Linearly Compressed Pages: A Main Memory Compression Framework with Low Complexity and Low Latency

Gennady Pekhimenko
Advisors: Todd C. Mowry and Onur Mutlu
Computer Science Department, Carnegie Mellon University
gpekhime@cs.cmu.edu
Submitted to the ACM Student Research Competition Grand Finals 2013

1 Introduction

Main memory, commonly implemented using DRAM technology, is a critical resource in modern systems. Its capacity must be sufficiently provisioned to prevent the target application's working set from overflowing into the backing store (e.g., hard disk, flash storage) that is slower by orders of magnitude compared to DRAM. The required main memory capacity in future systems is increasing rapidly due to two major trends: increased working sets for many applications, more sharing of main memory due to multi-core designs. DRAM already constitutes a significant portion of the system's cost and power budget. Therefore, adding more DRAM to meet the increasing main memory demand is ideally not desirable. To make matters worse, for signal integrity reasons, today's high frequency memory channels prevent many DRAM modules from being connected to the same channel, effectively limiting the maximum amount of DRAM in a system unless one resorts to expensive off-chip signaling buffers.

A promising technique to address the DRAM capacity problem is *data compression*. If a piece of data is compressed, it can be stored in a smaller amount of physical memory. Prior works have noted that in-memory data has significant redundancy and proposed different techniques to compress data in both caches [2,9,16,24] and main memory [1,6,8,23]. Depending on the compressibility of data, storing compressed data in main memory can be an effective way of significantly increasing memory capacity without incurring significant additional cost or power.

In this work, our goal is to take a closer look at designing compressed memory hierarchies. To this end, we aim to expore memory compression mechanisms that can increase main memory capacity (and decrease off-chip bandwidth consumption) without noticable increase in memory access latency.

2 Background and Motivation

Data compression is a widely used technique to improve the efficiency of storage structures and communication channels. By reducing the amount of redundancy in data, compression increases the effective capacity and bandwidth without increasing the system cost and power consumption significantly. One primary downside of data compression is that the compressed data must be decompressed before it can be used. Therefore, for latency-critical applications, using complex dictionary-based compression algorithms (such as Lempel-Ziv [26] or Huffman encoding [10]) significantly degrade performance due to their high decompression latency. However, to compress in-memory data, prior works have proposed simpler algorithms that have low decompression latencies while still achieving high compression ratios.

2.1 Compressing In-Memory Data

Several studies [2, 3, 16, 25] have shown that in-memory data has different exploitable patterns that allow for simpler compression techniques. Frequent value compression (FVC) [25] shows that a small set of values represent a majority of the values occurring in an application's working set. The proposed compression algorithm exploits this observation by encoding such frequently-occurring 4-byte values with fewer bits. Frequent pattern compression (FPC) [3] shows that a majority of words (4-byte elements) in memory fall under a few frequently occurring patterns. FPC compresses individual words within a cache line by encoding the frequently occurring patterns with fewer bits. A recently proposed compression technique, Base-Delta-Immediate (BDI) compression [16] observes that in many cases words co-located in memory have small differences in their values. Based on this observation, BDI compression encodes a block of data (e.g., a cache line) as a base-value and an array of differences that represent the deviation from the base-value for each individual word in the block.

Due to the simplicity of these compression algorithms, their decompression latencies are much shorter than those of dictionary-based algorithms: 5 cycles for FVC/FPC [3] and 1 cycle for BDI [16] in contrast to 64 cycles for a variant of Lempel-Ziv [26] used in IBM MXT [1]. These low-latency compression algorithms have been proposed in the context of on-chip caches, but can be also used as a part of a main memory compression framework, including the one we pro-

pose in this work.

2.2 Challenges in Main Memory Compression

The key requirement in building an efficient hardware-based main memory compression framework is low complexity and low latency. There are three challenges that need to be addressed to satisfy this requirement. Figures 1, 2 and 3 pictorially show these three challenges.

Challenge 1. Unlike on-chip last-level caches, which are managed and accessed at the same granularity (e.g., a 64-byte cache line), main memory is managed and accessed at different granularities. Most modern systems employ virtual memory and manage main memory at a large page granularity (e.g., 4kB or 8kB). However, processors access data from the main memory at a cache line granularity to avoid transferring large pages across the bus. When main memory is compressed, different cache lines within a page can be compressed to different sizes. The main memory address of a cache line is therefore dependent on the sizes of the compressed cache lines that come before it in the page. As a result, the processor (or the memory controller) should explicitly compute the location of a cache line within a compressed main memory page before accessing it (Figure 1), e.g., as in [8]. This computation not only increases complexity, but can also lengthen the critical path of accessing the cache line from the main memory and from the physically addressed cache (as we describe in Challenge 3). Note that this is not a problem for a system that does not employ main memory compression because, in such a system, the offset of a cache line within the physical page is the *same* as the offset of the cache line within the corresponding virtual page.

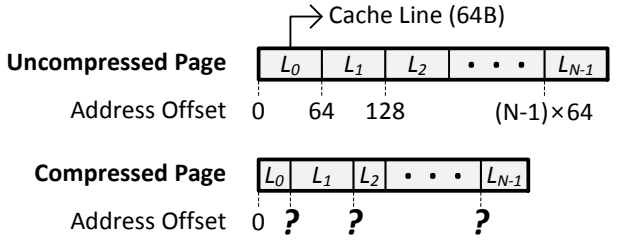


Figure 1: Challenge 1: Main Memory Address Computation

Challenge 2. Depending on their compressibility, different physical pages are compressed to different sizes. This increases the complexity of the memory management module of the operating system for two reasons (as shown in Figure 2). First, the operating system needs to allow mappings between fixed-size virtual pages and variable-size physical pages. Second, the operating system must implement mechanisms to efficiently handle fragmentation in main memory.

Challenge 3. The additional layer of indirection between virtual and physical addresses complicates on-chip cache management. On-chip caches (including L1 caches) typically employ tags derived from the physical address (instead of the virtual address) to avoid aliasing – i.e., two cache lines having the same virtual address but different physical address (homonyms) or vice versa (synonyms) [12, 18]. In such systems, every cache access requires the physical address of the corresponding cache line to be computed. As a result, the additional layer of indirection between the virtual and physical addresses introduced by main memory compression can lengthen the critical path of latency-critical L1 cache accesses (Figure 3).

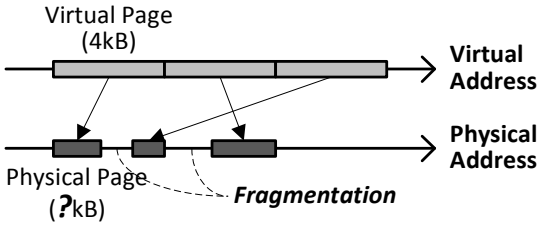


Figure 2: Challenge 2: Main Memory Mapping and Fragmentation

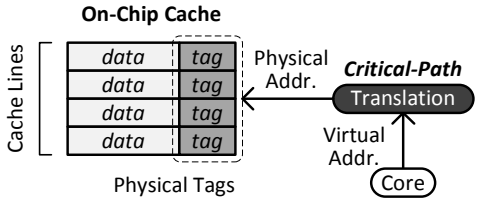


Figure 3: Challenge 3: Physically Tagged Caches

3 Related Work

Multiple previous works investigated the possibility of using compression for main memory [1,5,6,8,11,17,23]. Among them, two in particular are the most closely related to the design proposed in this work, because both of them are mostly hardware designs. Therefore, we describe these two designs along with their shortcomings.

Tremaine et al. [21] proposed a memory controller design, Pinnacle, based on IBM Memory Extension Technology (MXT) [1] that employed Lempel-Ziv compression [26] to manage main memory. To address the three key challenges, Pinnacle employs two techniques.

First, Pinnacle internally uses a 32MB cache managed at 1kB granularity, same as the granularity at which blocks are compressed. This cache reduces the number of accesses to main memory by exploiting locality in access patterns, thereby reducing the performance degradation due to the address computation (Challenge 1). However, there are drawbacks to this approach: 1) such a large cache adds significant cost to the memory controller, 2) the approach requires the main memory address computation logic to be present and used when an access misses in the 32MB cache, 3) if caching is not effective (e.g., due to lack of locality or larger-than-cache working set sizes), this approach cannot reduce the performance degradation due to main memory address computation. In addition to this, Pinnacle's decompression latency,

which is on the critical path of a memory access, is 64 processor cycles.

Second, to avoid complex changes to the operating system and on-chip cache-tagging logic, Pinnacle introduces a *real* address space between the virtual and physical address spaces. The real address space is uncompressed and is twice the size of the actual available physical memory. The operating system maps virtual pages to same-size pages in the real address space, which addresses Challenge 2. On-chip caches are tagged using the real address (instead of the physical address, which is dependent on compressibility), which effectively solves Challenge 3.

Ekman and Stenstrom [8] proposed a main memory compression design to address the drawbacks of MXT. To compress pages, they use a variant of the FPC technique [2], which has a much smaller decompression latency (5 cycles [3]) than the Lempel-Ziv compression in Pinnacle (64 cycles [1]). To avoid the long latency of a cache line's main memory address computation (Challenge 1), their design overlaps this computation with the last-level (L2) cache access. For this purpose, their design extends the page table entries to store the compressed sizes of all the lines within the page. This information is loaded into a hardware structure called the *Block Size Table* (BST, that is s at least twice as large as the TLB). On an L1 cache miss, the BST is accessed in parallel with the L2 cache to compute the exact main memory address of the corresponding cache line. While the proposed mechanism reduces the latency penalty of accessing compressed blocks by overlapping main memory address computation with L2 cache access, the main memory address computation is performed on *every* L2 cache access. This leads to significant wasted work and additional power consumption. To address Challenge 2, the operating system uses multiple pools of fixed-size physical pages. This reduces the complexity of managing physical pages at a fine granularity. Ekman and Stenstrom [8] do not address Challenge 3.

In summary, prior works on hardware-based main memory compression mitigate the performance degradation due to the main memory address computation problem (Challenge 1) by either adding large hardware structures that consume significant area and power [1] or by using techniques that require energy-inefficient hardware and lead to wasted energy [8].

4 Linearly Compressed Pages

The main shortcoming of prior approaches to main memory compression is that different cache lines within a physical page can be compressed to different sizes depending on the compression scheme. As a result, the location of a compressed cache line within a physical page depends on the sizes of all the compressed cache lines before it in the same page. This requires the memory controller to explicitly compute the main memory location of a compressed cache line before it can access it (Challenge 1).

To address this shortcoming, we propose a new approach to compressing pages, called the *Linearly Compressed Page* (LCP) [15]. The key idea of LCP is to *use a fixed size for compressed cache lines within a page* (which eliminates complex and long-latency main memory address calculation problem that arises due to variable-size cache lines), and still enable a page to be compressed even if not all cache lines within the page can be compressed to that fixed size (which enables high compression ratios).

There are two key design choices made in LCP to improve compression ratio in the presence of fixed-size compressed cache lines. First, the target size for the compressed cache lines can be different for different pages, depending on the algorithm used for compression and the data stored in the pages. The LCP-based framework identifies this target size for a page when the page is compressed for the first time (or recompressed). Second, not all cache lines within a page can be compressed to a specific fixed size. Also, a cache line which is originally compressed to the target size may later become uncompressible due to a write. One approach to handle such cases is to store the entire page in uncompressed format even if a single line cannot be compressed into the fixed size. However, this inflexible approach can lead to significant reduction in the benefits from compression and may also lead to frequent compression/decompression of entire pages. To avoid these problems, LCP stores such uncompressible cache lines of a page separately from the compressed cache lines (but still within the page), along with the metadata required to locate them.

Figure 4 shows the organization of an example Linearly Compressed Page, based on the ideas described above. In this example, we assume that the virtual page size is 4kB, the uncompressed cache line size is 64B, and the target compressed cache line size if 16B. As shown in the figure, the LCP contains three distinct regions.

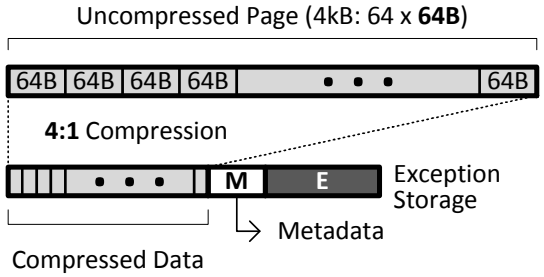


Figure 4: Organization of a Linearly Compressed Page

The first region, *the compressed data region*, contains a 16-byte slot for each cache line in the virtual page. If a cache line is compressible, the corresponding slot stores the compressed version of the cache line. However, if the cache line is not compressible, the corresponding slot is assumed to contain invalid data. In this design, we refer to such an uncompressible cache line as an "exception". The second region, *metadata*, contains all necessary information to identify and locate exceptions of a page. The third region, *the exception storage*, is the place where all the exceptions of the LCP are stored in their uncompressed form. The LCP design allows the exception storage to contain free space. In other words, not all

entries in the exception storage may store valid exceptions. This allows the memory controller to use the free space for storing future exceptions, and also simplifies the operating system page management mechanism.

4.1 Memory Compression Framework

The LCP-based main memory compression framework [1] consists of components that handle three key issues: 1) page compression, 2) handling a cache line read from main memory, and 3) handling a cache line writeback into main memory. Figure 5 shows the high-level design and operation of the LCP-based framework.

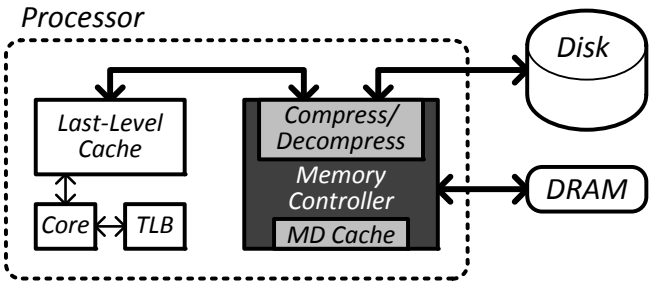


Figure 5: Memory request flow

The first component of LCP-based framework is concerned with compression of an uncompressed page of an LCP. The former happens when a page is accessed for the first time from disk and the latter happens when the size of an LCP increases beyond the original uncompressed size of a page (due to increase in the number of exceptions).

The second component of this framework is related to the operation of the memory controller. To avoid two accesses to main memory, we propose two optimizations that allow the controller to retrieve the cache line with the latency of just *one* main memory access in the common case: (1) a small *metadata cache* that caches the metadata of the recently accessed LCPs, and (ii) a speculative data access from the compressed region.

The third component of this framework deals with the operation of the memory controller when it receives a request for a cache line writeback. In this case, the memory controller attempts to compress the cache line using the compression scheme associated with the corresponding LCP, and based on the outcome can either save a new cache line in the corresponding page, or trigger a recompression event.

Our LCP-based main memory compression framework can be employed with any compression algorithm. The only requirement of LCP to the compression mechanism is that the compressed cache line should have a fixed size. This can be done by restricting the output sizes of the compressed mechanisms. Using this simple idea, we implemented a variant of FPC compression [2]. We limit the output sizes to four different sizes: 16B, 21B, 32B and 44B (similar to the fixed sizes used in [8]). We call this variant of FPC as *FPC-Fixed*. We also employ a recently proposed compression algorithm called Base-Delta-Immediate (BDI) compression [16] that already has a desired property of a fixed size after compression.

[1]The detailed description of the framework is provided in [15].

4.2 Key Design Ideas

LCP page organization (described in previous section) provides a low-complexity and low-latency mechanism to compute the main memory address of a compressed cache line, and hence addresses Challenge 1. To address the Challenge 2 (variable-size physical pages), we use the mechanism, similar to the one proposed by Ekman and Stenstrom [8]. To keep track of virtual pages that are stored in compressed format in main memory, we extend the page table entries to store information related to compression, e.g., compression encoding and compressed size. Our mechanism also allows the OS to manage main memory using a fixed number of predetermined physical page sizes – e.g., 512B, 1kB, 2kB, 4kB. For each one of the chosen sizes, the OS maintains a pool of allocated pages and a pool of free pages. This effectively solves Challenge 2. To address Challenge 3, we modify the cache tagging logic to use the tuple <physical page base address, cache line index within the page> for tagging cache lines. This tuple maps to a unique cache line in the system, and hence avoids the homonym and synonym problems without requiring the exact main memory address of a cache line to be computed.

4.3 LCP Optimizations

Enabling Memory Bandwidth Reduction. One potential benefit of main memory compression that has not been examined in detail by prior work on memory compression is bandwidth reduction. When cache lines are stored in compressed format in main memory, multiple consecutive compressed cache lines can be retrieved at the cost of retrieving a single uncompressed cache line. For example, when cache lines of a page are compressed to 1/4 their original size, four compressed cache lines can be retrieved at the cost of a single uncompressed cache line access. This can significantly reduce the bandwidth requirements of applications, especially those with good spatial locality. We exploit this idea to significantly reduce bandwidth consumption for many applications.

Zero Pages and Zero Cache Lines. Prior work [2, 7, 8, 16, 24] observed that in-memory data contains significant number of zeros at different granularity, e.g., zero pages and zero cache lines. Since this pattern is quite common, we propose two changes to the LCP-framework to more efficiently compress such occurrences of zeros: adding a special bit in page encoding to support zero pages, and a special bit per cache line stored in the metadata region to represent zero cache lines.

5 Results

We use an in-house, event-driven 32-bit x86 simulator whose front-end is based on Simics [13] for our CPU evaluations, and

multi2sim [22] AMD Evergreen ISA [4] simulator for GPU evaluations. Major simulation parameters are provided in our technical report [15]. We use benchmarks from the SPEC CPU2006 suite [19], four TPC-H/TPC-C queries [20], and an Apache web server. For GPU evaluations, we use a set of AMD OpenCL benchmarks. All results are collected by running a representative portion (based on PinPoints [14]) of the benchmarks for 1 billion instructions.

5.1 Effect on DRAM Capacity

Our LCP design aims to provide the benefit of higher effective main memory capacity without making main memory physically larger and without significantly increasing the memory access latency. Figure 6 compares the compression ratio of LCP against that of other compression techniques: *i)* Zero-Page compression, in which accesses to zero pages are served without any cache/memory access (similar to LCP's zero page optimization), *ii)* FPC [8], *iii)* MXT [1], and *iv)* Lempel-Ziv (LZ) [26].[2] We evaluate the LCP framework in conjunction with two compression algorithms: only BDI (as denoted by LCP (BDI)) and with BDI and FPC-fixed in which each page can be encoded with either BDI or FPC-fixed (denoted as LCP (BDI+FPC-fixed)).

We draw two conclusions from Figure 6. First, as expected, MXT, which employs the complex LZ algorithm, has the highest average compression ratio (2.30) of all practical designs and performs closely to our idealized LZ implementation (2.60). At the same time, LCP (with either BDI or BDI+FPC-fixed) provides a reasonably high compression ratio (up to 1.69 with BDI+FPC-fixed), outperforming FPC (1.59). Second, while the average compression ratio of Zero-Page-Compression is relatively low (1.29), it greatly improves the effective memory capacity for a number of applications (e.g., GemsFDTD, zeusmp, and cactusADM). This justifies our design decision of handling zero pages specifically at the TLB-entry level.

To show the general applicability of DRAM compression for different architectures, we perform an experiment to analyze the effect of LCP-based main memory compression for a GPU architecture (AMD Evergreen ISA). We observe that the average compression ratio for GPU applications can be even higher than for CPU ones (1.95 on average). We conclude that our LCP framework achieves the goal of high compression ratio.

5.2 Effect on Performance

Main memory compression can improve performance in two major ways: 1) reduced memory bandwidth requirements can enable less contention on the main memory bus, which is an increasingly important bottleneck in systems, 2) reduced memory footprint can reduce long-latency disk accesses. Evaluations using our LCP-framework (Figure 7a) show that the performance gains due to the bandwidth reduction more than compensate for the slight increase in memory access latency. We also evaluate the decrease in disk accesses (due to decrease in page faults) in Figure 7b.

[2] Our implementation of LZ performs compression at 4kB page-granularity and serves as an idealized upper boundary for the in-memory compression ratio. In contrast, MXT employs Lempel-Ziv at 1kB granularity.

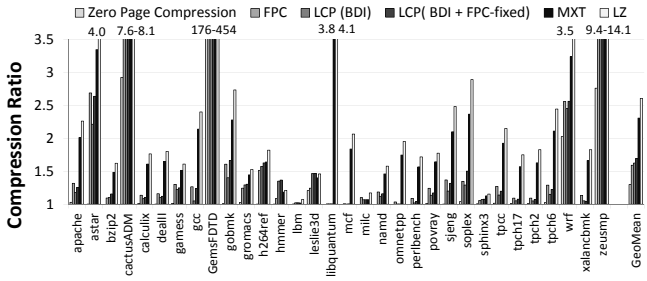
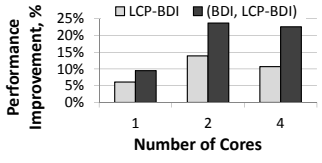
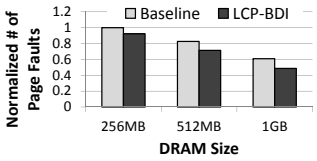


Figure 6: Main memory compression ratio.



(a) Average performance improvement (weighted speedup).[a]



(b) Number of page faults.[a]

[a] (BDI, LCP-BDI) has both LLC and memory compressed using BDI.

[a] Normalized to the baseline with 256MB.

Figure 7: Performance and the page faults number using LCP-BDI.

6 Contributions

As we described in this work, the primary challenge in incorporating compression in main memory is to devise a mechanism that can efficiently compute the main memory address of a cache line without significantly adding complexity, cost, or latency. Prior approaches to address this challenge are either relatively costly or energy inefficient.

We propose a new main memory compression framework to address this problem using an approach that we call *Linearly Compressed Pages* (LCP). The key ideas of LCP are to use a fixed size for compressed cache lines within a page (which simplifies main memory address computation) and to enable a page to be compressed even if some cache lines within the page are incompressible (which enables high compression ratios). We show that any compression algorithm can be adapted to fit the requirements of our LCP-based framework.

We evaluate the LCP-based framework using two state-of-the-art compression algorithms (FPC [2] and BDI [16]) and show that it can significantly increase effective memory capacity (by 69%) and reduce page fault rate (by 27%), leading to significant performance improvements. Based on our results, we conclude that the proposed LCP-based framework provides an effective way of designing low-complexity and low-latency compressed main memory.

References

[1] B. Abali, H. Franke, D. E. Poff, R. A. Saccone, C. O. Schulz, L. M. Herger, and T. B. Smith. Memory Expansion Technology (MXT): Software Support and Performance. *IBM J. Res. Dev.*, 45:287–301, 2001.

[2] A. R. Alameldeen and D. A. Wood. Adaptive Cache Compression for High-Performance Processors. In *ISCA-31*, 2004.

[3] A. R. Alameldeen and D. A. Wood. Frequent Pattern Compression: A Significance-Based Compression Scheme for L2 Caches. *Tech. Rep.*, 2004.

[4] AMD. Evergreen Family Instruction Set Architecture: Instructions and Microcode. In *www.amd.com*, 2011.

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

[6] F. Douglis. The Compression Cache: Using On-line Compression to Extend Physical Memory. In *Winter USENIX Conference*, 1993.

[7] J. Dusser, T. Piquet, and A. Seznec. Zero-Content Augmented Caches. In *ICS*, 2009.

[8] M. Ekman and P. Stenstrom. A Robust Main-Memory Compression Scheme. In *ISCA-32*, 2005.

[9] E. G. Hallnor and S. K. Reinhardt. A Unified Compressed Memory Hierarchy. In *HPCA-11*, 2005.

[10] D. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *IRE*, 1952.

[11] S. F. Kaplan. *Compressed caching and modern virtual memory simulation.* PhD thesis, 1999.

[12] E. Koldinger, J. Chase, and S. Eggers. Architectural Support for Single Address Space Operating Systems. Technical Report 92-03-10, University of Washington, 1992.

[13] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *Computer*, 35:50–58, February 2002.

[14] H. Patil et al. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. *MICRO-37*, 2004.

[15] G. Pekhimenko and et al. Linearly Compressed Pages: A Main Memory Compression Framework with Low Complexity and Low Latency. In *SAFARI Technical Report No. 2012-005*, 2012.

[16] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. Base-Delta-Immediate Compression: A Practical Data Compression Mechanism for On-Chip Caches. In *PACT*, 2012.

[17] V. Sathish, M. J. Schulte, and N. S. Kim. Lossless and Lossy Memory I/O Link Compression for Improving Performance of GPGPU Workloads. In *PACT*, 2012.

[18] A. J. Smith. Cache memories. *ACM Comput. Surv.*, 1982.

[19] SPEC CPU2006 Benchmarks. http://www.spec.org/.

[20] Transaction Processing Performance Council. http://www.tpc.org/.

[21] R. B. Tremaine, T. B. Smith, M. Wazlowski, D. Har, K.-K. Mak, and S. Arramreddy. Pinnacle: IBM MXT in a Memory Controller Chip. *IEEE Micro*, 2001.

[22] R. Ubal, J. Sahuquillo, S. Petit, and P. López. Multi2Sim: A Simulation Framework to Evaluate Multicore-Multithreaded Processors. In *SBAC-PAD*, 2007.

[23] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis. The Case for Compressed Caching in Virtual Memory Systems. In *USENIX Annual Technical Conference*, 1999.

[24] J. Yang, Y. Zhang, and R. Gupta. Frequent Value Compression in Data Caches. In *MICRO-33*, 2000.

[25] Y. Zhang, J. Yang, and R. Gupta. Frequent Value Locality and Value-Centric Data Cache Design. *ASPLOS-9*, 2000.

[26] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 1977.