Memory. *In Proceedings of the 7th IEEE Symposium on High-Performance Computer Architecture*, 2001.

[59] M. Ekman, P. Stenstrom. A robust main-memory compression scheme. *SIGARCH Computer Architecture News*, 2005.

[60] Julien Dusser, Andre Seznec. Decoupled Zero-Compressed Memory. *In Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, 2011.

[61] Gennady Pekhimenko, Vivek Seshadri, Yoongu Kim, Hongyi Xin, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, Todd C. Mowry. Linearly Compressed Pages: A Low Complexity, Low-Latency Main Memory Compression Framework. *Annual IEEE/ACM International Symposium on Microarchitecture*, 2013.

[62] 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.

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

[64] 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.

[65] Ali Shafiee, Meysam Taassori, Rajeev Balasubramonian, and Al Davis. MemZip: Exploring Unconventional Benefits from Memory Compression. *HPCA*, 2014.

[66] 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.

[67] R. S. de Castro, A. P. do Lago, and D. Da Silva. Adaptive Compressed Caching: Design and Implementation. *In SBAC-PAD*, 2003.

[68] S. Roy, R. Kumar, and M. Prvulovic. Improving system performance with compressed Memory. *In IPDPS '01: Proceedings of the 15th International Parallel & Distributed Processing Symposium*, page 66, Washington, DC, USA, 2001. IEEE Computer Society.

[69] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis. The case for compressed caching in virtual memory systems. *In ATEC '99: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 101–116, Berkeley, CA, USA, 1999. USENIX Association.

[70] Apple OS X Mavericks

http://www.apple.com/media/us/osx/2013/docs/OSX_Mavericks_Core_Technology_Overvie w.pdf.

[71] R. Dennard, F. Gaensslen, H. Yu, V. Rideovt, E. Bassous, A. Leblanc. Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions. *IEEE Journal of Solid-State Circuits*, 1974.

[72] S. Keckler, S. Life After Dennard and How I Learned to Love the Picojoule. *In Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011.

[73] A. Hartstein, V. Srinivasan, T. R. Puzak, P. G. Emma. On the Nature of Cache Miss Behavior: Is It √2? J. *Instruction-Level Parallelism*, 2008.

[74] Intel Core i7 Processors http://www.intel.com/products/processor/corei7/

[75] Andre Seznec. Decoupled sectored caches: Conciliating low tag implementation cost and low miss ratio. *International Symposium on Computer Architecture*, 1994.

[76] Avinash Sodani. Race to Exascale: Challenges and Opportunities. *Intl. Symp. Microarchitecture*, 12/2011.

[77] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, pages 114–117, Apr. 1965.

[78] PG&E Data Center Best Practices Guide.

[79] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely Jr., and Joel S. Emer. Adaptive insertion policies for high performance caching. *In 34th International Symposium on Computer Architecture (ISCA 2007)*, June 9-13, 2007, San Diego, California, USA. ACM, 2007.

[80] S. T. Srinivasan, R. D. Ju, A. R. Lebeck, and C. Wilkerson. Locality vs. Criticality. *In Proc. ISCA-28*, pp. 132-143, 2001.

[81] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's

General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News*, 2005.

[82] J. Liptay. Structural Aspects of the System/360 Model85 Part II: The Cache. *IBM Systems Journal*, 1968.

[83] J. Rothman and A. Smith The Pool of Subsectors Cache Design. *International Conference on Supercomputing*, 1999.

[84] D. Yoon, M. Jeong, Mattan Erez. Adaptive granularity memory systems: A tradeoff between storage efficiency and throughput. *In Proceeding of the 38th Annual International Symposium on Computer Architecture*, 2011.

[85] D. Weiss, M. Dreesen, M. Ciraula, C. Henrion, C. Helt, R. Freese, T. Miles, A. Karegar, R. Schreiber, B. Schneller, J. Wuu. An 8MB Level-3 Cache in 32nm SOI with Column-Select Aliasing. *Solid-State Circuits Conference Digest of Technical Papers*, 2011.

[86] ITRS. International technology roadmap for semiconductors, 2010 update, 2011. URL http://www.itrs.net

[87] CACTI: http://www.hpl.hp.com/research/cacti/

[88] Calculating memory system power for DDR3. *Technical Report TN-41-01*. Micron Technology, 2007.

[89] Alaa R. Alameldeen, Milo M. K. Martin, Carl J. Mauer, Kevin E. Moore, Min Xu, Mark D. Hill, David A. Wood, Daniel J. Sorin. Simulating a $2M Commercial Server on a $2K PC. *IEEE Computer*, 2003.

[90] A. Alameldeen, D. Wood. Variability in Architectural Simulations of Multi-threaded Workloads. *In Proceedings of the Ninth IEEE Symposium on High-Performance Computer Architecture*, 2003.

[91] Christian Bienia and Kai Li. PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors. *In Workshop on Modeling, Benchmarking and Simulation*, 2009.

[92] Vishal Aslot, M. Domeika, Rudolf Eigenmann, Greg Gaertner, Wesley B. Jones, and Bodo Parady. SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. *In Workshop on OpenMP Applications and Tools*, 2001.

[93] A. Seznec. A case for two-way skewed-associative caches. *In Proc. of the 20th annual Intl. Symp. on Computer Architecture*, 1993.

[94] A. Seznec. Concurrent Support of Multiple Page Sizes on a Skewed Associative TLB. *IEEE Transactions on Computers*, 2004.

[95] A. Seznec, F. Bodin. Skewed-associative caches. *Proceedings of PARLE' 93*, Munich, June 1993, also available as INRIA Research Report 1655.

[96] N. Binkert, B. Beckmann, G. Black, S. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. Hill, D. Wood. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 2011.

[97] G. Hamerly, E. Perelman, J. Lau, and B. Calder. SimPoint 3.0: Faster and More Flexible Program Analysis. *In Proceedings of the Work. on Modeling, Benchmarking and Simulation*, 2005.

[98] N.R. Mahapatra, J. Liu, K. Sundaresan, S. Dangeti, and B.V. Venkatrao. A limit study on the potential of compression for improving memory system performance, power consumption, and cost. *J. Instruction-Level Parallelism*, vol. 7, pp.1 -37 2005.

[99] N.R. Mahapatra, J. Liu, K. Sundaresan, S. Dangeti, and B.V. Venkatrao. The Potential of Compression to Improve Memory System Performance, Power Consumption, and Cost. *In Proceedings of IEEE Performance, Computing and Communications Conference*, Phoenix, AZ, USA, April 2003.

[100] J. Gandhi, A. Basu, M. Hill, and M. Swift. BadgerTrap: A Tool to Instrument x86-64 TLB Misses. *SIGARCH Computer Architecture News (CAN)*, 2014.

[101] graph500 --The Graph500 List: http://www.graph500.org.

[102] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. Popescu, A. Ailamaki, and B. Falsafi. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. *In the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2012.

[103] Coremark -- www.coremark.org

[104] Cloudsuite -- http://parsa.epfl.ch/cloudsuite/cloudsuite.html

[105] A. Gutierrez, R. Dreslinski, T. Wenisch, T. Mudge, A. Saidi, C. Emmons, and N. Paver. Full-system analysis and characterization of interactive smartphone applications. *In IISWC '11*.

[106] M. Burtscher and P. Ratanaworabhan. High throughput compression of double-precision floating-point data. *In DCC*. 2007.

[107] Y. Jin and R. Chen. Instruction cache compression for embedded systems. *Berkley Wireless Research Center, Technical Report*, 2000.

[108] C. Wu, A. Jaleel, W. Hasenplaugh , M. Martonosi , S. Steely, J. Emer. SHiP: signature-based hit predictor for high performance caching. *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011.

[109] D. Yoon, M. Jeong, M. Erez. Adaptive granularity memory systems: A tradeoff between storage efficiency and throughput. *In Proceeding of the 38th Annual International Symposium on Computer Architecture*, 2011.

[110] Arkaprava Basu, Derek R. Hower, Mark D. Hill, Michael M. Swift. Freshcache: Statically and dynamically exploiting dataless ways. *In ICCD*, 2013.

[111] J. Chang and G. S. Sohi. Cooperative caching for chip multiprocessors. *In ISCA-33*, 2006.

[112] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. *ICS-21*, 2007.

[113] Jorge Albericio, Pablo Ibáñez, Víctor Viñals, José M. Llabería. The reuse cache: downsizing the shared last-level cache. *In Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2013, pp. 310–321.

[114] Vijay Sathish, Michael J. Schulte, Nam Sung Kim, Lossless and lossy memory I/O link compression for improving performance of GPGPU workloads, *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, 2012.

[115] Arka Basu, Revisiting Virtual Memory, PhD dissertation, 2013.