Thesis for The Degree of Doctor of Philosophy

Lossy and Lossless Compression Techniques to Improve the Utilization of Memory Bandwidth and Capacity

Albin Eldstål-Ahrens

Division of Computer Networks and Systems
Department of Computer Science & Engineering
Chalmers University of Technology
Gothenburg, Sweden, 2022

Lossy and Lossless Compression Techniques to Improve the Utilization of Memory Bandwidth and Capacity

Albin Eldstål-Ahrens

Advisor:
Professor Ioannis Sourdis, Chalmers University of Technology

Co-Advisor:
Angelos Arelakis, Ph.D., ZeroPoint Technologies

Examiner:
Adjunct Professor Fredrik Dahlgren, Chalmers University of Technology

Thesis Opponent:
Professor Moinuddin K. Qureshi, Georgia Institute of Technology

Grading Committee:
Professor Stefanos Kaxiras, Uppsala University
Professor Andreas Moshovos, University of Toronto
Professor H. Peter Hofstee, Delft Technical University, IBM

Deputy Committee:
Risat Pathan, Chalmers University of Technology

ISBN 978-91-7905-607-0
Doktorsavhandlingar vid Chalmers tekniska högskola, Ny serie nr 5073.
ISSN 0346-718X

Technical Report No 209D
Department of Computer Science & Engineering
Division of Computer Networks and Systems
Chalmers University of Technology
Gothenburg, Sweden

This thesis has been prepared using LaTeX.
Printed by Chalmers Reproservice,
Gothenburg, Sweden 2022.

Abstract

Main memory is a critical resource in modern computer systems and is in increasing demand. An increasing number of on-chip cores and specialized accelerators improves the potential processing throughput but also calls for higher data rates and greater memory capacity. In addition, new emerging data-intensive applications further increase memory traffic and footprint. On the other hand, memory bandwidth is pin limited and power constrained and is therefore more difficult to scale. Memory capacity is limited by cost and energy considerations. This thesis proposes a variety of memory compression techniques as a means to reduce the memory bottleneck. These techniques target two separate problems in the memory hierarchy: memory bandwidth and memory capacity. In order to reduce transferred data volumes, lossy compression is applied which is able to reach more aggressive compression ratios. A reduction of off-chip memory traffic leads to reduced memory latency, which in turn improves the performance and energy efficiency of the system. To improve memory capacity, a novel approach to *memory compaction* is presented. The first part of this thesis introduces Approximate Value Reconstruction (*AVR*), which combines a low-complexity *downsampling* compressor with an LLC design able to co-locate compressed and uncompressed data. Two separate thresholds limit the error introduced by approximation. For applications that tolerate aggressive approximation in large fractions of their data, in a system with 1GB of 1600MHz DDR4 per core and 1MB of LLC space per core, AVR reduces memory traffic by up to 70%, execution time by up to 55%, and energy costs by up to 20% introducing at most 1.2% error in the application output. The second part of this thesis proposes Memory Squeeze (*MemSZ*), introducing a parallelized implementation of the more advanced Squeeze (*SZ*) compression method. Furthermore, MemSZ improves on the error limiting capability of AVR by keeping track of life-time accumulated error. An alternate memory compression architecture is also proposed, which utilizes 3D-stacked DRAM as a last-level cache. In a system with 1GB of 800MHz DDR4 per core and 1MB of LLC space per core, MemSZ improves execution time, energy and memory traffic over AVR by up to 15%, 9%, and 64%, respectively. The third part of the thesis describes L^2C, a hybrid lossy and lossless memory compression scheme. L^2C applies lossy compression to approximable data, and falls back to lossless if an error threshold is exceeded. In a system with 4GB of 800MHz DDR4 per core and 1MB of LLC space per core, L^2C improves on the performance of MemSZ by 9%, and energy consumption by 3%. The fourth and final contribution is FlatPack, a novel memory compaction scheme. FlatPack is able to reduce the traffic overhead compared to other memory compaction systems, thus retaining the bandwidth benefits of compression. Furthermore, FlatPack is flexible to changes in block compressibility both over time and between adjacent blocks. When available memory corresponds to 50% of the application footprint, in a system with 4GB of 800MHz DDR4 per core and 1MB of LLC space per core, FlatPack increases system performance compared to current state-of-the-art designs by 36%, while reducing system energy consumption by 12%.

Acknowledgement

The work underlying this thesis would not have been possible without the kind support of my colleagues and friends.

I am very grateful to my advisor Yiannis, for his firm guidance and advice during the ups as well as motivation during the downs. My co-advisor Angelos, whose expertise and humility have been both enlightening and crucial in reaching the finish line. My former co-advisor Pedro, thank you for offering balance and lightening the mood. Thanks to my former co-advisor Sally A. McKee, for your invaluable advice on writing, presentation and teaching.

I owe a great deal to my fellow Ph.D. students at CSE. Evangelos taught me everything I know about simulation, and laid the cornerstones of the infrastructure upon which this work is based. His dark sense of humor helps the rest of us feel more well-adjusted. Thank you Ahsen and Alirad, for all your help with the struggle of hardware design. Prajith and Stavros, who have been my guides to the business of being a PhD student. My neighbor Petros, for comedic relief and encouragement. Stefano, who reminds us all that there is also a life outside the 4th floor. Neethu and Panagiotis, who will bravely carry the torch onward.

Miquel Pericas and Lars Norén, for their generous help in my eternal quest for processing power. This really wouldn't have been possible without you.

Rolf Snedsböl, the joy of the office. If you are ever allowed to retire, we will all miss you. The CSE administrative staff who let people like me focus on the things we understand, especially Monica Månhammar who deserves more credit than three regular people.

My good friends from the Luleå Academic Computer Society (LUDD) and the ASCII Initiative, for their invaluable friendship, distraction and relief. Work like this cannot be completed without an outlet to laugh about it.

Finally, Lea, my loving partner in crime. Our struggles are one and the same, and this would have been infinitely more difficult without you. I cannot wait to see what comes next.

Albin Eldstål-Ahrens
Göteborg, February 2022

This work is supported by the Swedish Research Council (contract number 2014-6221) under the ACE project.

List of Publications

This thesis is based on the following publications:

I **Albin Eldstål-Damlin**, Pedro Trancoso and Ioannis Sourdis
"AVR: Reducing Memory Traffic with Approximate Value Reconstruction"
Proceedings of the 48th International Conference on Parallel Processing (ICPP). 2019.

II **Albin Eldstål-Ahrens** and Ioannis Sourdis
"MemSZ: Squeezing Memory Traffic with Lossy Compression"
ACM Trans. Archit. Code Optim (TACO). 17, 4, Article 40, 2020.

III **Albin Eldstål-Ahrens**, Angelos Arelakis and Ioannis Sourdis
"L^2C: Combining Lossy and Lossless Compression on Memory and I/O"
ACM Trans. Embed. Comput. Syst (TECS). 21, 1, Article 12, 2022.

IV **Albin Eldstål-Ahrens**, Angelos Arelakis and Ioannis Sourdis
"FlatPack: Flexible Compaction of Compressed Memory"
Submitted.

Contents

Abstract iii

Acknowledgement v

List of Publications vii

1 Introduction 1
- 1.1 Problem Statement . . . 2
- 1.2 Thesis Objectives and Contributions . . . 4
  - 1.2.1 Improve Memory Bandwidth Utilization . . . 4
    - 1.2.1.1 Related Work . . . 5
    - 1.2.1.2 Thesis Contributions . . . 5
  - 1.2.2 Improve Memory Capacity Utilization . . . 7
    - 1.2.2.1 Related Work . . . 7
    - 1.2.2.2 Thesis Contributions . . . 7
- 1.3 Thesis Outline . . . 8

2 AVR: Reducing Memory Traffic with Approximate Value Reconstruction 11
- 2.1 Related Work . . . 12
- 2.2 System Architecture . . . 14
  - 2.2.1 Memory Blocks . . . 15
  - 2.2.2 Metadata Table . . . 16
  - 2.2.3 Summarizing & Reconstruction . . . 16
  - 2.2.4 Last Level Cache . . . 19
  - 2.2.5 Memory Operations . . . 22
- 2.3 Evaluation . . . 24
  - 2.3.1 Experimental Setup . . . 24
  - 2.3.2 Hardware Overhead . . . 26
  - 2.3.3 Experimental Results . . . 26
- 2.4 Conclusion . . . 31

3 MemSZ: Squeezing Memory Traffic with Lossy Compression 33
- 3.1 Background . . . 34
  - 3.1.1 SZ Compression . . . 34
- 3.2 System Architecture . . . 35
  - 3.2.1 MemSZ Parallel Lossy Compressor . . . 37
  - 3.2.2 Error Limiter . . . 41

3.2.2.1 Metadata Table . . . 41
3.2.3 Last Level Cache . . . 42
3.2.3.1 On-Chip SRAM LLC . . . 42
3.2.3.2 3D-stacked DRAM LLC . . . 43
3.3 Evaluation . . . 44
3.3.1 Experimental Setup . . . 44
3.3.2 Hardware Overhead . . . 47
3.3.3 Experimental Results . . . 47
3.3.3.1 MemSZ with SRAM LLC . . . 47
3.3.3.2 Evaluation of individual MemSZ features . . . 50
3.3.3.3 MemSZ-DC evaluation . . . 52
3.4 Conclusion . . . 54

4 L²C: Combining Lossy and Lossless Compression on Memory and I/O 55
4.1 Related Work . . . 57
4.1.1 Memory Compression . . . 57
4.1.2 Link Compression . . . 58
4.1.3 Approximate Computing . . . 59
4.2 Background . . . 60
4.2.1 Statistical Cache Compression . . . 60
4.3 System Architecture . . . 60
4.3.1 Compression Methods . . . 62
4.3.1.1 Lossy Compression . . . 62
4.3.1.2 Lossless Compression . . . 64
4.3.2 Block Types . . . 65
4.3.3 Memory Layout . . . 66
4.3.4 Block Type Transition . . . 67
4.3.5 Block Metadata . . . 68
4.3.5.1 Metadata during transitions between block types 70
4.3.6 Last-Level Cache . . . 70
4.3.7 I/O Compression . . . 72
4.4 Evaluation . . . 73
4.4.1 Experimental Setup . . . 74
4.4.1.1 Memory Compression . . . 74
4.4.1.2 I/O Compression . . . 77
4.4.2 Results . . . 77
4.4.2.1 Memory Compression . . . 77
4.4.2.2 I/O Compression . . . 79
4.5 Conclusion . . . 81

5 FlatPack: Flexible Compaction of Compressed Memory 83
5.1 Background and Related Work . . . 85
5.1.1 Compression Algorithm . . . 85
5.1.2 Compression Granularity . . . 85
5.1.3 Block Compaction . . . 86
5.1.4 Address Translation and Page Compaction . . . 87
5.1.5 Metadata Handling . . . 87
5.1.6 Last-Level Cache Support . . . 88

5.1.7 Overheads of Existing Systems 88
5.2 System Architecture 90
5.2.1 Compression 91
5.2.2 Last-Level Cache 91
5.2.3 Lazy Evictions 92
5.2.4 Block Compaction 92
5.2.5 Minislots 93
5.2.6 Slot Assignment 94
5.2.7 Page Compaction 94
5.2.8 Interaction with the OS 95
5.2.9 Metadata 96
5.2.10 Block Migration 97
5.2.11 Page Migration 97
5.2.12 Memory Interleaving 97
5.3 Support for Lossy Compression 98
5.3.1 MemSZ Compression 99
5.3.2 Block Size 99
5.3.3 Block Placement 99
5.3.4 Block Transitions 100
5.4 Evaluation 100
5.4.1 Experimental Setup 100
5.4.2 Single-Core Experimental Results 102
5.4.3 Multi-Core Experimental Results 106
5.4.4 Latency Impact 109
5.4.5 Page Size Estimation 110
5.4.6 Sensitivity to System Configuration 111
5.4.7 FlatPack with Lossy Compression 111
5.5 Conclusion 113

6 Conclusion **115**
6.1 Summary 115
6.2 Contributions 116
6.3 Future Work 117

Bibliography **119**

Chapter 1

Introduction

Main memory is a critical resource in modern systems. Its capacity must be sufficient to avoid frequent page faults and its bandwidth high enough to accommodate the rates of requested data. The demand for both memory capacity and memory bandwidth is increasing as applications become more data-intensive and a larger number of cores is integrated on a single chip. However, simply scaling up memory size and bandwidth increases system cost and power consumption [1].

Memory bandwidth is a primary bottleneck in memory-intensive applications. Access latency increases as the main memory bus becomes saturated, causing a reduction in performance. Memory bandwidth is pin limited [2,3] and power constrained [4] and is therefore more difficult to scale than processing elements [5]. More expensive, 3D-stacked DRAM technologies alleviate the bandwidth problem, but due to power constraints cannot keep up with the increasing demand on data rates either [4].

Insufficient memory capacity is detrimental to system performance, as it leads to costly *page faults*. Page faults introduce a high latency [6], as well as additional traffic on the memory bus to swap data between main memory and persistent storage. Memory capacity is directly limited by cost and indirectly by energy consumption.

One way to increase the efficiency of the memory system is to reduce the volume of transferred data using compression. Data can then be transferred between the main memory and the processor in compressed form consuming less bandwidth and reducing energy cost. With a few exceptions, current hardware main memory compression is limited to lossless methods. Commercial examples of architectures that use memory compression are graphics processing units (GPUs) [7]. GPUs use application-specific compression, applied to texture and color data [8], and often solve the problem only for read-only data [9].

Memory capacity, on the other hand, is not improved by compression alone. In order for the physical memory footprint of compressed data to be reduced, *compaction* must be applied. Memory compaction is the reorganization of compressed data in memory, with the aim of minimizing unused space between compressed blocks. The benefit of compaction is a more efficient use of the available physical memory, which reduces the incidence of *page faults* and costly swapping to nonvolatile storage such as SSDs. Several approaches to memory

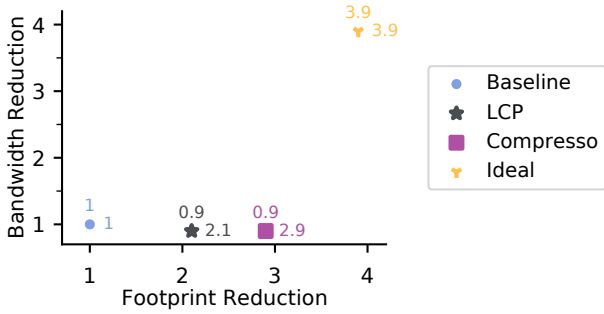


Figure 1.1: Trade-off between Memory Footprint Reduction (horizontal) and Memory Bandwidth Reduction (vertical) for state-of-the-art memory compaction systems. *Ideal* shows the achievable compression ratio of the data.

compaction have been proposed [10–17], with varying overhead on bandwidth.

Existing solutions for memory compression apply lossless low-latency compression algorithms, and achieve compression ratios of between 2× and 4× on unstructured data [18]. As illustrated in Figure 1.1, state-of-the-art memory compaction techniques can translate this into a capacity increase of 2× to 3×, but introduce traffic overheads which cancel out the bandwidth benefits of compression. As a result, no existing system capitalizes on the benefits of compression for both bandwidth and capacity simultaneously.

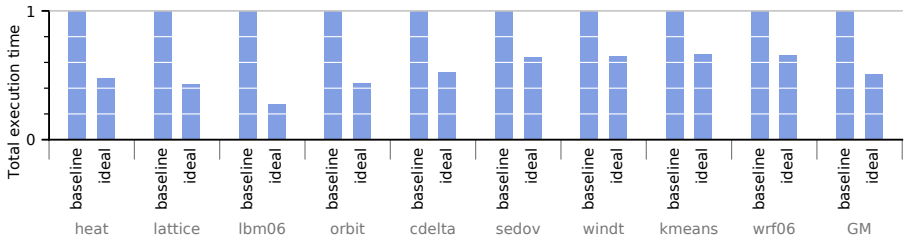
Some classes of applications, e.g., multimedia, scientific, forecasting, may allow for more aggressive compression as they tolerate approximations in parts of their data [19, 20] without introducing significant output error. This thesis proposes lossy memory compression as a means to increase compression ratio and thus reap greater benefits. Carefully selected portions of in-memory data are marked as approximable. Approximable data are divided into blocks and compressed before being written to off-chip memory, reducing the traffic on the bus. This is combined with lossless compression, which offers lower compression ratios but is safely applicable to all data. This hybrid compression is further combined with *memory compaction* to improve both bandwidth and capacity.

The remaining sections are organized as follows. Section 1.1 describes the problem statement underlying this thesis. Section 1.2 formulates its objectives, along with related work and thesis contributions. Section 1.3 contains an outline of the remainder of the thesis, which is divided into five chapters.

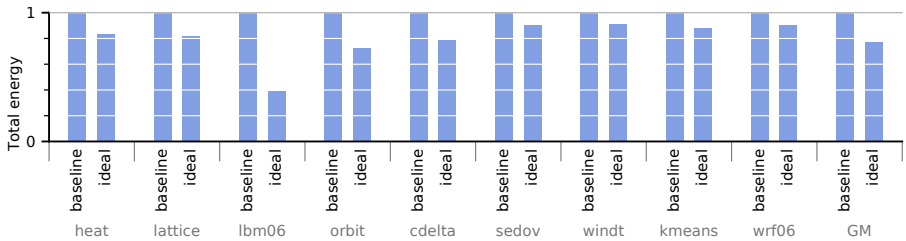
1.1 Problem Statement

Limited memory bandwidth and capacity lead to reduced system performance and increased energy consumption.

The primary drawback of limited memory bandwidth is increased latency of memory operations. Off-chip memory has a high latency compared to on-chip caches, and it increases further when the bus is saturated. Our experiments (Figure 1.2a) show that workloads with high memory intensity can increase

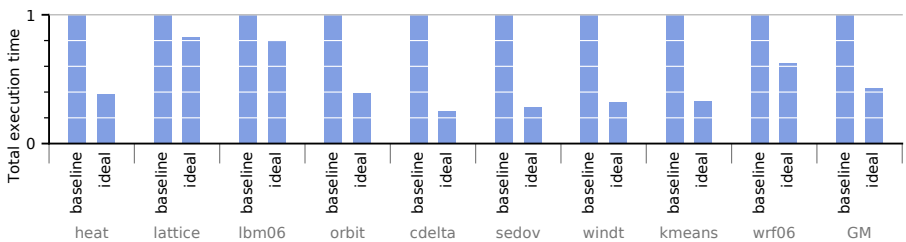


(a) Execution time, normalized to baseline.

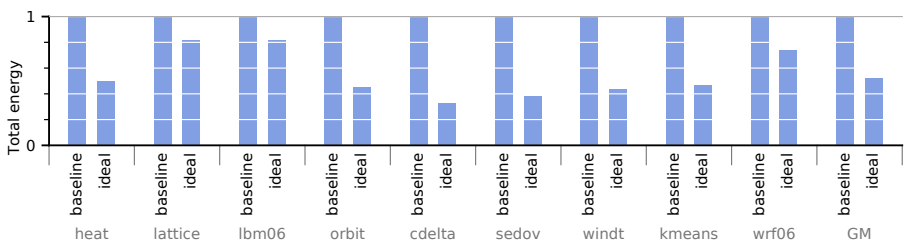


(b) Total system energy, normalized to baseline.

Figure 1.2: Comparison between a baseline system and an ideal system with unlimited memory bandwidth, for a range of applications. Both systems have 8 processor cores at 3.2GHz, an 8MB LLC and 8GB of dual-channel DDR1600.



(a) Execution time, normalized to baseline.



(b) Total system energy, normalized to baseline.

Figure 1.3: Comparison between a baseline system and an ideal system with unlimited memory capacity, for a range of applications.

their performance by an average of 59% if the memory bandwidth bottleneck is eliminated.

Memory capacity is critical to performance, since exhaustion of physical memory introduces *page faults* [6]. Page faults incur long latencies due to operating system overhead and access to nonvolatile storage. On the other hand, increased memory capacity leads to greater energy consumption in the form of leakage and DRAM refresh energy. Figure 1.3a illustrates the 57% average performance gain possible, if page faults are eliminated.

As a direct consequence of extended execution time, energy consumption increases. When processing elements are left idle waiting for memory operations, static power leakage leads to increased overall system consumption. Furthermore, the main memory itself has been estimated to account for roughly 45% of energy consumption in high-end server systems [21]. A large portion of DRAM energy consumption is leakage and refresh energy, which is also proportional to both memory capacity and execution time. In our experiments (Figures 1.2b and 1.3b), total system energy is reduced by an average of 22% when the bandwidth bottleneck is removed, and 48% when page faults are eliminated.

1.2 Thesis Objectives and Contributions

Improve memory bandwidth and capacity utilization using a combination of lossy compression, lossless compression, and memory compaction.

The objective of this thesis is twofold. First, to reduce traffic between processor and main memory. Second, to increase the effective capacity of main memory. By reducing traffic, applications can better utilize the limited bandwidth available. For memory-intensive applications, this will lead to reduced memory latency and improved system performance. Similarly, increased memory capacity reduces the number of *page faults* and thus increases system performance. Improved performance, by either mechanism, is expected to lead to energy efficiency benefits. This section outlines the specific challenges met, as well as the current state of the art.

1.2.1 Improve Memory Bandwidth Utilization

This thesis proposes a range of techniques for memory compression, i.e. compressing the data transferred between the processor and main memory. The primary technique investigated is *lossy memory compression*. By applying this lossy compression to large blocks, more aggressive compression ratios can be achieved compared to existing (lossless) memory compression systems. As a result, memory bandwidth can be utilized more efficiently.

1.2.1.1 Related Work

Several lossless memory compression techniques have been proposed aimed at improving memory bandwidth utilization. Various compression algorithms are used, such as dictionary-based [22], exploiting frequent patterns or zero-value blocks [23], similarities of words at the same bit position [18] or using a hybrid scheme of different lossless algorithms applied to different data [24]. However, lossless solutions have limited compression ratio between 2:1 and 4:1.

Managing the metadata needed for locating and handling the compressed data is also challenging as it may add considerable memory bandwidth overheads [25]. One solution to this is an on-chip cache for metadata [26]. Another way to reduce the metadata cost is to embed metadata in the compressed block [25,27].

The memory compression schemes listed above are lossless. Lossy compression is an example of *Approximate Computing*. Large classes of applications are inherently tolerant to approximations [19]. This enables a trade-off between the quality of their results and their performance and energy efficiency. This trade-off is exploited by various approximate computing techniques, some of them targeting the aforementioned memory bottlenecks in a lossy manner. Until recently, lossy compression has been limited to application specific compression, i.e., in GPUs [8], or truncating bits of individual values [9,28–30], which offered limited compression ratio (2:1 to 4:1).

Approximate load value prediction techniques reduce memory latency by providing a predicted value substantially faster than fetching the actual one from memory [31–33]. They may further improve memory bandwidth utilization by skipping some fetches. Value prediction techniques speculate that the values loaded by the same instruction may be identical or differ by a stride.

1.2.1.2 Thesis Contributions

Lossy Memory Compression One core objective of this thesis is to use *lossy compression* as a means to improve the performance of the memory system. Lossy compression offers greater compression ratios and thus increased benefits, but introduces additional challenges. *Approximate Value Reconstruction* (AVR), presented in Chapter 2, uses a low-complexity *downsampling* compression method. Individual values which are not compressed with adequate precision are stored alongside the compressed block, allowing for a variable compression ratio within acceptable precision bounds. Chapter 3 introduces *MemSZ* which implements an optimized variation of the more advanced Squeeze (SZ) compression algorithm [34]. An additional layer of lossless re-encoding is applied on top of the compression in order to maximize the achieved compression ratio.

Large Compression Blocks Compression ratio is directly related to the granularity of compression, the *block size*. In order to target a compression ratio of up to 16×, a block size larger than the typical single cache line is required. Operating on such large blocks requires adaptations in the memory system. AVR adapts a Decoupled Sectored Cache [35] design to create an LLC capable of storing both compressed blocks and uncompressed cache lines. This allows full compressed blocks of multiple cache lines to be read from memory once and serve several subsequent cache misses. In addition, a technique is

introduced to reduce the traffic overhead of cache evictions to compressed memory. MemSZ introduces an optimized compressed block format, which is possible to start decompressing as soon as the first memory transfer completes.

Error Limiting Mechanism Lossy compression deliberately allows for the introduction of inaccuracies in processed data. Different applications have different tolerance to such inaccuracy, also depending on which data are approximated. If applied improperly, small approximations in input values can have disproportionately large effects in application output. AVR employs two user-defined error thresholds, one for individual values and one for complete blocks. Any value or block which exceeds the corresponding error threshold is kept either at half precision or uncompressed, resulting in a variable compression ratio between 1× and 16×. MemSZ extends this mechanism with an additional error threshold, which limits the total accumulated error across a block throughout the application's lifetime. Blocks exceeding this threshold have compression permanently disabled, to reduce their impact on the application's output quality.

Low-latency decompression In a system with memory compression, decompression is on the critical path for memory reads and thus of crucial importance to system performance. The compression algorithm chosen must therefore offer low decompression latency. The downsampling compression used in AVR has a decompression latency of 12 cycles. MemSZ introduces a heavily parallelized implementation of the SZ compression algorithm, which is able to decompress a full block in at most 18 cycles. In comparison, a single main memory access to one cache line takes between 100 and 400 cycles and an LLC access takes 10-20 cycles. As a result, the total latency of reading out a compressed block and decompressing it is roughly 2× that of a regular LLC hit, and still far lower than that of an LLC miss.

Combined Lossy/Lossless Memory Compression Lossy compression is only applicable to a subset of application data, that which is approximation tolerant. Furthermore, approximable data can only be compressed lossily as long as the error limit can be respected. This limitation means that some data may be left entirely uncompressed, even though it exhibits some redundancy and thus can benefit from compression. Chapter 4 introduces L^2C, which covers this gap by devising a hybrid approach, where lossless compression is used as a fallback when lossy compression is infeasible. Non-approximable application data is compressed this way, improving the overall effect of memory compression. In addition, approximable blocks which have exceeded their error threshold may still exhibit some redundancy in their data. It is therefore beneficial to compress them losslessly rather than leave them entirely uncompressed.

Due to different requirements and characteristics, lossless compression methods are typically most suited to smaller block sizes. For this reason, combining lossless compression with large-block lossy compression introduces additional challenges. Metadata is needed to manage blocks of both sizes simultaneously, and the on-chip LLC needs support to store and access compressed blocks of multiple granularities. L^2C uses a block size of 256B for lossless compression, and adds support to the on-chip LLC to co-locate these smaller blocks with the

1kB blocks of MemSZ. In addition, the page metadata is extended to support any combination of lossy and lossless blocks as well as dynamic changes.

1.2.2 Improve Memory Capacity Utilization

In order for data compression to improve effective memory capacity, compressed data must be *compacted* in physical memory. This complicates address translation, as pages in memory are no longer of a fixed size or alignment. Furthermore, current state-of-the-art memory compaction solutions target memory capacity at the expense of memory traffic [13, 16]. The cause of this trade-off is data varying in compressibility over time, as well as non-uniform compressibility within each compressed page.

1.2.2.1 Related Work

A range of different techniques have been proposed for compacted organization of compressed data. Some designs remove the connection of pages entirely, storing sub-page blocks freely in main memory [10, 11]. Most maintain the relationship between blocks of the same virtual page, either packed within a single contiguous physical region [12, 13, 17] or dynamically adding disjoint regions on demand [14–16].

The current state of the art systems for memory compaction, LCP [13] and Compresso [16], both compress small (64B) blocks and assign them a suitable space within the physical allocation of the corresponding page. By packing compressed blocks when they are first compressed, however, the designs forego the ability to handle blocks changing size over time. A growing block exceeds its assigned space and must be stored elsewhere. A shrinking block leaves unused capacity within the assigned space, since it cannot be adjusted without additional data movement.

1.2.2.2 Thesis Contributions

Improve both Capacity and Bandwidth Compression of larger blocks opens up a novel approach to memory compaction. Since each block is compressed to a size which is a multiple of the Memory Access Granularity (MAG), compressed blocks can be fragmented and organized freely in physical memory. This allows blocks within a page to share the physical space allocated to that page, while remaining flexible to the varying compressibility of each block over time. Crucially, blocks can be reorganized as needed, with minimal traffic overhead. As a result, memory compaction can be maintained over time, responsive to varying compressibility of individual blocks as well as to uneven compressibility between adjacent blocks. An additional benefit is a reduced need for *page migrations*, when a compressed page changes size. To make use of this potential, Chapter 5 introduces *FlatPack*, a memory compaction scheme which combines with the compression system of L^2C to improve both memory capacity and memory bandwidth.

Combined Lossy/Lossless Compression As described above, lossy compression and lossless compression typically have differing characterstics. As a result, their most suitable block sizes also differ. This poses a challenge to

memory compaction, since any given page may consist of a variable number of compressed blocks. In addition, page metadata is complicated by the need for two separate block format and the need to support mixed block types within any compressed page. FlatPack supports the same hybrid compression as L^2C, and allows mixtures of blocks to be placed dynamically in physical memory. This grants the benefits of memory compaction to the hybrid compression system, yielding simultaneous improvements in both bandwidth and capacity.

1.3 Thesis Outline

The remainder of this thesis is organized as follows. Chapters 2, 3, 4, and 5 describe the four compression systems which make up the bulk of the contributions. Finally, Chapter 6 draws conclusions and discusses the impact of these findings.

Approximate Value Reconstruction: Chapter 2 introduces Approximate Value Reconstruction (*AVR*), which uses *downsampling* and *interpolation* for compression. A hardware compressor is introduced between the on-chip Last Level Cache (LLC) and the main memory controller. The on-chip SRAM LLC is adapted to store compressed data read from memory, facilitating reuse and increasing the effective capacity of the cache. The error introduced by lossy compression is limited by two discrete thresholds, allowing the user to control precision-performance trade-off.

Memory Squeeze: Chapter 3 introduces Memory Squeeze (*MemSZ*), which improves upon the preceding design in three ways. First, it includes the more advanced, more effective *SZ* compressor. Second, it adds another layer of error limiting. The accumulated error introduced by compression is monitored, giving the system the ability to detect highly error-sensitive data blocks and disable compression for them. Third, the LLC replacement policy is altered to reduce redundancy between compressed and uncompressed versions of the same data, increasing the storage efficiency.

Furthermore, MemSZ introduces an alternate memory compression architecture which utilizes 3D-stacked DRAM as an LLC, storing only uncompressed data. Approximable data is thus only compressed in main memory. The larger line size supported by a DRAM cache counteracts several of the challenges faced by AVR, yielding a less complex system while still providing the benefits of reduced off-chip traffic.

L^2C: Chapter 4 details the design of L^2C, a hybrid lossless and lossy memory compression system. It improves upon MemSZ by adding the capability of lossless compression, which has two main benefits. First, it allows non-approximable data to be compressed. Second, it allows for a low-compression fallback when lossy compression fails due to quality constraints. As a result, the overall compression ratio of the system is increased and fewer pages are left uncompressed. In addition to this, L^2C is applicable as a *link compression* system, for devices with tight I/O constraints.

FlatPack: Chapter 5 discusses FlatPack, a novel approach to *memory compaction*. Compaction of compressed data in main memory allows the system to increase its effective memory capacity. This, in turn, reduces the incidence of *page faults*, where data is swapped between main memory and the slower persistent storage. A compression system with memory compaction support can achieve this benefit without the added cost and energy consumption of additional physical memory. FlatPack uses L^2C as its basis for compression, and adds a dynamic compaction method. Compared to existing compaction systems, FlatPack is better able to handle varying compressibility over time and non-uniform compressibility within each memory page.

Chapter 2

AVR: Reducing Memory Traffic with Approximate Value Reconstruction

The performance of computer systems is largely dominated by their memory hierarchy as the gap between computing speed and data transfer speed keeps increasing [36]. Besides the long memory latency, memory bandwidth severely limits performance, energy efficiency and scalability of Chip Multiprocessors (CMPs) [5]. On one hand, the demand for higher memory bandwidth increases. Adding more cores on a chip and using specialized accelerators increases the potential processing throughput and calls for higher data rates. New emerging data-intensive applications further increase the need for large volumes of data to be transferred fast [2, 37, 38]. On the other hand, memory bandwidth is pin limited [2, 3] and power constrained [4] and is therefore more difficult to scale [5]. More expensive, 3D-stacked DRAM technologies alleviate the bandwidth problem, but due to power constraints cannot keep up with the increasing demand on data rates either [4].

One way to alleviate the memory bandwidth pressure is to reduce the volume of transferred data using compression. Data can then be transferred between the main memory and the processor chip in a compressed form consuming less bandwidth and reducing energy cost. With a few exceptions, hardware main memory compression is limited to lossless methods. Commercial examples of architectures that use memory compression are graphics processing units (GPUs) [7]. GPUs use application-specific compression, applied to texture and color data [8], and often solve the easy part of the problem, handling read-only data [9]. Current state-of-the-art, lossless memory compression techniques achieve on average a 2:1 to 4:1 compression ratio on unstructured data [18]. However, some classes of applications, i.e., commercial, multimedia, scientific, may allow for more aggressive compression as they inherently tolerate approximations in parts of their data [19, 20] without introducing significant error.

In the past, the performance of memory subsystems has been improved for approximation-tolerant applications. Load value prediction without fetching

the actual requested data has been used for improving memory latency and bandwidth [31–33], but has difficulties capturing irregular data variations. Approximate deduplication of individual cachelines increases cache capacity [39], however, multiple values need to match at cacheline granularity. A form of lossy compression has been applied in approximate computing, but is constrained to reducing precision of single values truncating their least significant bits [9,28–30] and therefore achieves limited compression ratio.

In this chapter, Approximate Value Reconstruction (AVR) is proposed for reducing data volumes transferred between processor chip and main memory in approximation tolerant applications, utilizing more efficiently the memory bandwidth. AVR goes beyond reducing the precision of individual values and compresses data in a lossy manner exploiting similarities between values while capturing their variance. In essence, AVR stores a "summary" of approximated data in memory, based on which values are approximately reconstructed in the processor chip. AVR addresses a number of challenges. Summarizing (compressing) and reconstructing (decompressing) the data needs to be generic, introduce low error, and add minimum latency and energy overheads. Moreover, managing (updating, re-packing, storing) compressed data needs to be efficient and impose low traffic overheads. In effect, AVR utilizes memory bandwidth better improving performance and energy efficiency.

AVR makes the following contributions:

- Aggressive, approximate memory compression exploiting similarities across values and reduces memory traffic improving execution time and energy efficiency.
- Improved effectiveness and minimized overheads of aggressive compression using the following techniques:
  - co-locating compressed memory blocks and uncompressed cachelines in the Last Level Cache (LLC);
  - handling LLC eviction in a lazy manner;
  - keeping track of badly compressing memory blocks;
  - selecting which data to store in LLC after decompression.

The remainder of this chapter is organized as follows. Section 2.1 discusses related work on lossless memory and cache compression as well as on approximate computing with focus on memory systems. Section 2.2 describes the proposed AVR architecture. Section 2.3 presents our evaluation results and Section 2.4 draws our conclusions.

2.1 Related Work

Prior work on related topics is discussed next. First, existing designs for lossless memory and cache compression are presented and subsequently an overview is provided on approximate computing techniques that improve the performance of memory systems.

Lossless Memory Compression: There is a plethora of memory compression techniques that improve memory capacity and bandwidth utilization.

Various compression algorithms are used, such as dictionary-based [22], exploiting frequent patterns or zero-value blocks [23], and more recently similarities of words at the same bit position [18]. However, applied to unstructured data, lossless solutions have limited compression ratio between 2:1 and 4:1, which is substantially lower than in AVR (4-8×). In general, lossless compression is orthogonal to AVR as it can be used in our design to compress data that are not approximated, or even on top of AVR approximately compressed data. Another aspect is the data placement in memory. Some approaches compact compressed data in memory to improve capacity [13]. Others, like AVR, avoid data compaction, allocating the worst case storage required for the uncompressed data and focus only on memory bandwidth [26, 27]. Finally, managing the metadata needed for locating and handling the compressed data is also challenging as it may add considerable memory bandwidth overheads [16, 25]. AVR uses a metadata table and a cache of it, as in [13], which is updated with the TLB and adds a few bytes of bandwidth overhead at every TLB miss; still techniques like Attache [25] could be used to further reduce the metadata cost.

Lossless Cache Compression: Lossless compression has been applied to caches, too. Besides the issues of encoding and compaction of variable size blocks [40], the compression and decompression latency constraints are tighter compared to memory compression. In the past, cache compression has been supported in various ways, for instance using value-centric caches [41]. Compacting compressed cache blocks has been tackled using decoupled super-blocks and sub-blocks [42], or super-blocks without decoupling tag and data arrays [43]. In general, cache compression cannot reduce memory traffic as it compresses single cachelines separately, rather than larger memory blocks of consecutive cachelines as performed by AVR. Consequently, as opposed to AVR, cache compression techniques applied to the LLC cannot reduce the number of memory accesses and hence cannot reduce memory traffic. Furthermore, AVR uses the LLC to store compressed memory blocks alongside the uncompressed lines, but does not attempt to compress individual cachelines. As a consequence, cache compression could be considered to compress the AVR LLC contents.

Approximate Computing: Large classes of applications are inherently tolerant to approximations [19]. This enables a tradeoff between the quality of their results and their performance and energy efficiency. This tradeoff is exploited by various approximate computing techniques, some of them targeting the aforementioned memory bottlenecks in a lossy manner.

Approximate load value prediction techniques reduce memory latency by providing a predicted value substantially faster than fetching the actual one from memory [31–33]. They may further improve memory bandwidth utilization by not always bringing the actual values at all. Value prediction techniques speculate that the values loaded by the same instruction may be identical or differ by a stride. However, this does not capture any irregular variance of data such as the variance in an image where neighboring pixels may have similar values but may not necessarily differ by a fixed stride. Approximate load value prediction is applied near the core (in parallel to the L1 cache) and is therefore orthogonal to the proposed AVR compression of memory traffic. Another fundamental difference compared to AVR and in general compared to compression is that load value prediction techniques aim primarily at reducing load latency rather than memory bandwidth because in the end they do fetch

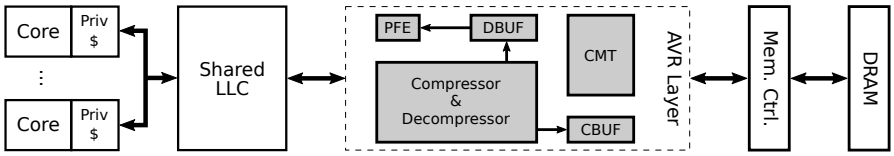


Figure 2.1: Toplevel block diagram of the AVR architecture.

the precise values from memory for error checking.

Reducing the precision of floating point [9, 28, 29] and fixed point [30] numbers has been used to alleviate the memory bandwidth bottleneck in deep neural networks [30], GPU workloads [9] and other approximation tolerant applications [28], thereby improving performance and energy efficiency. However, the compression ratio is still limited between 2:1 and 4:1 despite the loss of precision as these approaches do not exploit inter-value similarities to compress data. Closer to AVR, software techniques for lossy compression have been proposed, but have high complexity and latency and as a consequence cannot be used directly in hardware [34].

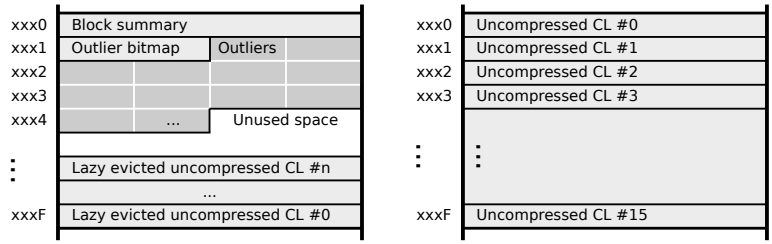
Approximate, lossy compression has been applied to caches, too. Doppelgänger deduplicates similar cachelines to compress data [39]. The subsequent Bunker cache design speculates similarities between cachelines solely based on their addresses without looking at their contents, proposing a less intrusive cache design but achieving lower compression ratio than Doppelgänger [44]. Both designs exploit similarities between cachelines. However, similar values need to have the same offset within their cachelines in order to match, which restricts deduplication opportunities.

2.2 System Architecture

Approximate Value Reconstruction (AVR) reduces the volume of data transferred between main memory and processor chip improving bandwidth utilization and in turn system performance and energy efficiency. Without loss of generality, AVR is applied to a Chip Multiprocessor as depicted in Figure 2.1.

In a nutshell, AVR handles a processor request to approximated data as follows. In case the requested cacheline misses in the LLC, the corresponding memory block, which compresses multiple cachelines including the requested one, is brought on-chip. The requested cacheline is then retrieved, stored in the LLC, and sent to the processor. The memory block is also stored in the LLC as is (compressed) so to avoid memory accesses at future requests to the block.

In general, AVR employs a number of optimizations to reduce its overheads. As mentioned above, compressed memory blocks are stored in the LLC trading LLC capacity for fewer memory accesses. In addition, every time there is an LLC eviction it would be wasteful to update the corresponding compressed block in memory. Instead, for as long as there is available space in the memory block, AVR evicts such cachelines lazily, writing them back to memory uncompressed. Then, when the space is exhausted, the block is compacted embedding all lazily evicted cachelines. Finally, the overheads of unsuccessful compression attempts are minimized by keeping a history of previous compression attempts per block.



(a) Compressed block (b) Uncompressed block

Figure 2.2: AVR Memory Block.

The AVR architecture requires the following additions: a compressor and decompressor module to *summarize* data before sending them to memory and to *reconstruct* data coming back from the memory; a metadata table for storing information about the compressibility of the memory blocks; finally the LLC design requires changes for storing compressed memory blocks in addition to normal cachelines. Next, each one of the above modules is discussed separately, after first presenting the format of the AVR memory blocks. At the end of the section, the AVR LLC and memory operations are discussed.

2.2.1 Memory Blocks

Similar to most techniques that focus on data approximations [28, 39, 45], AVR considers that the programmer annotates memory regions that can be approximated and hence compressed in a lossy manner. This annotation also includes the size of the region as well as the datatype of the approximable data. An additional OS system call allows allocated pages to be marked as approximate at the page table requiring an extra bit for every page table and translation lookaside buffer (TLB) entry as shown in Figure 2.3. The programmer may further indicate an upper error threshold for acceptable approximations. In our experiments, two thresholds are used, one for the relative error of each individual value and one for the average error of all values in a block. Currently, error thresholds are common for all approximations in a program, but they could be easily extended to thresholds per allocated memory region adding a respective field to the page table.

The AVR architecture does not consider improving memory capacity and therefore memory allocation is not affected. Compression is performed at the granularity of memory blocks composed of multiple cachelines as shown in Figure 2.2; a cacheline, i.e., 64B, being the granularity of accessing the main memory. In our implementation, a block is composed of 16 cachelines, in total a quarter of a physical 4KB page. AVR compresses the 16 cachelines of a block to a single cacheline *summary* aiming at a 16:1 compression ratio and at accessing the entire block with one memory request. The summary is stored in the first cacheline of the memory block as shown in Figure 2.2a. In case this compression produces approximations of some values that exceed a particular error threshold, these values are characterized as *outliers* and stored explicitly, uncompressed in the compressed block. The outliers are placed in order after the *summary* cacheline, together with a bitmap that indicates their

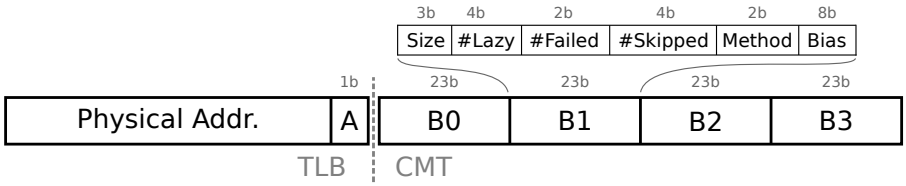


Figure 2.3: Format of a metadata table entry and TLB addition.

location in the uncompressed block (one bit per 32-bit value). This bitmap occupies half a cacheline if the block contains outliers. Summary, bitmap and outliers occupy in total 1-8 out of the 16 cachelines (2:1 worst case compression ratio). The remaining space of the memory block remains available for *lazy evictions*; that is for writing back dirty uncompressed cachelines of the block, when evicted from the LLC. Thereby, AVR avoids bringing a compressed block on-chip to be updated every time a dirty cacheline is evicted. This is possible until the block space is exhausted, then the block and the lazily evicted dirty uncompressed cachelines are fetched from memory for recompaction. In case a memory block fails to be compressed in 8 cachelines or is explicitly marked as not-approximable then it is stored uncompressed as shown in Figure 2.2b.

2.2.2 Metadata Table

Each compressible block requires some metadata information in order to be handled. Similar to previous approaches on memory compression [12, 13], these metadata are stored in main memory and cached on-chip in a TLB-like Compression Metadata Table (CMT) placed and accessed in parallel with the LLC. The CMT is updated in pair with the TLB. A CMT has four 23-bit entries per 4KB page, one per 1KB memory block as shown in Figure 2.3. CMT stores the following information about each memory block: its size (compressed in 1-7 lines or uncompressed), number of lazy evicted cachelines stored, compression method (and datatype of values), and a bias of its values. Finally, it maintains two counters to keep the history of previous compression attempts. The first one counts the number of consecutive failed compression attempts. Then, depending on that count, a number of recompression attempts (in block updates) are skipped to reduce the overhead of badly compressed blocks.

2.2.3 Summarizing & Reconstruction

Summarizing and approximately reconstructing memory blocks requires knowledge of the particular value representation used in the considered dataset. Our current implementation supports standard 32-bit floating-point and fixed point formats, but can be easily extended to support other representations, too. The core part of the compression is using fixed point arithmetic to reduce complexity. Consequently, memory blocks containing floating point numbers are converted to fixed point before compression and back to floating point after decompression. Figure 2.4 shows the block diagram of the AVR compressor and decompressor.

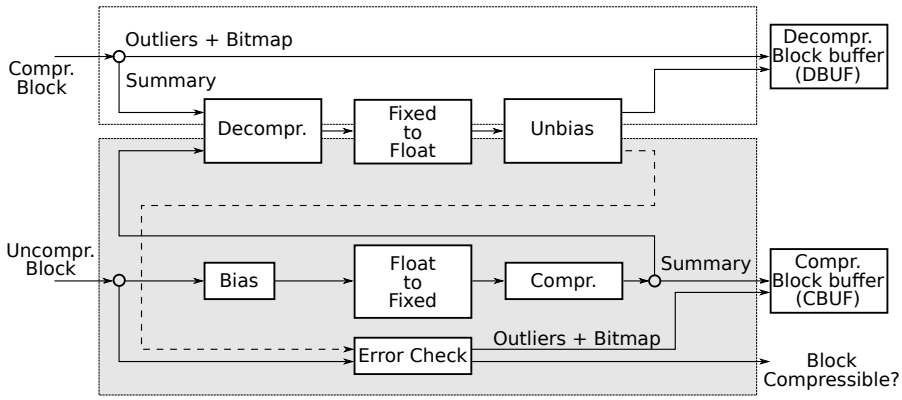


Figure 2.4: AVR compressor/decompressor module.

Incoming uncompressed blocks are fed to the compressor cacheline by cacheline in a pipelined fashion. Fixed point values are compressed directly. Floating point values are converted to fixed point after first having their exponent field biased to minimize loss of accuracy. Subsequently, a simple downsampling compressor is employed to generate the summary of the block replacing multiple (typically 16) uncompressed values with their average. In order to check the error of the approximated values and identify outliers, the compressed block summary is decompressed again, and if necessary converted back to floating point and unbiased. Then, each approximated value can be compared with its respective original uncompressed value stored in the input of the compressor. The result of this comparison identifies the outliers and produces the bitmap of their locations, which is part of the compressed block when outliers exist as shown in Figure 2.2a. This bitmap is also used to select and compact the outliers stored in the block. Thereby, the summary, bitmap and outliers of a block are produced and stored in the compressed block buffer (CBUF). Once compression completes, the metadata of the block are updated in the CMT.

Decompression is simpler. The summary of a compressed block is sent to the decompressor that produces its decompressed version and stores it to the decompressed block buffer (DBUF) after converting it to floating point and unbiasing, when needed. In addition, the outliers are placed according to their bitmap on the buffer replacing the respective decompressed values. The requested decompressed cachelines are then sent to the LLC. The remaining ones are kept in the buffer and future requests for cachelines of the same block are served from there. When the next block arrives for decompression, a prefetcher (PFE) selects a number of decompressed cachelines, not yet stored in the LLC, to be inserted in the LLC before being replaced by the new block under decompression.

Biasing & unbiasing: When dealing with extremely large or small floating-point numbers (large positive or negative exponent), the conversion to fixed-point format can cause a greater loss of precision. To avoid this, blocks are *biased* during compression. A *bias* value is determined, which, when added to the exponent of the values in the block, can bring the block's values into a

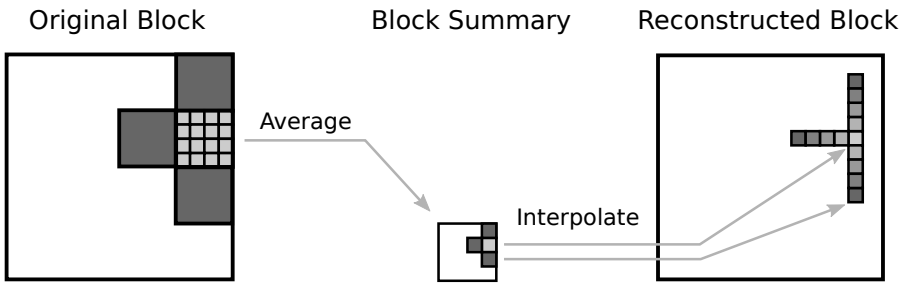


Figure 2.5: Downsampling and Reconstruction of a 2D block.

representable range. Biasing is not performed on blocks where either a) the selected bias would cause special values such as *NaN* or *Inf*, or b) the selected bias would cause over- or underflow of the exponent of any value. The bias is stored with the block's metadata and used during decompression to restore the original range of values. Biasing involves finding the maximum and minimum exponent of the values in a block, determining a suitable offset, and applying it to the exponents of the block. Biasing is pipelined and performed in 4 cycles. The inverse process (unbiasing) requires an 8-bit addition to all decompressed values and requires one cycle.

Float to fixed & fixed to float conversions: Converting from float to fixed point numbers and vice-versa is implemented as described in [46] requiring a single cycle.

Compression: Although various lossy compression algorithms can be considered, we opted for a method that is simple to implement. In AVR, memory blocks are compressed using downsampling [47]. This method entails dividing the block into a suitable number of sub-blocks and computing the average value of each sub-block. We aim for a 16:1 compression ratio and therefore sub-blocks of 16 values are used. In our attempt to find the best compression, a number of variations of the method are used in parallel. The main two variants differ in the considered placement of the values in the block before partitioning to sub-blocks; in particular, the first one considers the block as a square 2D array and the second as a linear 1D linear array. Figure 2.5 shows an example of 2D downsampling, where the compression is performed by averaging the values of a sub-block (light-grey) into a single value. For decompression, the average values are distributed evenly and bi-linear interpolation is applied to reconstruct the approximate values in-between. In our implementation, compression and decompression require 15 and 10 cycles, respectively.

Error calculation & Outliers selection: Lossy compression introduces errors in the approximated values. In order to limit this error, compression is skipped in case the error exceeds a particular threshold value. This is evaluated by comparing the original incoming uncompressed block with the approximately reconstructed block produced after compression and subsequent decompression. Two separate thresholds are used to control the approximation error of the compression operation: the relative error of each individual value may not exceed a percentage threshold T_1 and the average relative error across all values in the block may not be greater than a percentage threshold T_2.

These error thresholds are exposed as a tunable knob and in our experiments $T_1 = 2T_2$. Notice that this is an error mitigation strategy of low overhead and local decision.

The error per individual value is calculated in floating point format as follows[1]. For a value to be approximated with a relative error within T_1, a comparison between the original and approximated value should result in (i) the exact match of their signs and exponents and (ii) the difference of their mantissas not exceeding the Nth most significant bit (MSbit); for an error below $1/2^N$. The above comparisons are performed in a cycle and produce the bitmap of the values that are outliers. Subsequently, this bitmap is used for selecting and compacting the outliers and in parallel computing the average block error for the values that are not outliers. Selecting and compacting the outliers requires 16 cycles, one cycle per uncompressed cacheline. The relative error of each individual non-outlier value is required for computing the average error of the block. The sign and exponent are identical for the original and approximate values, otherwise they would be outliers. So, the average error is calculated by subtracting the mantissa bits of each original and approximated value. The average block error is the average of these subtractions for all the non-outliers values and computing it also fits in 16 cycles.

Prefetching decompressed cachelines: After decompressing a block, the requested cacheline(s) are stored in the LLC. Storing also the remaining cachelines could lead to the pollution of the LLC with unwanted cachelines. Consequently, they remain in DBUF until they are overwritten by another block. In the meantime, if one of these cachelines is requested it is sent directly to the LLC. When a new compressed block arrives for decompression, a prefetching engine (PFE) is consulted to decide whether any of the remaining decompressed cachelines in DBUF should be written in the LLC before they are replaced by the new block. The PFE employs a simple threshold strategy, prefetching all lines from a block where at least half have been explicitly requested.

Total compression and decompression latency: Based on the above and as confirmed by our synthesis results presented in Section 2.3, the total latency for compressing a block is 49 (processor) cycles, and for decompressing a block is 12 cycles. Decompression is more critical for system performance as it affects memory reads. Compression is less critical because it affects the write backs.

2.2.4 Last Level Cache

The AVR Last Level Cache (LLC) stores uncompressed cachelines (UCL) as well as compressed memory blocks. A compressed memory block may occupy one to eight LLC lines (64B), depending on its compressibility, it is therefore split in 64B compressed memory subblocks (CMS). When a memory block enters the processor chip and gets uncompressed, only selected uncompressed cachelines are stored in the LLC. The AVR LLC is decoupled in order to support the management of the LLC contents at two granularities, namely, that of a cacheline (64B) and that of a memory block (16 cachelines). Following the design of the Decoupled Sectored Caches [35], the AVR LLC decouples its tag array from the data array. On one hand, entries of the LLC data array have a

[1]For fixed point numbers a subtraction and a subsequent comparison would be required.

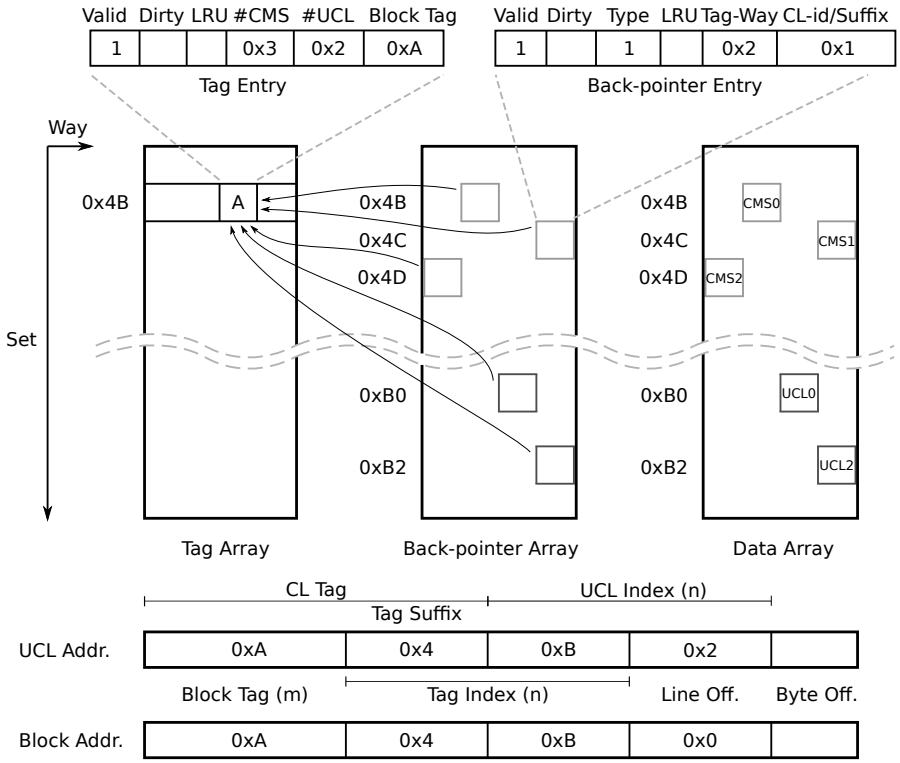


Figure 2.6: AVR Last Level Cache.

cacheline (64B) granularity. On the other hand, the tag array has a granularity of a memory block (16 cachelines). The decoupling of tag and data arrays is facilitated by a back-pointer array (BPA) which supports the indirection between every data array entry and a tag array entry to associate the data of a cacheline with its tag. In essence, each data array entry has a respective BPA entry at the same set and way, which maintains its state-bits and a pointer to its tag in the tag array. In contrast, a tag array entry can be shared among multiple data array entries.

LLC Functionality: Figure 2.6 illustrates the AVR LLC functionality using an example of a memory block with tag *A*. The memory block of this example, when compressed, occupies three cachelines, CMS0, CMS1, and CMS2; one for the summary of the block and two for the bitmap and the outliers. All three cachelines of the compressed block are stored in the LLC. In addition, two of its 16 uncompressed cachelines, UCL0 and UCL2 are also present in the LLC. The breakdown of a memory address is shown in Figure 2.6. After the 6-bits of byte offset, there is the 4-bit cacheline offset in the memory block. Let us consider that the LLC requires n-bits for indexing. Then, the tag array will use as index the n bits of the address after the cacheline offset (*tag index*) and store the remaining m most significant bits of the address as the memory *block tag* because it follows a memory block granularity. For example, the tag *A* for the memory block 0xA4B0 is placed in set 0x4B of the tag array. The same indexing is used for the placement of the compressed

memory subblock. The first CMS of the compressed block, CMS0, is placed in a way of set 0x4B, occupying the respective entry in both the data array and the BPA. The remaining parts of the compressed block, CMS1, and CMS2, are placed in the subsequent sets 0x4C and 0x4D. Uncompressed cachelines use the indexing of a conventional cache (UCL index), in particular, the n bits after the byte offset. For example, uncompressed cacheline UCL2, with address 0xA4B2, is placed in set 0xB2. This LLC design has two advantages. Firstly, each UCL and CMS is mapped to different LLC sets, thereby, not affecting the effective associativity of the cache. Secondly, a single tag entry is required for all of cachelines of a block, making the management of memory blocks simpler.

LLC Structure: Structurally, the AVR LLC, depicted in Figure 2.6, is based on the Decoupled Sectored Caches [35] and shares some common elements with Decoupled Compressed Cache [42]. A tag array entry stores the following fields:

- **Block Tag:** The memory block tag.
- **CMS count:** the number of subblocks/cachelines needed for storing the compressed memory block (3 bits).
- **UCL count:** number of uncompressed cachelines of the block stored in the LLC (4-bits).
- **Block state bits:** valid, dirty & least recently used (LRU).

The dirty bit indicates the compressed memory block is dirty. The tag LRU is updated when a UCL of the block is accessed and used for tag-entries replacement. A BPA entry stores the following:

- **CL-type:** one bit indicating a UCL or CMS.
- **CL-id:** for a UCL, this 4-bit field stores the *cacheline tag suffix* depicted in the address breakdown; for a CMS, 3 of these 4 bits store the CMS *offset in the compressed block*.
- **Tag-way:** the way of the tag array that stores the tag of the respective block.
- **CL state bits:** valid, dirty and LRU bits.

The tag suffix of an UCL is stored in the BPA because during a lookup it needs to match together with a block tag to complete a cacheline tag match. Instead, for the BPA entries that store a CMS, the compressed memory subblock number is serving the same purpose; that is when looking up the i-th subblock (cacheline) of the compressed block the *CL-id* of the matching BPA entry should be i. Finally, the CMS LRU bits are updated when any UCL of the block is accessed.

LLC Lookup & Allocation: A request for an LLC cacheline is served by accessing in parallel the DBUF and the LLC tag array. In case the requested cacheline is in the DBUF it is returned. Otherwise the tag array access will determine whether the cacheline is available in the LLC uncompressed or its compressed memory block is present. In the first case, an UCL lookup is

performed. Otherwise, in the second case a CMS lookup is performed. Figure 2.7 illustrates the AVR LLC lookups. Below we discuss each case in more detail.

A lookup for an uncompressed cacheline is performed as follows. The tag array is accessed using the tag index and in parallel the BPA and data array using the UCL index. The block tags in the set are matched. In parallel, the cacheline tag suffixes (*CL-id*) in the BPA set are matched for the entries in the set storing UCLs. Subsequently, the tag-way stored in each of the matching BPA entries is compared with the way of the matching block tag. There is a hit when a tag suffix matches and its tag-way points to a matching tag. The tag-way stored in the BPA entry must be equal to the way of the matching tag in order to ensure that a matching tag suffix points to its true tag, otherwise the cacheline stored in the BPA entry may have a different tag than the matching one.

A lookup for a compressed block in the LLC requires one or multiple accesses to the LLC, as many as the CMSs the block is composed of (*CMS count*). The tag array, BPA, and data array are accessed with the tag index. The block tags in the set are matched. In parallel, the entries in the BPA set that store CMSs compare their *CL-id* with zero. Here this field indicates the offset of the CMS in the compressed memory block and looking up for the first subblock requires *CL-id* to be zero. Subsequently, the tag-way stored in any of the matching BPA entries is compared with the way of the matching block tag. In this first access to the LLC, besides the first subblock (CMS), the *CMS count* is also retrieved to determine the total number of LLC accesses required for accessing the compressed block. If that number is more than one, the BPA and data array are accessed repeatedly until all parts of the block are read. At each access, *CL-id* needs to match the iteration increment, and tag-way should be the same as the matching tag entry.

When there is no available cacheline entry in the set, allocation for an UCL is performed choosing a victim cacheline based on the LRU bits stored in the BPA set. All cachelines in the set, UCLs and CMSs, compete equally. In case a CMS is evicted, then all the other CMSs of the same compressed block need to be evicted, too, and if dirty written back to memory. The tag entry of the block would remain if the LLC stores UCLs of the block. The absence of the compressed version of the block is indicated by setting to zero the field *CMS count* in its entry in tag array. Allocation for a tag entry is performed by choosing a victim tag in the set based on LRU. The LRU of a block tag is updated when one of its UCLs is accessed or when the block is recompressed. Finally, allocation for the CMSs of a block needs to be performed together at consecutive sets starting from the one indicated by the tag index.

2.2.5 Memory Operations

åWe explain next the AVR memory operations at the LLC and main memory level. More precisely, we explain how a request to the LLC and an LLC eviction are handled. The details of an LLC lookup and allocation are omitted as they were described above.

LLC Requests: A request to the LLC has the following possible outcomes as shown in Figure 2.7:

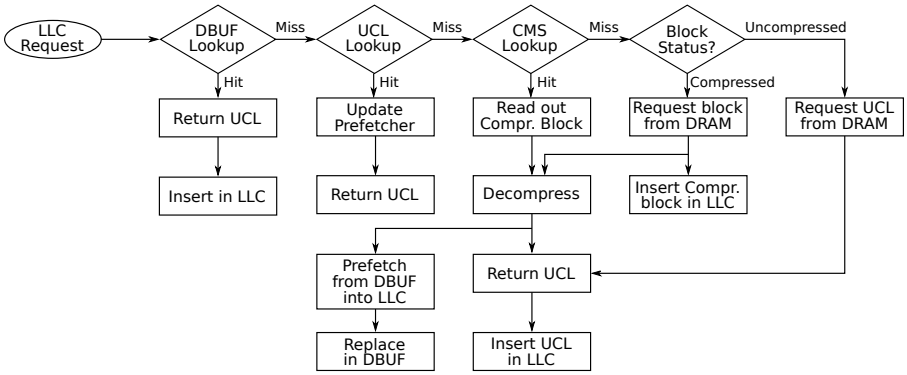


Figure 2.7: AVR LLC requests.

- The requested UCL may hit either in the LLC or in the decompressed buffer (DBUF). In the latter case the UCL is also written from DBUF to the LLC.
- There is a miss of the requested UCL, but a hit to the compressed memory block stored in the LLC. Then, the compressed block is read and decompressed in the AVR compressor block to retrieve the requested cacheline.
- In case both the UCL and the compressed block miss in the LLC, the compressed block containing the requested cacheline is requested from the main memory and upon arrival decompressed to retrieve the requested cacheline. Then, the compressed block is also stored in the LLC.

Note that at a new decompression, the decompressed block previously stored in the DBUF needs to be overwritten. Before overwriting the old block, the prefetcher is consulted to potentially save some of its UCLs, storing them in the LLC.

LLC Evictions: When a cacheline is replaced from LLC, then if clean no further action is required, if dirty, the cacheline is evicted and its type is checked first as shown in Figure 2.8.

In case of a dirty UCL, it is checked whether its compressed memory block is also stored in the LLC. If so, the compressed block is read from the LLC, decompressed, updated with the evicted dirty UCL, compressed again and stored back to the LLC. In case the compressed block is missing from the LLC (or the compression attempt fails), the metadata table is consulted to check whether there is space in the main memory to lazily store the dirty cacheline. If so, the dirty UCL is written back to the memory and the metadata entry is updated to reflect that. Otherwise, the compressed block is read from memory, decompressed, updated with all the dirty cachelines, as well as any lazy evicted lines, then compressed and written back to memory.

When bringing in a compressed block from memory, the metadata table is consulted to determine whether lazy evicted cachelines exist in memory. If so these lazy evicted cachelines are read from memory together with the block, and incorporated into the block after decompression. The block is immediately recompressed, marked dirty and stored in LLC.

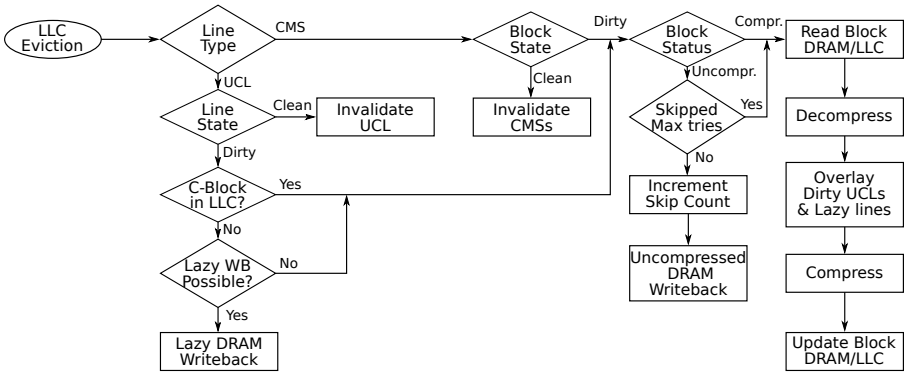


Figure 2.8: AVR LLC evictions.

When evicting a dirty CMS, the entire compressed block needs to be evicted, as partially storing it in the LLC is not useful. The dirty compressed block is first read from LLC and put in the AVR compressor/decompressor to be decompressed. Any dirty UCLs belonging to the block are read from LLC and overlaid on the decompressed block. The memory block is compressed again and written back to memory.

Note that before compressing a memory block that is currently uncompressed, because its last attempt for compression failed, its compression history and counter of skipped compressions is consulted. Based on these fields it is determined whether to proceed with the current compression or not. Accordingly, the above metadata fields are updated in the respective entry. If the recompression is skipped, the dirty UCL is written back to memory directly.

2.3 Evaluation

In this section we evaluate the effectiveness of the AVR architecture. We first describe our experimental setup, presenting the system configuration of our experiments and the benchmarks used. Then, we discuss the hardware overheads of the AVR architecture. Finally, we show our evaluation results and comparison with related designs in terms of performance, energy, and application output error.

2.3.1 Experimental Setup

We evaluated the AVR system using an in-house simulator, implemented on top of Pin [48], that employs an interval-based processor model, as proposed by Genbrugge et al. [49], and a cycle-accurate model of the memory hierarchy that uses DRAMSim2 for modelling main memory [50]. McPAT [51] and CACTI [52] were used to model power and latency of the system considering 32nm technology. The AVR compression hardware modules were implemented in RTL, synthesized using Synopsys to determine their operating frequency, latency and power consumption; this information was then fed to the simulation tool. The parameters of the simulated system are listed in Table 2.1.

Table 2.1: Simulation parameters

Parameter	Configuration
CPU	8 core, out-of-order, 4-way issue/commit @ 3.2GHz
L1 cache	64kB per core, 4-way, 1 cycle latency
L2 cache	256kB per core, 8-way, 8 cycle latency
L3 cache	8MB shared, 2 banks, 16-way, 15 cycle access latency
DRAM	8GB DDR4, 2 channels, 1600MHz

In order to correctly emulate the impact of the approximations in the overall application error, we not only emulate the memory accesses but we actually update the values of the memory contents accordingly by applying the construction and reconstruction methods to the data.

Besides the baseline system, AVR is further compared with (i) itself without marking any data as approximate so as to measure AVR overheads (*ZeroAVR*), (ii) a design that simply compresses approximate values to half-precision by truncating 16 bits similarly to what has been proposed in [9,28,30] (*Truncate*), and finally (ii) Doppelgänger [39], which is the closest and best performing related work on approximate data compression [39] (*Dganger*). As proposed, Doppelgänger is configured to have identical LLC data-array size and a 4× larger tag-array versus AVR, *i.e.* being able to index up to 4× more cachelines. Lossless compression techniques are considered orthogonal and so not included in the comparison; that is because the downsampled values and outliers of an AVR compressed block could be further compressed in a lossless way.

The benchmarks used in this evaluation are selected so each one of them (i) is able to execute until completion and generate an output, and (ii) can

Table 2.2: Benchmark Applications

Application	Approx.	Output	Footprint	Description
heat [53]	Temps	Temps	8.2MB/core	2D Thermodynamics application that iterates over a grid of values and computes the propagation of heat.
lattice [54]	P and M	Vel.+Pr.	5MB/core	2D Lattice-Boltzmann method simulation of air flow over a solid object.
lbm [55]	Velocities	Velocities	325MB/core	3D Lattice-Boltzmann method simulation of fluid flow over a sphere.
orbit [56]	Phys. data	Phys. data	376MB/core	3D simulation of the two-particle orbit problem
kmeans [57]	Topol. [58]	Clusters	5.5MB/core	Clustering algorithm, applied on a geographic elevation map.
bscholes [59]	Options	Prices	6MB/core	Financial forecasting, predicts future stock option prices based on historical parameters.
wrf [55]	Geo data	Temp.	90MB/core	Weather forecasting model.

Table 2.3: Application output error

	heat	lattice	lbm	orbit	kmeans	bscholes	wrf
dganger	0.4%	0.2%	22.3%	>100%	<0.05%	<0.05%	24.9%
truncate	0.2%	0.5%	0.6%	<0.05%	<0.05%	1.4%	4.2%
AVR	0.7%	0.6%	0.1%	<0.05%	1.2%	0.5%	8.9%

tolerate approximations in (parts of) its data. The above restricts us to using the benchmarks listed in Table 2.2; the table further presents the application domain, description of the approximated data-structures and output type as well as their memory footprint. The application code was analyzed to identify approximable data structures. In many cases, a large portion of the application's working set is dynamically allocated. For these cases, a wrapper was created to the *malloc* library call to allocate properly aligned space and register the address range as approximable. The input data sets used for our experiments are the standard input data sets provided with the benchmarks with the exception of (i) *lattice* for which we used a silhouette of a car as the input data set, and (ii) *k-means* where the input is topological data [58]. We use the mean of the relative errors for each output value as our quality metric. Benchmarks for approximate computing (AxBench [59]) considers 10% relative output error, but it is solely up to the application provider do define what is acceptable. Similar to previous works, AVR provides the means to control the data approximation error as a knob to constrain application output error.

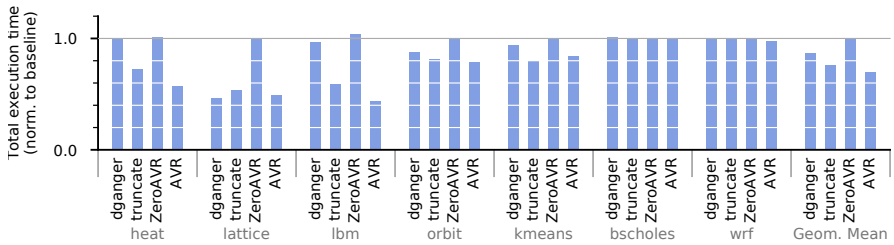
2.3.2 Hardware Overhead

AVR requires some extra hardware resources. The metadata stored in the CMT and the additional bit in the TLB add up to 93 bits per page. Compared to the unmodified TLB, which stores a virtual and a physical page address (52+36=88 bits), this is an overhead of roughly 2×. The AVR Tag array and the BPA add to the baseline set-associative LLC 18 bits per entry; that is in total 144kB and 3.2% overhead to the LLC. Moreover, the AVR compressor module occupies about 200k cells according to our synthesis report.

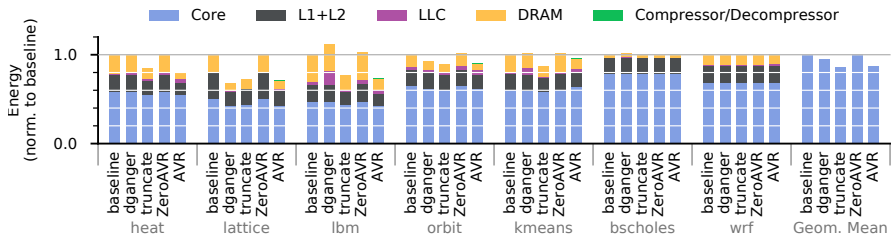
2.3.3 Experimental Results

We present next our experimental results for each benchmark comparing AVR with other related designs, namely, Doppelgänger, Truncate, and ZeroAVR (all results normalized to the baseline). The designs are evaluated in terms of execution time, system energy consumption, DRAM traffic, average memory access time (AMAT), and LLC misses per kilo-instruction (MPKI), as shown in Figures 2.9a, 2.9b, 2.9c, 2.9d and 2.9e as well as in terms of application output error shown in Table 2.3. Table 2.4 shows the AVR compression ratio (for Truncate compression ratio is 2:1) as well as the overall memory footprint versus the baseline. AVR approximate LLC requests and evictions are analyzed and shown in Figures 2.10a and 2.10b.

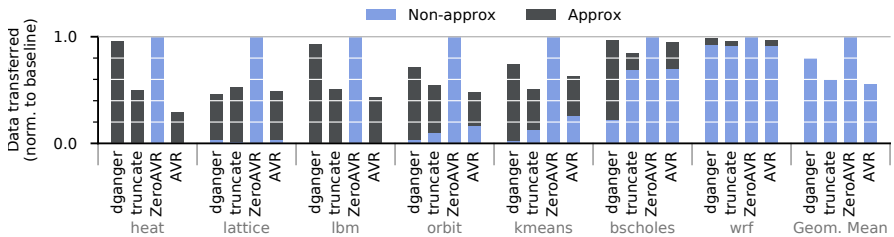
Before presenting the results of each application separately a few common observations are discussed. Analyzing the execution time and energy consumption of ZeroAVR, it is observed that AVR does not add significant overhead



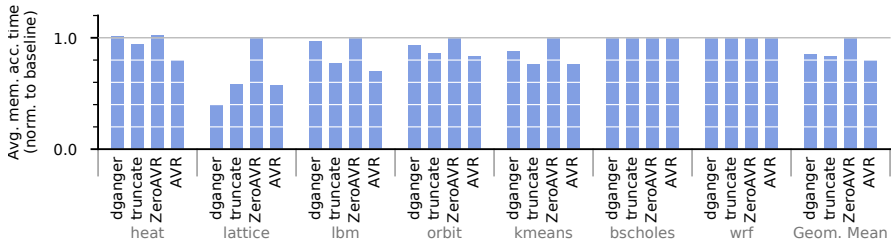
(a) Execution time



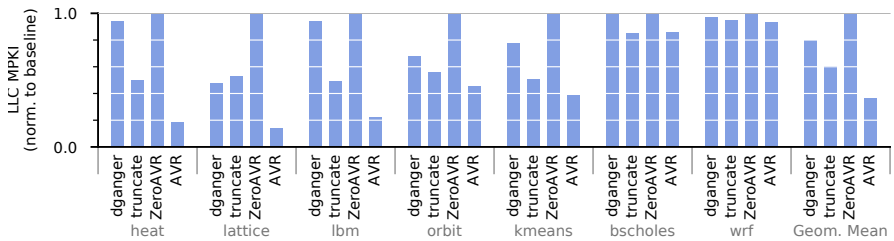
(b) System energy consumption



(c) Memory Traffic

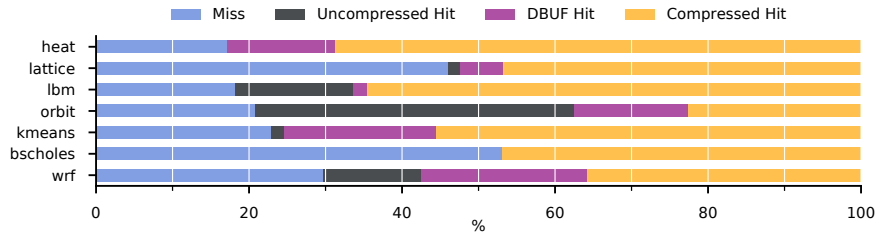


(d) Average Memory Access Time

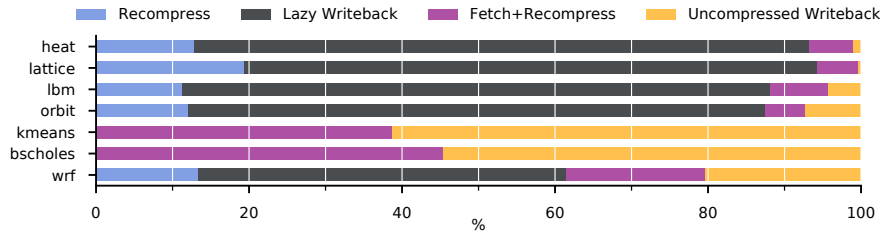


(e) LLC misses per kilo-instruction

Figure 2.9: Evaluation of the AVR design and comparison with competing designs.



(a) LLC Requests on approx. cachelines



(b) LLC Eviction of approx. cachelines[2]

Figure 2.10: Evaluation of the AVR LLC

when it does not approximate data (Figures 2.9a and 2.9b); only in *lbm*, ZeroAVR is 2% slower than baseline, adding similar energy overheads, mainly due to increased DRAM latency caused by changes in the memory access pattern. Moreover, its AVR Decoupled LLC performs similarly to the baseline LLC achieving the same MPKI as shown in Figure 2.9e. In our experiments AVR LLC devotes 2-16% of its capacity to compressed blocks.

Heat dataset exhibits excellent compression; about 8× smaller total memory footprint and a 10:1 compression ratio.AVR reduces execution time by 43% compared to the baseline introducing only 0.7% error. That is almost double the reduction compared to Truncate that has a 0.2% error. Doppelgänger shows no speedup as the data used by heat do not have significant locality and therefore having an "effectively" larger cache does not improve performance. Improvements in execution time lead to AVR and Truncate reduction of baseline energy cost by 18% and 15%, respectively. Furthermore, Doppelgänger introduces an energy overhead of 1% due to its LLC design. AVR reduces memory traffic by 71% compared to baseline. Truncate reaches 50% and Doppelgänger achieves a 4% reduction. AVR reduces memory latency by 20%. Truncate follows with a 5% reduction. This is confirmed by MPKI, where AVR has less than half the misses compared to Truncate as over half of its approximate LLC requests hit in compressed blocks in the LLC or in DBUF.

Lattice dataset is compressed by AVR by a factor of 9.6:1. AVR reduces execution time by 51% introducing 0.6% output error. Doppelgänger reduces

[2] *Recompress*: the evicted cacheline belongs to a compressed block available in LLC, which is updated and recompressed; *Lazy Writebacks*: the cacheline is evicted to memory uncompressed (lazily, without recompression) although it belongs to a compressed block (stored in memory); *Fetch+Recompress*: the compressed block, to which the evicted cacheline belongs, is read from memory and updated; *Uncompressed WB*: the evicted cacheline's block has failed to compress so the line is written back uncompressed.

Table 2.4: AVR compression ratio and footprint reduction

	heat	**lattice**	**lbm**	**orbit**	**kmeans**	**bscholes**	**wrf**
Compr. Ratio	10.5×	9.6×	15.6×	16.0×	2.3×	4.7×	3.4×
Mem. Footprint	12.6%	20.0%	7.9%	54.1%	58.5%	78.6%	89.6%

execution time by 54% with an error of 0.2%. This is because *lattice* can exploit the effectively larger Doppelgänger LLC. Furthermore, Truncate achieves a speedup of 47% with an output error of 0.5%. Energy consumption follows the performance trends. AVR reduces baseline energy by 23%. Doppelgänger and Truncate energy consumption is reduced by 27% and 23% of the baseline, respectively. AVR memory traffic is reduced by 51% compared to baseline. That is similar to Doppelgänger's 54%. Truncate reduces the memory traffic by 47%. It is noteworthy that the large gap in MPKI between AVR (14% of baseline) and competing designs (48% and 53% for Doppelgänger and Truncate, respectively) is not reflected in the memory traffic volume. This is caused by frequent lazy writebacks leading to an inflated amount of read traffic when memory space is exhausted. AMAT follows the execution time trends. Doppelgänger leads with a 60% reduction. AVR AMAT is down by 43% compared to baseline and Truncate follows with 42%.

Lbm has about 98% of its footprint approximable, and AVR reduces it more than 15×. AVR is better than Truncate. It reduces baseline execution time by 57% vs. 42% for truncate, has 0.1% application output error (Truncate error is 0.6%), similar energy savings, lower memory traffic (33% vs. 50% for Truncate) and lower memory latency (30% reduction for AVR versus Truncate's 23%) due to very low LLC MPKI. Doppelgänger yields an excessive 22.3% output error, with a 3% improvement in execution time and no effect on total energy. The high output error is caused by edge-cases in Doppelgänger's approximation, where cache-lines at the extreme edges of their respective expected value span are considered approximately equal even though their absolute values are very different.

Orbit sees its total footprint reduced to 54% by AVR as half of its data are approximate and compress almost perfectly. AVR and Truncate introduce negligible error, while Doppelgänger causes strong artefacts leading to a runaway error exceeding 100%. AVR reduces execution time to 79% of baseline, Truncate trails behind at 82% followed by Doppelgänger with 86%. Truncate achieves the largest improvement in energy totaling 89% of baseline, AVR follows closely with a total of 92% and Doppelgänger with 93%. In spite of the high compression ratio, AVR only reduces memory traffic to 52%, outperforming by a narrow margin Truncate's 54%, while traffic for Doppelgänger is 65%. Memory latency follows a trend similar to execution time, with AVR achieving a reduction to 84%, Truncate yielding 86% and Doppelgänger 90% of the baseline.

K-means has a 58% reduction in memory footprint at the cost of 1.2% error. AVR achieves the highest instructions per cycle (IPC) count among all design points, but has the second shorter execution time after Truncate. That is because the application requires extra iterations to converge for the AVR, which increases the total number of executed instructions. Note, that k-means is the only benchmark used where the workload may vary based on

the quality of the approximations, all other applications have a fixed number of instructions to execute. Doppelgänger matches the baseline execution time despite its slightly improved memory latency and reduced memory traffic and has negligible error. AVR reduces energy cost by 2%. Truncate, which runs an identical number of instructions to the baseline, reduces energy by 13%. Doppelgänger energy overhead is 3% due to its LLC design. AVR reduces memory traffic by 37% and Truncate by 50%. This difference is an artifact of AVR's higher number of executed instructions. Doppelgänger has a smaller reduction of memory traffic, 26% less than the baseline, primarily because its LLC performs better than the baseline, as confirmed by its MPKI results. Memory latency is the shortest for AVR and Truncate, each 23% lower than the baseline, Doppelgänger follows with 12%. It is noteworthy that 55% of the AVR approximate LLC requests hit in the compressed blocks stored in the LLC and another 20% in DBUF.

Blackscholes (bscholes) uses input data where some of the input fields are identical for multiple entries [59]. This has been exploited by the Doppelgänger design. About 30% of bscholes dataset is approximable and AVR reaches a compression ratio of 4.7:1. However, bscholes is not memory intensive. As a consequence, the evaluated designs have little impact. Nevertheless it is still interesting to discuss their behaviour. Indeed, the execution time of all designs is very close to the baseline as shown in Figure 2.9a. This holds also for the energy consumption. Truncate and AVR reduce memory traffic by 15% and 6%, respectively. Doppelgänger reduces traffic by 3%, does not improve memory latency and reduces MPKI by 1%.

WRF has only 15% of its data marked as approximable, most of them geographically ordered weather metrics. AVR compresses these data with a 3.4:1 ratio reducing total memory footprint by 10%. Still it is an interesting application to discuss as a case where approximation does not offer a large benefit. AVR reduces execution time by 2% introducing 8.9% error to the application output. It reduces memory traffic only by 3% and has no effect on memory access time. Its MPKI is 7% lower than the baseline as over 50% of the approximable LLC requests hit in compressed blocks in the LLC or in the DBUF. Truncate has similar performance. It reduces execution time by 1% with 4.2% output error, reduces memory traffic by 5% and memory latency is unaffected. Finally, Doppelgänger causes 24.9% error with negligible performance impact.

Figures 2.10a and 2.10b show the breakdown of the LLC requests and evictions. In general, about 40-80% of the LLC requests hit on the DBUF or on compressed blocks. The latter case adds extra latency to the LLC hits for reading and decompressing the compressed block before serving a request. More precisely, the average LLC latency when hitting on a compressed block in the LLC is 20-30 cycles for wrf and kmeans, 74 for bscholes, and 40-50 cycles for the other benchmarks, which is still significantly faster than a DRAM access. The analysis for the AVR LLC evictions is mixed among benchmarks. For kmeans and bscholes about 40% of the evictions require fetching the block from memory and recompressing introducing traffic overheads, the remaining evictions are uncompressed written-backs because the block has failed to compress. On the contrary, the other benchmarks exploit the AVR lazy evictions in 45% to 80% of the cases avoiding fetching the compressed block on chip. Even including

the lazily evicted cachelines, the average size of a block read from memory is similar to the one indicated by the compression ratio shown per benchmark in Table 2.4. That is about 5.1 memory accesses to read a block for kmeans, 3.4-3.8 for bscholes and wrf, and 1.2-2 for the other benchmarks. Finally, the reuse of blocks is indicative to the AVR performance gains; on average 7-10 unique cachelines of a block are used before eviction for bscholes and lattice and 13-16 cachelines for the other benchmarks.

In summary, for applications with high compression ratio (*heat, lattice, lbm*), AVR is better than competing designs. It achieves significant reduction in execution time (40-55%) and considerable energy savings (10-20%) with less than 1% output error. Memory traffic is also reduced for these applications by 50% to 70%, although in some cases less than expected based on the compression ratio. Orbit is an exception to this trend; although AVR achieves excellent compression ratio, execution time is reduced only by 20%. At medium compression ratio, i.e. in *k-means*, AVR has moderate performance gains (about 15%) despite increasing the number of executed instructions. At low compressibility, i.e. in *wrf*, AVR improvements are negligible as are its overheads. Moreover, in compute bound applications, i.e. *bscholes*, there is minimum impact. Note that AVR memory latency is substantially reduced and always lower than the compared approaches. Finally, when not approximating, AVR does not have notable overheads.

2.4 Conclusion

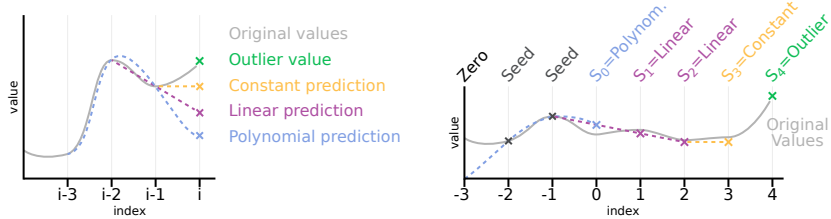
The AVR architecture improves the memory system using aggressive approximate compression. Thereby, AVR reduces memory traffic, utilizes more efficiently the off-chip bandwidth and achieves better performance and energy efficiency. AVR provides a low latency decompression scheme to reduce overheads in memory access time. Its LLC design stores both compressed and uncompressed data to increase its hit rate. AVR LLC evictions of compressible cachelines are handled in a lazy manner reducing the overhead of recompression. Moreover, keeping track of badly compressed blocks reduces unsuccessful compression attempts. Finally, the decompressed data selected to be stored in the LLC are carefully selected to avoid polluting the LLC with unwanted data. For applications with large part of the data being approximation-tolerant, in a system with 1GB of 1600MHz DDR4 per core and 1MB of LLC space per core, AVR reduces memory latency by up to 45%, memory traffic by up to 70%, and achieves up to 55% lower execution time, up to 20% lower energy with less than 1% error to the application output.

Chapter 3

MemSZ: Squeezing Memory Traffic with Lossy Compression

As laid out in Chapter 2, there is ample room for performance improvement by reducing traffic on the main memory bus. The AVR memory compression system was proposed, which uses *lossy compression* to achieve higher compression ratios for data which tolerate approximation. AVR uses a low-complexity *downsampling* compression algorithm to compress large (1kB) blocks of data. This large block size allows compression ratios of up to 16× to be supported. In order to limit the quality impact of approximation, AVR enforces two error thresholds during compression. Any individual value which deviates too far from the original is stored with at a fixed 2:1 compression ratio. If a full block exceeds the set threshold, the block is left completely uncompressed. This approach allows only *local* error control, i.e. deciding only based on the data currently being compressed. It does not account for accumulating error introduced when a block is compressed multiple times during its lifetime.

This chapter describes MemSZ, a more refined approach to lossy memory compression. MemSZ is based on the Squeeze (SZ) compression [34], a very effective, although sequential, algorithm introduced for compressing checkpointed data transferred between memory and disk. A brute-force hardware SZ implementation has a complexity of $O(n)$ and requires two processor cycles for compressing/decompressing a single value. Such latency would be prohibitive for memory compression. MemSZ introduces a new parallel version of SZ that is able to compress/decompress a block in $O(\sqrt{n})$ time, rather than in $O(n)$, in practice reducing the compression latency by 50× for the particular block size used. This low latency design of our compressor is then applied to an improved AVR design, which further offers better control of approximation error and a more efficient LLC replacement policy. MemSZ introduces a third error limiting mechanism, which tracks the accumulated error introduced when blocks are compressed more than once. Finally, in order to avoid AVR's LLC modifications, we apply the MemSZ compression to a system with a 3D-stacked DRAM cache with a line size that matches the granularity of the compressed



(a) The four function models of SZ (b) Reconstruction of a value sequence

Figure 3.1: The SZ compression scheme

memory blocks. As a result, both MemSZ designs offer significantly better compression ratio without increasing approximation error, in effect, reducing memory traffic and improving system performance and energy efficiency.

Concisely, MemSZ contributes the following:

- A new low latency parallel version of the SZ lossy compression algorithm that offers substantially higher compression ratio than previous lossy memory compression techniques;
- An extension of the existing AVR architecture, with a better cache replacement policy
- An additional error control mechanism, limiting accumulated error;
- A new architecture that combines memory compression with a 3D-stacked DRAM to more efficiently handle compression blocks.

The remainder of this chapter is organized as follows. Section 3.1 provides background information, and Section 3.2 describes the proposed MemSZ architecture. Section 3.3 presents our evaluation results and Section 3.4 draws our conclusions.

3.1 Background

MemSZ extends AVR, primarily by adding support for the more advanced Squeeze (SZ) lossy compression algorithm [34]. MemSZ adapts SZ for efficient hardware implementation by heavily parallelizing it. In this section, a detailed description of the SZ compression scheme is provided.

3.1.1 SZ Compression

SZ is an algorithm which lossily compresses a sequence of numeric values by describing each value X_i as a function of the three preceding values $[X_{i-3}, X_{i-2}, X_{i-1}]$, according to a predefined function model [34]. Four such function models are supported, as shown in Figure 3.1a: 1) A *constant* value is approximated as equal to the nearest preceding one (Equation 3.1), 2) a *linear* value is extrapolated from the preceding two values (Equation 3.2), 3) a *polynomial* value fits on the cubic curve described by the preceding three

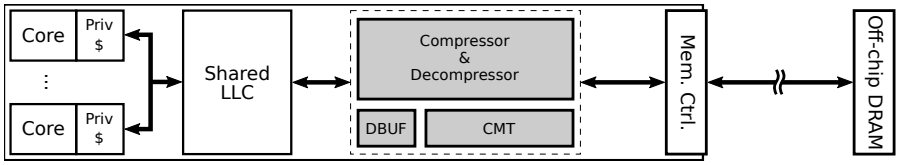


Figure 3.2: Top level lossy memory compression architecture used by AVR and MemSZ

values (Equation 3.3), or 4) if none of these models describes a value with an acceptable error, the value is an *outlier* and stored explicitly.

$$X_i^C = X_{i-1} \tag{3.1}$$

$$X_i^L = 2X_{i-1} - X_{i-2} \tag{3.2}$$

$$X_i^P = 3X_{i-1} - 3X_{i-2} + X_{i-3} \tag{3.3}$$

By adopting this classification, a sequence of values can be described using a two-bit *symbol* S_i per value, indicating one of the four functions used to generate the respective value, together with any outlier values, if option (4) is selected. To provide input for decompression, an initial set of *seed* values are explicitly stored. These values represent the first values in the sequence and allow decompression to begin with the first non-seed value.

Figure 3.1b illustrates how a sequence of values can be approximately reconstructed using only these *seeds*, the *symbols* chosen during compression, and the explicit *outlier* values stored. Inevitably, each value depends on up to three preceding values, forcing both compression and decompression to be sequential.

This sequential dependency between consecutive values increases the processing latency. Moreover, the computations needed for generating the polynomial value fit may be too complex to be performed in a single (processor) cycle. In the past, a hardware (FPGA) implementation of SZ, called GhostSZ has been proposed for accelerating the I/O compression [60]. GhostSZ compresses multiple streams of data in parallel to increase throughput, but the processing of each stream remains sequential. In essence, compressing or decompressing a single block of values is still *O(n)* and when implemented in ASIC would require at least 2 processor cycles per value due to SZ's complex arithmetic computations. As a consequence, previous SZ approaches are too slow and therefore impractical for memory compression. MemSZ introduces a new low latency parallel version of SZ able to compress/decompress a block of 256 values (16 cache lines) within 16 cycles.

3.2 System Architecture

Memory Squeeze (MemSZ) is a new lossy memory compression approach. It targets the parts of application datasets that tolerate approximations and substantially reduces their volume when transferred between main memory and processor chip. Thereby, MemSZ utilizes the off-chip bandwidth more efficiently

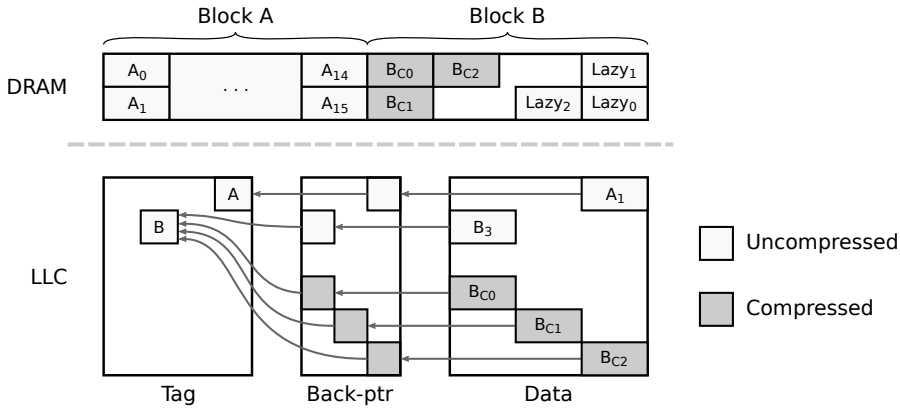


Figure 3.3: AVR's Decoupled Sectored Cache and main memory block layout.

and in turn improves system performance and energy efficiency. Without loss of generality, MemSZ is applied to a Chip Multiprocessor, between the main memory controller and the last level cache (LLC). A memory access to a location with compressed data brings on chip a compressed block of multiple (16) cache lines. After decompression, the requested cache line is stored in the LLC and sent to the processor, while the memory block is also stored in the LLC, to avoid memory accesses at future requests to the block.

MemSZ builds upon some of the AVR concepts described in Chapter 2. Data regions of applications are annotated by the programmer as approximable and marked as such in the page table and TLB. In addition, metadata information is maintained per memory block, stored in memory and cached on-chip in a *Compression Metadata Table* (CMT). Moreover, one of MemSZ's design alternatives (Figure 3.2) considers the same Decoupled SRAM LLC organization, storing both compressed and uncompressed data. Finally, like AVR, this design employs a 1kB Decompression Buffer (*DBUF*) to store the most recently decompressed block.

MemSZ extends our previous AVR design, as follows. First, it offers a new more effective lossy compressor, which improves compression ratio. For this purpose, we introduce the first parallel design of the Squeeze (SZ) algorithm [34]. Second, a more effective error limiting mechanism is used that is able to keep track of the error of a memory block, accumulated across multiple compressions. Third, it improves AVR's LLC replacement policy. Considering AVR's SRAM LLC on the processor die, MemSZ prioritizes the replacement of uncompressed lines versus compressed blocks; in doing so, LLC capacity is utilized more efficiently. Finally, as an alternative to AVR's modified Decoupled SRAM LLC, MemSZ also explores the use of a DRAM cache with a cache line size that matches the memory blocks considered for compression (1KB = 16×64B); then the DRAM cache needs to store only uncompressed data.

In the rest of this section, we describe the MemSZ compression and error handling as well as the two alternative LLC designs.

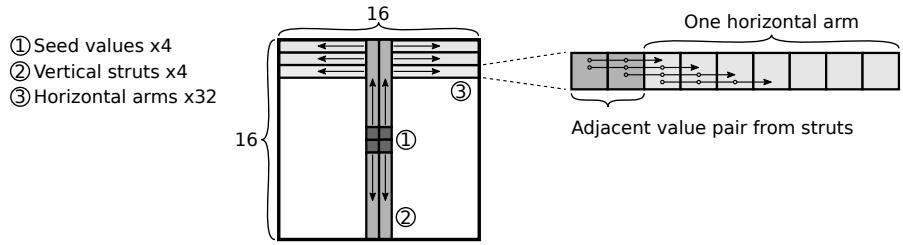


Figure 3.4: SZ Compression adapted to a 2D block

3.2.1 MemSZ Parallel Lossy Compressor

MemSZ introduces the first parallel hardware implementation of the SZ compression algorithm [34]. SZ is inherently sequential, using the three last produced values to compute the next value in the sequence. This linear dependency is an obstacle to low-latency implementation, since it limits parallelism. MemSZ breaks the linear sequence into a $(2\sqrt{n})$-way parallel operation, reducing the complexity of the algorithm from $O(n)$ to $O(\sqrt{n})$. In the following section, we present the optimizations behind this improvement.

The MemSZ compressor processes the values stored in a (memory) block in their native arithmetic representation, i.e. floating-point arithmetic is used for blocks of floating-point numbers, fixed-point arithmetic for fixed-point blocks. Our current implementation supports standard IEEE754 32-bit floating-point formats, but can be extended to integer and fixed-point.

Incoming blocks of 16 cache lines are arranged in a square of 16×16 values, as shown in Figure 3.4, before being fed to the compressor. In accordance with standard SZ, the generated compressed block consists of a two-bit *symbol* for each input value. Each two-bit *symbol* indicates the selected function used to reconstruct the input value, given its preceding values. The compressor also identifies individual *outlier* values which are not described by the summary with sufficient accuracy. These are explicitly stored alongside the symbol sequence. The proper location of each outlier is encoded in the compressed block using one designated *symbol*.

Decompression is performed in the opposite order. The summary is decoded, using each two-bit *symbol* to reconstruct a value based on its preceding neighbors. Outliers are stored explicitly in the compressed block, and re-inserted in the decompressed output. The complete decompressed block is stored in a decompressed block buffer (DBUF) and the requested cache line is inserted in the LLC.

As previously outlined, SZ describes each value in a sequence as a function of the preceding values. Four set function models are supported as described in Section 3.1.1, allowing each value to be described using a two-bit *symbol*. To start the sequence, a set of *seed* values are chosen and explicitly stored. Due to the dependency between the next produced value and the preceding ones, both SZ compression and decompression are sequential and have $O(n)$ latency, where n is the number of compressed/decompressed values. In fact, a direct hardware implementation of the SZ algorithm would require 2-3 processor cycles for each value generated. Such high latency is an obstacle for using SZ for memory

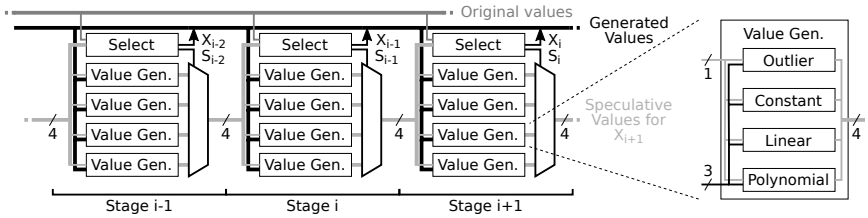
compression. To counteract this, MemSZ applies the following four techniques:

- Values are generated in multiple parallel sequences in two (block) dimensions requiring $O(\sqrt{n})$ steps.
- The two processing steps of (i) computing the four alternative functions output for the next value (ii) selecting the appropriate function (the one with the lowest error) are pipelined.
- Value generation of complex functions (i.e., polynomial) are also partitioned and performed in two consecutive pipeline stages.
- Generated values are fed forward in a *dataflow* fashion, allowing each computation to begin as soon as all dependencies are ready.

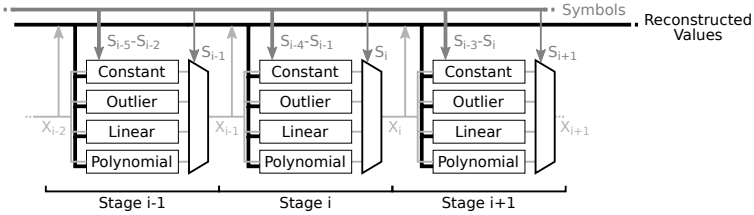
We use a subdivision of the 16×16 block into shorter sequences, which allows values to be generated in parallel. Both compression and decompression are divided into three phases as shown in Figure 3.4. In phase ①, the four values in the center of the block are used as *seed* values. These serve to start phase ②, consisting of four parallel vertical sequences of seven values each, called *struts*. The seeds and struts together form two complete columns of 16 values down the middle of the block. In phase ③, each horizontally adjacent pair of values from these columns serves to start two horizontal *arms*, making a total of 32. The arms are independent of each other and can therefore be processed in parallel starting as soon as the values they depend on are available – this may be well before phase 2 is complete. Using this method, the critical path to processing an entire block of 256 values is one full strut (7 operations) followed by one full arm (7 operations) for a total of $7+7 = 14 = 2 \cdot (\frac{1}{2}\sqrt{n}-1) = \sqrt{n}-2$ operations.

Compression: Figure 3.5a outlines a slice of our pipelined hardware implementation of an SZ compressor for a seven-value input sequence $[V_0 \dots V_6]$. A *Value Generator* module implements the four function model options (*outlier*, X^C, X^L, and X^P) to approximately reconstruct a single value given the three preceding values. In one cycle, all four options are speculatively generated for value V_{i-1} based on its preceding values. In the next cycle, each possible option (Outlier, Constant, Linear or Polynomial) for value V_{i-1} is used to speculatively generate values describing V_i. In parallel with this, a *Select* module chooses the most accurate reconstruction option X_{i-1} for value V_{i-1}. This yields the two-bit symbol S_{i-1} and resolves the speculation on V_i, resulting in its four options. These four options can then be used to speculatively generate X_{i+1} and so on.

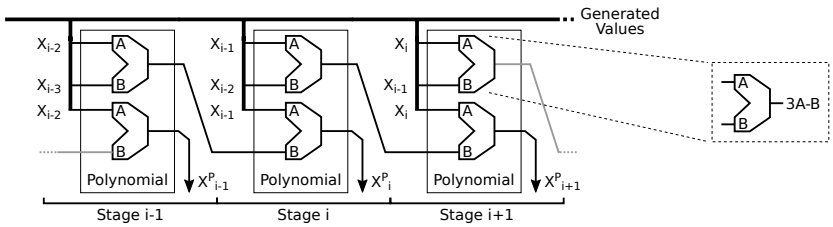
Since compression includes trying all function models for each value, compression latency is determined by the most complex function model. The *polynomial* function shown in Equation 3.3 is expressed as $X_i^P = 3X_{i-1} - 3X_{i-2} + X_{i-3}$. Considering that these are floating point computations, it would be too complex to fit it in a single cycle. In MemSZ, we break down the polynomial equation as $X_i^P = 3X_{i-1} - (3X_{i-2} - X_{i-3})$; the part in the parentheses can then be precomputed a cycle earlier, as soon as X_{i-2} is ready. By implementing the function $3A - B = 2A + A - B$ using a three-operand adder as described in [61], the polynomial function can be pipelined in two stages as illustrated in Figure 3.5c, without violating the clock period of our processor as shown in



(a) Pipeline for compression of a value sequence.



(b) Pipeline for decompression of a value sequence.



(c) Pipelining of the *polynomial* function

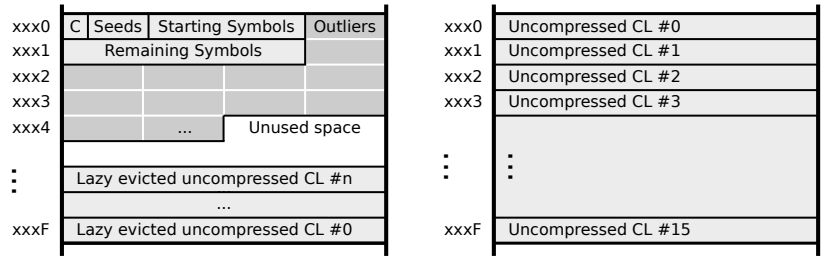
Figure 3.5: Design of the MemSZ compression pipeline.

Section 3.3.1. This allows the polynomial X_i^P to be ready in the same cycle as the other generated options for X_i, and the total latency for the compression of a full block is $2 \cdot (\frac{1}{2}\sqrt{n} - 1) + 2 = 14 + 2 = 16$ cycles, including two additional cycles, one for precomputing the first polynomial and one more for the function selection of the last generated value.

Decompression: Decompression is performed in a similar pipeline (Figure 3.5b), without the speculation. Since *constant* values require no further computation, these are forwarded up to three steps ahead in a single cycle. For example, if S_{i-1} is *constant* and S_i is *constant*, then $X_i = X_{i-1} = X_{i-2}$ and all three can be ready on the same cycle as X_{i-2}. Thus, several consecutive *constant* values can be reconstructed in a single cycle.

Because of this optimization, total decompression latency is variable. In the best case, the compressed block contains a large number of *constant* symbols and can be decompressed in 6 cycles. The longest latency is in a block containing an unbroken sequence of 14 *polynomial* values, which takes 16 cycles to decompress.

Symbol compression: The compression scheme outlined above requires a minimum of four *seeds* (stored in half precision) and 252 *symbols* to describe a block. The total size of this is $4 \cdot 16 + 2 \cdot 252 = 568$ bits $= 71$ bytes, which exceeds



(a) The MemSZ compressed memory block format

(b) Uncompressed memory block

Figure 3.6: MemSZ Memory Block.

the size of a cache line. As a result, the maximum achievable compression ratio would be 16 : 2 or 8×. In order to increase the compression ratio to 16 : 1, blocks with very few outliers undergo an additional pass of lossless encoding. The occurrences of each unique *symbol* are counted. New symbol encodings are assigned, such that the most common symbol requires only one 0 bit to be represented. The second-most common symbol is represented with two bits (10), outliers are given the three-bit code 111, and the remaining symbol is represented by 110. The total size of the compressed block with this encoding is the size of stored seeds and outliers plus the size of the re-encoded symbols plus a four-bit *dictionary* and a single bit to indicate that the block has been encoded this way. If this total size is less than one cache line, the block is stored using the smaller encoding. If the total size exceeds one cache line, the block is stored using the block format previously described. This re-encoding takes 3 cycles during compression, while it takes 2 cycles decode the first symbols before decompression can start.

Memory Blocks: To further facilitate low-latency decompression, we choose a format for the compressed block which allows decompression to begin as soon as the very first compressed memory subblock is available. This format is shown in Figure 3.6a, and packs the following into the very first 64-byte cache line of the block (CMS) : 1) A single bit C indicating if the symbols themselves are compressed, 2) The four seed values from the center of the block, at half precision, 3) The 28 two-bit symbols for the vertical struts, 4) Up to the 24 first outlier values, at half precision, 5) Any additional symbols for other positions (at least 4). Using this information, the decompressor can start reconstructing the center two columns of the 2D block on the very first cycle after these first 64B become available. The summary of a block compressed with *SZ* implicitly contains the required information to locate outliers, rendering AVR's outlier bitmap obsolete. As in the previous design, the empty space following a compressed memory block is used for lazy eviction of dirty cache lines, a technique designed to reduce and postpone recompression overhead, discussed in Chapter 2.

Total compression and decompression latency: Based on the above and as confirmed by our synthesis results presented in Section 3.3, the total worst-case latency for compressing a block is 32 processor cycles; that is 16 cycles for the actual compression and another 16 for packing the outliers. The

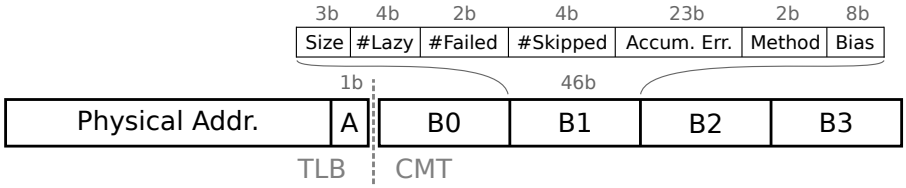


Figure 3.7: Format of a metadata table entry and TLB addition.

worst case latency for decompressing a block is 18 cycles, 2 for decoding the first symbols and 16 for decompressing the block. Decompression is more critical for system performance as it affects memory reads. Compression is less critical because it affects only the write backs.

3.2.2 Error Limiter

Lossy compression introduces errors in the approximated values. In order to limit this error, compression is skipped in case the error exceeds a particular threshold value. AVR employed the following two thresholds during compression to control the approximation error: the relative error of each individual value may not exceed the threshold T_1 and the average relative error across all values in the block may not exceed the threshold T_2. However, AVR's approach accounts only for the error introduced during the current compression. Oft-transferred blocks may accumulate error over their lifetime due to multiple compressions and AVR offers no mechanism to control this.

To limit the impact of accumulated error over time on the output quality of the application, MemSZ introduces an *error limiter* strategy. An accumulating counter is maintained for each block, updated with the average block error after each compression. If this accumulated error exceeds an absolute threshold T_3, compression is permanently disabled for the offending block.

MemSZ accounts for the two local thresholds T_1 and T_2 in two steps during compression. First, if none of the function models can approximate a value to within T_1, the value is kept as an outlier. Second, the average error of the compressed block is computed during compression and if it exceeds T_2 the block is left uncompressed.

The three error thresholds are exposed as knobs and in our experiments $T_1 = 2T_2$ while T_2 and T_3 are selected by profiling each individual application.

3.2.2.1 Metadata Table

Similarly to AVR, MemSZ uses a single bit in the page table and Translation Lookaside Buffer (TLB) to identify pages marked as approximable. Further metadata for each individual page is stored alongside the page table and cached on-chip in a dedicated Compression Metadata Table (CMT). These structures are shown in Figure 3.7. The CMT is updated in pair with the TLB. A CMT has four 46-bit entries per 4KB page, one per 1KB memory block as shown in Figure 3.7. CMT stores the following information about each memory block: its size (compressed in 1-7 lines or uncompressed), number of lazy evicted cache lines stored, compression method (and datatype of values), and a bias

of its values. In addition, it maintains two counters to keep the history of previous compression attempts. The first one counts the number of consecutive failed compression attempts. Then, depending on that count, a number of recompression attempts (in block updates) are skipped to reduce the overhead of badly compressed blocks. Finally, a running count of accumulated error is kept, to enable the *error limiter* outlined above.

3.2.3 Last Level Cache

After a memory access, keeping the entire accessed compressed block, rather than only the requested cache line, on-chip is important for reducing future memory accesses. This needs to be supported by the last level cache (LLC). MemSZ explores two alternative system designs with different last level cache (LLC) organizations. The first one, is based on AVR's Decoupled Sectored SRAM Cache placed on the processor die, where MemSZ proposes a new replacement policy. In an attempt to avoid AVR's LLC modifications, MemSZ also explores a DRAM LLC with a cache line size that matches the size of the memory blocks considered for compression. This second alternative uses a regular DRAM cache that stores uncompressed lines.

3.2.3.1 On-Chip SRAM LLC

A MemSZ system with an SRAM LLC on the processor die, employs AVR's Decoupled Sectored Cache, which is able to store uncompressed cache lines (UCL) and the compressed memory block (fragmented in compressed memory subblocks - CMS), as described in Chapter 2. Cache lines already accessed by the processor are stored as UCLs, while the rest of the memory block is compressed and stored in one or multiple CMS. Then, an LLC hit in the compressed block has a longer access time than a regular UCL hit because it requires decompression.

In the AVR system, there was a fair least recently used (LRU) replacement policy between UCL and CMS. This practically meant that cache lines already accessed by the processor could be found in LLC uncompressed and accessed again without a decompression overhead. However, it also meant that the LLC would often store a compressed memory block as well as (some) of its cache lines uncompressed (UCL), a redundancy which can waste LLC capacity.

MemSZ chooses a different trade-off between LLC capacity and access latency. It modifies the LLC replacement policy so that uncompressed approximable lines have higher priority to be replaced than compressed memory subblocks because they are redundant. Then, the LLC capacity is better utilized at the cost of a longer access latency to previously accessed lines. Normally, accessing an uncompressed cache line would cause updating its LRU bits. However, in case the LLC also contains the respective compressed block (indicated by the common tag entry), MemSZ updates the LRU bits of the CMSs instead. This marks the compressed block as recently used, delaying its replacement as opposed to the UCL. As a consequence, compressed blocks may remain on-chip longer than uncompressed lines representing the same data, thereby using the LLC capacity more efficiently.

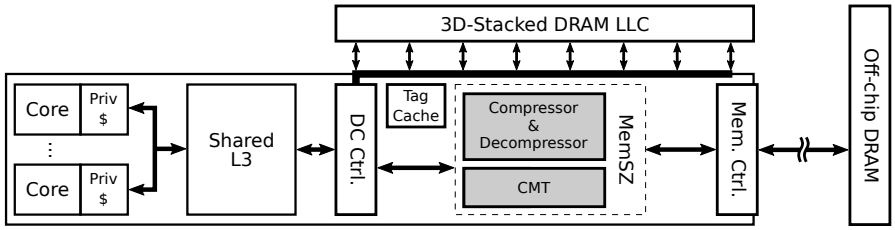


Figure 3.8: The architecture of MemSZ applied with a 3D-stacked DRAM cache (MemSZ-DC)

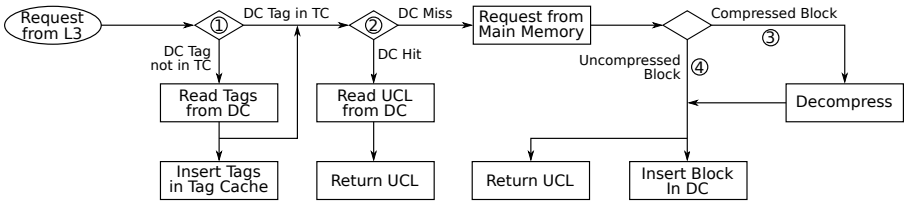


Figure 3.9: The handling of a request in a MemSZ system with a DRAM cache.

3.2.3.2 3D-stacked DRAM LLC

A system may use 3D-stacked DRAM, rather than SRAM, for implementing the last level cache. A DRAM cache offers larger capacity than an SRAM LLC and higher bandwidth than the off-chip main memory [62–66]. DRAM caches often use larger cache lines than the SRAM caches [62,65]. MemSZ exploits this characteristic and explores the use of a DRAM cache with line size equal to the size of the compressed memory blocks (1KB). Then, the memory blocks read from the memory can be stored uncompressed in the DRAM cache. In addition, making the DRAM cache inclusive eliminates the need of lazy evictions.

Our MemSZ system with DRAM cache (MemSZ-DC) is depicted in Figure 3.8. The compression-decompression remains between the main memory controller and the DRAM cache. The DRAM cache is organized as a set-associative cache with a 1kB block size. Tags for the DRAM cache are kept in a reserved area of the DRAM cache itself, with an SRAM tag cache on the processor die to store recently accessed tags.

A request from the shared (L3) cache for an approximable line (illustrated in Figure 3.9) can have the following outcomes:

① Miss in the Tag Cache, indicating that the DRAM cache lookup can not be performed. The relevant tags must be fetched from the DRAM cache before handling the request.

② Hit in last level DRAM cache, The line is read out and returned to L3

③ Miss in last level DRAM cache, compressed in main memory: The compressed block is read from main memory and decompressed. The requested line is returned to L3. The entire decompressed block is inserted in the last level DRAM cache.

④ Miss in last level DRAM cache, uncompressed in main memory: The entire block is read from main memory and inserted in DRAM cache. The requested line is returned to L3.

MemSZ-DC performs compression in the same manner as AVR, and maintains the same metadata. Compression failure (failure to meet the error threshold) is handled by storing the block uncompressed in main memory, and disabling compression for a set number of future attempts.

3.3 Evaluation

In this section we evaluate the effectiveness of the two proposed MemSZ architectures. We first describe our experimental setup, detailing the system configuration of our experiments and the benchmarks used. Then, we discuss the hardware overheads of the MemSZ architecture. Subsequently, we show the results of our evaluations and comparison with related designs in terms of performance, energy, and application output error. Results are first presented for designs with an SRAM LLC and then for designs with a 3D-stacked DRAM cache (MemSZ-DC). Finally, we evaluate the effect of the individual MemSZ features, namely: the compressor, error limiter, and replacement policy.

3.3.1 Experimental Setup

We evaluated the MemSZ system using an in-house simulator, implemented on top of Pin [48], that employs an interval-based processor model, as proposed by Genbrugge et al. [49], and a cycle-accurate model of the memory hierarchy that uses DRAMSim2 for modelling main memory and DRAM caches [50]. McPAT [51] and CACTI [52] were used to model power and latency of the system considering 32nm technology. The MemSZ compression hardware modules were implemented in RTL, synthesized using Synopsys to determine their operating frequency, latency and power consumption; this information was then fed to the simulation tool. The parameters of the simulated system are listed in Table 3.1.

In order to correctly emulate the impact of the approximations in the overall application error, we emulate not only the memory accesses but we actually update the values of the memory contents accordingly by applying the compression and reconstruction methods to the data.

Besides the baseline system, MemSZ is further compared with (i) the original design (*AVR*) [67], (ii) a design that simply compresses approximate values to half-precision by truncating 16 bits similarly to what has been proposed in [9, 28, 30] (*Truncate*), and (iii) Doppelgänger [39], which is the closest and best performing related work on approximate data compression (*Dganger*). As proposed, Doppelgänger is configured to have identical LLC data-array size and a 4× larger tag-array versus the other designs, i.e. being able to index up to 4× as many cache lines. Lossless compression techniques are considered orthogonal and so not included in the comparison; that is because the symbols and outliers of a MemSZ compressed block could be further compressed in a lossless way, in addition to compressing the non-approximable data which MemSZ leaves untouched. Finally, the individual features of

Table 3.1: Simulation parameters

Parameter	Configuration
CPU	8 core, out-of-order, 4-way issue/commit @ 3.2GHz
L1 cache	64kB per core, 4-way, 1 cycle latency
L2 cache	256kB per core, 8-way, 8 cycle latency
L3 cache	8MB shared, 2 banks, 16-way, 15 cycle access latency
L4 cache	256MB HBM2, 4 channels, 1066MHz, 8-way
Main Memory	8GB DDR4, 2 channels, 800MHz

MemSZ are evaluated by comparing it with (i) itself replacing its compressor with AVR's downsampling compressor (*MemDS*), (ii) itself with equal LRU priority between compressed and uncompressed lines (like in AVR) (*MemLRU*), (iv) itself without the error limiter (*MemUnL*).

MemSZ with DRAM cache (MemSZ-DC) is compared to (i) the baseline system and (ii) an approximating design which halves precision of data transferred across the main memory bus (*Trunc-DC*). Each of the three has an identical cache hierarchy and DRAM cache. AVR and Doppelgänger are not included in this comparison as they do not support a DRAM cache.

The benchmarks used in this evaluation are selected so each one of them (i) is able to execute until completion and generate an output, and (ii) can tolerate approximations in (parts of) its data. The above restricts us to using the benchmarks listed in Table 3.2; the table further presents the application domain, description of the approximated data-structures and output type as well as their memory footprint. In some applications, the selection of safely approximable data is affected by the technique applied. For this reason, the approximable footprint may differ between designs for the same application. Furthermore, some of the applications have a footprint smaller than the evaluated DRAM cache, and are therefore excluded from these experiments. The application code was analyzed to identify approximable data structures. In many cases, a large portion of the application's working set is dynamically allocated. For these cases, a wrapper was created around the *malloc* library call to allocate properly aligned space and register the address range as approximable.

One common source of memory traffic in scientific workloads is *checkpointing*. Checkpoints are occasional snapshots of the application's state, for the purpose of resuming execution after errors or outages. Such snapshots generate large bursts of data transfers to non-volatile storage, and contain approximable data from the application's working set. To reflect the effect of compression on these data, iterative benchmarks with checkpointing support have it enabled as indicated in Table 3.2.

The input data sets used for our experiments are the standard input data sets provided with the benchmarks with the exception of (i) *lattice* for which we used a silhouette of a car as the input data set, and (ii) *k-means* where the input is topological data [58]. Benchmarks for approximate computing (AxBench [59]) considers 10% relative output error, but it is solely up to the application provider do define what is acceptable. We use the mean of the relative errors for each output value as our quality metric. The only exception to this is *k-means*, whose output is discrete and strongly bounded. For this

Table 3.2: Benchmark Applications

Application	**Approx.**	**Output**	**Footprint / core**		**Checkp.**	**Description**
			Small	**Large**		
heat [53]	Temps	Temps	8.3MB	128.4MB	✓	Heat propagation through a 2D field of uniform material
lattice [54]	P and M	Vel.+Press.	5MB	160MB	✓	2D Lattice-Boltzmann simulation of air flow
lbm [55]	Velocities	Velocities	325MB	325MB		3D Lattice-Boltzmann simulation of fluid flow
orbit [56]	Phys. data	Phys. data	10MB		✓	3D simulation of the two-particle orbit problem
cdelta [56]	Phys. data	Phys. data	22MB		✓	Delta-function heat conduction model
sedov [56]	Phys. data	Phys. data	12MB		✓	Sedov explosion model
windt [56]	Phys. data	Phys. data	23MB		✓	Windtunnel with a step
kmeans [57]	Topol. [58]	Clusters	5.5MB	802MB	✓	Clustering, applied on a geographic elevation map.
bscholes [59]	Options	Prices	6MB	1368MB		Financial stock option price forecasting model
wrf [55]	Geo data	Temp.	90MB	90MB		Weather forecasting model

Table 3.3: Application output error

(a) SRAM LLC designs

	heat	lattice	lbm	orbit	cdelta	sedov	windt	kmeans	bscholes	wrf
dganger	0.4%	0.2%	0.1%	<0.05%	>100%	<0.05%	0.4%	<0.05%	<0.05%	56.8%
truncate	0.2%	0.4%	0.2%	<0.05%	<0.05%	<0.05%	0.1%	<0.05%	1.4%	4.2%
AVR	0.3%	0.5%	0.1%	<0.05%	4.8%	<0.05%	1.8%	0.7%	0.5%	8.9%
MemUnL	0.5%	0.5%	0.5%	<0.05%	3.8%	<0.05%	1.9%	1.0%	1.8%	crash
MemSZ	0.5%	0.5%	0.5%	<0.05%	2.6%	<0.05%	1.8%	0.6%	1.8%	0.6%

(b) DRAM LLC designs

	heat-large	lattice-large	lbm	kmeans-large	bscholes-large	wrf
Trunc-DC	<0.05%	0.1%	0.1%	<0.05%	0.6%	<0.05%
MemSZ-DC	<0.05%	0.3%	0.4%	0.2%	1.7%	<0.05%

application we normalize each individual error to the maximum possible error for that value, such that the maximum possible error is 100%. Similar to previous works, MemSZ provides tunable knobs to control the data approximation error and constrain application output error.

3.3.2 Hardware Overhead

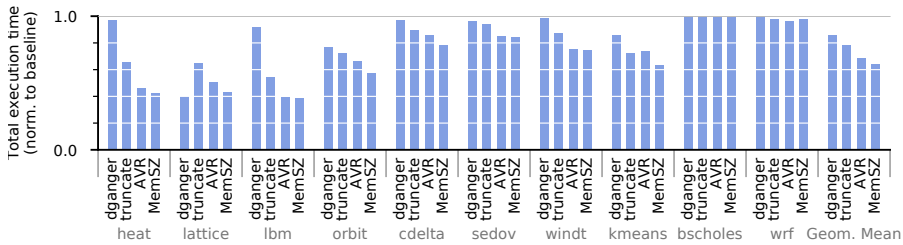
MemSZ requires additional hardware resources compared to a baseline cache hierarchy. The metadata stored in the CMT and the additional bit in the TLB add up to 173 bits per page. Compared to an unmodified TLB, which stores a virtual and a physical page address (52+36=88 bits), this is an increase to roughly 3× the original size. The MemSZ Tag array and the BPA add to the baseline set-associative LLC 18 bits per entry; that is in total 144kB and 3.2% overhead to the LLC. Moreover, the MemSZ compressor module occupies about 860k cells according to our synthesis report.

3.3.3 Experimental Results

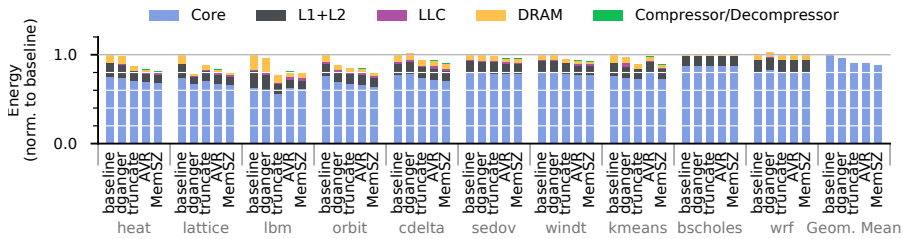
Next, we present our experimental results, comparing the two MemSZ designs against related approaches. The designs are evaluated in terms of execution time (Figures 3.10a and 3.12a), system energy consumption (Figures 3.10b and 3.12b), main memory traffic (Figures 3.10c and 3.12c), average memory access time (AMAT, Figures 3.10d and 3.12d), and LLC misses per kilo-instruction (MPKI, Figure 3.10e), as well as in terms of application output error (Tables 3.3a and 3.3b). Table 3.4 shows the compression ratios achieved by MemSZ and AVR as well as the reduction of the memory footprint versus the baseline. Finally, the individual features of the MemSZ architecture are evaluated separately in Figure 3.11.

3.3.3.1 MemSZ with SRAM LLC

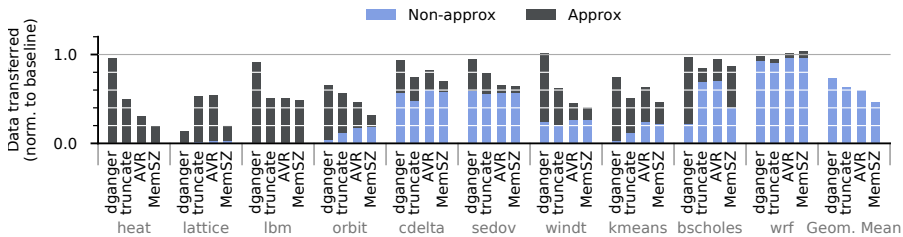
On average, MemSZ reduces the baseline execution time by 36% and system energy by 12%, the highest reduction across all the competing designs. That is mainly due to the fact that it has the lowest total memory traffic, with an average 46% of the baseline, as well as the lowest LLC MPKI, which is 43% of the baseline due to storing compressed memory blocks in the SRAM LLC. These two factors lead to a reduction of the baseline average memory



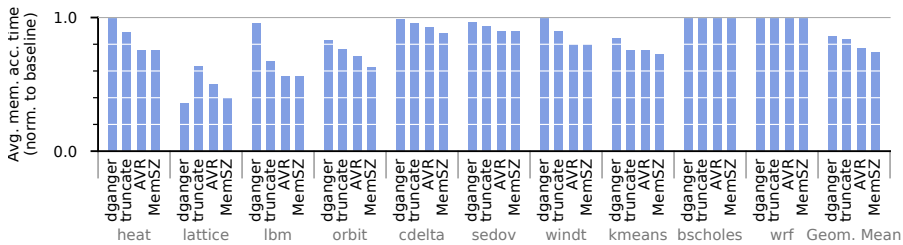
(a) Execution time.



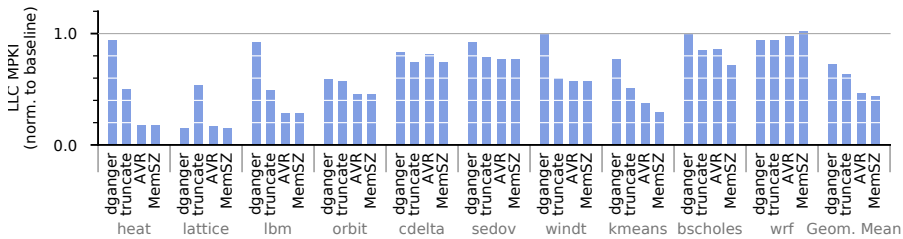
(b) System Energy Consumption.



(c) Memory Traffic.



(d) Average Memory Access Time (AMAT).



(e) LLC misses per kilo-instruction (MPKI).

Figure 3.10: Evaluation of the MemSZ design with SRAM LLC and comparison with competing designs.

access time (AMAT) by 26%. It is worth noting that for some benchmarks the baseline execution time is reduced by about 60% (heat, lattice, lbm) or by 40% (orbit, kmeans) and the energy consumption by up to 25% (heat, lattice, lbm, orbit) primarily due to very high memory traffic reduction, 2-5× in these cases.

The AVR design comes second with an average 31% reduction in execution time and 9% reduction in energy, which is 14% and 25% less impact than MemSZ on execution time and energy, respectively. AVR reduces memory traffic to 60% of the baseline as it uses a less efficient compressor, and has slightly higher MPKI than MemSZ due to its LLC replacement policy, which keeps redundant uncompressed lines and their compressed memory blocks. Overall its average AMAT is up to 25% higher, with an average of 4% higher than MemSZ.

Truncate offers an average 22% reduction in execution time and 9% in energy despite the limited compression ratio (2:1). Memory traffic is reduced to 64% of the baseline on average, and MPKI is at 63% of the baseline because it offers double the capacity for approximate LLC cache lines. This yields an AMAT of 84% on average for Truncate.

Doppelgänger has an average execution time that is only 14% lower than the baseline and system energy reduced only by 4%. It reduces MPKI (due to higher effective capacity) by 26%, which is lower than all other competing designs. This has an effect on memory traffic and AMAT, which are reduced by 26% and 14%, respectively.

Some of the benchmarks used have little approximate data or are already significantly improved/compressed by AVR. This has an impact on the above presented average results, however, for some other benchmarks MemSZ provides a significant improvement over the current state of the art. Below we analyze different groups of benchmarks separately and compare with the best previous designs.

As shown in Table 3.4a, for some benchmarks MemSZ improves compression ratio compared to AVR by 1.5× (heat, windt kmeans) or 2× (lattice, cdelta). For these benchmarks memory traffic of approximable data is reduced by about the same percentage. Note that in a couple of cases (cdelta, wind) the reduction in total memory traffic is less pronounced because the largest part of the traffic is non-approximable. On the other hand, MPKI for these benchmarks is less affected. For heat, lattice, and windt, AVR had already reduced the MPKI significantly and MemSZ maintains or slightly improves AVR's results. In cdelta and kmeans, MemSZ improves AVR's MPKI by about 10-25%. Reducing traffic and MPKI reduces AMAT compared to AVR by 5-25% for lattice, cdelta and kmeans, but does not show further improvement in heat and windt. Overall, for these five benchmarks MemSZ improves execution time compared to AVR by 2-15% and system energy by 1-9%. It is worth noting that the application error for all above benchmarks remains stable (lattice, windt) slightly changes ±0.2% (heat, kmeans), and in cdelta reduces to almost half of AVR's.

For other benchmarks, AVR's compression ratio is already close to the maximum 16:1 (lbm, orbit, sedov) and therefore MemSZ does not have significant room for improvement. Although for lbm and sedov, compared to AVR, MemSZ memory traffic and MPKI are not reduced, for orbit it reduces to $\frac{2}{3}$ due to the better LLC replacement policy, which yields significantly fewer writebacks as explained in Section 3.3.3.2. As a result, MemSZ does

Table 3.4: Compression efficacy

(a) MemSZ and AVR compression ratios. Truncate has a fixed 2× ratio.

	heat	lattice	lbm	orbit	cdelta	sedov	windt	kmeans	bscholes	wrf	GM
AVR	10.5×	6.8×	15.3×	16.0×	4.2×	15.6×	11.4×	2.3×	9.4×	2.3×	7.6×
MemSZ	15.8×	15.5×	16.0×	16.0×	9.2×	15.6×	15.5×	3.4×	6.3×[1]	1.6×	9.3×

(b) MemSZ and AVR memory footprints compared to baseline

	heat	lattice	lbm	orbit	cdelta	sedov	windt	kmeans	bscholes	wrf	GM
AVR	17.5%	27.2%	8.0%	48.1%	71.3%	72.5%	50.7%	61.8%	78.6%	93.1%	43.0%
MemSZ	14.3%	20.1%	7.7%	48.1%	66.4%	72.5%	49.5%	51.5%	54.4%[1]	95.4%	38.3%

not improve AVR's AMAT for lbm and sedov, but it does so by 21% for orbit. Execution time and system energy follow the same trend, AVR and MemSZ have similar performance and energy efficiency for lbm and sedov, but for orbit MemSZ reduces execution time by 14% and energy costs by 6%. Finally, MemSZ increases lbm's output error by 0.4% and maintains negligible error ($< 0.05\%$) in the other two benchmarks.

For wrf the approximable data are only a small fraction of the dataset (12%) and therefore there is little room for improvement. As a consequence, there are negligible ($\pm 2\%$) changes in performance and energy efficiency, however MemSZ reduces wrf's output error by 15× compared to AVR.

Finally, for bscholes AVR can approximate a smaller fraction of the dataset (27%) without crashing compared to MemSZ which approximates about 50% of the data and therefore achieves higher reduction of the memory footprint (46% vs. 21%) despite the lower compression ratio (6.3× vs. 9.4× of AVR). As a result, MemSZ reduces memory traffic and MPKI compared to AVR, but this has negligible effect in execution time and energy consumption because the performance of bscholes is not limited by memory bandwidth.

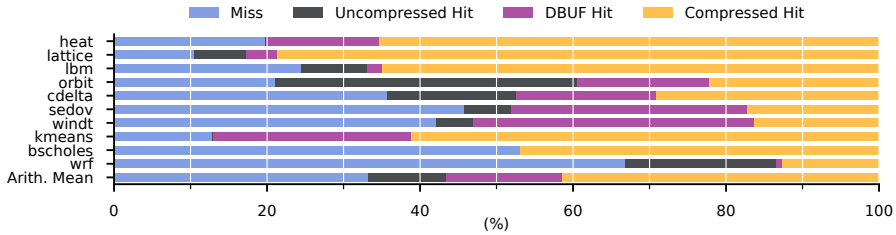
In summary, for applications with high compressibility and approximation tolerance (*heat, lattice, lbm, orbit, sedov, windt*), MemSZ outperforms competing designs. It achieves significant reduction in baseline execution time (up to 62%) and considerable energy savings (up to 25%) with less than 2% output error. Memory traffic is also reduced for these applications by up to 81%. Even when achieving medium compression ratios, i.e. in *kmeans*, MemSZ improves performance by up to 37%. MemSZ improves on the previous best design, AVR, by 23% in memory traffic, 7% in performance and 3% in energy on average.

3.3.3.2 Evaluation of individual MemSZ features

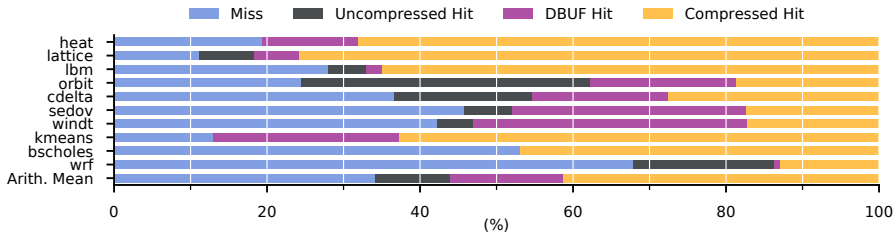
We investigate the effect of each of the three new primary MemSZ features by disabling one of them at a time. The resulting designs (i) *MemDS*: using the downsampling compressor of AVR, (ii) *MemLRU*: using AVR's LLC replacement policy, (iii) *MemUnL*: using no global error limiter (only AVR's local error checks), are compared to the original *AVR* design, lacking all these features, as well as with *MemSZ*, which has all features enabled.

Figure 3.11e shows this comparison for the key metrics of execution time,

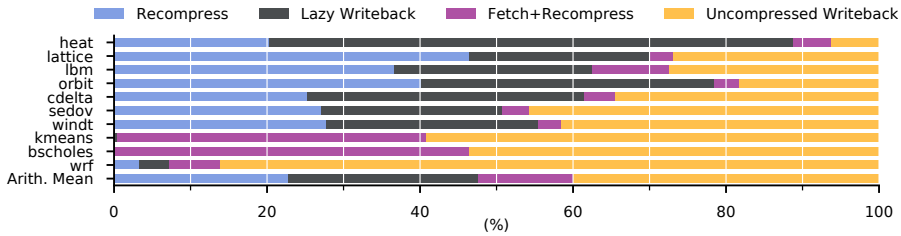
[1] MemSZ safely approximates a larger portion of the footprint compared to the other designs.



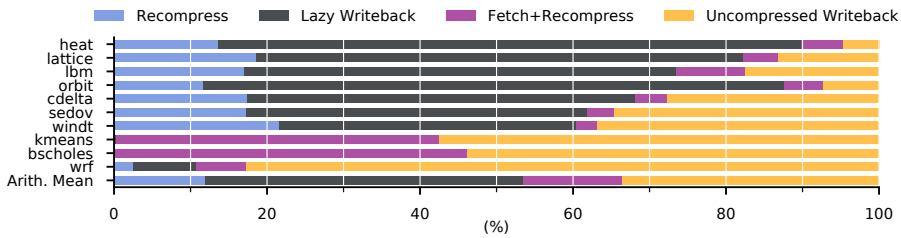
(a) MemSZ LLC Requests on approx. cache lines.



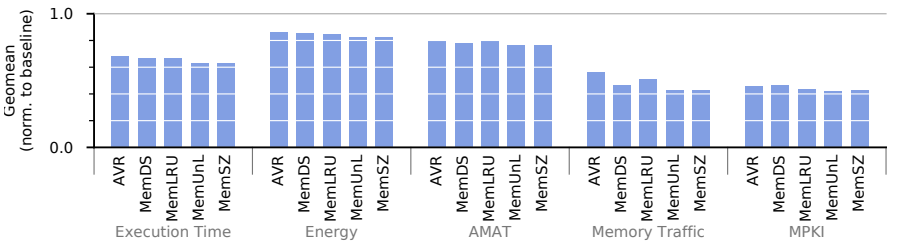
(b) MemLRU LLC Requests on approx. cache lines.



(c) MemSZ LLC Evictions of approx. cache lines.



(d) MemLRU LLC Eviction of approx. cache lines.



(e) MemSZ with each addition disabled.

Figure 3.11: Analysis of the compressor, error limiter, and LLC replacement policy introduced by MemSZ

total system energy, average memory access time, main memory traffic and LLC MPKI. Each metric is presented as a geometric mean across all the benchmarks from Table 3.2, normalized to the baseline. We observe that MemSZ outperforms AVR as well as all variations except *MemUnL*, losing 1% in execution time when enabling the error limiter and disabling some compression attempts. Table 3.3a illustrates the additional output error caused by disabling the error limiter. Note that when disabling the error limiter, for cdelta and kmeans the output error increases by about 50% and wrf crashes. Using the proposed MemSZ compressor rather than downsampling offers 10% lower memory traffic and 12% lower MPKI. Moreover, using MemSZ's LLC replacement instead of AVR's improves MPKI by 5% and reduces memory traffic by 15%.

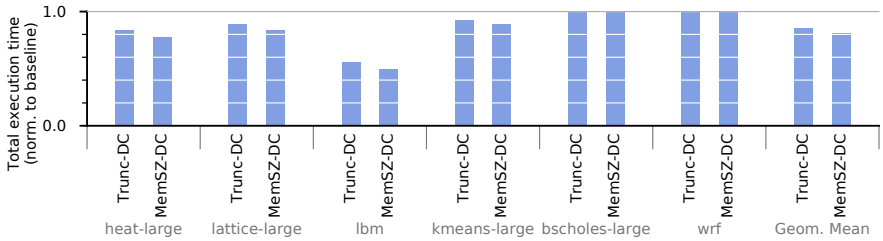
Taking a closer look at the impact of MemSZ LLC replacement policy in comparison to the one of AVR (MemLRU) we analyze the LLC requests and LLC evictions of the two designs in Figures 3.11a, 3.11b and Figures 3.11c and 3.11d, respectively.

As shown in Figures 3.11a and 3.11b, LLC requests may result in one of the following: a miss, a hit to an uncompressed line, a hit to the DBUF of the compressor, or a hit to a compressed memory block stored in the LLC. Compared to the old replacement policy, MemSZ slightly reduces the misses by 3% increasing the hits to the compressed blocks. This is more evident in heat, lattice, orbit, and sedov. In some cases like lbm and orbit the hits to uncompressed lines also increase.

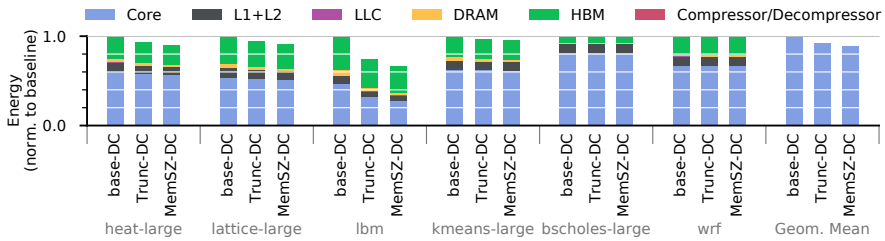
LLC evictions, shown in Figures 3.11c and 3.11d, can be handled in one of the following ways: update of the respective compressed memory block stored in the LLC (recompress), perform a lazy writeback of the evicted line uncompressed (when there is available space in the memory), the memory block may need to be fetched from memory to be updated (fetch+recompress), or there may be a common uncompressed writeback (if block is not compressed). Comparing the breakdown of evictions for the two designs we can observe the following. MemSZ increases the number of evictions served with a recompression by 89%, avoiding memory accesses. MemSZ also reduces lazy evictions by 40%, increasing the benefit of compression. Finally, the unwanted fetch+recompress cases are reduced by 4%.

3.3.3.3 MemSZ-DC evaluation

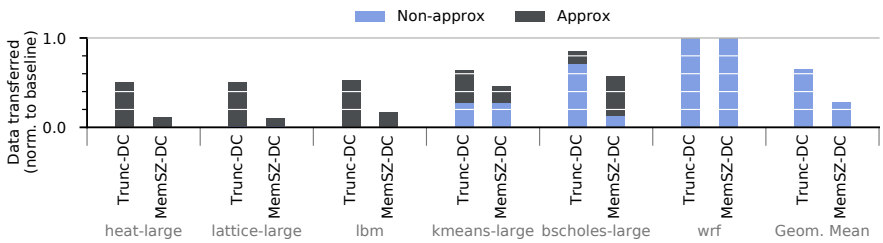
As expected, adding a DRAM cache (DC) to the system significantly reduces the traffic to main memory. This makes it more difficult to evaluate the proposed MemSZ-DC as it would require having longer running benchmarks and larger memory footprints making simulation times prohibitively long. As a result, our experiments show a smaller impact in systems with DC. We use all benchmarks with memory footprint larger than the DC size to evaluate our MemSZ approach with DC (MemSZ-DC) and compare it with a Truncate system with DC of the same size (Trunc-DC). Both designs are normalized to a baseline system with no compression and a DC of the same size. The other competing designs are not included in this comparison as they do not support DRAM caches. Figure 3.12 shows the execution time, system energy consumption, memory traffic and average memory access time (AMAT). MemSZ-DC reduces baseline memory traffic by 70% and Trunc-DC only by 35%. AMAT is less affected as MemSZ-



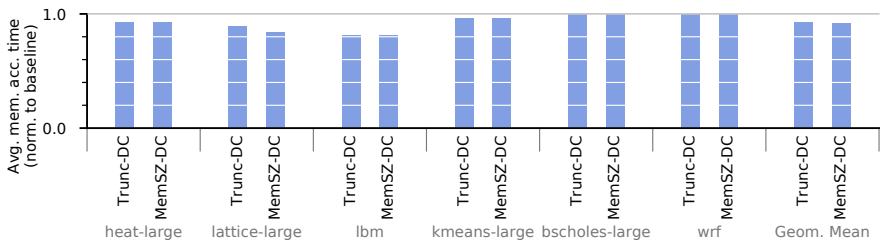
(a) Execution time.



(b) System Energy Consumption.



(c) Memory Traffic.



(d) Average Memory Access Time (AMAT).

Figure 3.12: Evaluation of the MemSZ design with DRAM LLC and comparison with competing designs.

DC reduces it to 92% of the baseline and Trunc-DC to 93%. This reduction in traffic and AMAT allows MemSZ-DC to reduce baseline execution time to 81% and energy consumption to 90% on average, while Trunc-DC reduces execution time to 85% and energy to 93% of the baseline. It is worth noting that the most pronounced improvements are achieved in heat, lattice and lbm, where memory traffic is reduced to 10-20% of the baseline traffic and execution time to 50-85% of the baseline.

3.4 Conclusion

MemSZ is a new more effective lossy memory compression approach. It introduces a new parallel design for the SZ algorithm, lowering compression latency by 50× and enabling for the first time its use in memory compression. The MemSZ architecture offers aggressive compression ratio (9.3:1 on average) reducing memory traffic and hence improving system performance and energy efficiency. MemSZ was evaluated in a multicore system with 1GB of 800MHz DDR4 per core and 1MB of LLC space per core. Compared to the previous AVR lossy memory compression approach, MemSZ has a better compressor offering up to 2× better compression ratio, up to 64% lower memory traffic, up to 15% lower execution time, and up to 9% lower system energy; on average the improvement is 23%, 23%, 7%, and 3%, respectively. Finally, MemSZ with a DRAM cache (MemSZ-DC) improves execution time, energy and off-chip memory traffic by up to 57%, 33% and 89%, and on average by 19%, 11%, and 71%, respectively.

Chapter 4

L^2C: Combining Lossy and Lossless Compression on Memory and I/O

The rapid increase of connected devices and data produced globally [68] drive numerous applications to become more data-intensive, overwhelming existing computing systems in various domains [2, 37, 38]. In high performance computing, server machines in data centers and supercomputers need to handle massive volumes of data supporting Big Data, Cloud Computing, streaming services and many other emerging applications. In the embedded domain, edge and Internet-of-Things (IoT) devices are expected to store, process and communicate data at high data rates under a tight power budget. In turn, the huge sizes and overwhelming rates of data put pressure on the memory and I/O bandwidth resources of systems and often become the bottleneck, limiting performance and wasting energy [5].

Chapters 2 and 3 outlined systems exploiting the novel *lossy memory compression* approach to reduce memory bandwidth. These systems use lossy compression to achieve greater compression ratios. Lossy compression has two main drawbacks: inaccuracies are introduced in the compressed data, and not all data are tolerant to such inaccuracies. This limits the applicability of lossy compression, and may require compression to be disabled if accumulated error exceeds acceptable levels [19, 20]. By contrast, while lossless compression is universally applicable, current lossless compression algorithms offer limited compression ratios (on average, between 2x and 4x) [18, 41, 69–72].

In addition to the main memory bus, another viable target for data compression is I/O traffic. Compression of data transferred through I/O ports has different design objectives as it strives for high throughput rather than low latency and handles data in larger blocks or in streams. In turn, the combination of latency tolerance and larger block sizes enables higher compression ratios. I/O compression offers better storage utilization and more efficient data transmission improving systems efficiency. I/O compression in embedded systems is often supported by custom hardware, hence is more expensive and with limited applicability, i.e., targeting wireless communications [73]. In the HPC

domain, IBM Power9 and Z15 offer user-controlled lossless-only compression acceleration [74], while software-based, hence slower, compression is used for check-pointing traffic [34].

This chapter describes L²C, a new holistic compression scheme aiming to more efficiently utilize the bandwidth resources of a processor chip. The main advantage of L²C is that it combines lossless and lossy compression to best fit the characteristics of different parts of a dataset and improve the impact of compression. In particular, L²C offers high-ratio, lossy compression for data that can be approximated and lower-ratio, but lossless compression for data that cannot. Thereby, it is an improvement over previous approaches that offer only lossy or only lossless compression. Combining lossless and lossy compression in the memory system is challenging as they exhibit radically different characteristics, which call for different design requirements. The compression ratio of a lossless method is 4-8× lower than that of a lossy one, as a consequence, using the same memory block size for both would either introduce excessive traffic overheads for the lossless or limit the effectiveness of the lossy method. On the other hand, supporting two different block sizes introduces challenges in the design of the memory system. L²C addresses these challenges to preserve the benefits of both compression alternatives. Another property of L²C is that it handles both memory and I/O traffic improving systems efficiency and simplifying integration in the uncore of a chip. However, reusing the same mechanism for memory and I/O compression introduces the challenge of supporting both low latency as well as high throughput compression, while remaining effective in handling small blocks.

The following contributions are made with L²C:

- The first approach that combines Lossy and Lossless compression algorithms in a memory system. L²C achieves this supporting:
  - two granularities of memory blocks, tailored to each compression method in order to increase its effectiveness and reduce overheads;
  - a cache structure and main memory layout that can store blocks of both granularities;
  - a mechanism to dynamically select the most suitable compression method;
  - a hybrid metadata format that supports the two methods and in addition is partially embedded with the data to reduce bandwidth overhead.
- Reusing the same compression mechanism for I/O traffic, too, to improve the efficiency of storage and networking functions, which is enabled by compressor designs that offer both high throughput and low latency.
- A thorough evaluation and comparison with state of the art compression techniques showing the benefits of combining lossy and lossless compression as well as the gains of reusing it for compressing I/O traffic.

The remainder of this chapter is organized as follows. Sections 4.1 and 4.2 discuss related work and background. Section 4.3 describes the proposed L²C architecture. Section 4.4 presents our evaluation results and Section 4.5 draws our conclusions.

4.1 Related Work

Prior work on related topics is discussed next. First, existing designs for memory compression are presented and subsequently a summary of I/O compression techniques applied in data collection systems is given. Finally, in relation with lossless compression methods, an overview is provided on approximate computing techniques that improve the performance of memory systems.

4.1.1 Memory Compression

A wide variety of memory compression techniques have been proposed for improving memory capacity and bandwidth utilization. They employ low latency algorithms and suggest different adjustments in the memory system to increase compression efficiency and minimize overheads.

Most existing designs use lossless compression algorithms to avoid introducing changes to the data. Some example of lossless algorithms applied to memory systems use dictionary-based compression [22], exploit frequent patterns and zero-value blocks [23], use similarities of words at the same bit position [18] or offer a hybrid scheme of different lossless algorithms applied to different data [24]. In spite of these varying approaches, lossless solutions have limited compression ratio between 2:1 and 4:1. Leveraging the fact that some applications can tolerate inaccuracies in parts of their data [19, 75], lossy algorithms, such as downsampling [67] and Squeeze [76], were introduced for memory compression to improve compression ratio up to 16:1. However, lossy approaches can be applied only to data that tolerate approximations and limits their applicability.

Besides the algorithm choice, another aspect is the data placement in memory. Some approaches compact compressed data in memory to improve capacity [13]. Others avoid the overheads of data compaction, allocating the worst case storage required for the uncompressed data and focus only on memory bandwidth improvements [26, 27, 67, 76].

Another important design choice is the granularity of the memory block size used for compression, especially when random access in the compressed form of the data is limited or not supported at all. Then, the block size defines a trade-off between the maximum supported compression ratio and the traffic overheads of fetching more data than requested. To exemplify, considering that a cache line (e.g. 64B) is the standard memory access granularity, selecting a block size of eight cache lines defines the maximum compression ratio to be 8:1. However if the average achieved compression ratio is 2:1 then that means that on average a memory access will bring four cache lines on-chip, at the risk of overhead in case of lacking locality. As a consequence, previous lossless memory compression solutions use small blocks of 2-4 cache lines and lossy ones use blocks of about 16 cache lines [67, 76]. Another overhead of larger block sizes is the fact that evicting a cache line from the chip requires the entire block to be present in order to get updated; this adds traffic overheads in case the block misses. In the past, the following two techniques have been used to reduce these overheads: the first one stores recently compressed blocks in the Last Level Cache of the processor and the second uses unoccupied memory space to evict dirty cache lines in their uncompressed form, postponing the

recompression of the block [67, 76].

Finally, managing the metadata needed for locating and handling the compressed data is also challenging as it may add considerable memory bandwidth overheads [16, 25]. One approach is to employ a metadata table and a cache of it, as in [13, 67, 76], which is updated with the TLB and adds a few bytes of bandwidth overhead at every TLB miss. Techniques like Attache [25] aim to further reduce the metadata cost by embedding the metadata directly in the compressed block.

L^{2}C strives to improve bandwidth utilization while avoiding compaction in main memory. Note that on the contrary, compaction in storage and networking I/O devices is one of L^{2}C objectives. L^{2}C is the first memory compression solution that addresses the challenge of combining lossy and lossless. It does so by adapting the memory system to support two block granularities; one for lossless and one for lossy compressed data. In addition, L^{2}C employs a mix between the two metadata approaches mentioned above, with *essential* metadata kept in a table along-side the TLB while *non-essential* metadata are embedded in the compressed block.

4.1.2 Link Compression

Compression has been a key technique for reducing I/O traffic in embedded as well as in HPC systems. The main design objective is high throughput and in the case of embedded systems low power is an additional requirement.

In distributed embedded data collection systems and IoT devices, compression fills a critical role due to tight constraints on power, communications and computational resources. Lossless compression has been applied to reduce the volume of off-device traffic [77], by exploiting application specific data properties [78], deduplication [79], prediction [80], and similarities between concurrent data streams [81]. General-purpose compression algorithms such as LZW have proved prohibitively expensive for such low-power devices [82] due to their excessive energy costs. A number of compression schemes have been proposed for embedded applications, utilizing data transformations [83], correlating multiple data sources [84], identifying particularly interesting (i.e. irregular) measurements [85], automatically adapting compression parameters to data features [86]. Moreover, a hybrid lossy and lossless scheme [73], the combination of which in I/O compression does not entail the challenges discussed for the memory compression counterpart.

In HPC applications, software-implemented lossy stream compression has been applied to high-volume I/O traffic without latency constraints [34] to alleviate the performance, energy and storage costs of saving checkpointed data. Moreover, IBM Power9 and z15 provide a user-controlled compressor accelerator in their DMA engine [74] to reduce data volume of DMA transfers.

In summary, embedded I/O compression techniques are mostly custom hardware designs, which increases the cost of the system and often limits their applicability to the particular targeted class of I/O devices. In the HPC domain, compression solutions are in some cases software-based, hence slower and less energy efficient, and in all cases controlled in the user space therefore cannot be exploited at regular memory and I/O operations.

L^{2}C exposes its proposed memory compression technique to compress I/O

traffic, too, in order to alleviate I/O bandwidth pressure and improve the efficiency of storage and networking systems functions. L^{2}C compression is generic, hardware accelerated and handled in a transparent way without user explicit control. Finally, reusing the same compression mechanism for memory and I/O saves systems energy and area.

4.1.3 Approximate Computing

The aforementioned lossy compression approaches can be considered part of the broader topic of Approximate computing as they introduce approximations to the data they handle. As such, they share in common some aspects such as the mechanisms for handling errors and identifying opportunities for approximation. Below, approximate computing techniques for improving the memory system are discussed.

Large classes of applications are inherently tolerant to approximations [19]. This enables a trade-off between the quality of their results and their performance and energy efficiency. This trade-off is exploited by various approximate computing techniques, such as computation acceleration [87], memoization [88], limited fault recovery [89], and data storage [90, 91].

Several approximate computing techniques target memory system bottlenecks. Approximate load value prediction reduces memory latency by predicting rather than fetching a value from memory [31–33]. Reducing the precision of floating point [9, 28, 29, 92] and fixed point [30, 93] numbers has been used to alleviate the memory bandwidth bottleneck in deep neural networks [30], GPU workloads [9, 29, 92, 94] and other approximation tolerant applications [28] improving performance and energy efficiency. However, the compression ratio is still limited between 2:1 and 4:1 despite the loss of precision as these approaches do not exploit inter-value similarities to compress data. Furthermore, Doppelgänger proposed to deduplicate similar cache lines to compress data [39].

A combined approach has been proposed to increase the compression ratio offering the option to reduce precision of individual values by truncating bits and then apply lossless compression on top [9]. The compression ratio remains at roughly 2:1, due to the limited impact of single-value precision reduction and is similar to existing lossless compression schemes, offering little benefit to outweigh quality loss of approximation. Precision reduction is distinct from full lossy compression, in that it only trivially reduces storage size for each individual value rather than identifying inter-value redundancy. Furthermore, the proposed design is implemented in a GPU architecture. While GPGPU techniques extend application support beyond graphics, it is nonetheless limited. L^{2}C takes a different approach, supporting lossless compression along-side more aggressive lossy compression in a general-purpose processor, as well as dynamically switching between the two. This is a more complex problem, due to the differing properties of the two compression methods.

In the past, applications [19] and (parts of) datasets [75] that tolerate approximations have been identified. Past lossy memory compression techniques used error thresholds for maintaining the introduced approximation error in check [67, 76] and evaluated the final error caused to the application output. They also kept track of the accumulated average error per block to limit the effect of repeated approximations on the same data [76]. L^{2}C follows the same

approach for handling the error introduced by lossless compression.

4.2 Background

L²C takes its basis in two existing compression systems: the lossy *MemSZ* presented in Chapter 3 and the lossless *Statistical Cache Compression* (SC²) [72]. Lossless compression is safe to apply to all application data, but generally offers limited compression ratio. Lossy compression is only applicable to select portions of data, but provides significantly higher compression potential. By combining these two approaches, L²C is able to reap the benefits of both. In this section, SC² is described in more detail.

4.2.1 Statistical Cache Compression

Statistical Cache Compression (SC²) is a lossless cache-compression scheme, rather than main memory, which is based on type-agnostic huffman-encoding [72]. A global *Value Frequency Table* (VFT) is populated during a sampling phase at the start of execution, forming the basis for an encoding tree. This encoding tree is then used to compress cache lines before they are written to LLC, increasing its capacity.

During the sampling phase, the VFT is populated by observing the last-level cache. The VFT is a set-associative cache structure, indexed by data values. It stores occurrence counters for the set of most frequently seen values. When a line is updated in LLC, each individual value in the cache line is *added* to VFT, i.e., its counter is incremented. When a cache line is evicted from LLC, each value in the line is *subtracted* from VFT, i.e. its counter is decremented.

Since the VFT is of finite capacity, not all possible values can be present at the same time. Newly observed values are inserted in the VFT, replacing the least-frequent value in its set. A special counter labeled *OTHER* is maintained with the sum of all replaced counts. This represents the frequency of any data value not explicitly present in the VFT.

When the sampling phase ends, the frequencies collected in VFT are used to build a huffman tree, assigning variable-length codes to each of the observed data values. This process assigns shorter codes to the most frequently seen values, based on the assumption that common values during sampling will remain common during the rest of execution.

During execution, any line to be inserted in the LLC is compressed using the generated encoding. Known values are replaced with their variable-length code. Values not assigned an explicit encoding are stored as-is, prefixed by the code assigned to *OTHER*. The global state (VFT) being shared between all compressed blocks removes the need to embed the huffman dictionary in the compressed block. This allows SC² to be applied to blocks of arbitrary size, with no reduction in compression efficiency.

4.3 System Architecture

L²C is a hybrid compression scheme which combines lossless compression with more aggressive, lossy compression. Lossy compression has the potential for

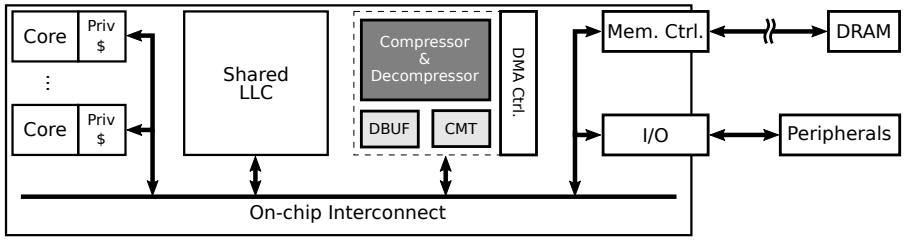


Figure 4.1: Top-level view of the L^{2}C memory compression architecture. The compressor module is placed next to the DMA controller, with access to the on-chip interconnect.

higher compression ratios, but is limited to data annotated by the developer as *approximable*. Lossless compression offers more modest benefits, but is safe to apply to all data, even as a fallback for approximable data. The hybrid nature of L^{2}C offers benefits over either approach. Lossless compression is available for all data. For data which is marked approximable, lossy compression is employed as a primary technique. If lossy compression fails due to quality constraints, L^{2}C falls back to lossless compression. This approach makes L^{2}C applicable and beneficial to any application able to tolerate lossy memory compression.

L^{2}C adds a hardware compressor in the uncore of a processor chip as depicted in Figure 4.1. It uses the MemSZ [76] and SC^2 [72] compression methods for lossy and lossless compression, respectively. The L^{2}C compressor module includes a buffer that stores the most recently decompressed data (DBUF) and a cache of the metadata table (CMT) to handle the compression/decompression process. Similar to MemSZ the LLC is designed as a decoupled sectored cache able to store compressed blocks alongside the normal uncompressed data. Moreover, the L^{2}C LLC and memory support two block type of different granularity to fit the requirements of the two compression modes.

The compressor is located next to a Direct Memory Access (DMA) controller and connected to the on-chip interconnect allowing it to interact with data transfers between the Last Level Cache (LLC), Memory controller and system I/O ports. This placement allows both memory and I/O compression. In turn, this enables L^{2}C to use the same compressor for both memory and I/O compression, the latter case controlled by the DMA.

Briefly, a memory access in the L^{2}C system, is handled as follows. L^{2}C extends the page table to include metadata information about the allocated pages, including the annotation of *approximable* pages, in other words pages that can be compressed in a lossy manner. A memory access is marked as approximable or not after the TLB access. Metadata is read out in parallel with the LLC being accessed. At the LLC, an access may hit either compressed or uncompressed data; otherwise (LLC miss), an access to the main memory is triggered. The metadata indicates the size and compression state of the fetched data. Moreover, LLC evictions are handled lazily by first attempting to update the block if it resides in the LLC; if not, an uncompressed write-back is attempted, if compression has left any unused space, otherwise, the block is fetched from memory to be recompressed.

In general, data in memory are grouped into larger blocks of multiple

cache lines. These blocks are kept in memory in compressed form. When a dirty cache line is evicted from the LLC, the compressed block it belongs to is eventually updated to include the fresh data. At maximum compression ratio, a block of 1kB (16 cache lines) fits in 64B (one cache line). Moreover, L²C can automatically downgrade blocks from lossy to lossless compression in cases where insufficient precision can be preserved. This allows the benefits of compression to be retained, at a reduced level, rather than leaving the data uncompressed. Finally, blocks which are not explicitly marked as approximable are only compressed losslessly.

In the remaining of the section, we describe the system design in more detail. First, the design of the compressor is presented. Subsequently, the L²C memory block format and memory layout are discussed. After that, it is explained how transition between block types are handled, and metadata information is organized. Then, the design of the last level cache (LLC) is described. Finally, the L²C I/O compression support is explained.

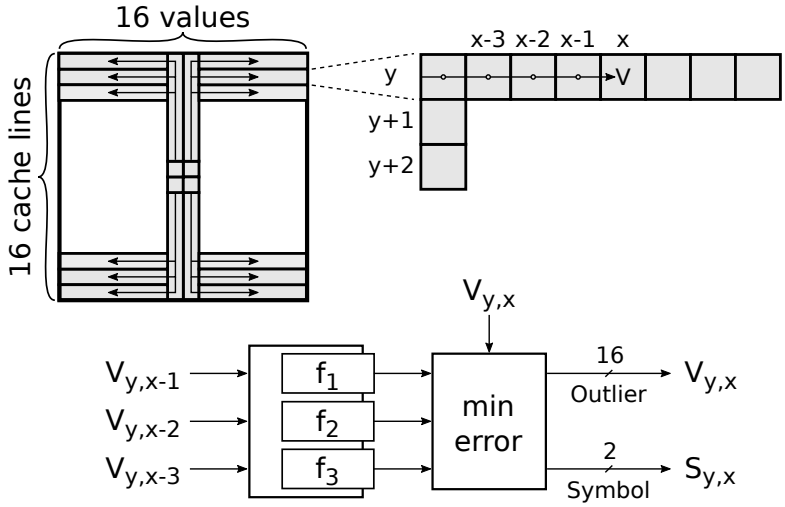
4.3.1 Compression Methods

The main feature of L²C is the application of two separate compressors, unified in a hybrid design. In this article, we present and evaluate using the MemSZ lossy compressor [76] and the SC² lossless compressor [72]. MemSZ represents the state of the art in lossy memory compression, offering compression ratios of up to 16×. SC² is designed for cache compression, which requires low latency and hardware complexity. These features also make it suitable for memory compression. Both parts of the L²C compressor are pipelined allowing high throughput. Without loss of generality, L²C can be implemented using any combination of block compressors. It is also trivial to extend L²C to support multiple lossy or lossless compressors and choose the most successful method for any given block.

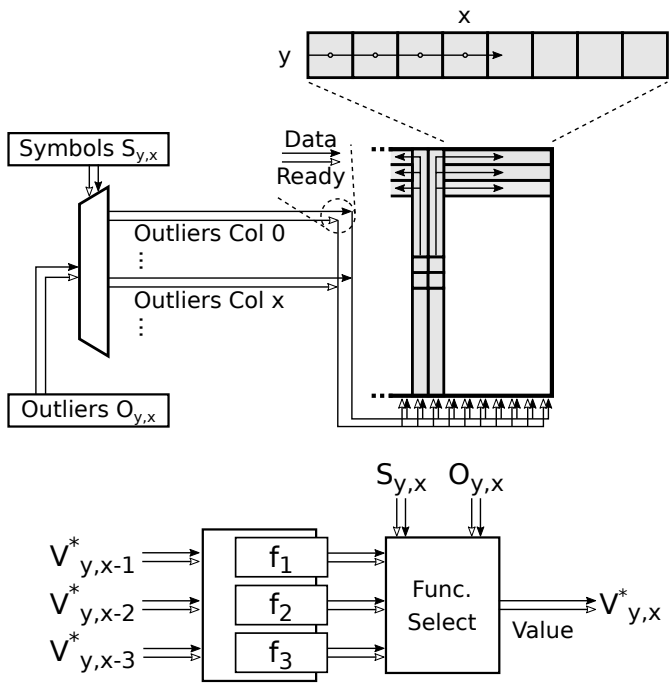
4.3.1.1 Lossy Compression

The lossy part of the L²C compressor is based on the SZ lossy compression algorithm [34], which compresses sequences of values by describing each consecutive value as a function of the preceding values. This is done by computing three different fixed functions (constant, linear or polynomial), comparing their respective error and selecting and storing the best option (two bits) in place of the value (32 bits). MemSZ introduces several performance improvements to SZ and applies it to 1kB blocks for memory compression [76]. Data blocks are processed in a square arrangement, allowing for greater parallelism both during compression and decompression as illustrated by Figure 4.2. The maximum achievable compression ratio for a 1kB block is 16 : 1.

The process of lossy compression of a 16 cache line block, outlined in Figure 4.2a, is designed to maximize parallelism. The 16 cache lines are arranged as rows in a square block. Four *seed* values are taken from the center of the block. The block is divided into 32 parallel sequences, starting vertically from the seeds in both directions and then spreading out toward the sides. Within each sequence, the compressor attempts to describe each value V strictly as a function of the preceding three values. If one of the available functions (constant, linear, or polynomial) successfully describes the value, a

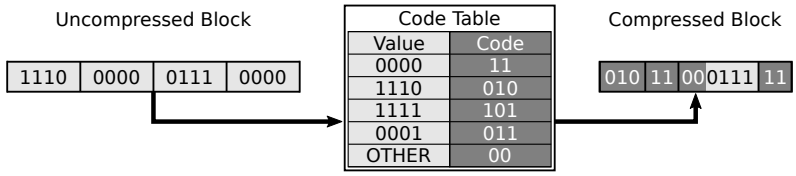


(a) Lossy MemSZ compression. Every 32-bit input value V is replaced by a 2-bit symbol S.

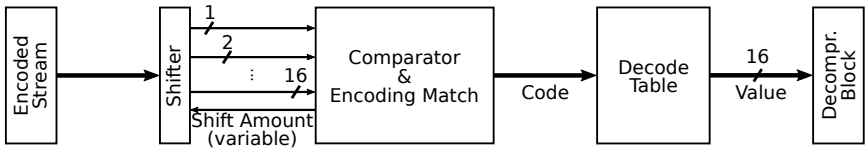


(b) MemSZ decompressor. Outlier placement (left) is carried out in parallel with symbol decompression (right). Dataflow signalling allows decompression to take place out of sequence.

Figure 4.2: Lossy compression scheme employed by L^2C. A data block is processed as several parallel sequences. Each value in a sequence is encoded as a function of the preceding values.



(a) SC^2 compression of 4-bit values. More common values are assigned shorter codes.



(b) Decompression of 16-bit values. A comparator identifies a single valid code from the front of the bitstream.

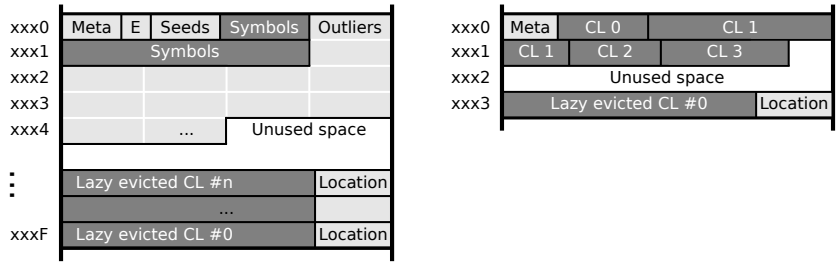
Figure 4.3: Lossless SC^2 compression scheme employed by L^{2}C.

two-bit *symbol* identifying that function is enough to represent the value. If none of the functions is successful, the value is an *outlier*, and is marked by a special symbol. The outlier value itself is stored at reduced precision (16 bits) in the compressed block. After this process, the completed compressed block consists of the seed values, the set of two-bit symbols and a collection of all identified outlier values. Compression of 1kB is completed in 16 cycles.

Decompression is illustrated in Figure 4.2b. It is optimized for minimal latency, and carried out in two parallel processes: Distribution of outlier values and decompression of symbols. Distribution of outliers is performed by decoding the sequence of symbols, identifying the location of outlier values, as well as their order. The outlier values are first assigned to their proper column. Each column is then populated, starting with the most critical center and progressing outward. The decompression of the two-bit symbols is performed in the same order as they were compressed; seed values are placed in the center of the block and 32 parallel sequences spread out vertically. Outliers may be placed throughout the block out of synchronization with these sequences, and the three decompression functions introduce differing dependencies and latencies. To exploit these irregularities, a dataflow-enabled pipeline design is used. Any one value to be decompressed is processed as soon as all its dependencies are in place. The variable decompression latency of a block, which is critical for memory reads and thus for performance, is at most 16 cycles.

4.3.1.2 Lossless Compression

The lossless L^{2}C compressor is based on the Statistical Cache Compressor (SC^2) [72], which employs huffman-encoding. SC^2 is an inter-block compression scheme that uses a single, global, symbol table to establish the encoding, as described in Section 4.2.1. Hence, it does not need to add any other overhead per block and is therefore well suited to compressing blocks of arbitrary size. Figure 4.3 illustrates an example SC^2 compression operation, where each value



(a) L-block compressed using MemSZ. (b) s-block compressed using SC^2.

Figure 4.4: L^2C Memory Block formats. Large blocks (*L-blocks*) are lossily compressed, Small blocks (*s-blocks*) are losslessly compressed.

of the uncompressed block (4 bits in Figure 4.3a) is looked up in the Code Table and replaced by the associated code. If the value is not found, it is maintained in uncompressed form preceded by the code for OTHER. The compression outcome is a compressed block of variable width. L^2C applies SC^2 compression using 16-bit value symbols and offers compression ratios of up to 4 : 1. SC^2 compression employs canonical Huffman codes: the codes follow the numerical sequence property, i.e., codes of the same length are numerically sequential. This is important during decompression.

The lossless L^2C decompressor is also based on the SC^2 decompressor [72] and is depicted in Figure 4.3b. Decompressing Huffman-encoded streams is inherently sequential because coded values are of variable length, thus it is not known where the next coded value starts in the encoded stream. Importantly, Huffman codes follow the prefix property, i.e., a code cannot be prefix of another code. Hence, when a bit sub-sequence matches a code, the next bit in the encoded stream determines the beginning of the next code.

The SC^2 decompressor works as follows: Part of the compressed block is inserted to a shifter. The 16 most significant bits of the bit-sequence within the shifter are inserted to the Comparator and Encoding Match engine. For each code length (1b, 2b, 3b, ..., and 16b), this engine performs numerical comparisons of the inserted bit sequence and the base value of the respective code length (i.e., the first assigned code for this length). A code of length x is matched within the bit sequence, when the comparison of x bits yields true result and the comparison of x+1 bits yields false. The matched code length determines the shift amount in the shifter and decoding can proceed with the next coded value in the stream. In parallel, the matched code is looked up in the Decode Table and the associated value is output and attached to the decompressed block. This process is repeated until all values are decompressed in the block. The decompression latency is 14 cycles per cache line at 1GHz, parallelizable for larger blocks.

4.3.2 Block Types

The two compression schemes employed by L^2C differ in their utility and application. The lossy compressor is geared toward high compression ratios, necessitating large blocks. This is in part due to a fixed per-block data

overhead, in the form of *seed values* which must be included uncompressed in the compressed block. The lossless compressor, by contrast, has no such fixed overheads. Its compressed blocks consist only of re-encoded values from the original data. This allows it to be applied to blocks of any size.

The optimal block size for any memory compression scheme depends on two factors: the maximum achievable compression ratio and the minimal transfer unit of the memory bus. An undersized block may compress to a size smaller than the minimal transfer unit, leading to transfers larger than necessary. Conversely, an oversized block may compress below expectation, leading to extraneous data transferred. For these reasons, the optimal block size is such that the maximum expected compression ratio results in a compressed size equal to the minimum compression size.

The minimum transfer unit of a typical system is one cache line. The lossy compression employed by L^{2}C is designed for a maximum compression ratio of 16 : 1, and is thus applied to blocks of 16 cache lines. We refer to these large blocks as *L-blocks*. The lossless compression using 16-bit values has a theoretical maximum compression ratio of 16 : 1 (compressing each 16-bit value to a single-bit encoding), but typically achieves compression ratios between 2 : 1 and 4 : 1 on non-constant data. For this reason, L^{2}C applies lossless compressed data to blocks of 4 cache lines. We label these small blocks *s-blocks*. An *s-block* is a quarter of an *L-block*, which is convenient for their alignment and management. As L^{2}C combines these block types, a 1kB region of memory can either be one *L-block* or four *s-blocks*. Figure 4.4 illustrates the format of each block type. Both types contain a small amount of embedded block metadata, which is further described in Section 4.3.5.

The L-block is specifically organized to allow decompression to begin as soon as the first line is available. A single bit E indicates that the rest of the line has been losslessly encoded to save space. This is followed by a set of *seed* values, from which all SZ sequences begin. The first line also contains an initial set of two-bit *symbols* representing compressed values, as well as a number of *outliers* sufficient to start decompressing the center columns of the block. The remaining lines of the compressed block contains the rest of the symbols and any remaining outliers. The s-block format is simpler, consisting only of the compressed cache lines.

Both types of blocks leave unused space at the end of their allocation in physical memory, which is used for *lazy evictions*. When a compressed block is only available off-chip, any dirty uncompressed cache line evicted from LLC will be stored in this space. In order to reconstruct a block with lazily evicted cache lines, the location of each dirty line must be maintained. For approximable data, data precision is reduced by a few bits to encode the proper location of the cache line. In non-approximable data, the evicted cache line is compressed and the location information is appended to the end of the cache line.

4.3.3 Memory Layout

The use of multiple compression schemes with differing block sizes necessitates a flexible memory layout for compressed data. A memory location may be in one of three different states:

1. Compressed lossily as part of a 1kB L-block

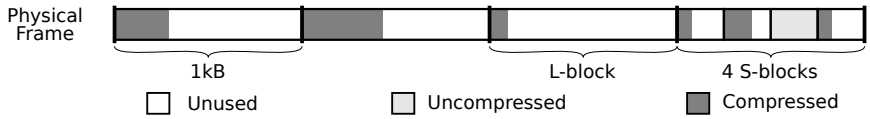


Figure 4.5: Main memory with a mixture of block types. Each 1kB space is one L-block or four s-blocks.

2. Compressed losslessly as part of a 256B s-block

3. Uncompressed as part of an uncompressed 256B s-block

L-blocks are aligned to 1kB boundaries while s-blocks always appear in groups of four, each aligned to 256B. Figure 4.5 illustrates L- and s-blocks coexisting in physical memory. This alignment serves dual purposes. First, the address of a cache line can be trivially translated into the physical address of the corresponding compressed block. Second, it allows an L-block to transition into four s-blocks if lossy compression fails, without affecting neighboring blocks outside the 1kB allocation. This type of transition is central to L^2C, enabling a fallback to less aggressive compression rather than leaving data uncompressed.

4.3.4 Block Type Transition

During the execution of a program, the same memory region may be dynamically selected to be compressed in a lossy or lossless manner as long as it is indicated to be approximable. The transition between lossy L-blocks and lossless s-blocks is described below.

When lossy compression of an L-block is attempted and fails, MemSZ leaves the full block uncompressed. This leads to wasted compression potential, since the data may still exhibit some amount of redundancy. L^2C leverages this potential by transitioning the L-block into four s-blocks and applying lossless compression. In effect, data compressibility determines a block's place within a hierarchy of compression states, from lossy L-block via lossy s-block and down to completely uncompressed s-block. Figure 4.6 illustrates the logic governing transitions between these states.

Uncompressed data may, with updates, become compressible again. SC^2 compression is applicable to blocks of any size, and L^2C uses this property to determine the compressibility of individual cache lines. A *back-off* counter ① associated with uncompressed s-blocks keeps track of the number of individual and compressible cache lines written back to the block. When the counter reaches its maximum, the s-block is expected to be compressible and a transition ② is attempted.

Analogously, after some number of updates to a compressed s-block, it is possible that compressibility changes and lossy compression becomes viable. L^2C uses the lossless compressibility of the s-blocks as an indicator for this (Figure 4.6). Every group of four s-blocks shares a *transition count* ③, which is incremented when a compressed s-block is written back to memory. If any s-block fails compression, the transition count is cleared. Once a sufficient number of consecutive lossless compression attempts have been successful, a transition ④ to a single L-block is attempted.

Transition to a lower compression state (i.e. L-block to S-Block or s-block to uncompressed data) is straight-forward. Such a transition occurs only when compression fails, and thus all data is already available on-chip. Conversely, any transition toward a higher compression state involves reading multiple cache lines from memory, in order to compress a larger block. In the worst case, this consists of three compressed s-blocks totalling nine cache lines. To reduce this traffic overhead, L^{2}C postpones the transition attempt until the next cache miss for this block. Because miss resolution requires one uncompressed cache line or one compressed s-block from memory, this reduces the total overhead of the transition. In addition, any compressed blocks which are already on-chip in the LLC do not need to be transferred.

4.3.5 Block Metadata

One hurdle faced by memory compression systems is the overhead of metadata. Certain information about a compressed block may be necessary in order to manipulate the block in memory or bring it on-chip for processing. This additional information is too large to keep on-chip in its entirety, and must therefore be stored in main memory.

To reduce the traffic overhead of such metadata, L^{2}C divides the compression metadata into two categories. *Essential* metadata are necessary even when the corresponding block is not on-chip, in order to fetch or update it. *Non-essential* metadata are only needed once the block is on-chip, and are embedded in the compressed block as illustrated in Figure 4.4.

Non-essential metadata are only needed when the full compressed block is also available for processing. This information consists of the size of the compressed block excluding lazily evicted cache lines, which is necessary in order to decompress the block. In addition, L-blocks encode the *compression method* used, to be able to differentiate between data types and potentially support other compression schemes.

L^{2}C uses a Compression Metadata Table (CMT) as an on-chip cache for essential compression metadata. CMT has a structure corresponding to the existing Translation Lookaside Buffer, and is updated in tandem with it on TLB misses. Each quarter-page is described either as one L-block or four s-blocks. Four unused bits (labeled F) in the regular Page Table Entry (PTE) are used to encode this state. An additional TLB bit is used to mark approximable pages. A CMT entry comprises 64 bits for one page, and is organized as illustrated in Figure 4.7.

s-blocks are afforded four bits of CMT space. These four bits are used to encode three fields: a two-bit *size* field, a 1-bit *transition* counter (described below), and a 3-bit *back-off counter* used to delay compression for uncompressed blocks. Since the size and transition fields are only needed for compressed blocks and the counter is only needed for uncompressed blocks, these two sets are overlapped. A single bit C is used to distinguish between the two states.

L-blocks have 16 bits of essential metadata, divided into two fields: a four-bit *size* field and a twelve-bit counter of *accumulated error*. The twelve-bit counter is a floating-point (4-bit exponent and 8-bit mantissa) representation of the accumulated error introduced by lossy compression.

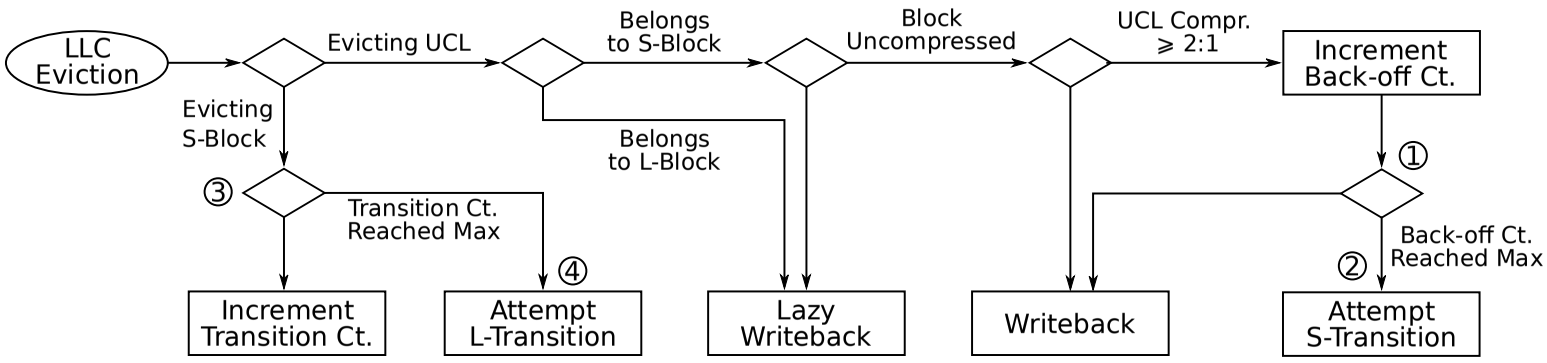


Figure 4.6: Back-off and transition behavior. Blocks transition from Uncompressed to lossless s-Block to lossy L-Block as compressibility tests succeed.

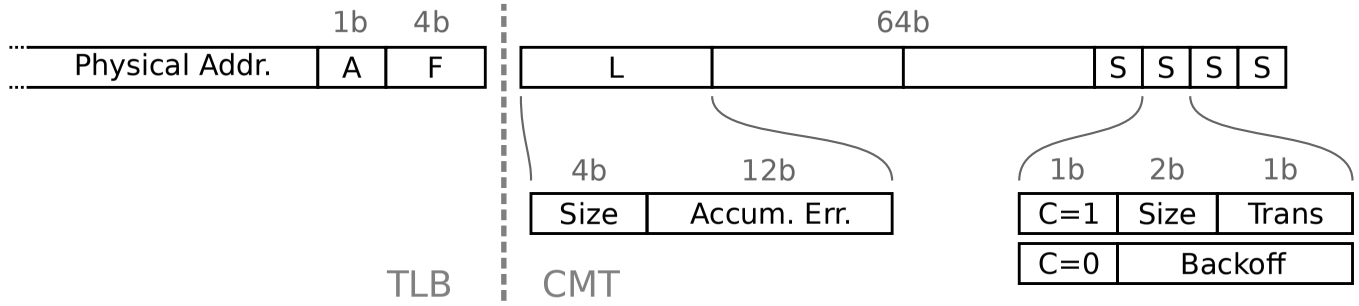


Figure 4.7: Metadata table format. s-block metadata is encoded differently for Compressed (C=1) and Uncompressed (C=0) blocks.

4.3.5.1 Metadata during transitions between block types

Metadata encoding is complicated by the multiple compression states a single block may have. One cause of transition is a failure to compress. L-blocks which fail lossy compression transition into four s-blocks. s-blocks which fail lossless compression transition into uncompressed data. The opposite transitions are carried out when compression is retried successfully. These retries are controlled using *back-off counters*.

Transition from uncompressed to compressed s-block is tracked using the metadata for s-blocks, as discussed above. When an uncompressed eviction occurs, the compressibility of the cache line is tested. The *back-off* counter of the corresponding s-block is incremented if the evicted line has an individual compressibility at or above 2 : 1.

Transition from s-block to L-block (for pages annotated as approximable, i.e. allowing L-block lossy compression) is controlled by four *transition* bits spread out across the metadata of the s-blocks. These bits encode a counter of consecutive successful s-block compression attempts, indicating that the data is compressible. Overlapping the metadata bits this way works, since the *transition* counter is only valid if all four s-blocks have been successfully compressed.

The *Accumulated Error* counter associated with an approximable L-block must be maintained even when the block temporarily transitions to s-blocks or is left uncompressed due to failed compression. This is done by including three bits of the counter in the *non-essential* metadata embedded in each s-block, if that block is compressed. If an s-block is uncompressed, the three bits are instead embedded as the least significant bit of each of the first three data words.

4.3.6 Last-Level Cache

Support for two separate memory block sizes also raises the need for similar support in the last-level cache. Resolving LLC misses by fetching compressed blocks from memory introduces traffic overhead because blocks may be larger than one cache line. In order to benefit from the extra fetched data, it must be kept on-chip for as long as possible. If the data exhibits spatial locality, the fetched block acts as a form of prefetching, at reduced traffic cost.

L²C uses a Decoupled Sectored Last-level Cache [35] to store compressed and uncompressed data on-chip simultaneously. Tags are decoupled from data entries as illustrated in Figure 4.8 and associated using a special *back-pointer array*. This allows multiple data entries representing the same 1kB address space to share the same tag. For example, a 1kB region of physical memory may be present in the LLC as one compressed L-block and three uncompressed cache lines, simultaneously. Three separate indexing functions are used for data placement: One for compressed L-blocks ($Index_L$), one for compressed s-blocks ($Index_S$) and one for uncompressed data ($Index_U$).

L- and s-blocks all consist of one or more cache line sized CMSs. All CMSs belonging to a single compressed block are placed in consecutive LLC sets. Since a tag never represents both L- and s-blocks simultaneously, the two use similar indexing functions. If the L-block indexing function $Index_L(A)$ indicates that the compressed data for a tag A should start in set X, then

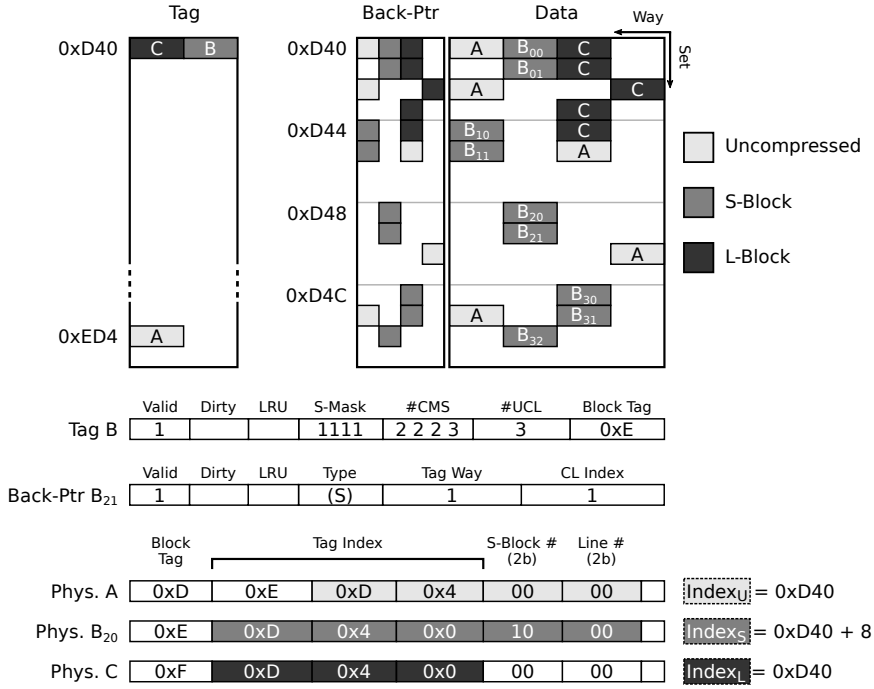


Figure 4.8: Conceptual view of the L^{2}C decoupled sectored cache and its three data indexing functions. s-blocks are placed at 4-set intervals.

$Index_S(A)$ would also place the first s-block for that same tag starting at X. The second s-block is placed at $X+4$, the third at $X+8$ and the last at $X+12$. This way, L-compressed blocks and S-compressed blocks have similar behavior in the LLC. The indexing functions $Index_L(A)$ and $Index_S(A)$ are chosen to minimize interference between compressed and uncompressed data belonging to the same block, i.e. the uncompressed indexing function $Index_U(A)$ is unlikely to return the same index as $Index_L(A)$ or $Index_S(A)$.

Figure 4.8 illustrates a slice of the LLC with data from three 1kB memory blocks (A, B and C) present. A is uncompressed, B is compressed as four s-blocks and C is compressed as a single L-block. Their respective physical addresses are such that the indexing functions $Index_U(A) = Index_S(B) = Index_L(C) = 0xD40$, and they thus contend for the same 16 sets in LLC. The uncompressed cache lines from A are placed based on their individual addresses. The four s-blocks $B_0 - B_3$ start at four-set intervals, while the compressed L-block is placed in five consecutive sets starting at $0xD40$. Any compressed data for A or uncompressed data for B and C are placed in other sets..

The LLC supports three types of lookups (Uncompressed, S-Block, or L-Block). Lookups work similarly to a standard Decoupled Sectored Cache. The tag index is computed from the sought physical address. Based on the type of lookup (Uncompressed, S-Block, or L-Block), the corresponding indexing function ($Index_U$, $Index_S$, $Index_L$, respectively) is used to identify the proper set in the back-pointer/data arrays. Tag and BP lookups are then performed in parallel. If a Tag entry and a BP entry are both located, a tag match is

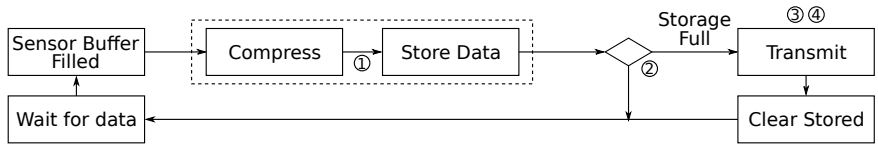


Figure 4.9: Execution flow of a data collection application which benefits from compressed I/O.

confirmed using the *tag way* stored in each BP entry as well as the *block tag* from the physical address. If these comparisons all match, both tag and data have been successfully located.

L^{2}C uses a single tag to represent each contiguous 1kB region of physical memory, in both compressed and uncompressed forms. The tag entry is extended with additional fields to support the two block sizes. A four-bit mask indicates which s-blocks are present in the LLC. An 8-bit counter field is used to indicate the number of data entries present for each compressed block (four 2-bit counters for s-blocks or a single 3-bit counter for an L-block).

Compressed data has the potential to offer greater utility compared to their size. To exploit this, replacements are performed with a modified Least-Recently-Used (LRU) mechanism. When an uncompressed cache line is updated (via write-back from the L2 cache), its LRU is normally updated to record that it has been used recently. If the tag entry indicates that a compressed copy of the same block is present in the cache, the LRU counter of the *compressed block* is updated in stead of that of the UCL. This way, compressed blocks are prioritized over their uncompressed (and redundant) counterparts during cache replacements.

The decoupled sectored cache organization allows L^{2}C to store any combination of compressed and uncompressed data on-chip. The accompanying metadata enables lookups of compressed data, increasing the effective capacity of the cache. As an additional benefit, this enables the reuse of compressed blocks, thus amortizing their memory traffic overhead.

4.3.7 I/O Compression

The placement of the L^{2}C compressor, attached to the on-chip interconnect and next to the Direct Memory Access (DMA) controller, also enables the compression of I/O traffic. L^{2}C can direct through the compressor any data transfer between two memory-mapped regions. In DMA-capable systems, the on-chip DMA controller is programmed to initiate the data movement, while in systems without DMA, a processor core performs this task. This covers both data input (e.g sensor devices) and bidirectional devices (e.g. local storage, network interfaces). L^{2}C enables transparent compression at high bandwidth.

I/O-heavy applications which can benefit from compression include data aggregation services and remote sensor networks. These networks typically consist of low-power devices with limited performance and communication resources. Nodes of this type are strongly power constrained, and may rely on a small battery and unreliable power harvesting techniques (e.g. solar cells, RF energy harvesting). For this reason, energy efficiency is a high priority. The device typically spends as much time as possible in a low-power state,

periodically waking up to collect and transmit data.

Figure 4.9 illustrates the execution flow of a simple embedded application. Data is collected and buffered in an off-chip sensor, while the processor itself is in a low-power sleep state. An interrupt wakes the processor when the buffer is full. The processor triggers a data transfer (via DMA or software mechanisms) to bring the sensor data on-chip. The data is stored in persistent storage, and the processor returns to its sleep state. When local storage is full, a batch of data is transmitted via radio for central aggregation. The benefits of data compression in such a system are fourfold:

① Execution time is reduced, allowing longer sleep periods.

② Longer periods of data can be logged in local storage, reducing the frequency of transmission.

③ Radio transmission and relay energy is reduced, due to smaller payloads.

④ Radio bandwidth is saved.

The data transfer from sensor to processor, be it via DMA or software mechanism, is uncompressed at the source, but passes through the compressor after arriving on-chip. As a result, the data is compressed before being written to storage, saving both time ① and storage space ② . In addition, this allows the collection period to be extended before local storage space is exhausted. Once it is, energy ③ and bandwidth ④ savings are compounded; less frequent radio transmissions, each containing more sensor data.

In addition to these benefits, digital sensors for natural phenomena (e.g. air pressure, temperature, pollutants, radiation) have finite precision, introducing some amount of quantization during data acquisition. Lossy compression can be used to exploit this approximation tolerance.

By placing the L^2C compressor appropriately, compression can be applied to memory-mapped peripherals such as built-in sensors. Attaching the compressor to the on-chip interconnect as illustrated in Figure 4.1 also allows compression to be applied to external peripherals.

I/O compression differs from memory compression by the property that data is compressed *exactly once*. As a result, block type transitions will never occur. For this reason, the I/O compressor does not need to prioritize L-blocks over s-blocks. Instead, both lossy and lossless compression are attempted, choosing whichever achieves a better compression ratio.

4.4 Evaluation

In this section we evaluate the efficiency of L^2C. We first describe our experimental setup, detailing the system configuration of our experiments and the benchmarks used. Two separate evaluations are described: one applying L^2C for memory compression and one for I/O compression. Then, experimental results from each evaluation are presented.

Table 4.1: Simulation parameters.

(a) System parameters.

Parameter	Configuration
CPU	4 core, O-o-O, 4-way issue @ 3.2GHz
L1 cache	64kB per core, 4-way, 1 cycle latency
L2 cache	256kB per core, 8-way, 8 cycle latency
L3 cache	4MB shared, 16-way, 15 cycle latency
Main Memory	16GB DDR4, 1 channels, 800MHz
VFT	7kB, 8-way, 16-bit values

(b) Compressor properties.

Parameter	Compressor	Decompressor
SC^2 latency	18 cycles	42 cycles
SC^2 leakage power	33.6mW	0.4mW
SC^2 dyn. energy	0.576nJ	0.592nJ
MemSZ latency	16 cycles	8-16 cycles
MemSZ leakage power	28.8mW	144.5mW
MemSZ dyn. energy	3.94nJ	17.5nJ

4.4.1 Experimental Setup

Our evaluation of L^2C is twofold. First, we evaluate its use as a Memory Compression scheme, using a processor and memory simulator. Separately, we evaluate the potential of L^2C as a I/O Compression scheme by applying it to a selection of real-world datasets.

4.4.1.1 Memory Compression

We evaluated L^2C for memory compression in an in-house simulator, implemented on top of Pin [48]. The simulator employs an interval-based processor model, as proposed by Genbrugge et al. [49]. The memory hierarchy was modelled at cycle granularity, using DRAMSim2 for main memory [50]. McPAT [51] and CACTI [52] were used to model power and latency of the system considering 32nm technology. The MemSZ compression hardware modules were implemented in RTL, synthesized using Synopsys Design Compiler to determine their operating frequency, latency and power consumption; the same parameters for SC^2 are taken from [72] which were measured with the same technology node. These factors are used as configuration information for the simulations. The general properties of the simulated system are listed in Table 4.1a. The power and latency of each compressor are outlined in Table 4.1b.

As explained in Section 4.3, the developer is responsible for the annotation of *approximable* data structures. For this evaluation, we manually add annotations to the source code of each benchmark based on experimentation to find safe approximations. Table 4.2a summarizes the type of approximated data for each application.

In order to emulate the impact of the approximations on the overall application error, we emulate not only the memory accesses but also update the

values of the memory contents accordingly. This is done by applying a software implementation of the compression and reconstruction methods to the data. Lossless compression is applied to all non-code pages mapped into the process. This includes heap, stack, and data segments of the application itself, as well as those of shared libraries.

Besides the baseline system, L^{2}C is further compared with (i) the lossy-only *MemSZ* [76] and (ii) a variation using only lossless SC^2 compression (*Lossless*). As all three compressing systems use the same decoupled sectored cache design, they are configured identically apart from the employed compression mechanism. This similarity allows the isolation of lossy compression, to study its impact compared to a system with only lossless compression capability.

Each simulation is executed in the following steps: i) A *warmup* period of 50M instructions is carried out to warm up the cache hierarchy; ii) at the end of this warmup period, 10% of the compressible system memory is randomly sampled to train the SC^2 and populate the VFT. This emulates a longer sampling period. Furthermore, all compressible data in memory is compressed at the end of the warmup period, simulating an application with compressed input data; iii) the application is executed until it has finished generating output data.

One common source of memory traffic in scientific workloads is *checkpointing*. Checkpoints are occasional snapshots of the application's state, for the purpose of resuming execution after errors or outages. Such snapshots generate large bursts of data transfers to non-volatile storage, and contain approximable data from the application's working set. To reflect the effect of compression on these data, iterative benchmarks with checkpointing support have it enabled as indicated in Table 4.2a.

The input data sets used for our experiments are the standard input data sets provided with the benchmarks with the exception of (i) *lattice* for which we used a silhouette of a car as the input data set, and (ii) *k-means* where the input is topological data [58].

Compression metadata has been identified as a significant source of memory traffic [25]. To evaluate this factor, our simulations include both the traffic of regular page table information (via TLB misses) and the additional transfer of essential compression metadata.

Benchmarks for approximate computing (AxBench) considers 10% relative output error [59]. Due to its strongly application-dependent nature, it is solely up to the application provider to define what is an acceptable error level. We evaluate and present output error using the mean relative error across the output dataset. The only exception to this is *k-means*, whose output is discrete and strongly bounded. For this application we normalize each individual error to the maximum possible error for that value, such that the maximum possible error is 100%. Similar to previous works, L^{2}C provides tunable knobs to control the data approximation error and constrain application output error. These knobs allow an application provider to adjust the trade-off between output error and performance/energy improvement. Specifically, two quality thresholds are configurable. One is local to each compression attempt, controlling which values are outliers. The second is maintained over the entire execution time, limiting accumulated approximation error.

Table 4.2: Workloads used to evaluate L²C.

(a) Benchmark Applications.

Application	Approx.	Output	Footprint / core	Checkp.	Description
heat [53]	Temps	Temps	8.3MB	✓	Heat propagation through a 2D field
lattice [54]	P and M	Vel.+Press.	5MB	✓	2D Lattice-Boltzmann simulation of air flow
lbm [55]	Velocities	Velocities	325MB		3D Lattice-Boltzmann simulation of fluid flow
orbit [56]	Phys. data	Phys. data	10MB	✓	3D simulation of the two-particle orbit problem
cdelta [56]	Phys. data	Phys. data	22MB	✓	Delta-function heat conduction model
sedov [56]	Phys. data	Phys. data	12MB	✓	Sedov explosion model
windt [56]	Phys. data	Phys. data	23MB	✓	Windtunnel with a step
kmeans [57]	Topol. [58]	Clusters	5.5MB	✓	Iterative clustering algorithm
wrf [55]	Geo data	Temp.	90MB		Weather forecasting model

(b) Datasets used to evaluate L²C for Link Compression.

Dataset	Domain	Type	Size	Description
height [58]	Geo survey	2D spatial	1024×1024 samples	Geographical height map
aqua [95]	Geo survey	2D spatial	$8 \times 512 \times 1024$ samples	Sea surface properties
gb6 [96]	Astronomical survey	2D spatial	2048×2048 samples	Radiotelescope imagery
strang [97]	Geo survey	Time series	187176 samples	Solar radiation measurement at $60°N15°E$
hand [98]	HCI	Time series	80×400000 samples	Hand positions for gesture detection
mitbih [99]	Medical	Time Series	$9 \times 2 \times 650000$ samples	Two-channel ECG recordings
ampds [100]	Energy distribution	Time series	12×1051200 samples	Energy consumption data from a residential building
air [101]	Meteorological	Time series	13×121641 samples	Air quality measurements
gas [102]	Scientific	Time series	19×786432 samples	Carbon monoxide sensor in physics experiment
hydra [103]	Mechanical	Time series	18×1048576 samples	Condition monitoring of hydraulic system

4.4.1.2 I/O Compression

The benefits of I/O compression (reduced execution time, reduced communication duration, reduced communication bandwidth, improved storage efficiency) are directly proportional to the achieved compression ratio. For this reason, we evaluate the use of L^{2}C as a I/O compression scheme by applying it to a selection of real-world datasets as outlined in Table 4.2b.

The datasets can be generally divided into two categories: Spatial and Time series. Spatial data represent a snapshot of samples from different locations, such as a topological survey. This type of data is typically seen at centralized collection points, such as coordinating nodes or database servers, where data are collated from multiple distributed sources. Time series represent multiple samples from the same sensor, such as a continuous energy consumption measurement. This type of data is typically seen in the individual sensor node, such as an implanted medical device.

To evaluate the efficiency of L^{2}C for I/O compression, each dataset is compressed using the three evaluated compression schemes: *Lossless*, *MemSZ* and *L^{2}C*. We present the achieved compression ratio of each system as well as the resulting approximation error.

4.4.2 Results

In the following section we present the results of both evaluations. First, we show detailed statistics acquired from simulations of memory compression. Subsequently, we show the compressibility of the datasets used to evaluate L^{2}C for I/O Compression.

4.4.2.1 Memory Compression

The primary characteristic differentiating the various compression schemes is the achieved compression ratio for any given dataset. Table 4.3a shows the compression ratio of each application's footprint at the end of execution. While neither lossy nor lossless alone show a clear advantage, it is clear that a hybrid approach is able to reap the benefits of each. L^{2}C consistently achieves a higher compression ratio than either of the two competing designs. Table 4.3b shows the compression ratio for the approximable subset of the footprint. We observe that lossy compression is up to 7 *times* more effective than lossless compression for the annotated data. MemSZ does, however, leave blocks uncompressed if they fail to meet quality requirements under lossy compression. L^{2}C falls back to lossless compression for these blocks, achieving a higher overall compression ratio. This effect is most pronounced in *lattice*, where L^{2}C achieves a 49% higher compression ratio compared to lossy compression alone.

The main benefit of memory compression lies in reduced traffic on the main memory bus. Figure 4.10c shows the total memory traffic for each design, normalized to the traffic of the baseline system. Traffic is broken down by data type: non-approximable data, approximable data, page table traffic, and metadata traffic. We find that metadata traffic comprises at most 3.9% of total traffic, twice as much as the regular page table traffic. On average, L^{2}C reduces the total traffic volume by 73%. This is an improvement of 18% compared to MemSZ and 56% over Lossless.

Table 4.3: Compression efficacy of the three memory compression systems.

(a) Compression ratio, all data.

	heat	lattice	lbm	orbit	cdelta	sedov	windt	kmeans	wrf	GM
Lossless	2.5×	2.5×	2.2×	3.1×	2.9×	3.4×	2.7×	1.9×	1.5×	2.4×
MemSZ	1.5×	1.1×	4.8×	1.8×	1.1×	1.3×	1.0×	1.3×	1.2×	1.5×
L²C	3.2×	2.6×	7.2×	4.1×	3.1×	4.3×	2.8×	2.5×	1.6×	3.2×

(b) Compression ratio, approximable data.

	heat	lattice	lbm	orbit	cdelta	sedov	windt	kmeans	wrf	GM
Lossless	2.8×	1.9×	2.2×	3.7×	2.6×	3.7×	2.1×	2.3×	3.0×	2.6×
MemSZ	15.9×	5.1×	15.9×	14.9×	9.2×	15.8×	15.9×	3.6×	4.4×	9.6×
L²C	16.0×	7.6×	15.9×	14.9×	9.2×	15.8×	15.9×	3.9×	5.3×	10.4×

(c) Mean relative application output error.

	heat	lattice	lbm	orbit	cdelta	sedov	windt	kmeans	wrf
Lossless	0%	0%	0%	0%	0%	0%	0%	0%	0%
MemSZ	0.12%	0.24%	0.05%	0%	0.01%	0%	0%	0.05%	<0.01%
L²C	0.13%	0.25%	0.06%	0%	<0.01%	0%	<0.01%	0.05%	<0.01%

(d) Fraction of memory traffic caused by L²C block transitions.

	heat	lattice	lbm	orbit	cdelta	sedov	windt	kmeans	wrf
L²C	0.000%	2.207%	0.000%	0.000%	0.006%	0.000%	0.000%	0.001%	0.064%

One potential cause of traffic overhead is the transition from multiple s-blocks to a single L-block. To attempt such a transition, multiple s-blocks must be read from main memory. Table 4.3d shows the fraction of total memory traffic caused by such reads. The maximum 2.2% is found in *lattice*, while the remaining benchmarks see at most a fraction of a percent of overhead.

The reduced traffic on the main memory bus yields lowered latency for memory accesses, which is particularly important for memory reads. Figure 4.10d shows the Average Memory Access Time (AMAT) for instructions with memory input operands, normalized against the baseline AMAT. On average, L²C reduces baseline AMAT by 36%, improving on MemSZ by 5% and Lossless by 17%.

Another benefit of the three compressing designs is that they are able to maintain compressed data in the LLC, increasing its apparent capacity. Figure 4.10e shows the LLC Misses per Kilo-Instruction (MPKI) normalized to the baseline system. L²C reduces average MPKI by 69%. This is a 16% improvement over MemSZ and 49% over Lossless.

Execution time is affected both by the reduced memory latency and the improved LLC miss rate. Figure 4.10a shows the execution time achieved by each system, normalized to that of the baseline system. We observe that L²C equals or surpasses both competing designs in all tested applications. L²C reduces execution time by an average 50%, improving on MemSZ by 9% and Lossless by 26%.

The reduced execution time coupled with reduced DRAM activity translate into a reduction of total system energy. Figure 4.10b shows the total energy consumption of each design, broken down by system component. The energy consumption follows the same trend as memory traffic, with L²C achieving an

average reduction of 16%. This is 3% and 5% better than MemSZ and Lossless, respectively. Notably, Lossless is closer in energy consumption than the other metrics, owing to the less complex compressor/decompressor.

Finally, each application's output error is presented in Table 4.3c. We find that for the majority of the benchmarks, approximation introduces less than 0.05% relative error compared to the baseline output. L^2C differs from MemSZ by at most 0.01%. This is due to cache interference effects causing slight differences in eviction timing, leading to small variations in lossy compression outcome.

Across the tested applications, we see clear indications that the improvements gained by lossy and lossless compression have significant overlap. A hybrid approach is able to achieve the benefits of both methods, where each is most suitable. L^2C surpasses MemSZ by also compressing the non-approximable traffic, and outperforms Lossless by applying more aggressive compression to the subset of data which tolerate it.

We observe that the traffic reduction achieved by L^2C equals or surpasses MemSZ and Lossless in all the tested benchmarks. Of note is that two of the tested benchmarks benefit more from the modest Lossless compression across all data than from more aggressive MemSZ compression on only the approximable subset. This illustrates that the memory footprint of each subset is of lesser importance than the memory activity induced by each. Compression is most beneficial on blocks which normally bounce between main memory and LLC, and this is highly application-dependent.

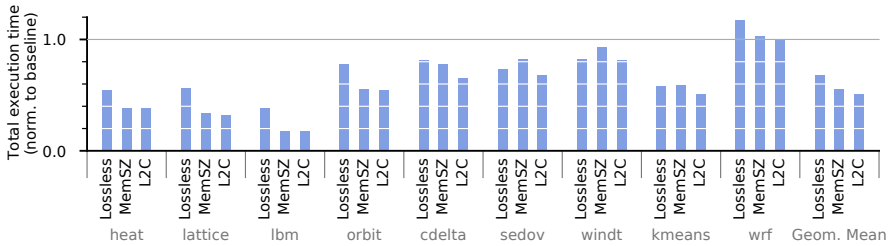
Wrf and *orbit* illustrate a data pattern which defeats the heuristic used by L^2C to determine compressibility of s-blocks. A subset of non-approximable data has interspersed cache lines showing at least 2:1 compressibility, but four-line blocks alternate between being compressible and incompressible. Each time a compressible line is written back to an uncompressed s-block, the block's *back-off* counter is incremented, bringing the block closer to a retry. The result is a large number of *failed* block writebacks which ultimately lead to new *retry* fetches.

Heat, *lattice* and *lbm* make up another interesting subset of applications, those with only or almost only approximable memory traffic. For such applications, the only room for L^2C to improve upon MemSZ is in approximable blocks which have failed lossy compression. As shown in Table 4.3b, only *lattice* has any significant opportunity like this, and L^2C successfully exploits it. *Kmeans* and *wrf* also show MemSZ leaving blocks uncompressed, which are successfully compressed by L^2C.

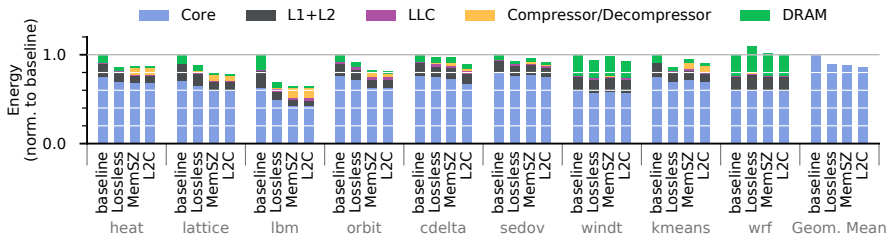
Sedov and *windt* both benefit more from lossless compression than lossy, in terms of memory traffic. This is a by-product of approximation tolerance. While these applications both process a large data footprint of regular data, not all of it is safe to approximate. As a result, a large portion of their memory traffic is compressible but only using lossless compression. In these applications, Lossless performs better than MemSZ, while L^2C capitalizes on the strengths of both.

4.4.2.2 I/O Compression

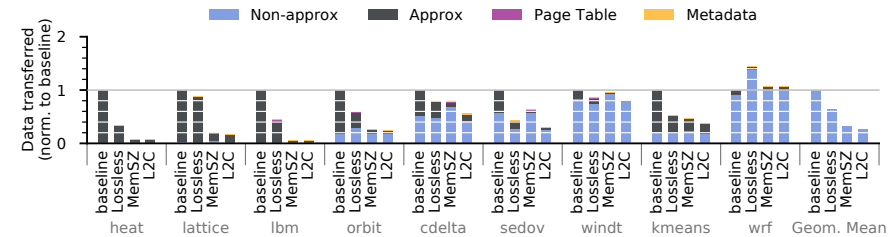
As explained in Section 4.3.7, the primary metric of interest for I/O compression is the achieved compression ratio. Table 4.4a shows the results for the three



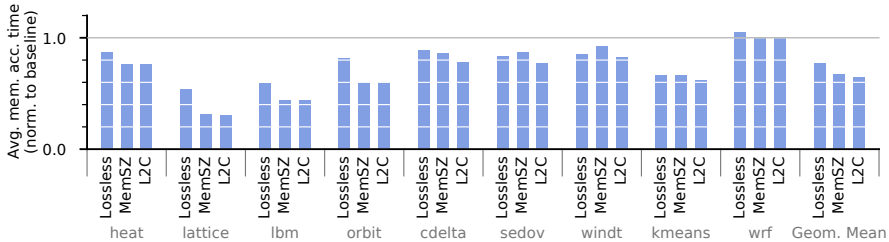
(a) Execution time.



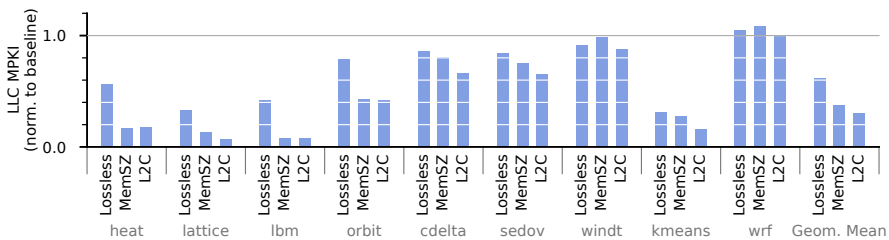
(b) System Energy Consumption.



(c) Memory Traffic.



(d) Average Memory Access Time (AMAT).



(e) LLC misses per kilo-instruction (MPKI).

Figure 4.10: Evaluation of the L²C memory compression design and comparison with competing designs.

Table 4.4: I/O compression efficacy.

(a) Achieved compression ratio.

	height	**aqua**	**gb6**	**strang**	**hand**	**mitbih**	**ampds**	**air**	**gas**	**hydra**	**GM**
Lossless	2.34×	1.54×	1.03×	2.67×	2.25×	1.61×	1.51×	1.44×	1.05×	1.52×	1.62×
MemSZ	3.59×	6.38×	2.48×	1.95×	2.17×	2.03×	7.44×	1.03×	10.35×	8.20×	3.55×
L^2C	3.93×	6.39×	2.48×	2.87×	2.36×	2.31×	7.55×	1.45×	10.35×	8.58×	3.96×

(b) Relative approximation error.

	height	**aqua**	**gb6**	**strang**	**hand**	**mitbih**	**ampds**	**air**	**gas**	**hydra**
MemSZ	0.33%	0.44%	0.33%	0.07%	0.35%	0.32%	0.25%	<0.01%	0.40%	0.36%
L^2C	0.33%	0.44%	0.33%	0.04%	0.09%	0.32%	0.25%	<0.01%	0.40%	0.36%

evaluated compression schemes, *Lossless*, *L^2C*, and *MemSZ*. Due to its hybrid nature, L^2C equals or surpasses MemSZ in all cases. This is because any block which MemSZ can compress successfully will be compressed identically in L^2C. The remaining blocks are guaranteed equal or better compression, since MemSZ leaves them uncompressed while L^2C applies additional compression. The same holds true against *Lossless*. Notably, *strang* and *hand* exhibit better compressibility with lossless than lossy in some blocks. Since L^2C chooses the most effective compressor, it yields a higher total compression ratio than MemSZ. On average, L^2C achieves a compression ratio of 3.96:1. MemSZ manages 3.55:1 and Lossless reaches 1.62:1.

A similar trend is observed in the introduced approximation error. Table 4.4b shows the mean relative error caused by compressing each dataset. In spite of its higher compression ratio, L^2C introduces no extra error compared to MemSZ. This is because all lossily compressed blocks are compressed identically between the two, introducing the exact same error. In *strang* and *hand*, a by-product of selecting lossless compression when beneficial is that error is also reduced. No tested dataset suffers more than 0.4% error.

4.5 Conclusion

L^2C is a hybrid lossy/lossless memory and I/O compression scheme, the first of its kind. It combines general-purpose lossless compression with state-of-the-art lossy compression to improve the bandwidth efficiency of both the system memory bus and processor I/O traffic. In memory compression experiments in a system with 4GB of 800MHz DDR4 per core and 1MB of LLC space per core, L^2C achieves average memory-footprint compression of 3.2:1 across all benchmarks (up to 7.2:1 on a single one), improving by 33% over a pure-lossless solution. On approximable data, L^2C achieves an average compression ratio of 10.4:1 (up to 16:1), which is an 8% improvement over the current state-of-the-art lossy memory compression. Furthermore, compared to the best previous work, L^2C reduces off-chip memory traffic at least by 18%, execution time by 9% and total system energy by 3%. When applied to a set of real-life datasets for I/O compression, L^2C achieves an average of 4:1 compression, surpassing lossy and lossless single-method compressors by 10% and 241%, respectively.

Chapter 5

FlatPack: Flexible Compaction of Compressed Memory

In addition to memory bandwidth, as tackled by AVR, MemSZ and L^{2}C, memory capacity is a critical resource in modern systems. Memory capacity must be sufficient to avoid frequent page faults and its bandwidth high enough to accommodate the rates of requested data. The demand for both memory capacity and memory bandwidth is increasing as applications become more data-intensive and a larger number of cores is integrated on a single chip. However, simply scaling up memory size and bandwidth increases system cost and power consumption [1].

Data compression has the potential to offer a better cost-performance tradeoff in computing systems by more efficiently utilizing the capacity and bandwidth resources of main memory. Previous techniques are able to achieve improvements in either capacity or bandwidth but usually not in both. Some designs (such as AVR, MemSZ and L^{2}C) use *memory compression* to reduce memory traffic without considering capacity improvement [67, 76]. Others aim primarily at *memory compaction* to increase capacity [13, 16], exploit free prefetching effects [13, 104], but introduce significant traffic overheads to manage compacted memory [16].

There are several reasons for the traffic overheads of managing a compressed and compacted memory. First, compressed blocks require additional *metadata* to be accessed. Second, compressed blocks have variable size and therefore may cross the boundaries of a regular memory location requiring two *split accesses*. Even if they do not span across the access boundaries, writing a compressed block back to memory often requires a *read-modify-write (RMW)* operation in order to preserve data of neighboring blocks. Another source of traffic overheads and inefficiency is the *change in compressibility of data* and hence in their size during execution, i.e., blocks growing (or even shrinking) during runtime. This variation in compressibility leads to inefficiencies in existing compaction systems, which compact blocks into sequential spaces of rigid size. A growing block may be stored uncompressed as an *exception*, in space specially reserved

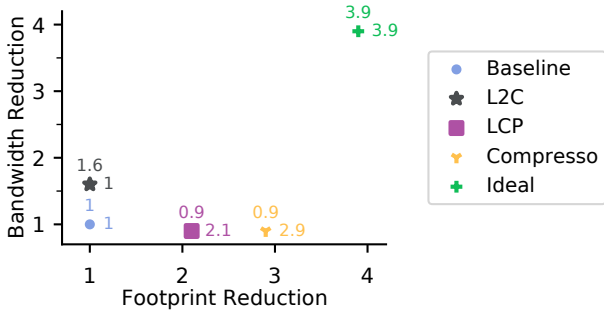


Figure 5.1: Bandwidth reduction vs. Footprint reduction for three state-of-the-art memory compression systems with identical lossless compressors. *Ideal* shows the achievable compression, *LCP* and *Compresso* are compacting systems, L^2C performs no compaction.

for this purpose [13,16]. Alternatively, the change causes a *page overflow* which requires the entire page to be brought on-chip, repacked and migrated, inducing a memory traffic overhead. In our experiments, an average of 15% of cache line updates lead to an increase in compressed size, causing an *exception* at the expense of memory capacity and traffic.

Although some of the above issues, e.g. metadata [25], have been addressed in the past, the bandwidth benefits - if any - of current state of the art memory compaction designs are far from the achieved raw compression ratio. As illustrated in Figure 5.1, the required memory traffic of such designs is comparable to the traffic of a baseline with no compression, 2×-4× higher than the theoretical minimum indicated by the achieved compression ratio. In effect, existing memory compaction techniques consume significant bandwidth and hence limit system performance and energy efficiency.

This chapter introduces FlatPack, a novel technique aimed at reducing the memory bandwidth overheads of memory compaction. The key observation behind its design is that existing systems are unable to efficiently handle dynamically varying compressibility of data, which leads to excessive page overflows and hence excessive memory traffic. FlatPack mitigates this problem allowing compressed blocks to be fragmented within a page and share expansion space providing the flexibility to be reorganized independently without disturbing other blocks. FlatPack's flexible reorganization can be performed in response to all size changes without introducing additional data movement. Furthermore, by fragmenting blocks at the Memory Access Granularity (MAG), FlatPack reduces RMW traffic, since most compressed memory writes only affect one block. FlatPack's reorganization is performed in hardware by the memory controller, without intervention by system software or the operating system. In effect, FlatPack's flexibility to handle variability in block size reduces data movement and memory traffic improving system performance and energy efficiency.

Concisely, FlatPack is a novel flexible memory compaction approach that aims to reduce the traffic overheads and makes the following contributions:

- a flexible format of compressed pages that allows compressed blocks to be fragmented and share expansion space in the physical memory region

of the page offering efficient memory compaction.

- a hardware mechanism that enables the memory controller to exploit the above format and dynamically reorganize data within the page, without software intervention and with minimal data movement.
- a thorough evaluation of FlatPack and comparison with current state of the art memory compaction designs to measure the significant reduction in memory traffic and impact of FlatPack in performance and energy efficiency of the system.

The remainder of this chapter is organized as follows. Section 5.1 discusses background and related work. Section 5.2 describes the proposed FlatPack architecture. Section 5.4 presents evaluation results and Section 5.5 draws our conclusions.

5.1 Background and Related Work

A number of systems have been proposed to compress and compact main memory, to save bandwidth or increase memory capacity. Any memory compression or compaction design is subject to a number of design choices, which are detailed below. The two designs currently at the forefront of memory compaction are Linearly Compressed Pages (LCP) [13] and Compresso [16]. This section outlines the design parameters of these and other related approaches, as well as a summary of the design choices employed by FlatPack.

5.1.1 Compression Algorithm

A number of algorithms have been proposed for compression in the memory hierarchy. The primary requirement for a suitable compression algorithm is low decompression latency, to minimize performance impact. Lossless compression schemes typically offer compression ratios up to 2× to 4× on real-world data [18,41,69–72]. For applications which tolerate approximation, lossy compression offers more aggressive compression ratios of up to 16× [67, 76, 93] or allows for bandwidth optimizations in combination with lossless compression [94]. Deduplication has been proposed as an alternative to compression [105]. The more complex Lempel-Ziv algorithm has also been employed, using an additional cache to hide its decompression latency [10].

FlatPack is compatible with any block compression algorithm, without loss of generality. The system is evaluated here with the SC^2 compression scheme, which offers a competitive compression ratio and low-latency operation [72].

5.1.2 Compression Granularity

One way to differentiate memory compression systems is based on compression granularity, i.e. the size of the data block being compressed as one unit. Two basic approaches are possible, each with different characteristics. *1) Compressing individual cache lines* at the granularity of the Last Level Cache (LLC) [12–16, 27, 93, 105, 106]. The benefit of this is that there is no overhead from fetching unused compressed data. On the other hand, general purpose

DRAM is restricted to a minimum Memory Access Granularity (MAG), which is typically tuned to be the size of one cache line. As a result, single cache line compression has limited potential for bandwidth reduction. Similarly, the MAG prevents individual compressed blocks from being updated in memory, forcing the memory controller to perform a Read-Modify-Write (RMW) sequence. *2) Compressing multiple cache lines* together, making up a larger compression block [10, 11, 17, 26, 67, 76, 94, 107]. The benefit of this is that the compressed block may exceed the MAG, and thus memory transfers are more efficiently utilized. Conversely, these systems do not support random access of individual cache lines within a compressed block. This complicates writebacks to memory and enforces a form of prefetching of all co-compressed cache lines.

LCP and Compresso both compress single cache lines. FlatPack compresses blocks of four cache lines, in order to be able to improve memory bandwidth utilization and reduce the need for RMW operations. In addition, the larger compressed blocks are key to enabling a flexible approach to block compaction.

5.1.3 Block Compaction

While memory compression may give a bandwidth benefit, capacity improvement requires compressed blocks to be *compacted* in physical memory. Some systems forgo compaction altogether, aiming only to improve bandwidth utilization [26, 27, 67, 76, 93, 94] or to make space for error correcting codes [106]. Several compaction approaches have been proposed in literature. *1) LCP packing* is the simplest form of compaction, assigning an identically sized space for each block within a page [13, 15, 17]. This simplifies address calculation, at the expense of wasted space for blocks with higher compressibility. *2) Line Packing* is employed by Compresso and others, packing the compressed blocks of a page together while supporting more than one block size [12, 14, 16, 107]. This allows for greater benefit when compressibility varies, eliminating more wasted space. *3) Block Fragmentation* as employed by MXT and related designs compresses very large blocks (1kB) and fragment compressed blocks in fixed-size sectors in memory without limitation to their page sharing relationships [10, 11]. Such free placement yields a large number of physical memory addresses stored as metadata, which can be a significant overhead. *4) BCD Deduplication* divides physical memory into several large arrays, each storing a different part of all compressed blocks [105].

FlatPack allocates a contiguous space for each logical page, thus keeping a single physical address in its metadata. Compressed blocks are fragmented at MAG granularity and placed freely within that space. The fragmentation and location of compressed blocks is dynamic, automatically adapting to the changing compressibility of data. Furthermore, it allows blocks within a page to differ in size. This automatic reorganization of blocks is performed by the memory controller, in hardware. Since it occurs only on block writeback, blocks are reorganized without any additional data movement. FlatPack stores the first part of each compressed block in a fixed location within the page, which allows memory access to begin in parallel with further address calculation for the compressed block.

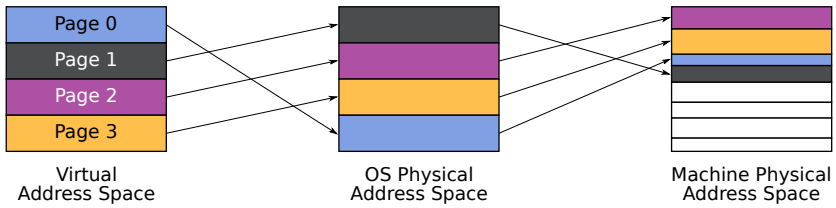


Figure 5.2: The three address spaces used for memory compaction. A regular system uses the OSPA space for physical memory.

5.1.4 Address Translation and Page Compaction

A standard, uncompressed memory hierarchy has one level of address translation. The *Virtual Address* (VA) space of each process is translated into the *Operating System Physical Address* (OSPA) space, where pages are the same size and uncompacted. Any memory compaction system must modify or supplement the address translation by introducing an additional *Machine Physical Address* (MPA) space, as illustrated in Figure 5.2. This is a necessity to support blocks or pages of non-uniform size and gain memory capacity.

The implementation of this additional MPA space depends on the exact organization of compacted memory. Two principal approaches are taken by existing systems, illustrated in Figure 5.3. *1) Contiguous pages* are used by LCP and others, allocating contiguous memory space for a single compressed page, smaller than the system's normal page size [12, 13, 17]. This approach requires only one MPA to locate each page, but relies on a relatively static compressibility. Reduced compressibility can cause page overflows, requiring the full compressed page to be migrated to a larger allocation. *2) Fragmented pages* is an alternative approach, where a compressed page is allocated as a set of smaller chunks. The benefit of this organization is the ability to dynamically append chunks of physical memory to a given compressed page [14–16]. Employed by Compresso and others, this approach is better able to deal with compressibility changes, but requires more metadata to locate multiple chunks in physical memory.

FlatPack uses fixed contiguous compressed pages for their simplicity and reduced metadata. Compressibility changes are handled by improved flexibility in block placement within each compressed page. By allowing blocks to grow and shrink, FlatPack reduces the number of page overflows.

5.1.5 Metadata Handling

All compression schemes require additional metadata to manage compressed blocks and pages. This metadata describes block sizes, compression methods, placement in physical memory and other auxiliary compression information which is not application data. A number of approaches exist for organization and transfer of compression metadata. *1) Metadata in main memory, separated from the compressed data* is the most common organization [10, 11, 15–17, 26, 93, 105]. An on-chip metadata cache is typically employed to reduce the traffic and latency overhead of finding the metadata for accessed pages. To further reduce latency, the metadata table can be accessed in parallel with the page table [12, 67, 76, 107]. *2) Metadata embedded in the compressed block* has

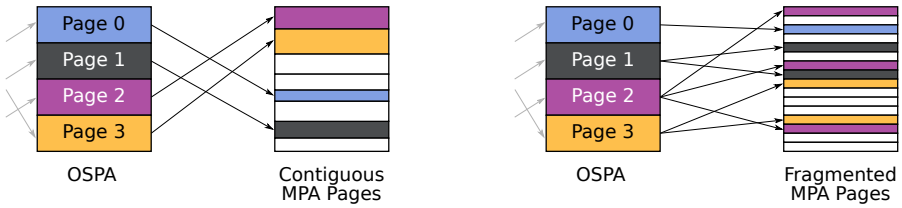


Figure 5.3: The two principal approaches to page compaction. Contiguous pages require a single MPA, fragmented pages may require multiple.

been proposed as a method to reduce the bandwidth overhead of metadata traffic [25,27,94,106]. This is unsuitable for memory compaction, as it decouples the page-level metadata from some of the blocks within the page. *3) Metadata embedded in the compressed page* adds this ability, and avoids fragmenting the physical address space [13, 14].

Compresso uses a separate metadata table accessed on demand, LCP embeds metadata in its compressed pages. Both designs employ a small metadata cache.

FlatPack uses a separate metadata table, accessed in parallel with the page table. An on-chip metadata cache is kept updated in tandem with the TLB. This approach guarantees that when requests reach the LLC or memory controller, the block and page compression metadata is available on-chip.

5.1.6 Last-Level Cache Support

A number of memory compression systems also modify or use the LLC in ways designed to optimize or support memory compression. One approach is to unify LLC and main memory compression by storing only compressed blocks in the cache [11, 105]. This increases the effective capacity of the LLC, without additional SRAM cost. Another approach is to allow the LLC to store compressed blocks alongside regular uncompressed cache lines [26, 67, 76]. This allows large-block memory compression schemes to offset the traffic overhead of reading large blocks from memory and benefit from spatial locality. Compression systems which compress at the granularity of single cache lines may pack multiple of these together, in order to satisfy the memory access granularity. Such extra cache lines can be decompressed into an otherwise unmodified LLC [13, 16] or a small special-purpose cache [14] in order to gain some bandwidth benefits. LCP and Compresso both work with an unmodified LLC, and insert all valid overfetched data. IBM MXT adds an off-chip cache level in order to hide the latency of decompression of its large blocks.

FlatPack compresses blocks consisting of multiple MAGs of data. To mitigate the traffic overhead of LLC misses, it employs a modified Decoupled Sectored Cache [35] to co-locate both compressed blocks and uncompressed cache lines in the LLC similarly to AVR, MemSZ, and L^{2}C [67, 76].

5.1.7 Overheads of Existing Systems

Current state of the art memory compaction systems, Compresso and LCP, suffer from two primary types of overhead. First, rigid and static assignment

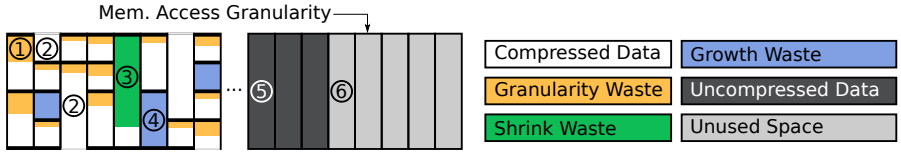


Figure 5.4: A classification of memory capacity waste for a single physical page in a memory compaction system. The unused space at the end may be used for future uncompressed data.

of compressed space for each block leads to deteriorating compaction as compressibility changes. While Compresso allows for more than one block size per page, it is unable to handle blocks changing size over time. Growing blocks are left uncompressed while shrinking blocks continue occupying their initially assigned space. Over time, this deterioration leads to a page overflow forcing the system to recompress and recompact the page. The second major source of overhead is the compression granularity of single cache lines. Compresso and LCP both fetch individual compressed cache lines from memory on an LLC miss, which is rounded up to the memory access granularity (MAG). As a result, one full MAG is transferred from memory for each cache line, regardless of compressibility. In some cases, this overfetching can be beneficial, if it contains additional complete compressed cache lines. This gives a modest prefetching effect, which can offset part of the bandwidth overhead.

Figure 5.4 shows an example of a compacted page using the Compresso compaction scheme. Each compressed block is assigned a space of one of a few fixed sizes and stored there. The illustration is also applicable to LCP which uses a similar layout but maps all compressed blocks to a single size. This rigid granularity leads to some wasted capacity (*granularity waste* ①), since the actual size of the block is likely to be smaller than the assignment. A sub-MAG block size prevents a compressed block from being updated in a single operation, requiring a Read-Modify-Write sequence. If a compressed block crosses a MAG boundary ②, it requires two *split memory accesses* to read or modify. If a block's compressed size shrinks during execution, the same assigned space is used. This leads to additional wasted capacity (*shrink waste* ③). If a block's compressed size grows, it cannot be stored in the existing space assigned to it. That space cannot be eliminated without additional data movement, and thus remains unused (*growth waste* ④). Then, the data for the block is stored uncompressed in a dedicated separate region of the page ⑤. Finally, the physical space allocated to a compressed page is also limited to a set of fixed sizes, leading to allocated but unused space ⑥. This overallocated space is not wasted, but kept in reserve for block growth. If this space is exhausted, the compressed page overflows and must be migrated.

FlatPack mitigates capacity waste by compressing larger blocks and fragmenting them in physical memory. Each page is divided into *slots* at the memory access granularity. Each compressed block is fragmented into MAG-sized parts and a number of slots are assigned to it. When a block's compressed size changes, slots can be released and reassigned as needed, without additional data movement overhead. This eliminates growth and shrink waste, by dynamically adapting space assignment to each block.

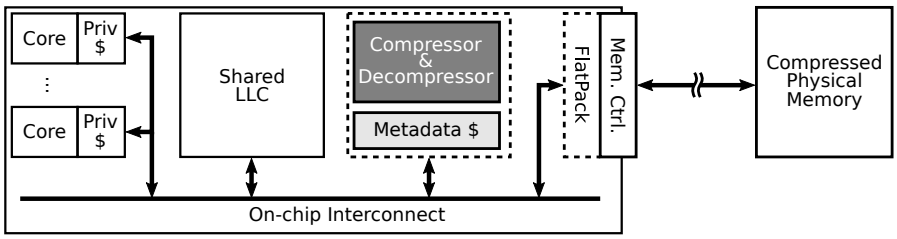


Figure 5.5: The FlatPack memory system that utilizes a modified LLC and adds compressor hardware and a metadata cache.

5.2 System Architecture

Currently published state-of-the-art memory compression systems primarily target one of two bottlenecks of main memory: memory bandwidth or memory capacity. Systems which maintain data compacted in main memory do so at the expense of additional memory traffic [13, 16]. This is mainly due to *data movement*, e.g., page migration required to reorganize a compressed page in order to eliminate fragmentation and handle varying compressibility. Conversely, the most straight-forward compression approach to reducing memory traffic leaves memory capacity unimproved [26, 27, 67]. This eliminates the bandwidth overhead of page reorganization, and simplifies address translation logic.

FlatPack aims to combine these approaches in order to gain the benefits of both. Using a novel compaction scheme, FlatPack is offers memory capacity improvements on par with current state-of-the-art systems. By being flexible to compressibility changes, as well as allowing size skew between blocks, FlatPack is able to dynamically reorganize physical memory on demand with fewer costly page migrations.

The basis of this flexibility is the fragmentation of compressed blocks, at the Memory Access Granularity (MAG) dictated by the physical memory controller. MAG-sized *slots* in physical memory are assigned and reassigned on demand, as blocks shrink and grow. By compressing blocks which are larger than the MAG, the compressed data can be fragmented and flexibly placed. This allows a compressed page to support a wide variety of block sizes, and allows blocks to change size over time as long as the average compressibility across the page does not increase. Crucially, blocks are able to change size independently of each other. Reorganization of a block is performed on demand whenever the block is written back to memory, and thus introduces no additional data movement.

Figure 5.5 shows a top-level overview of the FlatPack architecture. A specialized compressor and decompressor hardware module is added, as well as a small on-chip metadata cache. The shared Last-Level Cache (LLC) is designed to store both uncompressed cache lines and complete compressed blocks, using a Decoupled Sectored Cache organization [35]. The flexible packing and organization is performed in hardware by a module situated on the core side of the Memory Controller. This module implements a Finite State Machine which receives FlatPack block and page operations from the LLC and uses metadata to translate them into individual requests to the standard memory controller. Allocation of physical memory on the page level

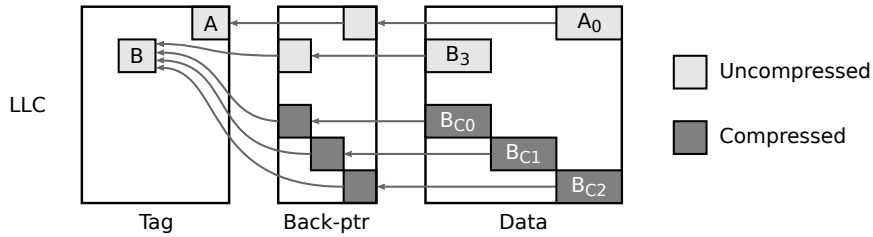


Figure 5.6: The Decoupled Sectored Cache design employed by FlatPack. The uncompressed cache lines B_x share a tag with the compressed version of the block stored in B_{Cy}.

is performed by a software runtime.

Similarly to other memory compaction systems, FlatPack introduces an additional layer of address translation. Virtual addresses are translated using a Page Table into the OS Physical Address (OSPA) space. OSPA is used for cache tags, but does not represent physical memory. FlatPack organizes data in the Machine Physical Address (MPA) space, using variable-size compressed pages. In MPA space, compressed pages are compacted to maximize the available memory capacity.

This section describes each component of the FlatPack system design in further detail.

5.2.1 Compression

The main feature of FlatPack is the ability to dynamically pack and repack compressed blocks in physical memory on demand. To avoid bandwidth waste, compaction is performed at Memory Access Granularity (MAG), which is typically 64 bytes and equal to an LLC cache line. Lossless compression typically offers between 2× and 4× compression ratio on real-world data [18, 41, 69–72]. In order to generate compressed blocks which can be fragmented into multiple MAGs, FlatPack compresses blocks of 256B into compressed sizes between 32B (1/2 MAG) and 256B. This gives a theoretical maximum compression ratio of 8×.

FlatPack is compatible with any block compression algorithm able to compress 256 bytes of data. In this thesis, we evaluate using SC^2 [72]. SC^2 uses a global Value Frequency Table (VFT) to assign shorter encodings to frequently appearing bit sequences. The VFT content is created dynamically by profiling a small random part of memory, while encoding transition is implemented using a similar mechanism presented in the original SC^2 work. Originally designed for cache compression, SC^2 exhibits high compression ratio for real-life datasets, as well as low-latency compression and decompression. These properties make it suitable for memory compression as well.

5.2.2 Last-Level Cache

The use of multi-MAG memory blocks introduces a potential bandwidth overhead from overfetching. Compressing multiple neighboring cache lines as one

block before writing to memory requires the block to always be transferred in its entirety. Reading multiple MAGs worth of compressed data from memory to serve a single LLC miss trivially leads to additional data transfer(s) compared to an uncompressed system. One way to alleviate this is to store the compressed block on-chip, exploiting spatial locality to allow the same block to serve future LLC misses. Similarly to previous works [67, 76], FlatPack employs a *Decoupled Sectored Cache* (DSC) for this purpose [35]. A DSC decouples the cache tags from the data, allowing multiple cache lines to share a tag. This is accomplished by introducing an array of Back Pointers, each associating a data entry with its tag as illustrated in Figure 5.6. Our implementation is extended with the ability to store both uncompressed single cache lines and compressed blocks, co-located in the same data array similarly to AVR and MemSZ [67, 76]. The main benefit of this co-location is that LLC requests can be served either using uncompressed data (like a regular set-associative LLC) or compressed data, adding decompression in order to avoid a costly memory access.

5.2.3 Lazy Evictions

Another challenge introduced by large memory blocks is updates. As the MAGs within a block are compacted completely, there is no ability to update a single compressed cache line in memory without also updating the full block. As a result, writebacks of dirty lines from LLC directly into compressed memory cannot be performed. In order to recompress a block which is not available on chip, the full compressed block must be read from memory. *Lazy Evictions* mitigate this overhead by delaying the actual recompression using the empty space left in memory by compressed blocks [67]. If there is free space in physical memory, the dirty cache line is written back uncompressed and the block's metadata is updated to reflect this. The next time the block is fetched from memory, all lazily evicted cache lines are also read, and the block is recompressed to include the dirty data. The end result is that the dirty cache line is written to memory once and read from memory once, which is a lesser traffic overhead compared to reading the full compressed block for recompression and then writing it back to memory.

5.2.4 Block Compaction

The variable size of compressed memory blocks necessitates a dynamic method for compaction in physical memory. Assigning a fixed space to each block risks introducing fragmentation as the block changes size. A shrinking block will leave parts of its space unused. A growing block must be relocated, leaving its original space unused. FlatPack breaks this fixed structure by dividing physical space into MAG-sized *slots* and dynamically assigning one or more slots to each block when needed. As blocks shrink and grow over time, their slot allotments change as needed, while unused slots remain available for other blocks within the same page. Crucially, a compressed block which changes size can be moved without affecting other blocks, and thus without additional data movement.

Figure 5.7 shows a FlatPack-compressed page of 16 compressible blocks. The compressed page occupies a section of physical memory, of a fixed size.

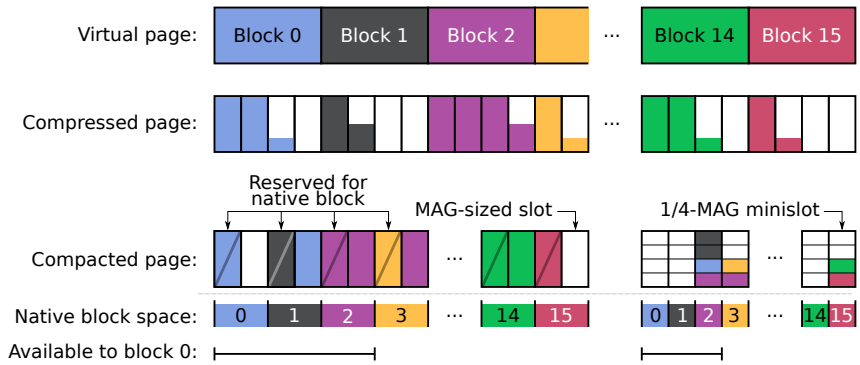


Figure 5.7: The layout of a compacted physical page, compared to an uncompacted compressed page. Slots native to a block N are also available to neighboring blocks N-1 and N-2.

This allocation is divided into MAG-sized *slots*. Each logical block is assigned an equal number of slots where it is *native*. The first of these slots is reserved for the native block itself. Any other slot native to a block N is made available to store the block N or its neighboring blocks $N-1$ and $N-2$ in a circular fashion. For example, parts of block 14 may be stored in a memory slot native to blocks 14, 15 or 0.

By compressing blocks of four MAGs, FlatPack generates compressed blocks which may exceed a single MAG. By dividing the compressed block into MAG-sized pieces, placement is flexible within the block's native space and that of the two following blocks. When a block is written to memory, on-chip metadata allows selection of a suitable slot set for the block. These properties allow a page's physical allocation to be reorganized on the fly without additional data movement.

Another advantage of this organization is that it supports pages where block sizes are non-uniform and variable. A FlatPack compacted page with a fixed size is able to support a wide variety of block size combinations, as well as handling compressibility variation over time. As long as the average compressibility across the entire page remains stable, the memory controller is able to manage individual blocks growing and shrinking without incurring additional reorganization traffic and without intervention by system software.

5.2.5 Minislots

Dividing blocks along the memory access granularity introduces one important drawback, as all compressed sizes are effectively rounded up to multiples of the MAG. In the worst case, this means over-using and over-transferring (MAG-1) bytes per compressed block, wasting memory capacity and bandwidth on the order of one MAG per block. A majority of blocks will have some amount of such *granularity waste*, as it is unlikely that the compressed size will be an exact multiple of the MAG. FlatPack mitigates this problem by supporting finer granularity in a subset of its slots in physical memory. Depending on the allocated physical page size, each block is allotted two or four *minislots*.

Each minislot is one quarter of the size of a regular slot, and made available to neighboring blocks in exactly the same way as regular slots. Figure 5.7 illustrates a compressed page with four minislots native to each block. Block 0 has compressed to two full-sized slots and the remainder fit within a single minislot. Without minislot support, the block would occupy three full-sized slots, with $\frac{3}{4}$ slot granularity waste. By prioritizing minislots in the same MAG as existing data, free minislots are also concentrated, reducing fragmentation. Contrary to full-sized slots, updating a minislot may require a read-modify-write operation.

5.2.6 Slot Assignment

When a block is evicted from the last-level cache, it must be written back to main memory. Due to data being updated, the block may have changed its size. As a result, physical packing of compressed blocks is updated on block writeback.

As explained in Section 5.2.4, the space of a compressed page in physical memory is divided into *slots*, with each slot being assigned to at most one block. Any given block is able to make use of slots from three separate locations: its own native space, as well as the respective native spaces of the following two blocks. For example, block 5 of a page is able to use the slots native to blocks 5, 6, and 7. The very first slot native to block 5 is reserved for that block, and thus always contains the first part of the block's data. Successive MAG-sized parts of the block are placed greedily in the first available slots. The fixed use of the first slot allows memory access to begin immediately on a cache miss, in parallel with address calculation for any remaining data.

After placing all full MAG-sized parts of the block in full-sized slots, a remainder is likely to exist which requires less than a full slot of physical space. To reduce granularity waste, FlatPack attempts to place this remainder data in one or more of the *minislots* reserved at the end of the page. Similarly to the full-sized slots, each minislot has one *native* block, and is made available to the native block and the two preceding ones. Placement of remainder data in minislots prioritizes data packing, to leave as many slots as possible unoccupied. Since the block is packed and written to memory in its entirety, there is no need for additional metadata to maintain the order of data within a block; slot are filled in their logical order.

5.2.7 Page Compaction

In order to maximize capacity gains, a memory compaction system is designed to minimize the allocated space for any given page. FlatPack uses fixed-size MPA pages with a set of predefined sizes organized in accordance with Table 5.1. A pool of allocated and free pages for each size is maintained by a software runtime. Each OSPA page is assigned one MPA page from one of these pools, and the assignment is stored in a separate metadata table, leaving the OSPA assignment in the regular page table. As a result, OS address translation and cache tagging logic remain unmodified.

The assignment of MAG-sized slots to native blocks sets the lower bound of a page's size to one slot per block. Because an uncompressed block fits

Table 5.1: FlatPack compressed page sizes and organization.

Page Size	512B	1kB	1.5kB	2kB	2.5kB	3kB	3.5kB	4kB
Full Slots	0	0	16	16	32	32	48	64
Per Block	0	0	1	1	2	2	3	4
Minislots	32	64	32	64	32	64	32	0
Per Block	2	4	2	4	2	4	2	0

in 4 MAGs, this would limit compression ratio to 4×, and would leave no block size flexibility at that size. To counteract these problems, FlatPack increases granularity for very small pages and introduces a 512 byte page with a compression ratio of 8×. Compressed pages of 512B or 1kB have no full-sized slots, and are instead entirely composed of minislots. In a 512B page, each block is native to two minislots. This finer granularity allows small pages to remain flexible to block size variation.

One drawback of fixed-size compressed pages is *Page Migration*. Page migrations occur when a compressed page exceeds its allocated size (page overflow) or is deemed able to fit in a smaller size (page shrink). To minimize the number of page size changes, FlatPack employs a runtime mechanism to estimate the page's future size at the time of initial allocation. Page allocation occurs in two stages. Initially, a new page is introduced into the page table by the operating system, and mapped to a read-only zero-data page. On the first writeback from the LLC (modifying the data in memory) a unique compressed page is allocated in MPA space. The size of this allocation is based on the compressibility of the written-back data. The compressed size of that data is extrapolated to the full page's size. If this expected page size exceeds one slot per block, a margin is added for flexibility. Finally, the estimate is rounded up to the nearest supported page size and used for the initial allocation.

5.2.8 Interaction with the OS

FlatPack can use memory ballooning in order to handle the variable memory size due to compression. Memory ballooning is a common feature in virtualization environments, and has been proposed for compressed memories by IBM [108] and more recently in Compresso [16].

In a virtualization environment, the Hypervisor controls the amount of available memory to the guest OS through a memory balloon software driver implemented in every guest OS. In essence, the Hypervisor can use the balloon driver to reclaim memory from one guest OS and provide it to others depending on the runtime memory needs of the respective virtual machines. The reclaimed memory from the respective guest OS is reserved by the balloon driver and cannot be allocated by the OS itself.

In a compressed memory system, the OS starts with a memory size $M = C \times P$, for maximum compression ratio C and physical capacity P. In the beginning of execution the memory is still uncompressed, thus the overcommitted memory $(C-1) \times P$ is reserved by the memory ballooning software driver. As the system allocates memory, the FlatPack runtime compresses the data and manages pools of allocated and free memory pages. New free memory pages are released

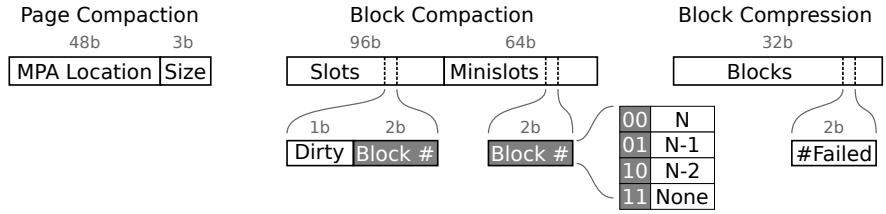


Figure 5.8: Metadata for a compacted FlatPack page. Two bits are used to encode which block occupies any given slot.

to the OS by *deflating* the balloon driver. This way, free memory pages can be directly allocated by the OS. If memory compressibility deteriorates due to page overflows, the balloon driver is *inflated*. This triggers the OS's memory reclamation to free up allocated memory using the paging mechanism. The reclaimed addresses are communicated to the FlatPack runtime system, freeing the corresponding physical memory.

FlatPack can be combined with other transparent and less transparent implementations to interact with the OS and release to it the free space created from compression.

5.2.9 Metadata

The described flexibility of placement requires supporting metadata to maintain page organization. Figure 5.8 summarizes the metadata used by FlatPack, which can be divided into three parts. The first part concerns page compaction. The MPA page location is 48 bits wide and indicates where in physical memory the compressed page resides. An additional 3 bits indicates the page size.

The second part of the metadata maintains block compaction. Any full-size slot can be in one of three states: *unused*, *used for compressed data*, or *used for lazily evicted data*. In addition, a slot may be occupied by one of three blocks (the native block N or one of the nearby blocks $N-1$ and $N-2$). By limiting each slot to one of three blocks, FlatPack can encode this information using three bits per slot. A minislot, similarly, may be either *unoccupied* or contain compressed data from one of three blocks. This requires two bits of metadata per minislot. The most complex page size (3.5kB) contains 48 full-size slots and 32 minislots. 16 full-size slots are reserved for their native blocks and always occupied, requiring no metadata. Consequently, each page requires at most $32 \times 3 + 32 \times 2 = 160$ bits of page compaction metadata.

The third and final part contains metadata for individual blocks. Each block has two bits associated with it, used to count failed compression attempts in order to reduce the frequency of retries. With 16 blocks in one page, this block metadata comprises 32 bits per page. In total, FlatPack requires a maximum of 243 bits of metadata per compressed page.

Metadata are managed on the page level, and cached on-chip in a Metadata Cache which has the same capacity as the system Translation Lookaside Buffer (TLB). Metadata are fetched from memory in parallel with TLB misses, which guarantees that block and page metadata are readily available by the time a request reaches the LLC.

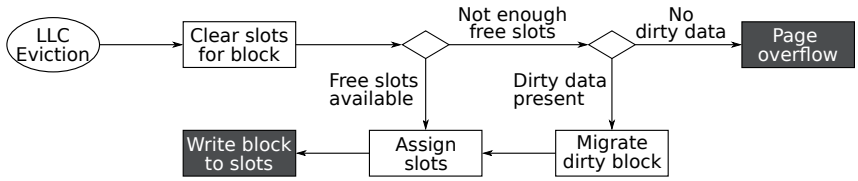


Figure 5.9: Overview of block writeback logic. Repacking of a block occurs on LLC eviction, when the block changes size.

5.2.10 Block Migration

Figure 5.9 illustrates the logic flow of block writebacks. In the common case, sufficient slots and minislots are available in main memory, and the block can be written back directly. However, If a block cannot be packed successfully, slots must be freed up to accommodate it. The page metadata distinguishes compressed data from lazily evicted cache lines (i.e. uncompressed, dirty data). Blocks with lazily evicted cache lines consume more space in memory than necessary, to delay costly recompression. If such data prevents packing of a neighboring compressed block, the offending block is brought on-chip for recompression immediately, as shown in Figure 5.10. This way, physical space can be used for lazy evictions, and made available for actual compressed data on demand.

5.2.11 Page Migration

If no slots can be freed to pack an evicted block, the size of the compressed page is insufficient to contain its data. As a result, the data must be migrated to a larger page. The process of page growth is straight-forward. The page and block metadata are consulted to determine the smallest sufficient allocation size. Similarly to the initial allocation, an additional margin is added to the estimated size, to allow for flexibility to future growth. A new compressed page is set up and all blocks are read from physical memory. Finally, the compressed blocks are transferred to the newly allocated location.

Conversely, data updates during execution may cause blocks to shrink. In order to maximize the capacity gain of compaction, it is beneficial to detect and adapt to such changes. Each page's metadata is sufficient to identify pages whose data could fit within a smaller page allocation. When such a page is detected, FlatPack initiates a page migration to shrink the page and increase memory capacity.

5.2.12 Memory Interleaving

So far, the description of FlatPack has considered systems with one memory controller. However, address interleaving across multiple memory controllers can also be supported as follows. The minimum interleaving granularity is 4×MAG, keeping a single block access to a single channel. In addition, interleaving across n controllers requires FlatPack to consider groups of n pages together, rather than a single page as described above. Then, blocks within the group of pages with the same ($block\_id \bmod n$) are handled together as described in

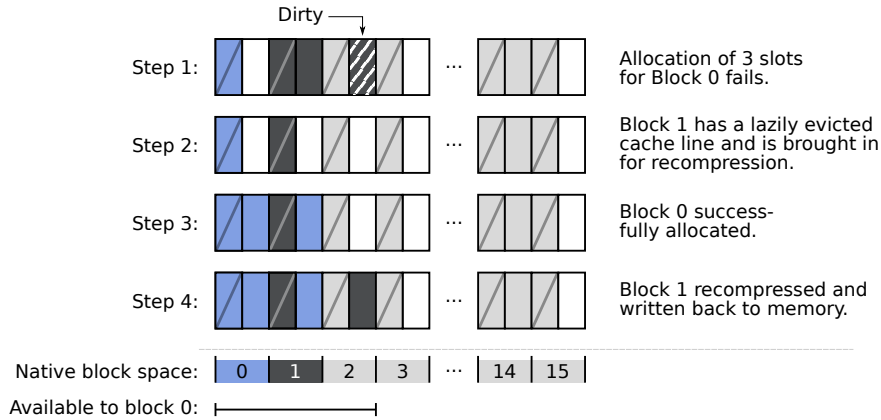


Figure 5.10: Block migration to support a growing block. Block 0 (3 MAGs) needs to be written back. In order to fit, block 1 is read from memory and recompressed to eliminate lazily evicted data.

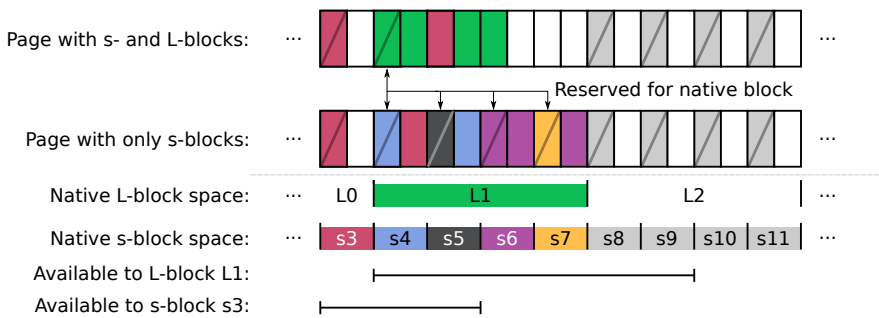


Figure 5.11: Mixture of L-blocks and s-blocks in the same FlatPack page

Section 5.2.4 as if they belonged to the same page, stored on the same channel. This requires metadata of all pages in the same group to be brought on chip together.

5.3 Support for Lossy Compression

As explored in previous chapters, certain classes of applications are able to tolerate inaccuracies in their data. This tolerance for approximation can be exploited to reach higher compression ratios using *lossy compression*. AVR (Chapter 2) and MemSZ (Chapter 3) outline two different lossy compression mechanisms and apply them to memory compression.

FlatPack supports hybrid lossy and lossless compression, using the LLC and metadata mechanisms of L^2C (Chapter 4) combined with its own memory compaction scheme.

5.3.1 MemSZ Compression

For lossy compression, FlatPack uses the compression scheme introduced by MemSZ , a highly parallelized implementation of SZ [34]. MemSZ arranges numeric data in a square block and divides it into several linear value sequences. A sequence is compressed by describing each individual data value as a function of the three preceding values, with a small selection of supported functions. This way, the block can be encoded as a few starting values (*seeds*) followed by two bits per data value indicating which function best describes that value. By accepting approximate matches within a configurable tolerance, MemSZ enables lossy compression. MemSZ is designed for compression ratios of up to 16×.

5.3.2 Block Size

The two compression schemes employed by FlatPack have differing characteristics. The lossy compressor is geared toward high compression ratios. In addition, its block format contains an overhead of fixed data per block. As a result, a large block size is necessary. The lossless compressor, by contrast, has no such fixed overheads. Its compressed blocks consist only of re-encoded values, in order. This allows it to be applied to blocks of any size.

The optimal block size for any memory compression scheme depends on two factors: the maximum compression ratio and the Memory Access Granularity (MAG). An undersized block may compress to a size smaller than the memory access granularity, leading to transfers with overhead. Conversely, an oversized block may compress less efficiently than expected, leading to extraneous data transferred. For these reasons, the optimal block size is such that the maximum expected compression ratio results in a compressed size equal to the memory access granularity.

The MAG of a typical memory system is one cache line. The lossy compression employed by L^2C is designed for a maximum compression ratio of 16 : 1, and is thus applied to blocks of 16 MAGs. These large blocks are referred to as *L-blocks*. SC^2 lossless compression using 16-bit values has a theoretical maximum compression ratio of 16 : 1 (compressing each 16-bit value to a single-bit encoding), but typically achieves compression ratios between 2 : 1 and 4 : 1 on non-trivial data. For this reason, FlatPack applies lossless compressed data to blocks of 4 MAGs. These small blocks are labeled *s-blocks*. An *s-block* is a quarter of an *L-block*, which is beneficial to alignment and management. A 1kB region of memory can be represented either as one *L-block* or as four *s-blocks*.

The on-chip management of s- and L-blocks is identical to the system employed by L^2C, which was described in more detail in Chapter 4

5.3.3 Block Placement

This support for multiple block granularities has consequences for block placement in compacted memory. Any given 1kB portion of the logical address space is stored in physical memory as either a single compressed L-block or four compressed s-blocks. FlatPack manages the addition of L-blocks by allowing each L-block to reside in any slot allowed for its constituent s-blocks.

Figure 5.11 illustrates an example of this dual placement. If the second kB of a page is compressed losslessly, it is divided into s-blocks 4, 5, 6, and 7. These four s-blocks, collectively, are able to occupy not only their native slots, but also those of neighboring blocks 8 and 9. If this 1kB is instead compressed with lossy compression, it is represented as a single L-block. The data for this L-block is also allowed to be placed in slots native to any of the s-blocks 4-9.

Each s-block stored in physical memory has one *reserved* slot, which is not made available to store other blocks. The same is true for each L-block, which has one reserved slot at the start of its native space. The rest of the slots native to an L-block are made available to neighboring blocks following the same restrictions as lossless FlatPack.

5.3.4 Block Transitions

Compressed memory blocks representing approximable data may transition between L- and s-blocks during execution. This can happen for two reasons. First, an L-block may fail recompression due to quality constraints, and thus transition into four s-blocks. Second, if lossless compression is successful for several consecutive updates, lossy compression may be attempted to merge four s-blocks into an L-block.

These transitions can be carried out with minimal traffic overhead. Transition from L- to s-blocks introduce no traffic overhead, since such transitions can only occur when the full 1kB of data is on chip and lossy compression has failed. Transition from s-blocks to L-block is triggered by a sequence of successful s-block compressions, indicating that compressibility of the 1kB block is good. In order to attempt the transition, up to three s-blocks need to be read from memory.

Figure 5.11 illustrates a placement issue which can arise when transition from L- to s-blocks takes place. Consider the L-block L1, which has a single reserved slot and is native to the same space as s-blocks 4-7. If this L-block fails lossy compression and must transition into four s-blocks, one slot per s-block must be reserved. In the illustrated example one of these slots is occupied by the neighboring block s3, but must be reserved for s5 as part of the transition. FlatPack resolves this collision by reading the data from the offending slot, and assigning space for it among the shared slots created during the transition.

5.4 Evaluation

In this section we evaluate the efficiency of FlatPack. First, the experimental setup is described, detailing the system configuration and the benchmarks used. Then, evaluation results from single core and multicore experiments are presented.

5.4.1 Experimental Setup

We evaluated FlatPack in an in-house simulator, implemented on top of Pin [48]. The simulator employs an interval-based processor model [49]. The memory hierarchy is modelled at cycle granularity, using DRAMSim2 for main memory [50]. McPAT [51] and CACTI [52] were used to model power and latency

Table 5.2: Simulation parameters.

Parameter	Configuration
Simulation Duration	1×10^9 instructions per core
CPU	O-o-O, 4-way issue @ 3.2GHz
L1 cache	64kB per core, 4-way, 1 cycle latency
L2 cache	256kB per core, 8-way, 8 cycle latency
L3 cache	1MB per core shared, 16-way, 15 cycle latency
Main Memory	4GB DDR4 per core, $\frac{1}{4}$-channel per core, 800MHz
Avail. Memory	50% of application footprint, at least 8× LLC
Page Fault Latency	8.6µs [6]
Compressor	SC^2 lossless, 16-bit values
SC^2 VFT	7kB, 8-way, 16-bit values

of the system considering 32nm technology. Operating frequency, latency and power consumption parameters for the SC^2 compressor are based on a place-and-route implementation for the same technology node [72]. These factors are used as configuration information for the simulations. The general properties of the simulated system are listed in Table 5.2.

Besides the baseline system, FlatPack (F) is further compared with (i) Compresso [16], labeled C, and (ii) LCP [13], labeled L. All three compressing designs use the same SC^2 compressor. The SC^2 compressor requires training data specific to each application to be effective. For each application, a profiling run is performed, where a randomized 10% subset of application data is gathered. This dataset is used to train the compressor for all designs, ensuring consistent compressor behavior. For evaluation we use the complete set of applications of SPECspeed 2017 [109], Graph500 [110], as well as ForestFire and PageRank from the SNAP suite [111].

Compression is applied to all non-code pages. This includes heap, stack, and data segments of the application itself, as well as those of shared libraries. All benchmarks use their respective *ref_speed* input data provided with SPEC 2017.

A large benefit of memory compaction is the ability to avoid costly *page faults*, when data is swapped into memory from the slower nonvolatile storage. To investigate the effect of each design on page faults, the uncompressed baseline is profiled to determine its *active footprint* (the total number of unique pages in memory actually read or written) during the simulated phase of execution, as shown in Table 5.3. This baseline active footprint is used as a basis for page fault modelling, where the available physical memory is limited to 50% of the baseline footprint or at least 8× the LLC capacity (a minimum capacity limit of 32MB). This capacity limit is applied to all designs, including the baseline. Page faults are modeled as memory traffic and an additional delay of 8.6µs to account for OS handling and nonvolatile storage latency [6].

Each simulation is executed in the following steps: i) SC^2 compressor's Value Frequency table (VFT) is populated with data from the profiling run; ii) The application is run through its initialization phase; iii) the application's main phase is executed for one billion instructions per core, and statistics are gathered. The end of the initialization phase is manually selected such that

Table 5.3: Benchmarks and their active memory footprints.

Application	Footprint / core	Application	Footprint / core
bwav [109]	247MB	omnt [109]	16MB
cact [109]	521MB	perl [109]	20MB
cam4 [109]	59MB	pop2 [109]	73MB
deep [109]	688MB	roms [109]	133MB
exc2 [109]	16MB	wrf [109]	48MB
foton [109]	2320MB	x264 [109]	26MB
gcc [109]	33MB	xbmk [109]	16MB
imag [109]	34MB	xz [109]	525MB
lbm [109]	292MB	ffire [111]	11MB
leela [109]	16MB	gp500 [110]	255MB
mcf [109]	150MB	pgrank [111]	10MB
nab [109]	31MB		

statistics are collected during the application's primary processing phase.

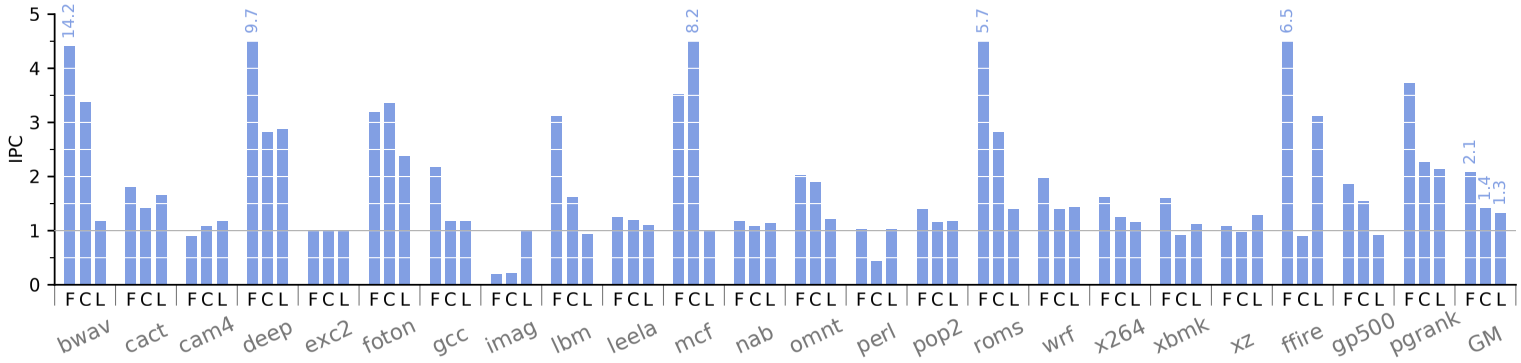
In the following sections, we present our evaluation in two separate sets, a single-core system and a multi-core system where four instances of the same application are run side by side on four identical processors.

5.4.2 Single-Core Experimental Results

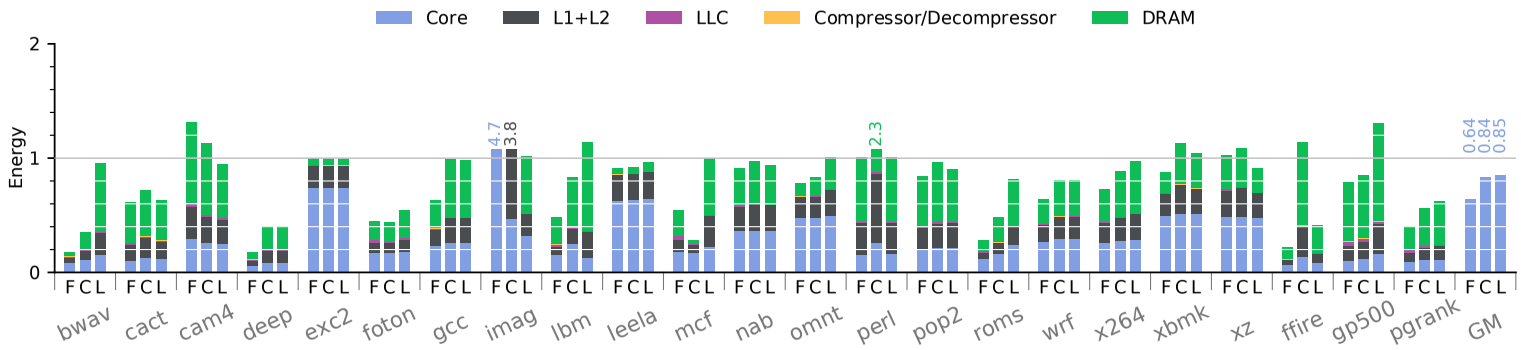
The primary effect of memory compaction is a reduction in physical memory footprint. Figure 5.13b shows the achieved footprint for each design, broken down into categories. The *data* footprint consists only of the compressed data, excluding any compaction-related waste. Each design also introduces some *metadata* storage overhead, and a varying amount of *capacity waste*. *Granularity waste* is a result of limited block granularity. *Shrink waste* is caused by a compressed block shrinking, leaving some of its original allocation unused. *Growth waste* is caused by a compressed block growing to become an *exception*, leaving its old allocation entirely unutilized. *Uncompressed* waste represents blocks being left uncompressed for organizational reasons, even though their actual compressibility is better than 1:1. Finally, *overallocation* is unused space due to the granularity of page allocation. While this space is not wasted, it consumes memory capacity.

Compresso yields a final footprint 35% of the baseline, closely followed by FlatPack at 37%. LCP brings the footprint down to 49%. The major differences between FlatPack and Compresso are seen in *foton* (caused by reduced compressibility) and *xz* (caused by overallocation of physical pages). This single case of large overallocation indicates that adjacent blocks (which compete for memory slots) are growing together, exceeding the ability for dynamic placement. At the expense of additional metadata bits, FlatPack can be modified to allow a wider span of slot placements per block.

The raw compression ratio (an average of 3.9× across all applications) illustrates the average compressibility for the full memory footprint. This compression ratio sets the upper bound of compaction, and functions as an *ideal* reduction, as shown in Figure 5.14a. This ideal serves to illustrate the capacity overhead of each design, when compared to the achieved total footprint. The figure also illustrates a non-compacting design from literature, *MemSZ* [76],

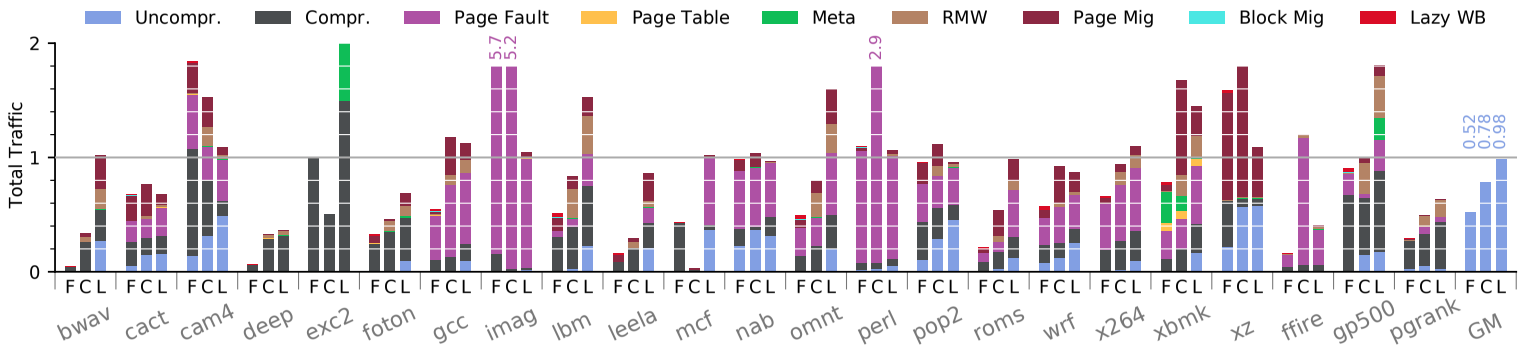


(a) Normalized Instructions Per Cycle (IPC).

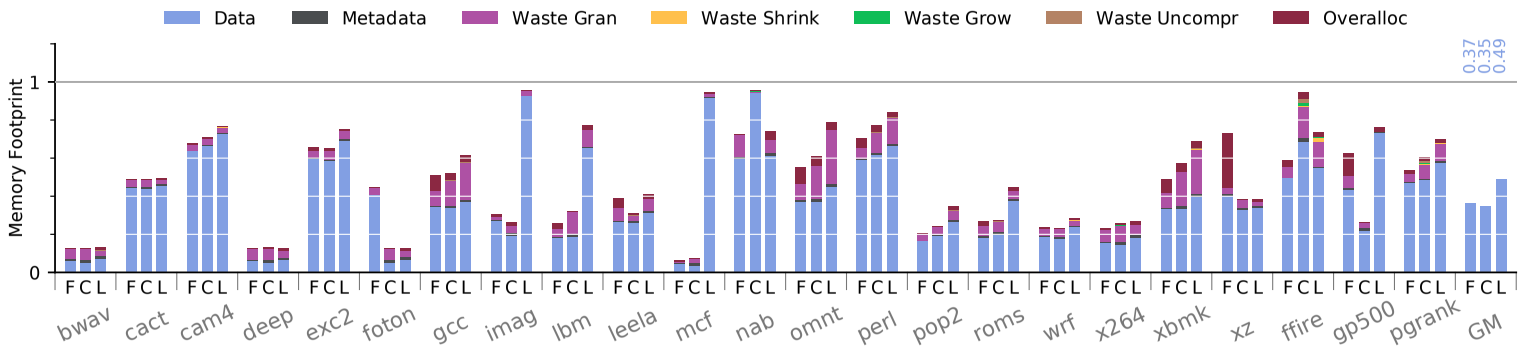


(b) Normalized system energy consumption.

Figure 5.12: Single-core results for (F)latPack, (C)ompresso and (L)CP.

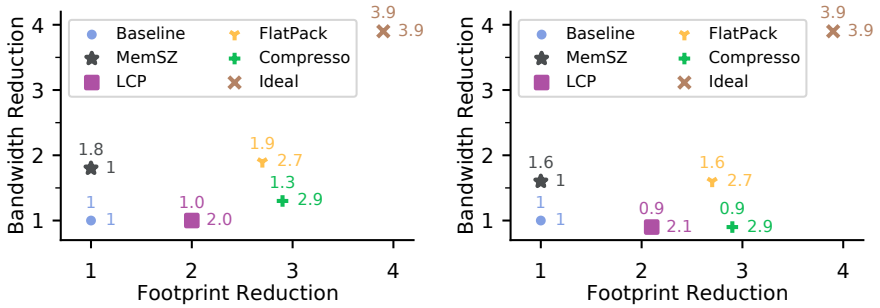


(a) Normalized memory traffic.



(b) Normalized physical footprint.

Figure 5.13: Single-core results for (F)latPack, (C)ompresso and (L)CP. (continued)



(a) Single-core (b) Multi-core

Figure 5.14: Bandwidth and Footprint improvements. *MemSZ* is a non-compacting memory compression system, here configured to use the same lossless SC^2 compression.

configured to use the same compressor as described above. Compresso achieves 74% of the ideal footprint reduction, FlatPack follows with 70% and LCP manages 53%. The largest factor in this overhead, in a majority of applications, is *granularity waste*. Compared to the baseline footprint, FlatPack introduces 4.6% of granularity waste. LCP and Compresso add 5.0% and 5.3%, respectively.

Memory compression also has the potential to reduce off-chip memory traffic, by transferring compressed blocks over the main memory bus. Like the footprint, the extent of this reduction is also bounded by the compression ratio. Many memory compaction schemes do not target traffic reduction, instead using additional traffic for the data movement required to support an adaptive compaction scheme. Figure 5.13a shows the volume of data transferred across the bus, broken down by cause. Various overheads are introduced by the evaluated compaction systems. *Metadata* traffic varies both due to the size of the metadata itself and its access patterns. *Read-Modify-Write* operations are necessary to update compressed data in memory at a finer granularity than the MAG, and take the form of additional reads from main memory. *Page Migration* is necessary to grow compressed pages upon overflow, or shrink them if possible. *Block Migration* traffic is induced by single compressed blocks being brought on-chip to update them with dirty data. *Lazy Writebacks* are used by FlatPack to delay block migrations. These overheads of memory compaction also prevent compaction systems from reaching a traffic reduction proportional to the ideal compression ratio.

FlatPack reduces the mean memory traffic to 52% of the baseline. Compresso also reduces traffic, to 78% on average. LCP achieves an average 2% reduction. Similarly to the memory capacity, the raw compression ratio also indicates an ideal memory traffic reduction. FlatPack reduces memory traffic by 1.9×, which is 32% of the ideal. Compresso achieves a 1.3× reduction in traffic, 9.7% of the ideal. The modest 1.02x improvement of LCP corresponds to 0.6% of the ideal. One significant traffic overhead from page compaction is *page migrations*. Normalized to the baseline's total memory traffic, FlatPack spends 2.3% on page migrations. The corresponding metric for LCP is 4.9% and for Compresso 7.1%. *Xbmk* exhibits a highly irregular access pattern, even on the page level. As a result, it sees a significantly increased amount of *page table* traffic in the baseline. In the compacting designs, this also carries over into *metadata* traffic, with FlatPack's metadata overhead being proportional

to TLB misses.

Page faults make up a large portion of baseline memory traffic. The principal goal of reducing these page faults is achieved by all three designs. LCP reduces the average total number of page faults to 66%, Compresso to 68% and FlatPack achieves a reduction to 72%. Notably, Compresso increases the number of page faults in two benchmarks, due to page migrations. FlatPack introduces additional page faults in *cam4*, *imag* and to a lesser extent in *perl*. *Imag* exhibits little spatial locality which penalizes both the large compression blocks of FlatPack and the fragmented page allocation of Compresso.

Performance benefits from memory compaction stem from the reduction of costly page faults as well as reduced memory traffic which leads to lower memory latency. Figure 5.12a shows the IPC across execution for each design and benchmark. FlatPack increases system IPC by an average 107%, Compresso by 41% and LCP by 32%. The greatest performance boosts correlate to significant reductions in total memory traffic, indicating memory-bounded applications.

Finally, reduced execution time and memory activity lead to reductions in system energy as illustrated in Figure 5.12b. FlatPack reduces average system energy to 64%. Compresso reaches 84% and LCP achieves an average of 85%.

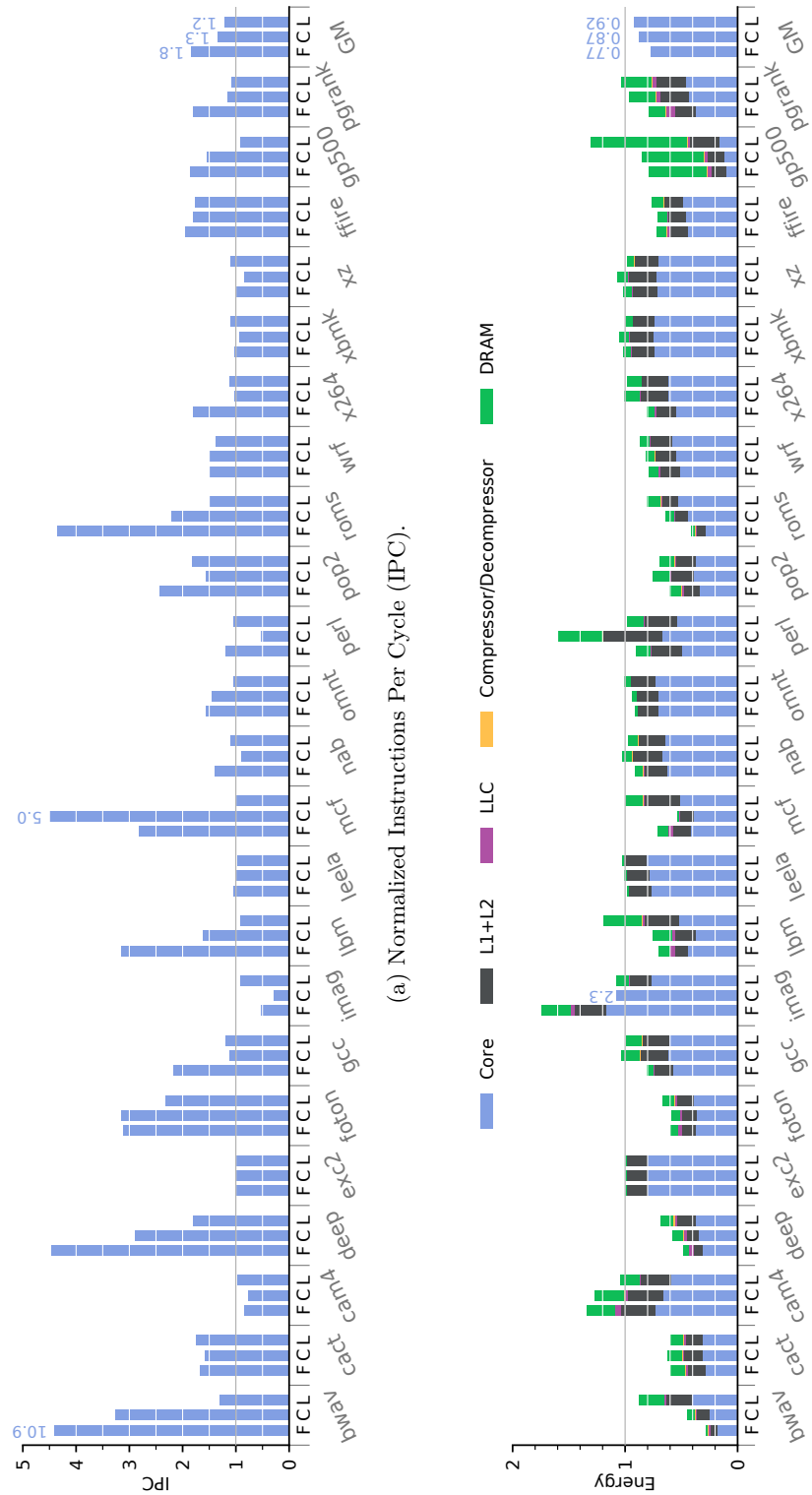
In summary, the previous state-of-the-art Compresso improves system performance and energy by 41% and 16%, respectively. FlatPack roughly doubles these benefits, offering 107% better performance and 36% lower energy consumption than a baseline system. FlatPack improves performance and energy consumption by 46% and 24%, respectively, compared to the second best design. Meanwhile, FlatPack maintains a memory capacity improvement within 6% of Compresso.

5.4.3 Multi-Core Experimental Results

The achieved memory footprint for each design is shown in Figure 5.16b. As in the single-core evaluation, Compresso and FlatPack achieve similar totals, of 34% and 38%, respectively. LCP achieves a normalized footprint of 48%. As with the single-core system, the main overhead of all three designs is *granularity waste*. Granularity waste introduced by FlatPack accounts for 4.7% of the mean baseline footprint. The same metric is 5.3% waste for Compresso and 5.1% for LCP. The flexibility of block placement in FlatPack is insufficient in *xz*, as in the single-core evaluation. This leads to a disproportionate overallocation, as larger pages are required to support only a subset of large blocks which are competing internally.

The average raw compressibility is compared to the achieved reduction in footprint and traffic in Figure 5.14b. The ideal footprint reduction is 3.9×, determined by the raw compression ratio. Compared to this ideal, Compresso reaches 76%, FlatPack follows at 68% and LCP achieves 54%.

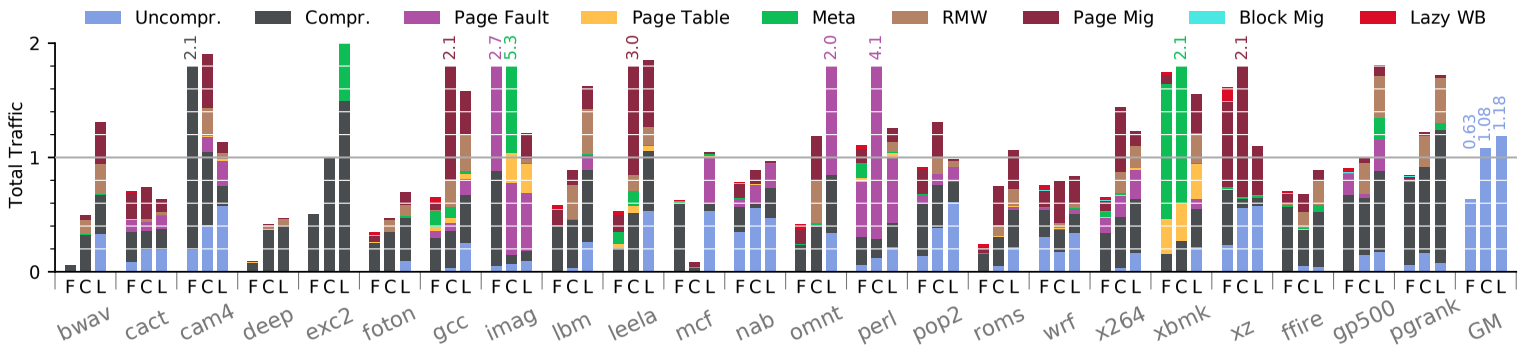
Figure 5.16a shows the total memory traffic of each design, broken down by cause. In total, FlatPack achieves a 1.6× reduction in memory traffic. This is 20% compared to the ideal. Compresso and LCP both add traffic overheads of 0.08× (4.2× ideal) and 0.13× (4.5× ideal), respectively. *Cam4* shows little spatial locality in its access pattern, illustrating the drawback of FlatPack's large compression blocks; while full compressed blocks are read from memory, parts of the data remain unused and become overhead. The same affects Compresso, to



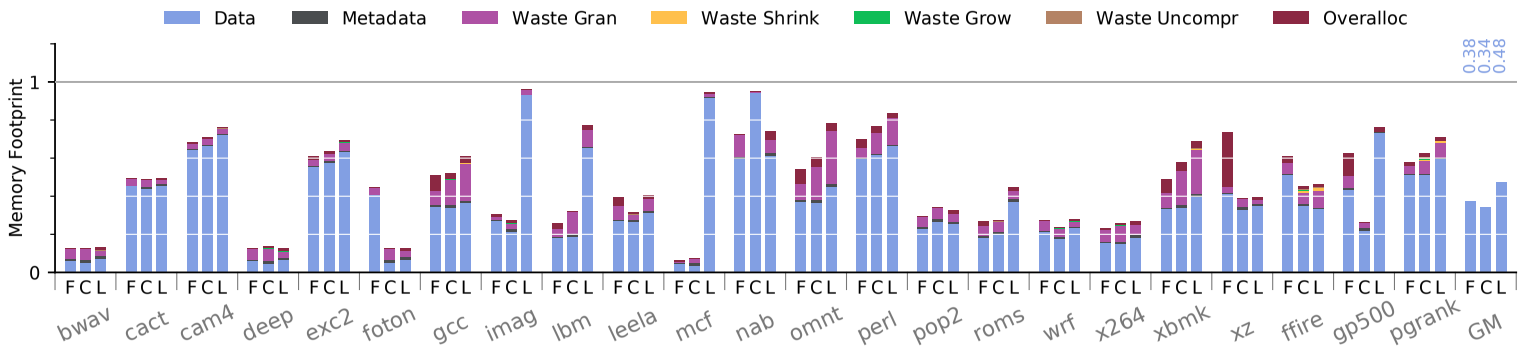
(a) Normalized Instructions Per Cycle (IPC).

(b) Normalized system energy consumption.

Figure 5.15: Multi-core results for (F)latPack, (C)ompresso and (L)CP.

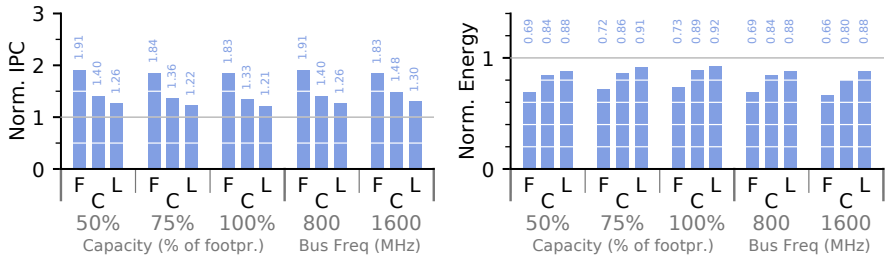


(a) Normalized memory traffic.



(b) Normalized physical footprint.

Figure 5.16: Multi-core results for (F)latPack, (C)ompresso and (L)CP. (continued)



(a) Normalized system performance

(b) Normalized system energy

Figure 5.17: Normalized means of system metrics under (F)latPack, (C)ompresso, and (L)CP for varying memory capacity and memory bandwidth.

a lesser extent, as compressed overfetching does not serve as useful prefetching. A similar effect is seen in *imag*, where FlatPack's compressed traffic is increased by overfetching. *Page migration* remains a significant overhead in the multi-core system. Normalized to the baseline's total memory traffic, FlatPack spends 4.0% on page migrations. The corresponding metric for LCP is 8.5% and for Compresso 16.3%. By reducing page migration overhead by 2×-4×, and eliminating a majority of RMW traffic, FlatPack offers a significant traffic benefit compared to the competing designs. FlatPack reduces memory traffic by 42% compared to Compresso, the next-best design. FlatPack reduces the mean number of page faults to 40%, while Compresso reaches 45% and LCP achieves a reduction down to 61%.

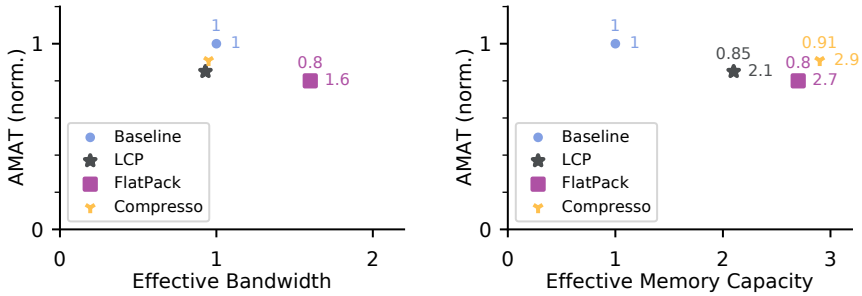
The end goal of footprint and memory traffic reductions is performance improvement. Figure 5.15a shows the mean IPC achieved by each design across execution. FlatPack increases performance by 83%, with Compresso following at 34% and LCP offering a performance benefit of 22% over the baseline.

Finally, total system energy consumption benefits from the performance increase and reduced memory activity. Figure 5.15b summarizes energy consumption for the 4-core systems. FlatPack brings the total system energy down to 77%, while Compresso and LCP reach 87% and 92%, respectively.

In summary, Compresso improves system performance and energy by 34% and 13%, respectively. FlatPack doubles these benefits, offering 83% better performance and 23% lower energy consumption than a baseline system. FlatPack improves performance and energy consumption by 36% and 12%, respectively, compared to Compresso.

5.4.4 Latency Impact

The additional decompression introduced into the critical path of memory accesses may have an impact on system performance. The decompression latency of FlatPack is similar to the access latency of the last-level cache, and thus the latency of a compressed cache hit is roughly double that of a regular, uncompressed hit. In addition, when serving a cache miss, multiple MAGs of data may be read from memory and must be decompressed before the hit can be served. These effects have the potential to increase the Average Memory



(a) AMAT vs. Effective Bandwidth. (b) AMAT vs. Effective Capacity.

Figure 5.18: Average Memory Access Time (AMAT) weighed against the bandwidth and capacity effects of the investigated memory compaction systems.

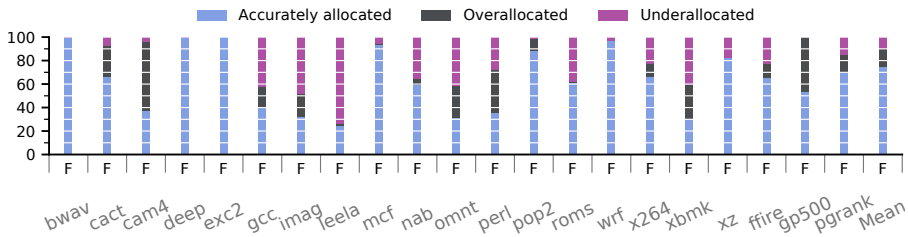


Figure 5.19: Physical pages broken down by the success of initial page size estimation.

Access Time (AMAT) by introducing additional latency. On the other hand, memory compaction systems aim to improve system performance by reducing the incidence of slow page faults. Figure 5.18a shows the memory access latency reduction and the *effective bandwidth* of each investigated system. Figure 5.18b shows the effective memory capacity of each system. In our experiments, all compacting systems reduce the AMAT compared to the baseline. This indicates that the benefits of compression and compaction do not come at the expense of an overall latency increase.

5.4.5 Page Size Estimation

One cause of page faults is the uncertainty inherent in the initial allocation of each physical page. Typically, data need to be written back to memory before the whole page has been populated with data. As a result, a physical allocation is required before the actual compressibility of the page is known. FlatPack employs an estimation mechanism based on the first written-back cache line. The compressed size of this line is extrapolated to a full page, and a small margin is added before rounding up to the nearest supported page size. Figure 5.19 shows a breakdown of all physical pages in each tested application. *Accurately allocated* pages are such that the initial estimation is equal to the eventual size of the page. On average, 75% of pages are accurately allocated using the first estimation. *Overallocated* pages had an initial estimation which

Table 5.4: Approximable Benchmark Applications.

Application	Domain	Approx.	Footprint / core
heat [53]	2D Therm.	Temps	8.3MB
lattice [54]	2D CFD	P and M	5MB
lbm06 [55]	3D CFD	Velocities	325MB
orbit [56]	3D Phys.	Phys. data	10MB
cdelta [56]	Thermodynamics	Phys. data	22MB
sedov [56]	Hydrodynamics	Phys. data	12MB
windt [56]	Hydrodynamics	Phys. data	23MB
kmeans [57]	Clustering	Topol. [58]	5.5MB
wrf06 [55]	Weather	Geo data	90MB

Table 5.5: Application Output Error for FP-X.

heat	lattice	lbm06	orbit	cdelta	sedov	windt	kmeans	wrf06
0.13%	0.25%	0.06%	0%	<0.01%	0%	<0.01%	0.05%	<0.01%

was larger than the first full size of the page, and thus required no migration. 15% of pages are of this type. The remaining 10% are *underallocated*, pages whose initial estimate was smaller than their actual size once filled, and thus required migration before being fully populated with data.

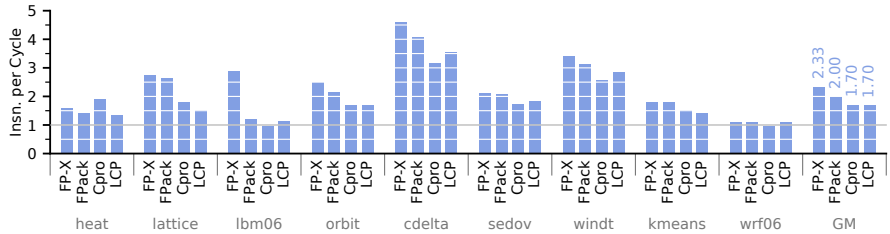
5.4.6 Sensitivity to System Configuration

To further evaluate the impact of system parameters on FlatPack, we also present the result of two separate sensitivity analyses, shown in Figure 5.17. The first sweeps across available memory capacity, with limits set at 50%, 75% and 100% of the baseline footprint. While the benefits of compaction are reduced overall, the trend between the tested designs remains. The second analysis illustrates the impact of memory bandwidth, comparing an 800MHz memory bus to a 1600MHz one. We find that the performance and energy benefits of all three systems remain stable.

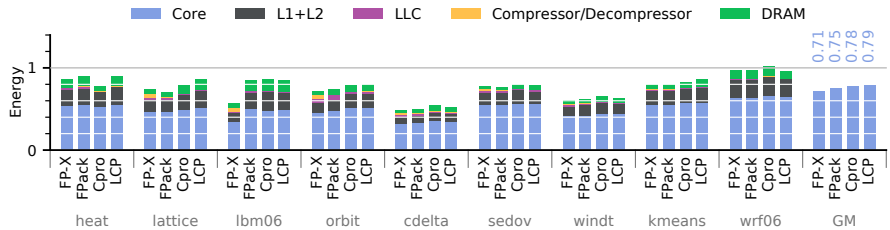
5.4.7 FlatPack with Lossy Compression

To evaluate the effectiveness of FlatPack with lossy compression support, we use a selection of approximation-tolerant applications as summarized in table 5.4. The source code of each application has been annotated to label approximable data structures. The data approximation caused by lossy compression is introduced into the data during simulation, and applications are run to completion in order to generate output. The modified FlatPack with both lossy and lossless compression (*FP-X*) is compared to the original lossless-only FlatPack (*FPack*), Compresso (*Cpro*), LCP (*LCP*), and finally a baseline system with no compression or compaction.

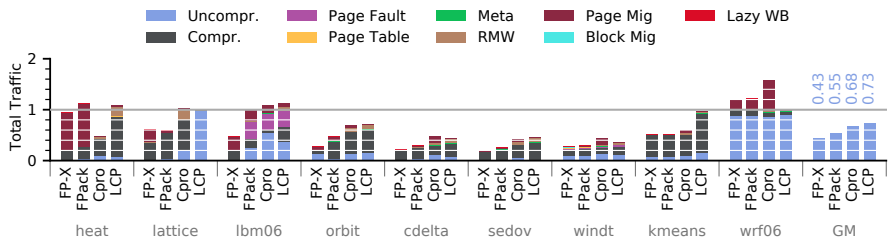
Figure 5.20a shows the performance of the evaluated systems, normalized to the non-compressing baseline system. Lossy FlatPack achieves an IPC 2.33× that of the baseline. Lossless FlatPack follows with a 2.00× IPC, Compresso with 1.70× and LCP with 1.70×.



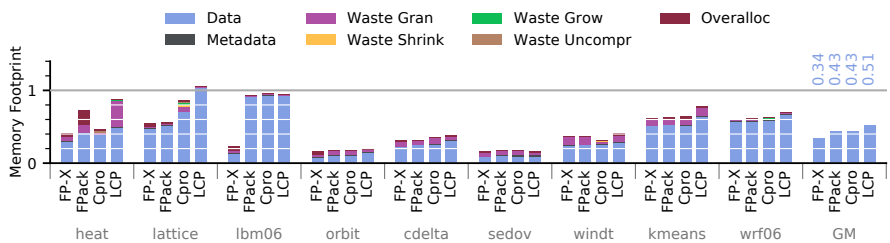
(a) Normalized Instructions Per Cycle (IPC).



(b) Normalized system energy consumption.



(c) Normalized memory traffic.



(d) Normalized physical footprint.

Figure 5.20: Multi-core results for FlatPack with lossy and lossless compression (FP-X), FlatPack with lossless compression (FPack), Compresso (Cpro), and LCP

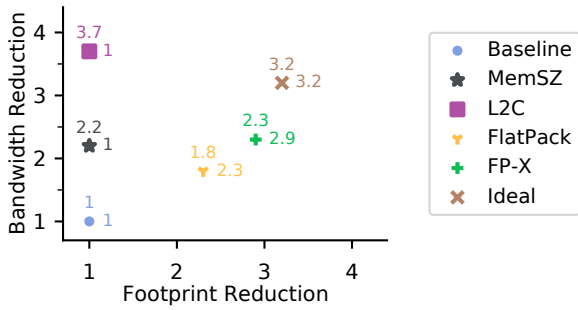


Figure 5.21: Bandwidth and footprint improvements of FlatPack compared to other lossy memory compression systems.

As a consequence of the reduced execution time, energy consumption is also reduced. Figure 5.20b shows a breakdown of total system energy. Lossy FlatPack achieves a mean reduction of 32%, while lossless FlatPack manages 28%, and Compresso and LCP achieve 25% and 24%, respectively.

One important cause of the performance improvement is reduction of traffic on the memory bus, shown in Figure 5.20c. Lossy FlatPack reduces total traffic by 57%. Lossless FlatPack achieves a 46% reduction, followed by Compresso at 32% and LCP at 27%.

Figure 5.20d shows the achieved memory footprints of the evaluated designs. On average, Lossy FlatPack manages a 2.95× compaction, followed by lossless FlatPack and Compresso both at 2.31×, and LCP at 1.94×.

The lossy MemSZ compression employed by FP-X introduces inaccuracy in the final application output. Table 5.5 shows the deviation between approximated output and baseline output for each application. We find that the total error does not exceed 0.25%.

Finally, Figure 5.21 shows the footprint and bandwidth reduction of FP-X compared to MemSZ, L^{2}C, and FlatPack. L^{2}C achieves the greatest memory bandwidth reduction thanks to its hybrid compression, and storing compressed data in LLC allows it to reduce the cache miss rate and surpass the ideal set by the raw compression ratio. FP-X is able to capitalize on lossy compression for both bandwidth and footprint reduction, surpassing the lossless-only FlatPack.

5.5 Conclusion

FlatPack is a novel approach to memory compaction, which allows compressed blocks to be packed fragmented within a page and share expansion space. It uses a hardware mechanism to dynamically reorganize pages when blocks are updated, without introducing any additional data movement. In a multicore system with 4GB of 800MHz DDR4 per core and 1MB of LLC space per core, FlatPack offers memory capacity increases on par with state-of-the-art compaction systems, while improving on their memory bandwidth utilization by up to 67%. By leveraging compression and compaction to reduce data movement and costly page faults, FlatPack is shown to improve performance and energy consumption of a single-core system by 107% and 36%, respectively

and in a multi-core system, the improvements are 83% and 23%, respectively. Compared to the best previous work, FlatPack improves performance by 36-46% and energy by 12-24%, while achieving a comparable memory capacity.

By adding support for lossy compression to FlatPack, memory capacity can be improved by an additional 28% compared to lossless FlatPack. Memory traffic is reduced by an additional 21%, leading to a 17% performance improvement and a 6% energy reduction.

Chapter 6

Conclusion

The memory hierarchy is an increasingly important bottleneck in general-purpose computing systems. The limited bandwidth of the bus is exposed to saturation which leads to increased latency of memory requests. The capacity of main memory is limited by cost and energy constraints, and leads to costly page faults when exhausted. Compression is a potential solution to both of these limitations. By transmitting compressed data over the bus, the hardware-limited bandwidth can be utilized more efficiently, overall latency reduced and performance improved. By compacting compressed data in physical memory, the effective capacity can be increased, mitigating page faults.

Existing compression systems typically target only one of these bottlenecks, aiming to reduce bandwidth or increase memory capacity. The straight-forward approach to bandwidth reduction requires no compaction in memory, and thus does not affect memory capacity. State-of-the-art memory compaction systems introduce additional traffic, cancelling out the bandwidth benefits of their compression.

This thesis presents several compression techniques aimed at mitigating these bottlenecks, primarily by identifying approximation-tolerant data and applying *lossy compression* to them.

6.1 Summary

Chapter 2 presented Approximate Value Reconstruction (AVR), a memory compression system which uses lossy compression to relieve the bandwidth pressure on the main memory bus. In order to achieve compression ratios of up to 16×, AVR compresses large (1kB) contiguous blocks using a low-complexity *downsampling* compressor. AVR uses a Decoupled Sectored design for the Last-Level Cache, modified to allow both compressed and uncompressed data to be co-located. Two separate, local, error thresholds are maintained, allowing the application to set an acceptable level of approximation.

Chapter 3 proposed MemSZ, an extension to AVR. MemSZ introduces a highly parallelized implementation of the Squeeze (SZ) lossy compression scheme. An additional, global, error threshold ensures that accumulated approximation error does not exceed acceptable levels. The related MemSZ-DC implements memory compression, and uses a 3D-stacked DRAM cache to

reduce the number of needed modifications to on-chip components such as the SRAM LLC.

Chapter 4 introduced L^2C, a hybrid lossy and lossless memory compression system. The main improvement over MemSZ is the ability to use lossless compression for data which are non-approximable, or data whose accumulated approximation error have exceeded acceptable levels. L^2C extends the cache design of MemSZ to support multiple granularities of compression, in order to best fit the differing characterstics of the two supported compression schemes.

Chapter 5 detailed FlatPack, a novel memory compaction scheme designed to improve both memory capacity and memory bandwidth. FlatPack exploits the large compression blocks of AVR, MemSZ and L^2C to achieve greater flexibility in physical memory. By fragmenting large blocks at the Memory Access Granularity (MAG), they can be reorganized in physical memory without affecting other blocks. Crucially, FlatPack can respond to growing and shrinking blocks, reassigning free space to whichever block requires it.

6.2 Contributions

AVR is the first lossy memory compression scheme for general-purpose architectures. It uses the following techniques to more efficiently utilize the available main memory bandwidth:

- Low-complexity lossy compression of large blocks;
- Co-locating compressed blocks with uncompressed cache lines in the LLC;
- Handling LLC evictions in a *lazy* manner, reducing the overhead of block recompression;
- Managing blocks which do not compress well;
- Selectively storing decompressed data in the LLC to reduce the overhead of decompression.

For applications with significant fractions of approximation-tolerant data, AVR reduces memory traffic by up to 70%, improving system performance by up to 55% and lowering energy consumption by up to 20% with a maximum of 1% error introduced in application output.

MemSZ is a more advanced memory compression system, which extends upon AVR in the following ways:

- MemSZ implements a more advanced lossy compression method, based on SZ.
- A better cache replacement policy reduces duplication of data, utilizing LLC capacity more efficiently.
- A new, global, error control mechanism limits accumulated error over the lifetime of a block.
- MemSZ can be combined with a 3D-stacked DRAM Cache, allowing for fewer modifications of on-chip components.

Compared to a baseline system with no compression, MemSZ reduces memory traffic by up to 81%, improves performance by up to 62%, and reduces total system energy consumption by up to 25%.

L^2C is the first general-purpose memory compression system to combine lossy and lossless compression. In addition to memory compression, L^2C is applicable to I/O traffic. The key techniques to support these features are:

- Support for two granularities of memory blocks, tailored to each compression method;
- A cache structure and main memory layout supporting blocks of both granularities;
- A mechanism to dynamically select the most suitable compression method for each block;
- A hybrid metadata format supporting the two block granularities, partially embedding metadata with compressed blocks to reduce traffic overhead.

L^2C is able to compress data which AVR and MemSZ had to leave uncompressed, leading to additional benefits. Compared to MemSZ, L^2C reduces memory traffic by 18%, increases performance by 9% and reduces system energy by 3%.

FlatPack is a flexible memory compaction system which extends L^2C with the ability to reduce physical memory footprint. While existing memory compression systems target either bandwidth or footprint, FlatPack is able to improve both by reducing the traffic overhead of compaction. FlatPack makes the following contributions:

- A flexible format of compressed pages, allowing compressed blocks to be fragmented and share expansion space in physical memory;
- A hardware mechanism enabling the memory controller to utilize that format and dynamically reorganize data within a physical page, without software intervention and without affecting unrelated blocks.

FlatPack is able to reduce baseline memory traffic by 48%. In a multi-core system, FlatPack improves mean system performance by 107% and energy consumption by 36%. Compared to Compresso, the previous state-of-the-art in memory compaction, FlatPack achieves a footprint within 6% and improves mean system performance and energy consumption by 36% and 12%, respectively. Memory traffic is reduced by up to 67%.

6.3 Future Work

The systems proposed in this thesis open up several avenues of future research. This section contains a few suggestions for inquiry.

The principal trade-off of all approximate computing techniques is reduced quality for increased performance or energy efficiency. This trade-off poses a risk, compared to the traditional precise paradigm. As a result, approximate computing is not yet widely accepted as a feasible option outside of a few specific domains. One way to reduce this risk is more robust error models and stricter guarantees on error bounds. AVR and MemSZ both offer mechanisms

to limit the error of approximation, but base these mechanisms on estimates rather than precise measurement. A beneficial future direction of research is mechanisms to model the exact relationship between input approximation and output error, which is highly application dependent.

A simpler shortcoming of the proposed designs lies in the annotation of approximable data, and the distinction between approximable and non-approximable data. The AVR family of compression systems all rely on the application developer to manually provide these annotations, as well as determining proper values for the various error thresholds. This additional work is another hurdle for the adoption of lossy compression in specific and approximate computing in general. With sufficient source-code level metadata, it should be possible to devise an automatic profiling system which is able to identify the best targets for approximation, based on input constraints. This would allow the developer to make the more direct decision of determining an acceptable *output* error, rather than experimentally determining the proper *input* error thresholds required to achieve that goal.

A trivial addition to improve the effects of AVR, MemSZ, L^2C and FlatPack is the support for multiple compression algorithms. Some compressors such as *Frequent Pattern Compression* (FPC) and *Global Base-Delta-Immediate* (GBDI) target specific data types [112, 113], while others like *Bit-Plane Compression* (BPC) are type-agnostic [18], and systems like HyComp attempt to select the most effective among multiple supported compressors [41]. Similarly, alternative lossy compressors like ZFP may offer better compressibility depending on application characteristics [114]. This type of support would require a few bits of metadata (indicating the current compression method of each compressed block) as well as hardware modules for each supported method. Decompression latency and energy is not affected by such additions, and compression with all supported methods can be performed in parallel. The main benefit is the ability to dynamically choose the most effective compression algorithm at runtime, which increases the potential compression ratio.

Finally, there is potential for co-design between lossy memory compression and other approximation techniques. For example, AVR stores outliers at half precision. An arithmetic unit with sufficient information could reduce computation on these values to half-precision, thus reducing energy consumption without introducing additional error.

Bibliography

[1] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. W. Keller, "Energy management for commercial servers," *Computer*, vol. 36, no. 12, pp. 39–48, 2003.

[2] J. Ahn *et al.*, "Pim-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture," in *ISCA*. ACM/IEEE, 2015, pp. 336–348.

[3] Y. Zhou and D. Wentzlaff, "Mitts: Memory inter-arrival time traffic shaping," in *ISCA*. ACM/IEEE, 2016, pp. 532–544.

[4] M. O'Connor *et al.*, "Fine-grained dram: energy-efficient dram for extreme bandwidth systems," in *MICRO*, 2017, pp. 41–54.

[5] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin, "Scaling the bandwidth wall: Challenges in and avenues for cmp scaling," in *ISCA*, 2009, pp. 371–382.

[6] J. Yang and J. Seymour, "Pmbench: A micro-benchmark for profiling paging performance on a system with low-latency SSDs," in *Information Technology-New Generations*. Springer, 2018, pp. 627–633.

[7] NVIDIA, "Nvidia tegra x1: Nvidia's new mobile superchip," whitepaper, 2015.

[8] M. Doggett, "Texture caches," in *MICRO*, vol. 32, no. 3. IEEE, 2012, pp. 136–141.

[9] V. Sathish *et al.*, "Lossless and lossy memory I/O link compression for improving performance of GPGPU workloads," in *PACT*, 2012, pp. 325–334.

[10] 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*, vol. 45, no. 2, pp. 271–285, 2001.

[11] E. Hallnor and S. Reinhardt, "A unified compressed memory hierarchy," in *11th International Symposium on High-Performance Computer Architecture*, 2005, pp. 201–212.

[12] M. Ekman and P. Stenstrom, "A robust main-memory compression scheme," in *SIGARCH C.A. News*, vol. 33, 2005, pp. 74–85.

[13] G. Pekhimenko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Linearly compressed pages: a low-complexity, low-latency main memory compression framework," in *MICRO*. IEEE, 2016, pp. 172–184.

[14] Y. Cao, L. Chen, and Z. Zhang, "Flexible memory: A novel main memory architecture with block-level memory compression," in *2015 IEEE International Conference on Networking, Architecture and Storage (NAS)*, 2015, pp. 285–294.

[15] J. Zhao, S. Li, J. Chang, J. L. Byrne, L. L. Ramirez, K. Lim, Y. Xie, and P. Faraboschi, "Buri: Scaling big-memory computing with hardware-based memory expansion," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 3, p. 31, 2015.

[16] E. Choukse, M. Erez, and A. R. Alameldeen, "Compresso: Pragmatic main memory compression," in *MICRO*. IEEE, 2018, pp. 546–558.

[17] E. Choukse, M. B. Sullivan, M. O'Connor, M. Erez, J. Pool, D. Nellans, and S. W. Keckler, "Buddy compression: Enabling larger memory for deep learning and hpc workloads on gpus," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 926–939.

[18] J. Kim, M. Sullivan, E. Choukse, and M. Erez, "Bit-plane compression: Transforming data for better compression in many-core architectures," in *ISCA*. ACM/IEEE, 2016, pp. 329–340.

[19] V. K. Chippa *et al.*, "Analysis and characterization of inherent application resilience for approximate computing," in *DAC*, 2013, pp. 1–9.

[20] J. R. Goldschneider, "Lossy compression of scientific data via wavelets and vector quantization," Ph.D. dissertation, Univ. of Washington, 1997.

[21] M. Ware, K. Rajamani, M. Floyd, B. Brock, J. C. Rubio, F. Rawson, and J. B. Carter, "Architecting for power management: The ibm power7 approach," in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, Jan 2010, pp. 1–11.

[22] L. Benini, D. Bruni, B. Ricco, A. Macii, and E. Macii, "An adaptive data compression scheme for memory traffic minimization in processor-based systems," in *ISCAS*, vol. 4. IEEE, 2002, pp. IV–IV.

[23] J. Dusser and A. Seznec, "Decoupled zero-compressed memory," in *Int. Conf. on HiPEAC*. ACM, 2011, pp. 77–86.

[24] S. Kim, S. Lee, T. Kim, and J. Huh, "Transparent dual memory compression architecture," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017, pp. 206–218.

[25] S. Hong, P. J. Nair, B. Abali, A. Buyuktosunoglu, K. Kim, and M. Healy, "Attaché: Towards ideal memory compression by mitigating metadata bandwidth overheads," in *MICRO*. IEEE, 2018, pp. 326–338.

[26] R. Kanakagiri, B. Panda, and M. Mutyam, "MBZip: Multiblock data compression," *TACO*, vol. 14, no. 4, pp. 1–29, 2017.

[27] A. Shafiee, M. Taassori, R. Balasubramonian, and A. Davis, "Memzip: Exploring unconventional benefits from memory compression," in *HPCA*, 2014, pp. 638–649.

[28] A. Jain *et al.*, "Concise loads and stores: The case for an asymmetric compute-memory architecture for approximation," in *MICRO*. IEEE, 2016, pp. 1–13.

[29] A. Angerd *et al.*, "A framework for automated and controlled floating-point accuracy reduction in graphics applications on gpus," *TACO*, vol. 14, no. 4, pp. 1–25, 2017.

[30] P. Judd *et al.*, "Proteus: Exploiting numerical precision variability in deep neural networks," in *ICS*. ACM, 2016, pp. 1–12.

[31] J. San Miguel *et al.*, "Load value approximation," in *MICRO*. IEEE, 2014, pp. 127–139.

[32] B. Thwaites *et al.*, "Rollback-free value prediction with approximate loads," in *PACT*, 2014, pp. 493–494.

[33] A. Yazdanbakhsh *et al.*, "RFVP: Rollback-free value prediction with safe-to-approximate loads," *TACO*, vol. 12, no. 4, p. 62, 2016.

[34] S. Di and F. Cappello, "Fast error-bounded lossy hpc data compression with sz," in *IPDPS*. IEEE, 2016, pp. 730–739.

[35] A. Seznec, "Decoupled sectored caches: conciliating low tag implementation cost and low miss ratio," in *ISCA*. ACM/IEEE, Apr 1994, pp. 384–393.

[36] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *SIGARCH C.A. News*, vol. 23, no. 1, pp. 20–24, 1995.

[37] J. Ahn *et al.*, "A scalable processing-in-memory accelerator for parallel graph processing," in *ISCA*. ACM/IEEE, 2015, pp. 105–117.

[38] M. Pavlovic, Y. Etsion, and A. Ramirez, "On the memory system requirements of future scientific applications: Four case-studies," in *IISWC*, 2011, pp. 159–170.

[39] J. San Miguel *et al.*, "Doppelganger: A cache for approximate computing," in *MICRO*. IEEE, 2015, pp. 50–61.

[40] S. Sardashti, A. Seznec, and D. Wood, "Yet another compressed cache: a low-cost yet effective compressed cache," *TACO*, vol. 13, no. 3, p. 27, 2016.

[41] A. Arelakis, F. Dahlgren, and P. Stenstrom, "HyComp: a hybrid cache compression method for selection of data-type-specific compression methods," in *MICRO*. IEEE, 2015, pp. 38–49.

[42] S. Sardashti *et al.*, "Decoupled compressed cache: Exploiting spatial locality for energy-optimized compressed caching," in *MICRO*. IEEE, 2013, pp. 62–73.

[43] B. Panda *et al.*, "Synergistic cache layout for reuse and compression," in *PACT*, 2018, pp. 1–13.

[44] J. San Miguel *et al.*, "The bunker cache for spatio-value approximation," in *MICRO*. IEEE, 2016, pp. 1–12.

[45] A. Sampson *et al.*, "Enerj: Approximate data types for safe and general low-power computation," *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 164–174, 2011.

[46] L. Saldanha *et al.*, "Float-to-fixed and fixed-to-float hardware converters for rapid hardware/software partitioning of floating point software applications to static and dynamic fixed point coprocessors," *Des. Aut. for Emb. Sys.*, vol. 13, no. 3, pp. 139–157, 2009.

[47] E. Meijering, "A chronology of interpolation: from ancient astronomy to modern signal and image processing," *Proc. of IEEE*, vol. 90, 2002.

[48] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *ACM SIGPLAN Notices*, vol. 40, 2005, pp. 190–200.

[49] D. Genbrugge, S. Eyerman, and L. Eeckhout, "Interval simulation: Raising the level of abstraction in architectural simulation," in *HPCA*, Jan 2010, pp. 1–12.

[50] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A cycle accurate memory system simulator," *IEEE CAL*, vol. 10, no. 1, pp. 16–19, 2011.

[51] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO*. IEEE, 2009, pp. 469–480.

[52] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP lab.*, vol. 27, pp. 22–31, 2009.

[53] J. Quinn Michael, "Parallel programming in c with mpi and openmp," ISBN 0-07-058201-7, Tech. Rep., 2004.

[54] S. Ansumali *et al.*, "Minimal entropic kinetic models for hydrodynamics," *Europhysics Letters*, vol. 63, no. 6, p. 798, 2003.

[55] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *SIGARCH C.A. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006.

[56] University of Chicago ASCF Center, "Flash4 user's guide," 2018, online; Accessed 2019-04-18. [Online]. Available: http://flash.uchicago.edu/site/flashcode/user_support/flash4_ug_4p6.pdf

[57] "1D K-Means, Open Source," 2018. [Online]. Available: https://github.com/eldstal/kmeans

[58] Lantmäteriet, "Swedish topological survey hdb 50+ västra götaland, zone 63_3," 2016. [Online]. Available: https://www.lantmateriet.se/

[59] A. Yazdanbakhsh *et al.*, "Axbench: A multiplatform benchmark suite for approximate computing," *IEEE Design & Test*, vol. 34, no. 2, pp. 60–68, 2017.

[60] Q. Xiong *et al.*, "Ghostsz: A transparent fpga-accelerated lossy compression framework," in *FCCM*. IEEE, 2019, pp. 258–266.

[61] J. Sohn and E. E. Swartzlander, "A fused floating-point three-term adder," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 61, no. 10, pp. 2842–2850, Oct 2014.

[62] E. Vasilakis, V. Papaefstathiou, P. Trancoso, and I. Sourdis, "Decoupled fused cache: Fusing a decoupled LLC with a DRAM cache," *TACO*, vol. 15, no. 4, pp. 65:1–65:23, 2019.

[63] C. Huang and V. Nagarajan, "Atcache: Reducing dram cache latency via a small sram tag cache," in *PACT*. ACM, 2014, pp. 51–60.

[64] D. Jevdjic *et al.*, "Unison cache: A scalable and effective die-stacked dram cache," in *MICRO*. IEEE, 2014, pp. 25–37.

[65] Y. L. et al, "A fully associative, tagless dram cache," in *ISCA*, ser. ISCA '15. New York, NY, USA: ACM, 2015, pp. 211–222.

[66] H. Jang, Y. Lee, J. Kim, Y. Kim, J. Kim, J. Jeong, and J. W. Lee, "Efficient footprint caching for tagless dram caches," in *HPCA*. IEEE, 2016, pp. 237–248.

[67] A. Eldstål-Damlin, P. Trancoso, and I. Sourdis, "AVR: Reducing memory traffic with approximate value reconstruction," in *ICPP*, 2019, pp. 1–10.

[68] J. Gantz and D. Reinsel, "The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east," December 2012.

[69] A. R. Alameldeen and D. A. Wood, "Frequent pattern compression: A significance-based compression scheme for l2 caches," *Dept. Comp. Scie., Univ. Wisconsin-Madison, Tech. Rep*, vol. 1500, 2004.

[70] X. Chen, L. Yang, R. P. Dick, L. Shang, and H. Lekatsas, "C-pack: A high-performance microprocessor cache compression algorithm," *IEEE transactions on very large scale integration (VLSI) systems*, vol. 18, no. 8, pp. 1196–1208, 2010.

[71] G. Pekhimenko, V. Seshadri, O. Mutlu, M. A. Kozuch, P. B. Gibbons, and T. C. Mowry, "Base-delta-immediate compression: Practical data compression for on-chip caches," in *PACT*, 2012, pp. 377–388.

[72] A. Arelakis and P. Stenstrom, "SC2: a statistical compression cache scheme," in *ACM SIGARCH Computer Arch. News*, vol. 42. IEEE Press, 2014, pp. 145–156.

[73] C. J. Deepu *et al.*, "A hybrid data compression scheme for power reduction in wireless sensors for IoT," *IEEE Trans. on Biomedical Circuits and Systems*, vol. 11, no. 2, pp. 245–254, 2017.

[74] B. Abali, B. Blaner, J. J. Reilly, M. Klein, A. Mishra, C. B. Agricola, B. Sendir, A. Buyuktosunoglu, C. Jacobi, W. J. Starke, H. Myneni, and C. Wang, "Data compression accelerator on IBM POWER9 and z15 processors : Industrial product," in *ISCA '20*. IEEE, 2020, pp. 1–14.

[75] A. Malek *et al.*, "Odd-ecc: on-demand dram error correcting codes," in *MEMSYS*, 2017, pp. 96–111.

[76] A. Eldstål-Ahrens and I. Sourdis, "MemSZ: Squeezing memory traffic with lossy compression," *ACM TACO*, vol. 17, no. 4, pp. 40:1–40:25, Nov. 2020.

[77] M. Vecchio *et al.*, "Adaptive lossless entropy compressors for tiny iot devices," *IEEE Transactions on Wireless Communications*, vol. 13, no. 2, pp. 1088–1100, 2014.

[78] G. Campobello *et al.*, "An efficient lossless compression algorithm for electrocardiogram signals," in *2018 26th European Signal Processing Conference (EUSIPCO)*, 2018, pp. 777–781.

[79] R. Vestergaard *et al.*, "Generalized deduplication: Lossless compression for large amounts of small iot data," in *European Wireless 2019; 25th European Wireless Conference*, 2019, pp. 1–5.

[80] D. Blalock *et al.*, "Sprintz: Time series compression for the internet of things," *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, vol. 2, no. 3, Sep. 2018. [Online]. Available: https://doi.org/10.1145/3264903

[81] B. R. Stojkoska and Z. Nikolovski, "Data compression for energy efficient iot solutions," in *2017 25th Telecommunication Forum (TELFOR)*, 2017, pp. 1–4.

[82] K. B. Adedeji, "Performance evaluation of data compression algorithms for iot-based smart water network management applications," *Journal of Applied Science & Process Engineering*, vol. 7, no. 2, pp. 554–563, 2020.

[83] A. Moon *et al.*, "Lossy compression on IoT big data by exploiting spatiotemporal correlation," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, 2017, pp. 1–7.

[84] A. Khelifati *et al.*, "Corad: Correlation-aware compression of massive time series using sparse dictionary coding," in *2019 IEEE International Conference on Big Data (Big Data)*, 2019, pp. 2289–2298.

[85] A. Ukil *et al.*, "Adaptive sensor data compression in iot systems: Sensor data analytics based approach," in *IEEE Int. Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2015, pp. 5515–5519.

[86] ——, "Iot data compression: Sensor-agnostic approach," in *2015 Data Compression Conference*, 2015, pp. 303–312.

[87] H. Esmaeilzadeh *et al.*, "Neural acceleration for general-purpose approximate programs," in *MICRO*. IEEE, 2012, pp. 449–460.

[88] C. Alvarez, J. Corbal, and M. Valero, "Fuzzy memoization for floating-point multimedia applications," *IEEE Trans. Comput.*, vol. 54, no. 7, pp. 922–927, 2005.

[89] M. de Kruijf *et al.*, "Relax: An architectural framework for software recovery of hardware faults," in *ISCA*. ACM/IEEE, 2010, pp. 497–508.

[90] D. Jevdjic, K. Strauss, L. Ceze, and H. S. Malvar, "Approximate storage of compressed and encrypted videos," in *ASPLOS*, 2017, pp. 361–373.

[91] A. Sampson *et al.*, "Approximate storage in solid-state memories," *ACM TOCS*, vol. 32, no. 3, p. 9, 2014.

[92] A. Angerd *et al.*, "A GPU register file using static data compression," in *ICPP*. ACM, 2020.

[93] A. Ranjan, A. Raha, V. Raghunathan, and A. Raghunathan, "Approximate memory compression," *IEEE TVLSI*, vol. 28, no. 4, pp. 980–991, 2020.

[94] S. Lal, J. Lucas, and B. Juurlink, "SLC: Memory access granularity aware selective lossy compression for gpus," in *DATE*. IEEE, 2019, pp. 1184–1189.

[95] NASA/JPL, "JPL SMAP level 3 CAP sea surface salinity standard mapped image 8-day running mean v4.3 validated dataset," 2019, online; Accessed 2020-01-18.

[96] T. McGlynn *et al.*, "Skyview: The multi-wavelength sky on the internet," in *Symposium-International Astronomical Union*, vol. 179. Cambridge University Press, 1998, pp. 465–466.

[97] Swedish Meteorological and Hydrological Institute, "STRÅNG - a mesoscale model for solar radiation," 2020, online; Accessed 2020-01-18. [Online]. Available: http://strang.smhi.se/

[98] S. Lobov *et al.*, "Latent factors limiting the performance of sEMG-interfaces," *Sensors*, vol. 18, no. 4, p. 1122, 2018.

[99] A. L. Goldberger *et al.*, "PhysioBank, PhysioToolkit, and PhysioNet: Components of a new research resource for complex physiologic signals," *Circulation*, vol. 101, no. 23, pp. e215–e220, 2000 (June 13).

[100] S. Makonin *et al.*, "Electricity, water, and natural gas consumption of a residential house in Canada from 2012 to 2014," *Scientific Data*, vol. 3, no. 160037, pp. 1–12, 2016.

[101] S. De Vito *et al.*, "On field calibration of an electronic nose for benzene estimation in an urban pollution monitoring scenario," *Sensors and Actuators B: Chemical*, vol. 129, no. 2, pp. 750–757, 2008.

[102] J. Fonollosa *et al.*, "Reservoir computing compensates slow response of chemosensor arrays exposed to fast varying gas concentrations in continuous monitoring," *Sensors and Actuators B: Chemical*, vol. 215, pp. 618–629, 2015.

[103] N. Helwig *et al.*, "Condition monitoring of a complex hydraulic system using multivariate statistics," in *IEEE I2MTC*, 2015, pp. 210–215.

[104] Y. Zhang and R. Gupta, "Enabling partial cache line prefetching through data compression," in *2003 International Conference on Parallel Processing, 2003. Proceedings.* IEEE, 2003, pp. 277–285.

[105] S. Park, I. Kang, Y. Moon, J. H. Ahn, and G. E. Suh, "BCD deduplication: Effective memory compression using partial cache-line deduplication," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 52–64. [Online]. Available: https://doi.org/10.1145/3445814.3446722

[106] D. J. Palframan, N. S. Kim, and M. H. Lipasti, "Cop: To compress and protect main memory," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 682–693. [Online]. Available: https://doi.org/10.1145/2749469.2750377

[107] L. Seiler, D. Lin, and C. Yuksel, "Compacted cpu/gpu data compression via modified virtual address translation," *Proc. ACM Comput. Graph. Interact. Tech.*, vol. 3, no. 2, Aug. 2020. [Online]. Available: https://doi.org/10.1145/3406177

[108] P. A. Franaszek and D. E. Poff, "Management of guest os memory compression in virtualized systems," U.S. Patent US20 080 307 188A1, 2007.

[109] "SPEC CPU 2017," Standard Performance Evaluation Corporation, 2017. [Online]. Available: https://www.spec.org/cpu2017

[110] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray Users Group (CUG)*, vol. 19, pp. 45–74, 2010.

[111] J. Leskovec and R. Sosič, "Snap: A general-purpose network analysis and graph-mining library," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 8, no. 1, pp. 1–20, 2016.

[112] M. Burtscher and P. Ratanaworabhan, "FPC: A high-speed compressor for double-precision floating-point data," *IEEE Transactions on Computers*, vol. 58, no. 1, pp. 18–31, 2008.

[113] A. Angerd, A. Arelakis, V. Spiliopoulos, E. Sintorn, and P. Stenström, "GBDI: Going beyond base-delta-immediate compression with global bases," in *2022 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Washington, DC, USA: IEEE Computer Society, 2022.

[114] G. Sun and S. Jun, "ZFP-V: Hardware-optimized lossy floating point compression," in *ICFPT*. IEEE, 2019, pp. 117–125.