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

Gennady Pekhimenko[†]
gpekhime@cs.cmu.edu

Vivek Seshadri[†]
vseshadr@cs.cmu.edu

Yoongu Kim[†]
yoongukim@cmu.edu

Hongyi Xin[†]
gohongyi@gmail.com

Onur Mutlu[†]
onur@cmu.edu

Michael A. Kozuch[⋆]
michael.a.kozuch@intel.com

Phillip B. Gibbons[⋆]
phillip.b.gibbons@intel.com

Todd C. Mowry[†]
tcm@cs.cmu.edu

[†]Carnegie Mellon University [⋆]Intel Labs Pittsburgh

September 08, 2012

Abstract

Data compression is a promising technique to address the increasing main memory capacity demand in future systems. Unfortunately, directly applying previously proposed compression algorithms to main memory requires the memory controller to perform non-trivial computations to locate a cache line within the compressed main memory. These additional computations lead to significant increase in access latency, which can degrade system performance. Solutions proposed by prior work to address this performance degradation problem are either costly or energy inefficient.

In this paper, we propose a new main memory compression framework that neither incurs the latency penalty nor requires costly or power-inefficient hardware. The key idea behind our proposal is that if all the cache lines within a page are compressed to the same size, then the location of a cache line within a compressed page is simply the product of the index of the cache line within the page and the size of a compressed cache line. We call a page compressed in such a manner a *Linearly Compressed Page* (LCP). LCP greatly reduces the amount of computation required to locate a cache line within the compressed page, while keeping the hardware implementation of the proposed main memory compression framework simple.

We adapt two previously proposed compression algorithms, Frequent Pattern Compression and Base-Delta-Immediate compression, to fit the requirements of LCP. Evaluations using benchmarks from SPEC CPU 2006 and five server benchmarks show that our approach can significantly increase the effective memory capacity (69% on average). In addition to the capacity gains, we evaluate the benefit of transferring consecutive compressed cache lines between the memory controller and main memory. Our new mechanism considerably reduces the memory bandwidth requirements of most of the evaluated benchmarks (46%/48% for CPU/GPU on average), and improves overall performance (6.1%/13.9%/10.7% for single-/two-/four-core CPU workloads on average) compared to a baseline system that does not employ main memory compression.

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. First, applications are becoming more data intensive with increasing working sets. Second, with more cores integrated on a single chip, more applications are concurrently run on the system, sharing the available main memory capacity. 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 [27, 2, 16, 10] and main memory [1, 8, 25, 6, 5]. 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.

One of the primary concerns with data compression is that *decompression* lies on the critical path of accessing any compressed data. Sophisticated compression algorithms [29, 11] typically achieve high compression ratios at the expense of large decompression latencies which can significantly degrade performance. To counter this problem, prior work [28, 3, 16] has proposed specialized compression algorithms that exploit some regular patterns present in in-memory data, and has shown that such specialized algorithms have similar compression ratios as more complex algorithms, while incurring much lower decompression latencies.

Although prior works have used such simpler algorithms to compress cache lines in on-chip caches [27, 2, 3, 16], applying these techniques directly to main memory is difficult for three reasons.[1] First, when a virtual page is mapped to a compressed physical page, the virtual page offset of a cache line can be different from the corresponding physical page offset since each physical cache line is expected to be smaller than the virtual cache line. In fact, where a compressed cache line resides in a main memory page is dependent on the sizes of the compressed cache lines that come before it in the page. As a result, accessing a cache line within a compressed page in main memory requires an additional layer of indirection to compute the location of the cache line in main memory (which we will call the *main memory address*). This indirection not only increases complexity and cost, but also can increase the latency of main memory access (e.g., up to 22 integer additions or optimistically 6 cycles for one prior design [8]), and thereby degrade system performance. Second, compression complicates memory management, because the operating system has to map fixed-size virtual pages to variable-size physical pages.[2] Third, modern processors employ on-chip caches with tags derived from the physical address to avoid aliasing between different cache lines (physical addresses are unique, virtual addresses are not). When memory is compressed, the cache tagging logic needs to be modified to take the main memory address computation off the critical path of latency-critical L1 cache accesses.

Fortunately, simple solutions exist for the second and third problems as we describe in Section 4. However, prior approaches [1, 8] for mitigating the performance degradation due to the additional computations required to determine the main memory address of a cache line are either costly or inefficient. One approach (IBM MXT [1]) adds a large (32MB) uncompressed cache managed at the granularity at which blocks are compressed (1kB). Since this large cache helps to avoid some requests that need to access main memory (if locality is present in the program), this reduces the number of requests that suffer the latency penalty of the main memory address computation (see Section 2) at the expense of the significant additional area and power cost of such a large cache. Another approach is to overlap a cache line's main memory address computation with the last-level cache access [8]. However, this approach wastes power since not all accesses to the last-level cache result in an access to main memory.

Our goal in this work is to build a main memory compression framework that neither incurs the latency penalty nor requires power-inefficient hardware. To this end, we propose a new approach to compress pages, which we call *Linearly Compressed Pages* (LCP). The key idea of LCP is that if all the cache lines within a page are compressed to the *same size*, then the location of a cache line within a compressed page is simply the product of the index of the cache line and the compressed cache line size. Based on this idea, each compressed page has a target compressed cache line size. Cache lines that cannot be compressed to this target size are called *exceptions*. All exceptions, along with the metadata required to locate them, are stored separately on the same compressed page. We adapt two previously proposed compression algorithms (Frequent Pattern Compression [2] and Base-Delta-Immediate Compression [16]) to fit the requirements of LCP, and show that the resulting designs can significantly improve effective main memory capacity on a wide variety of workloads.

[1]Note that our goal, similarly to that of other works on main memory compression [8, 1], is to have a main memory compression mechanism that compresses data at the granularity of last-level cache line, because main memory is accessed at the cache line granularity. At the same time, we aim to find a mechanism that is compatible with compression employed in on-chip caches, minimizing the number of compression/decompressions performed.

[2]We assume that main memory compression is made visible to the operating system (OS). Unless the main memory is completely managed by hardware, it is difficult to make it transparent to the OS. In Section 2.3, we discuss the drawbacks of a design that makes main memory compression mostly transparent to the OS [1].

One potential benefit of compressing data in main memory that has not been explored by prior work on main memory compression is memory bandwidth reduction. When data is stored in compressed format in main memory, multiple consecutive compressed cache lines can be retrieved for the cost of accessing a single uncompressed cache line. Given the increasing demand on main memory bandwidth, such a mechanism can significantly reduce the memory bandwidth requirement of applications, especially those with high spatial locality. We propose a mechanism that allows the memory controller to retrieve multiple consecutive cache lines with a single access to DRAM, with negligible additional cost. Evaluations show that our mechanism provides significant bandwidth savings, leading to improved system performance.

The contributions of this paper are as follows.

- We propose a new approach to compress pages, Linearly Compressed Pages, that efficiently addresses the associated main memory address computation problem, i.e., the problem of computing the main memory address of a compressed cache line, with much lower cost and complexity compared to prior work.
- We show that any compression algorithm can be adapted to fit the requirements of our LCP-based framework. We evaluate our design with two state-of-the-art compression algorithms, FPC [2] and BDI [16], and show that it can significantly increase the effective main memory capacity (by 69% on average).
- We evaluate the benefits of transferring compressed cache lines over the bus between DRAM and the memory controller and show that it can considerably reduce memory bandwidth consumption (46%/48% for CPU/GPU on average), and improve overall performance by 6.1%/13.9%/10.7% for single-/two-/four- core CPU workloads, compared to a system that does not employ main memory compression.

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 [29] or Huffman encoding [11]) significantly degrade performance due to their high decompression latency. However, to compress in-memory data, prior works have proposed simpler algorithms that have short decompression latencies while still achieving high compression ratios.

2.1 Compressing In-Memory Data

Several studies [28, 3, 16] have shown that in-memory data has different exploitable patterns that allow for simpler compression techniques. Frequent value compression (FVC) [28] 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 frequent-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 used in IBM MXT [1]. However, these low-latency compression algorithms have been proposed in the context of on-chip caches, not main memory. Unfortunately, in order to apply such compression algorithms to main memory, the following challenges must be addressed.

2.2 Challenges in Main Memory Compression

There are three challenges that need to be addressed to employ hardware-based compression for main memory. Figures 1, 2 and 3 pictorially shows 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.

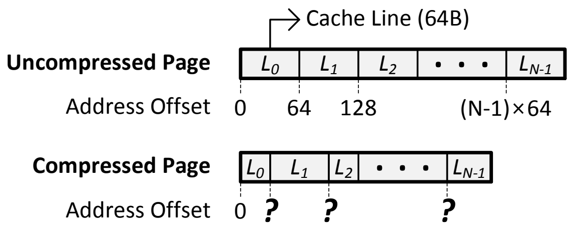

Cache Line (64B)
Uncompressed Page
L_0 L_1 L_2 • • • L_{N-1}
Address Offset 0 64 128 (N-1)×64
Compressed Page
L_0 L_1 L_2 • • • L_{N-1}
Address Offset 0 ? ? ?

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.

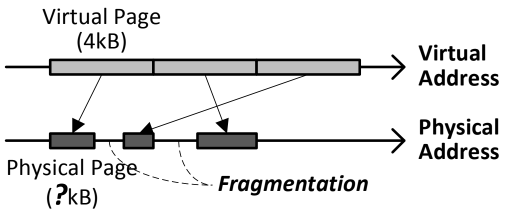


Figure 2: Challenge 2: Main Memory Mapping and Fragmentation

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) [14, 17]. 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).

2.3 Prior Work on Memory Compression

Multiple previous works investigated the possibility of using compression for main memory [1, 8, 25, 13, 5, 6]. Among them, two in particular are the most closely related to the design proposed in this paper, because both of them are mostly hardware designs. Therefore, we describe these two designs along with their shortcomings.

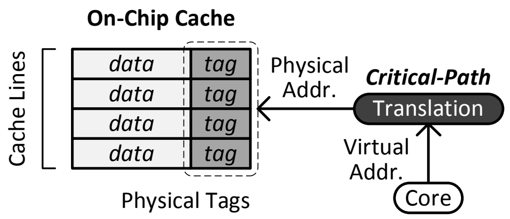


Figure 3: Challenge 3: Physically Tagged Caches

Tremaine et al. [23] proposed a memory controller design, Pinnacle, based on IBM Memory Extension Technology (MXT) [1] that employed Lempel-Ziv compression [29] to manage main memory. To address the three problems described in the previous section, 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. 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. On a miss in the 32MB cache, Pinnacle maps the corresponding real address to the physical address of the compressed block in main memory, using a memory resident mapping-table managed by the memory controller. Following this, Pinnacle retrieves the compressed block from main memory, performs decompression and sends the data back to the processor. Clearly, the additional access to the memory-resident mapping table on every cache miss significantly increases the main memory access latency. In addition to this, Pinnacle's decompression latency, which is on the critical path of a memory access, is 64 processor cycles.

Ekman and Stenstrom [8] proposed a main memory compression design to address the drawbacks of MXT. In their design, the operating system maps the uncompressed virtual address space directly to a compressed physical address space. To compress pages, they use a variant of the Frequent Pattern Compression technique [2], which has a much smaller decompression latency (5 cycles [3]) than the Lempel-Ziv implementation 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). 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. Moreover, the size of the BST is at least twice the size of the translation look-aside buffer (TLB) (number of entries is the same). This increases both the complexity and power consumption of the system significantly. 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. The paper does 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]. Our goal in this work is to address these shortcomings of prior work and develop a simple yet efficient main memory compression design. To this end, we propose a new main memory compression framework that simplifies the nature of the main memory address computation, thereby significantly reducing the associated latency (compared to prior approaches) without requiring energy-inefficient hardware structures. We describe the design and operation of our new framework in the subsequent sections.

3 Linearly Compressed Pages

In this section, we provide the basic idea behind our proposal, Linearly Compressed Pages (LCP), and a brief overview of our LCP-based main memory compression framework. We describe the design and operation of the framework in more detail in Section 4.

3.1 LCP: Basic Idea

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.

To address this shortcoming, we propose a new approach to compressing pages, called the *Linearly Compressed Page* (LCP). 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).

Since all the cache lines within a page are compressed to the same size, then the location of a compressed cache line within the page is simply the product of the index of the cache line within the page and the size of the compressed cache line – essentially a linear scaling using the index of the cache line (hence the name *Linearly Compressed Page*). LCP greatly simplifies the task of computing a cache line's main memory address. For example, if all cache lines within a page are compressed to 16 bytes, the byte offset of the third cache line (index within the page is 2) from the start of the physical page is $16 \times 2 = 32$.

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. Our LCP-based framework identifies this target size for a page when the page is compressed for the first time (or recompressed), as we will describe in Section 4.7. 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.

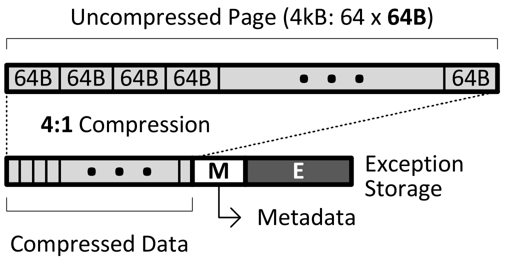


Figure 4: Organization of a Linearly Compressed Page

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 our 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. We provide more details on what exactly is stored in the metadata region in Section 4.2. The third region, *the exception storage*, is the place where all the exceptions of the LCP are stored in their uncompressed form. Our LCP design allows the exception storage to contain free space. In other words, not all entries in the exception storage may store valid exceptions. As we will describe in Section 4, this allows the memory controller to use the free space for storing future exceptions, and also simplifies the operating system page management mechanism. Now that we have described our LCP organization, we will provide a brief overview of the main memory compression framework we build using LCP.

3.2 Memory Compression Framework: Overview

Our LCP-based main memory compression framework consists of components that handle three key issues: 1) page compression/recompression, 2) handling a cache line read from main memory, and 3) handling a cache line writeback into main memory. We briefly provide an overview of each one of these components below. Section 4 presents a detailed description of our design.

3.2.1 Component 1: Page (Re)Compression

The first component of our framework is concerned with compression of an uncompressed page and recompression 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). In both these cases, the operating system (with the help of the memory controller) first determines if the page is compressible using the compression algorithm employed by the framework (described in Section 4.7). If the page is compressible, the OS allocates a physical page of appropriate size and stores the compressed page (LCP) in the corresponding location. It also updates the relevant portions of the corresponding page table mapping to indicate that the page is compressed along with the compression scheme (details in Section 4.1).

3.2.2 Component 2: Handling Cache Line Read

The second component of our framework is related to the operation of the memory controller when it receives a read request for a cache line within an LCP. A naïve way of reading a cache line from an LCP would require at least two accesses to the corresponding page in main memory. First, the memory controller accesses the metadata in the LCP to determine if the cache line is stored in the compressible state or not. Second, depending on the result, the controller accesses the cache line either in the compressed format from the compressed data region, or in the uncompressed format from the list of exceptions. 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. First, we add a small *metadata cache* to the memory controller that caches the metadata of the recently accessed LCPs. This allows the controller to avoid the first main memory access to the metadata in cases when the metadata is present in the cache. Second, in cases when the metadata is not present in the metadata cache, the controller speculatively assumes that the cache line is stored in the compressed form and *first* accesses the data corresponding to the cache line from the compressed data region. The controller then *overlaps* the latency of decompression of the cache line with the access to the metadata of the LCP. In the common case, when the speculation is correct (i.e., the cache line is actually stored in compressed form), this approach significantly reduces the latency of serving the read request. In case of a misspeculation (uncommon case), the memory controller issues another request to retrieve the cache line from the list of exceptions.

3.2.3 Component 3: Handling Cache Line Writeback

The third component of our 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. Depending on the original state of the cache line (compressible or uncompressible), there are four different possibilities: 1) the cache line was compressible and stays compressible, 2) was uncompressible and stays uncompressible, 3) was uncompressible but becomes compressible and 4) was compressible but becomes uncompressible. In the first two cases, the memory controller simply overwrites the old data with the new data at the same location associated with the cache line, as determined by the metadata stored in the page. In case 3, the memory controller frees the exception slot for the cache line and writes the compressible data in the compressed data region of the LCP. (Section 4.2 provides more details on how the exception storage is managed.) In case 4, the memory controller checks if there is a free exception slot to store the uncompressible cache line. If so, it stores the cache line in the corresponding location. If there are no free exception slots in the exception storage of the page, the

Symbol	Description	Per-Page?
$\mathscr{V}$	Virtual page size	No
$\mathscr{C}$	Uncompressed cache line size	No
n	Number of cache lines in a page	No
$\mathscr{M}$	Size of the metadata region	No
$\mathscr{P}$	Compressed physical page size	Yes
$\mathscr{C}^*$	Compressed cache line size	Yes
n_{avail}	Number of slots available to store exceptions	Yes

Table 1: Symbols used in the discussion

memory controller traps to the operating system, which migrates the page to a new location. In both cases 3 and 4, the memory controller appropriately modifies the LCP metadata corresponding to the cache line.

4 Detailed Design

As described in Section 2, there are three main challenges that have to be addressed to implement a hardware-based compression scheme in main memory: 1) providing a low-complexity and low-latency mechanism to compute the main memory address of a compressed cache line, 2) supporting variable-size physical pages in the operating system, and 3) modifying the cache tagging logic to take a cache line's physical address computation off the critical path of the cache access. In the previous section, we provided a high-level overview of the Linearly Compressed Page (LCP) organization and the operation of our main memory compression framework. In this section, we provide a detailed description of LCP, along with the changes to the memory controller, operating system and on-chip cache tagging logic. In the process, we explain how our proposed design addresses each one of the above three challenges.

In the following discussion, we assume that the virtual page size of the system is $\mathscr{V}$, and the cache line size is $\mathscr{C}$. Therefore, the number of cache lines within a virtual page is $\frac{\mathscr{V}}{\mathscr{C}}$, which we denote as n. Table 1 shows the list of symbols used in our discussion along with their meaning.

4.1 Page Table Entry Extension

To keep track of virtual pages that are stored in compressed format in main memory, the page table entries need to be extended to store information related to compression. In addition to the information already maintained in the page table entries, each virtual page in the system is associated with the following pieces of metadata: 1) a bit that indicates if the page is mapped to a compressed physical page (LCP), `c-bit`, 2) a field that indicates the compression scheme used to compress the page, `c-type`, 3) a field that stores the size of the LCP, `c-size`, 4) an extended physical page base address to allow LCPs to start with an address not aligned with the virtual page size, `c-base`. The number of bits required to store `c-type`, `c-size` and `c-base` depends on the exact implementation of the framework. We provide the corresponding values for our implementation in Section 7.

When a virtual page is compressed, i.e., the `c-bit` is set, all the compressible cache lines within the page are compressed to the same size, say $\mathscr{C}^*$. The value of $\mathscr{C}^*$ is uniquely determined by the compression scheme used to compress the page, i.e., the `c-type`. We will defer the discussion on how to determine the `c-type` for a page to Section 4.7. Now, assuming the `c-type` (and hence, $\mathscr{C}^*$) is already known for a page, we describe the corresponding LCP organization in more detail.

4.2 LCP Organization

As mentioned in Section 3, an LCP is divided into three regions: 1) the compressed data region, 2) the metadata region, and 3) the exception storage.

4.2.1 Compressed Data Region

The compressed data region is a contiguous array of n slots each of size $\mathscr{C}^*$. Each one of the n cache lines in the virtual page are mapped to one of the slots, irrespective of whether the cache line is compressible or not. Therefore, the size of the compressed data region is $n\mathscr{C}^*$. This organization simplifies the computation required to determine the physical address for the compressed slot corresponding to a cache line. More specifically, the location of compressed slot for the i^{th} cache line in the virtual page is computed using the following equation:

$$\text{Main memory location of } i^{th} \text{ line} = \texttt{c-base} + i\mathscr{C}^* \tag{1}$$

where `c-base` is the physical page base address and $\mathscr{C}^*$ is the compressed cache line size. Clearly, computing the main memory address of a cache line requires one multiplication and one addition independent of i. This computation requires a lower latency and simpler hardware than prior approaches (e.g., up to 22 additions in the design proposed in [8]), thereby efficiently addressing Challenge 1 (main memory address computation). Our approach also has fixed latency to compute main memory address as address computation operations performed are independent of cache lines relative location in the page.

4.2.2 Metadata Region

The metadata region of an LCP contains two parts. The first part stores two pieces of information for each cache line in the virtual page: 1) a bit indicating if the cache line is compressible, i.e., whether the cache line is an *exception*, `e-bit`, and 2) the index of the cache line in the exception storage (the third region), `e-index`. If the `e-bit` is set for a cache line, then it means that the corresponding cache line should be accessed in the uncompressed format from the location `e-index` in the exception storage. The second part of the metadata region is a bit vector to track the state of the slots in the exception storage. If a bit is set, it indicates that the corresponding slot in the exception storage is used by some incompressible cache line within the page.

The size of the first part depends on the size of `e-index`, which in turn depends on the number of exceptions allowed per page. Since the number of exceptions cannot exceed the number of cache lines in the page ($\mathscr{C}$), we will need at most $1 + \lceil \log n \rceil$ bits for each cache line in the first part of the metadata. For the same reason, we will need at most n bits in the bit vector in the second part of the metadata. Therefore, the size of the metadata region ($\mathscr{M}$) is given by

$$\mathscr{M} = n(1 + \lceil \log n \rceil) + n \text{ bits} \tag{2}$$

Since n is fixed for the entire system, the size of the metadata region ($\mathscr{M}$) is same for all compressed pages (64B in our implementation).

4.2.3 Exception Storage

The third region, the exception storage, is the place where all incompressible cache lines of the virtual page are stored. If a cache line is present in the location `e-index` in the exception storage, its main memory address is given by the following equation.

$$\text{Main memory address of an } \textit{exception} \; = \texttt{c-base} + n\mathscr{C}^* + \mathscr{M} + \texttt{e-index}\mathscr{C} \tag{3}$$

The number of slots available in the exception storage, denoted by n_{avail}, is dictated by the size of the physical page ($\mathscr{P}$) allocated by the operating system for the corresponding LCP. The following equation expresses the relation between the physical page size ($\mathscr{P}$), the compressed cache line size ($\mathscr{C}^*$) which is determined by `c-type`, and the number of available slots in the exception storage (n_{avail}).

$$\mathscr{P} = n\mathscr{C}^* + \mathscr{M} + n_{avail}\mathscr{C} \tag{4}$$

$$\Rightarrow n_{avail} = \frac{\mathscr{P} - (n\mathscr{C}^* + \mathscr{M})}{\mathscr{C}} \tag{5}$$

As mentioned before, the metadata region contains a bit vector which is used to manage the exception storage. When the memory controller assigns an exception slot to an incompressible cache line, it sets the corresponding bit in the bit vector to indicate that the slot is no longer free. If the cache line later becomes compressible and no longer requires the exception slot, the memory controller resets the corresponding bit in the bit vector. In the next section, we describe the operating system memory management policy which determines the physical page size ($\mathscr{P}$) allocated for an LCP and hence, the number of available exception slots (n_{avail}).

4.3 Operating System Memory Management

The second challenge related to main memory compression is to provide operating system (OS) support for managing variable-size compressed physical pages – i.e., LCPs. Depending on the compression scheme employed by the framework, different LCPs may be of different sizes. Allowing LCP of arbitrary sizes will require the OS to keep track of main memory at a very fine granularity. It may also lead to free space getting fragmented across the entire main memory at a fine granularity. As a result, the OS needs to maintain large amounts of metadata to maintain the locations of individual pages and the free space which also leads to increased complexity.

To avoid this problem, our mechanism allows the OS to manage main memory using a fixed number of pre-determined 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. When a page is compressed for the first time or recompressed due to overflow (described in a later section), the OS chooses the smallest available physical page size that will fit the compressed page. For example, if a page is compressed to 768B, then the OS allocates a physical page of size 1kB. Once the physical page size is fixed, the available number of exceptions for the page, n_{avail}, can be determined using Equation 5.

4.4 Changes to the Cache Tagging Logic

As mentioned in Section 2.2, modern systems employ physically-tagged caches to avoid the homonym and synonym problems. However, in a system that employs main memory compression, using the physical (main memory) address to tag cache lines puts the main memory address computation on the critical path of cache access (Challenge 3). To address this challenge, 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.

4.5 Changes to the Memory Controller

In addition to the changes to the memory controller operation described in Section 3.2, our LCP-based framework requires two additional hardware structures to be added to the memory controller: 1) a small metadata cache to accelerate main memory lookups in an LCP, 2) compression/decompression hardware to actually perform the compression and decompression of cache lines.

4.5.1 Metadata Cache

As described in Section 3.2.2, adding a small metadata cache to the memory controller can significantly reduce the number of main memory accesses required to retrieve a compressed cache block. This cache stores the metadata region of recently accessed LCP's so that the metadata for subsequent accesses to such recently-accessed LCPs can be retrieved directly from the cache. In our study, we show that a small 512-entry metadata cache can eliminate 90% of the metadata accesses on average (Section 8.4.1). The additional storage required for the 512-entry cache is 32kB.

4.5.2 Compression/Decompression Hardware

Depending on the compression scheme employed with our LCP-based framework, the memory controller should be equipped with the hardware necessary to compress and decompress cache lines using the corresponding scheme. Although our framework does not impose any restrictions on the nature of the compression algorithm, it is desirable to have compression schemes that have low complexity and decompression latency – e.g., Frequent Pattern Compression (FPC) [2] and Base-Delta-Immediate Compression (BDI) [16]. In Section 4.7, we provide more details on how to adapt any compression algorithm to fit the requirements of LCP and also the specific changes we made to FPC and BDI as case studies of compression algorithms that we adapted to the LCP framework.

4.6 Handling Page Overflows

As described in Section 3.2.3, when a cache line is written back to main memory, it is possible for the cache line to switch from the compressible state to the incompressible state. When this happens, the memory controller should explicitly find a slot in the exception storage to store the uncompressed cache line. However, it is possible that all the slots in the exception storage are already used by other exceptions in the LCP. We call this scenario a *page overflow*. A page overflow increases the size of the LCP and leads to one of two scenarios: 1) the LCP requires a physical page size which is not larger than the uncompressed virtual page size (type-1 page overflow), and 2) the LCP requires a physical page size which is larger than the uncompressed virtual page size (type-2 page overflow).

Type-1 page overflow simply requires the operating system to migrate the LCP to a physical page of larger size. The OS first allocates a new page and copies the data from the old location to the new location. It then modifies the mapping for the virtual page to point to the new location. Since the data in the old page is still valid, this operation may not have to stall the thread for read requests to the page. However, if there are no additional cores to run the OS handler or if a write request is generated for the page, the application may have to stall till the migration operation is complete. In any case, in our evaluations, we stall the application for 10000 cycles when a type-1 overflow occurs.

In a type-2 page overflow, the size of the LCP exceeds the uncompressed virtual page size. Therefore, the OS attempts to recompress the page, possibly using a different encoding (`c-type`) that fits well with the new data of the page. Depending on whether the page is compressible or not, the OS allocates a new physical page to fit the LCP or the uncompressed page, and migrates the data to the new location. The OS also appropriately modifies the `c-bit`, `c-type` and the `c-base` in the corresponding page table entry. Clearly, a type-2 overflow requires significantly more work from the OS than the type-1 overflow. However, we expect page overflows of type-2 to be occur extremely rarely. In fact, we never observed a type-2 overflow in our simulations.

4.7 Compression Algorithms

Our LCP-based main memory compression framework can be employed with any compression algorithm. In this section, we describe how to adapt a generic compression algorithm to fit the requirements of the LCP framework. Subsequently, we describe the two compression algorithms that we used to evaluate our framework, namely, Frequent Pattern Compression [2] and Base-Delta-Immediate Compression [16].

4.7.1 Adapting a Compression Algorithm to Fit LCP

Every compression scheme is associated with a compression function and a decompression function. Let f_c and f_d be the compression and decompression functions associated with the compression scheme to be employed with the LCP framework.

To compress a virtual page into the corresponding LCP using the compression scheme, the memory controller carries out three steps. In the first step, the controller compresses every cache line in the page using f_c and feeds the sizes of each compressed cache line to the second step. In the second step, the controller analyzes the distribution of compressed cache line sizes and identifies a particular target compressed cache line size for the LCP. In the third and final step, the memory controller classifies any cache line whose compressed size is less than or equal to the target size as compressible and all other cache lines as incompressible (exceptions). The memory controller uses this classification to generate the corresponding LCP based on the organization described in Section 3.1. The fixed target size is chosen such that the final compressed size of the LCP is minimized.

To decompress a compressed cache line of the page, the memory controller reads the fixed-target-sized compressed data and feeds it to the function f_d. Since the size of the uncompressed cache line ($\mathscr{C}$) is already known to the memory controller, it only takes the first $\mathscr{C}$ bytes of the uncompressed data and discards the remaining bytes output by the function f_d.

4.7.2 FPC and BDI Compression Algorithms

Although any compression algorithm can be employed with our framework using the approach described above, it is desirable to use compression algorithms that have low complexity hardware implementation and low decompression latency so that the overall complexity of the design is minimized. For this reason, we adapt two state-of-the-art compression algorithms, specifically designed for such design points to compress cache lines, to fit our framework: 1) Frequent Pattern Compression [2], and 2) Base-Delta-Immediate Compression [16].

Frequent Pattern Compression (FPC) is based on the observation that a majority of the words accessed by applications fall under a small set of frequently occurring patterns [3]. FPC compresses each cache line one word at a time. Therefore, the final compressed size of a cache line is dependent on the individual words within the cache line. To minimize the search time for the fixed target compressed cache line size during compression of a page using FPC, we limit our search to four different target sizes: 16B, 21B, 32B and 44B (similar to the fixed sizes used in [8]). We call this variant of FPC as *FPC-Fixed*.

Base-Delta-Immediate compression, a recently proposed compression algorithm, is based on the observation that in most cases, words co-located in memory have small differences in their values, a property referred to as *low dynamic range* [16]. BDI encodes cache lines with such low dynamic range using a base value and an array of differences (Δs) of words within the cache line from the base value. The size of the final compressed cache line only depends on the size of the base and the size of the Δ. To employ BDI with our framework, the memory controller attempts to compress a page with different versions of the Base-Delta encoding as described in the BDI paper [16] and chooses the combination that minimizes the final compressed page size.

For our GPU workloads, multiple values are typically packed into a word – e.g., three components of a color. As a result, in a number of cases, we observed cache lines for which the most significant byte of the values within the cache line are different while the remaining bytes are fixed. The original BDI algorithm will not be able to compress such cache lines as the differences between the words will be large. However, if words of such cache lines are shifted cyclically (by one byte), they can then be compressed using BDI. We call this modification to BDI as BDI-rotate and evaluate it in Section 8.6.

5 LCP Optimizations

In this section, we propose two simple optimizations to our proposed LCP-based framework: 1) Memory bandwidth reduction, and 2) Exploiting zero pages and cache lines.

5.1 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.[3] 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 with a single uncompressed cache line access. This can significantly reduce the bandwidth requirements of applications, especially those with good spatial locality. We propose two mechanisms that exploit this idea.

In the first mechanism, when the memory controller needs to access a cache line in the compressed data region of LCP, it obtains the data from multiple consecutive compressed slots (which add up to the size of an uncompressed cache line). However, not all cache lines that are retrieved in this manner may be *valid*. To determine if an additionally-fetched cache line is valid or not, the memory controller consults the metadata corresponding to the LCP. If a cache line is not valid, then the corresponding data is not decompressed. Otherwise, the cache line is decompressed and the stored in the cache.

The second mechanism is an improvement over the first mechanism, where the memory controller additionally predicts if the additionally-fetched cache lines are *useful* for the application. For this purpose, the memory controller uses hints from a stride prefetcher [12]. In this mechanism, if the stride prefetcher suggests that an additionally-fetched cache line is part of a useful stream, then the memory controller stores that cache line in the cache. This approach has the potential to prevent cache lines that are not useful from polluting the cache. Section 8.5 shows the benefits of this approach.

5.2 Zero Pages and Zero Cache Lines

Prior work [7, 27, 2, 8, 16] 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. First, one value of the page compression encoding (`c-type`, say 0), is reserved to indicate that the entire page is zero. When accessing data from a page with `c-type` =0, the processor can avoid any last-level cache (LLC) or DRAM access by simply zeroing out the allocated cache line in the L1-cache. Second, to compress zero cache lines more efficiently, we add another bit per cache line to the first part of LCP metadata. This bit, which we call the `z-bit`, indicates if the corresponding cache line is zero. Using this

[3]Other prior works [9, 26, 21] looked at the possibility of using compression for bandwidth reduction between the memory controller and DRAM. While significant reduction in bandwidth consumption are reported, all the prior works achieve this reduction at the cost of increased memory access latency, as they have to both compress and decompress a cache line for every request.

approach, the memory controller does not require any main memory access to retrieve a cache line with the `z-bit` set (assuming a metadata cache hit).

6 Integration with Compressed Last-Level Caches

While our LCP-framework does not require any compression in the last-level cache (LLC), it might be desirable to consider LLC compression together with main memory compression for two reasons. First, different applications may have different performance bottlenecks, e.g., limited bandwidth or LLC capacity. Hence, compressing data at both levels of the memory hierarchy can improve overall performance significantly by either improving an application's cache hits or reducing its bandwidth requirements or both. Second, if the same compression algorithm is employed to compress cache lines both in LLC and in main memory, cache lines can be migrated between the two levels without requiring any additional compression or decompression, leading to significant improvements in energy efficiency and latency. Our framework facilitates seamless integration of LLC compression and main memory compression by ensuring that compression algorithm is applied at the cache line granularity.

To understand the benefits of integrating main memory compression with LLC compression, we evaluate a set of designs where both LLC and main memory can be either uncompressed or compressed with different compression mechanisms, e.g., BDI and FPC. We present the results of these evaluations in Section 8.2.

7 Methodology

We use an in-house, event-driven 32-bit x86 simulator whose front-end is based on Simics [15] for CPU evaluations, and multi2sim [24] AMD Evergreen ISA [4] simulator for GPU evaluations. All configurations have private L1D caches and shared L2 caches. Major simulation parameters are provided in Table 2. For CPU evaluations, we use benchmarks from the SPEC CPU2006 suite [19], four TPC-H/TPC-C queries [22], 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 of the benchmarks for 1 billion instructions (CPU) or to completion (GPU).

CPU Processor	1–4 cores, 4GHz, x86 in-order
CPU L1-D cache	32kB, 64B cache-line, 2-way, 1 cycle
CPU L2 cache	2 MB, 64B cache-line, 16-way, 20 cycles
Main memory	4 Banks, 8 kB row buffers, row hits/conflicts: 168/408 cycles
LCP Design	Type-1 Overflow Penalty: 10000 cycles
GPU Processor	1–16 cores, 8 warps per core, 1.4GHz
GPU L1-D cache	16kB, 256B cache-line, 2-way, 1 cycle
GPU L2 cache	64 kB, 256B cache-line, 4-way, 5 cycles

Table 2: Major Parameters of the Simulated Systems.

Metrics. We measure performance of our benchmarks using IPC (instruction per cycle), effective compression ratio (effective DRAM/cache size increase, e.g., a compression ratio of 1.5 for 2MB cache means that a compression scheme employed with this cache achieves the size benefits of that of a 3MB cache), and L2 MPKI (misses per kilo instruction). For multi-programmed workloads we used the weighted speedup performance metric: [18] ($\sum_i \frac{IPC_i^{shared}}{IPC_i^{alone}}$). For bandwidth consumption we used BPKI (bytes transferred over bus per thousand instructions [20]).

Parameters of Evaluated Schemes. As reported in respective previous works, we used a decompression latency of 5 cycles for FPC [3] and 1 cycle for BDI [16]. For BDI-rotate, we assumed a 2-cycle latency due to the additional rotation step.

8 Results

8.1 Effect on DRAM Capacity

Our LCP design aims to provide the benefit of increased effective main memory capacity without making memory physically larger and without significantly increasing the access latency. Figure 5 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) [29].[4] We evaluate the LCP framework in conjunction with two compression algorithms: BDI, and BDI+FPC-fixed, in which each page can be encoded with either BDI or FPC-fixed (Section 4.7.2).

We draw two conclusions from Figure 5. 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., GemsFDFD, zeusmp, and cactusADM) . This justifies our design decision of handling zero pages specifically at TLB-entry level.

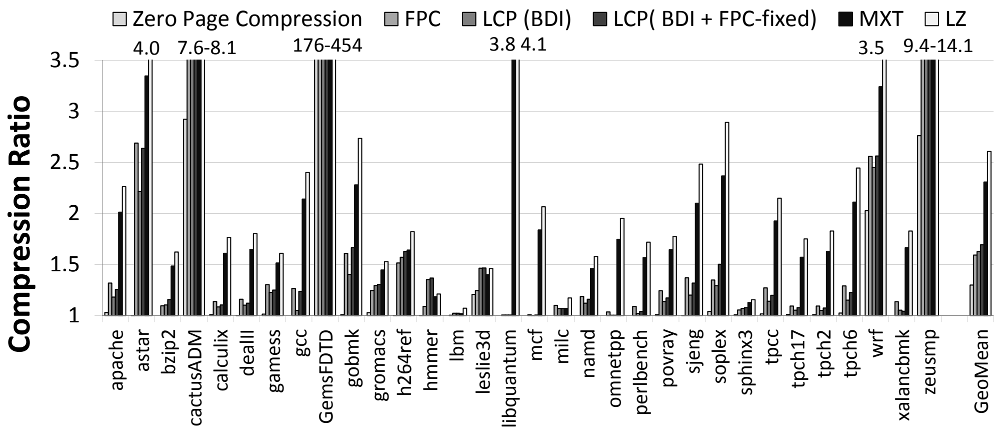


Figure 5: Effect of compression on main memory footprints.

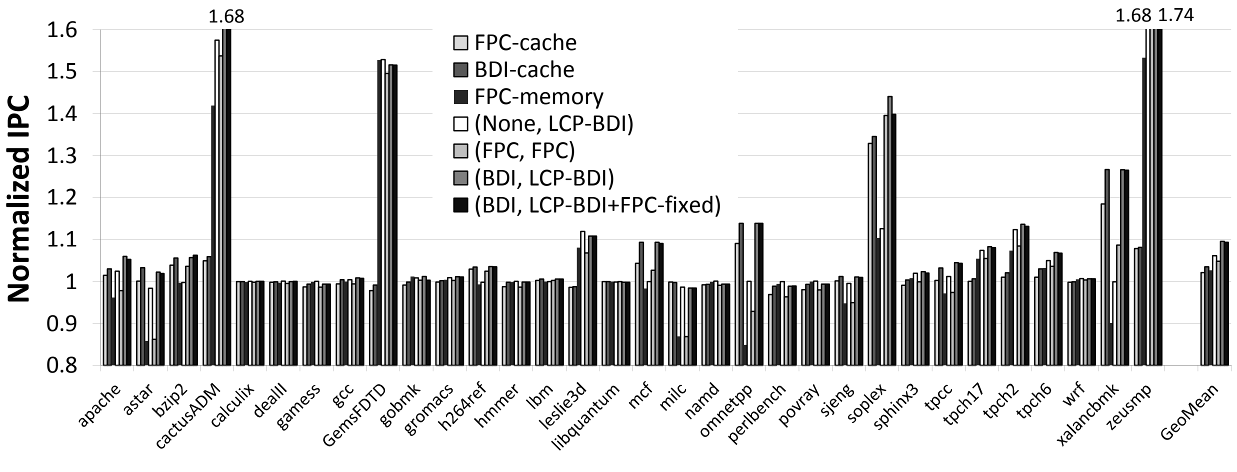


Figure 6: Performance comparison (IPC) of different compressed designs.

[4]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.

8.2 Effect on Performance

Main memory compression can improve performance in two major ways: 1) reduced memory footprint can reduce long-latency disk accesses, 2) reduced memory bandwidth requirements can enable less contention on main memory bus, which is a increasingly important bottleneck in systems. In our system performance evaluations, we do not take into account the former benefit as we do not model disk accesses (i.e., we assume that the uncompressed working set fits entirely in memory). However, we do evaluate the performance improvement due to memory bandwidth reduction (including our optimizations for compressing zero values described in Section 5.2). Evaluations using our LCP-framework show that the performance gains due to the bandwidth reduction more than compensate for the slight increase in memory access latency.

In our experiments for both single-core and multi-core systems, we compare eight different schemes that employ compression either in the last-level cache or main memory or both. Table 3 describes the eight schemes.

No.	Label	Description
1	(None, None)	Baseline with no compression
2	FPC-Cache	LLC compression using FPC [2]
3	BDI-Cache	LLC compression using BDI [16]
4	FPC-Memory	Main memory compression (Ekman and Stenstrom [8])
5	LCP-BDI	LCP-framework with BDI
6	(FPC, FPC)	Designs 2 and 4 combined
7	(BDI, LCP-BDI)	Designs 3 and 5 combined
8	(BDI, LCP-BDI+FPC-Fixed)	Design 3 combined with LCP-framework using BDI+FPC-Fixed

Table 3: List of evaluated designs.

8.2.1 Single-Core Results

Figure 6 shows the performance of single-core workloads using all our evaluated designs normalized to the baseline (None, None). We draw three major conclusions from the figure.

First, compared against an uncompressed system (None, None), the LCP design using BDI compression (LCP-BDI) improves performance by 6.1%. This DRAM-only compression scheme outperforms all LLC-only compression schemes and the DRAM-only compression scheme proposed by Ekman and Stenstrom [8] that uses the FPC algorithm (FPC-memory). We conclude that our LCP framework is effective in improving performance by compressing main memory.

Second, the performance improvement of combined LLC and DRAM compression is greater than that of LLC-only or DRAM-only compression alone. For example, LCP-BDI improves performance by 6.1%, whereas (BDI, LCP-BDI) improves performance by 9.5%. Intuitively, this is due to the orthogonality of the benefits provided by cache compression (retains more cache lines that otherwise would have been evicted) and DRAM compression (brings in more cache lines that would otherwise have required separate memory transfers on the main memory bus). We will provide more analysis of this observation when analyzing the effect on bandwidth in Section 8.3. We conclude that our LCP framework integrates well with and complements cache compression mechanisms.

Third, a high compression ratio does not always imply an improvement in performance. For example, while GemsFDTD is an application with a highly compressible working set in both the cache and DRAM, its performance does not improve with cache-only compression schemes (due to the extra decompression latency), but improves significantly with while significantly improving for DRAM-only compression schemes. In contrast, cache-only compression is significantly beneficial for omnetpp, whereas DRAM-only compression is not. This difference across applications can be explained by the difference in their memory access patterns. We observe that when temporal locality is critical for the performance of an application (e.g., omnetpp and xalancbmk), then cache compression schemes are typically more helpful. On the other hand, when applications have high spatial locality and less temporal locality (e.g., GemsFDTD has an overwhelming streaming access pattern with little reuse), they benefit significantly from the bandwidth compression provided by the LCP-based schemes. Hence, if the goal is to improve performance of a wide variety of applications, that may have a mix of temporal and spatial locality, our results suggest that LCP-based designs with both DRAM and LLC compressed are the best option. We conclude that combined LLC and DRAM compression that takes advantage of our main memory compression framework benefits a wide variety of applications.

8.2.2 Multi-Core Results

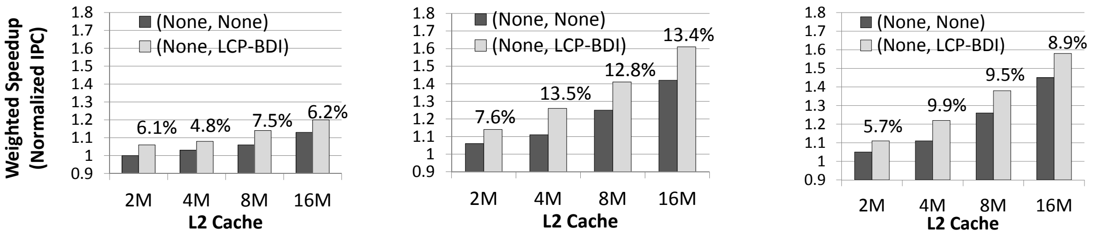
When the system has a single core, the memory bandwidth pressure may not be large enough to take full advantage of bandwidth compression. However, in a multi-core system where multiple applications are running concurrently, savings in bandwidth (reduced number of memory bus transfers) may significantly increase the overall system performance.

To study this effect, we conducted experiments using 100 randomly generated multiprogrammed mixes of applications (for both 2-core and 4-core workloads). Our results show that bandwidth compression is indeed more critical for multi-core workloads. Using our LCP-based design, LCP-BDI, the average performance improvement[5] is 13.9% for 2-core workloads and 10.7% for 4-core workloads. We summarize our multi-core performance results in Table 4.

Figure 7 shows the effect of varying the last-level cache size on the performance benefit of our LCP-based design (using BDI compression in main memory) both for single core and multi-core systems across all evaluated workloads. LCP-based designs outperform the baseline across all evaluated systems, even when the L2 cache size of the system is as large as 16MB. We conclude that our memory compression framework is effective for a wide variety of core counts and last-level cache sizes.

Cores	LCP-BDI	(BDI, LCP-BDI)	(BDI, LCP-BDI+FPC-fixed)
1	6.1%	9.5%	9.3%
2	13.9%	23.7%	23.6%
4	10.7%	22.6%	22.5%

Table 4: Average performance improvement (weighted speedup) using LCP-based designs.



a) 1-core b) 2-core c) 4-core

Figure 7: Effect of varying cache size on performance.

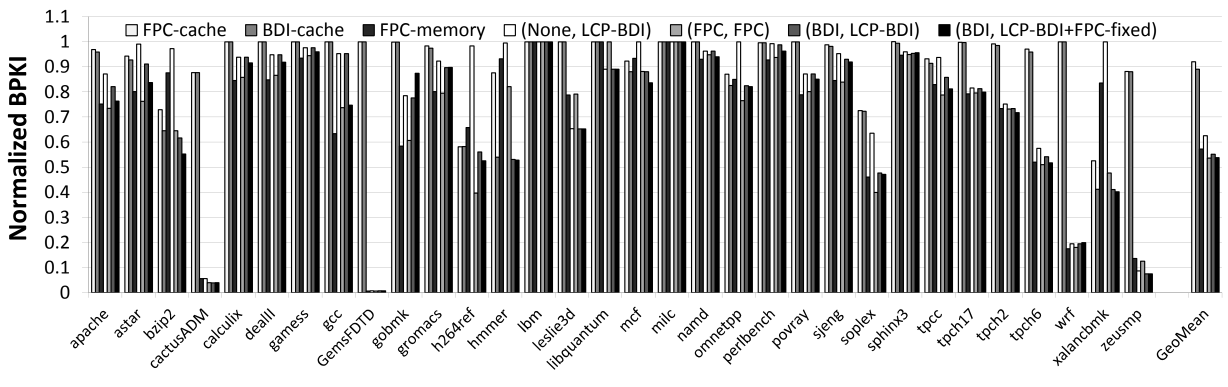


Figure 8: Effect of main memory compression on memory bandwidth.

[5]Normalized to the performance of the baseline system without any compression.

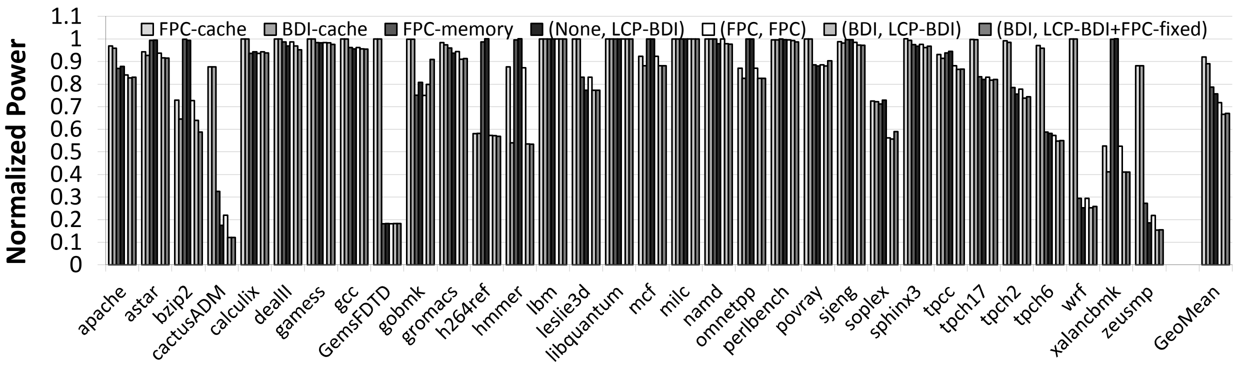


Figure 9: Effect of main memory compression on power consumption of bus between memory controller and DRAM.

8.3 Effect on Bus Bandwidth and Power

When DRAM pages are compressed, the traffic between the LLC and DRAM can also be compressed. This can have multiple positive effects: *i)* reduction in the average latency of memory accesses, which can lead to improvement in the overall system performance, *ii)* decrease in the bus power consumption due to the decrease in the number of transfers.

Figure 8 shows the reduction in main memory bandwidth between LLC and DRAM (in terms of bytes per kiloinstruction, normalized to a system with no compression) using different compression designs. Two major observations are in order.

First, DRAM compression schemes are more effective in reducing bandwidth usage than cache compression schemes. This is because cache-only compression schemes reduce bandwidth consumption by reducing the number of LLC misses but they cannot reduce the bandwidth required to transfer a cache line from main memory. Overall, combined cache-DRAM compression schemes such as (FPC, FPC) and (BDI, LCP-BDI+FPC-fixed) decrease bandwidth consumption by more than 46% by combining the reduction in both LLC misses and bandwidth required to transfer each cache line.

Second, there is a strong correlation between bandwidth compression and performance improvement (Figure 6). Applications that show a significant reduction in bandwidth consumption (e.g., GemsFDFD, cactusADM, soplex, zeusmp, leslie3d, tpc*) also see large performance improvements. There are some noticeable exceptions to this observation, e.g., h264ref, wrf and bzip2. Although the memory bus traffic is compressible in these applications, main memory bandwidth is not the bottleneck for their performance.

8.3.1 Effect on Main Memory Bus Power

By reducing the number of data transfers on the memory bus, a compressed main memory design also reduces the power consumption of the memory bus. Figure 9 shows the reduction in consumed power[6] by the main memory bus with different compression designs. We observe that DRAM compression designs outperform cache compression designs, and LCP-based designs provide higher reductions than previous mechanisms for main memory compression. The largest power reduction, 33% on average, is achieved by combined cache compression and LCP-based main memory compression mechanisms, i.e. (BDI, LCP-BDI) and (BDI, LCP-BDI+FPC-fixed). Even though we do not evaluate full system power due to simulation infrastructure limitations, such a large reduction in main memory bus power consumption can have a significant impact on the overall system's power, especially for memory-bandwidth-intensive applications. We conclude that our framework for main memory compression can enable significant memory power savings.

8.4 Analysis of LCP Structures and Parameters

8.4.1 Effectiveness of the Metadata Cache

The metadata (MD) cache is a critical structure in the LCP framework as it helps the memory controller to avoid accesses to the LCP metadata (Section 4.5.1). Figure 10 shows the hit rate of a 512-entry (32kB) MD cache for an

[6]Normalized to the power of the baseline system with no compression.

LCP design that uses the BDI+FPC-fixed compression scheme for the single-core system.[7] We draw two conclusions from the figure.

First, the average hit ratio is high (88% on average), indicating that the use of MD cache can significantly reduce the number of LCP metadata accesses to main memory. This also justifies the absence of significant performance degradation using the LCP framework (Figure 6) even for applications that do not benefit from compression. Second, some applications have significantly low MD cache hit rate, especially, sjeng and astar. Analysis of the source code of these applications revealed that accesses of these applications exhibit very low locality. As a result, we also observed a low TLB hit rate for these applications. Since TLB misses are costlier than MD cache misses (former requires multiple memory accesses), the low MD cache hit rate does not lead to significant performance degradation for these applications.

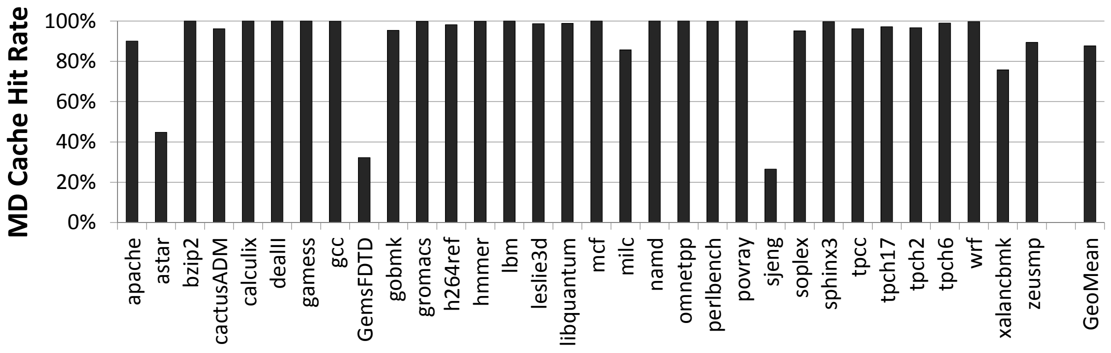


Figure 10: Effectiveness of the metadata cache.

8.4.2 Analysis of Page Overflows

As described in Section 4.6, page overflows can stall an application for a considerable duration. As we mentioned in that section, we did not encounter any type-2 overflows (the more severe type) in our simulations. Figure 11 shows the number of type-1 overflows per instruction. The y-axis uses a log-scale as the number of overflows per instruction is very small. As the figure shows, on average, less than one type-1 overflow occurs every million instructions. Although such overflows are more frequent for some applications (e.g., soplex and tpch2), our evaluations show that this does not degrade performance in spite of adding a 10000 cycle penalty for each type-1 page overflow. In fact, these applications gain significant performance from our LCP design. The main reason for this is that the benefits of bandwidth reduction far outweighs the performance degradation due to type-1 overflows. We conclude that page overflows do not prevent the proposed LCP framework from providing good overall performance.

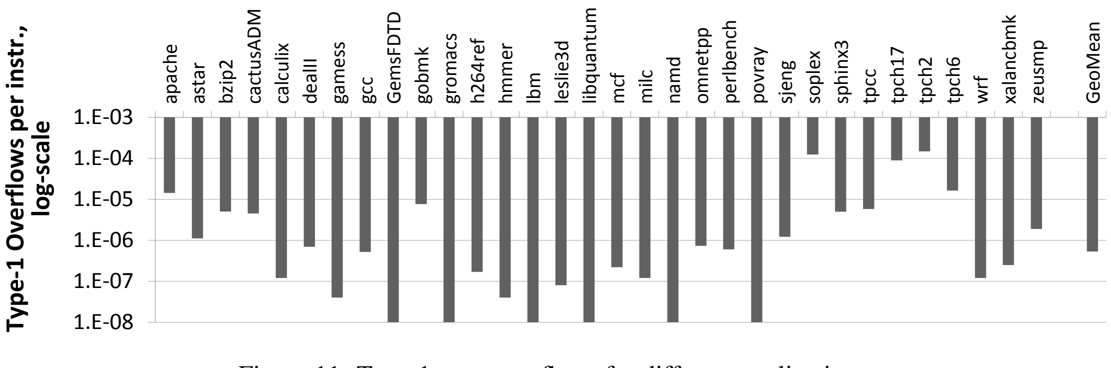


Figure 11: Type-1 page overflows for different applications.

[7]Other previously discussed designs have similar hit rate.

8.4.3 Number of Exceptions

The number of exceptions in the LCP framework is critical for two reasons. First, it determines the size of the physical page required to store the LCP. The higher the number of exceptions, the larger the required physical page size. Second, it can affect an application's performance as exceptions require three main memory accesses on an MD cache miss (Section 3.2). We studied the average number of exceptions (across all compressed pages) for each application. Figure 12 shows the results of these studies.

The number of exceptions varies from as low as 0.02/page for GemsFDTD to as high as 29.2/page in milc (17.3/page on average). The average number of exceptions has a visible impact on the compression ratio of applications (Figure 5). An application with high compression ratio also has relatively few exceptions per page. Note that we do not restrict the number of exceptions in an LCP. As long as an LCP fits into a physical page not larger than the uncompressed page size (i.e., 4kB in our system), it will be stored in the compressed form irrespective of how high the number of exceptions are. This is why applications like milc have a large number of exceptions per page. We note that etter performance is potentially achievable by either statically or dynamically limiting the number of exceptions per page, but a complete evaluation of the design space is a part of our future work.

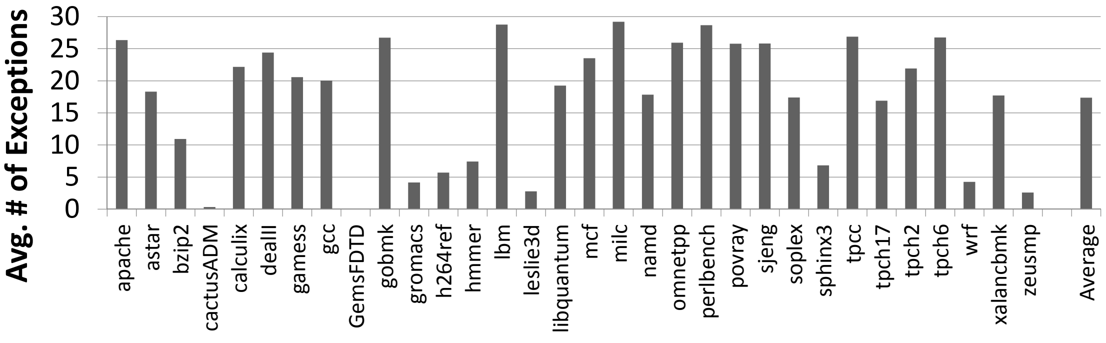


Figure 12: Average number of exceptions per page for different applications

8.5 Comparison to Stride Prefetching

Our LCP-based framework improves performance due to its ability to transfer multiple compressed cache lines using a single memory request. Since this benefit resembles that of prefetching cache lines into the last-level cache (LLC), we compare our LCP-based design to a system that employs a stride prefetcher [12]. Figures 13 and 14 compare the performance and bandwidth consumption of three systems: 1) one that employs stride prefetching, 2) one that employs LCP, and 3) one that employs LCP along with hints from a prefetcher to avoid cache pollution (Section 5.1). Two conclusions are in order.

First, our LCP-based designs (second and third bars) outperform the stride prefetcher for all but a few applications (e.g., libquantum). The primary reason for this is that a stride prefetcher can considerably increase the memory bandwidth consumption of an application due to inaccurate prefetch requests. On the other hand, LCP obtains the benefits of prefetching without increasing (in fact, while significantly reducing) memory bandwidth consumption.

Second, the effect of using prefetcher hints to avoid cache pollution is not significant. The reason for this is that our systems employ a large, highly-associative LLC (2MB 16-way) which is less susceptible to cache pollution. Evicting the LRU lines from such a cache has little effect on performance.

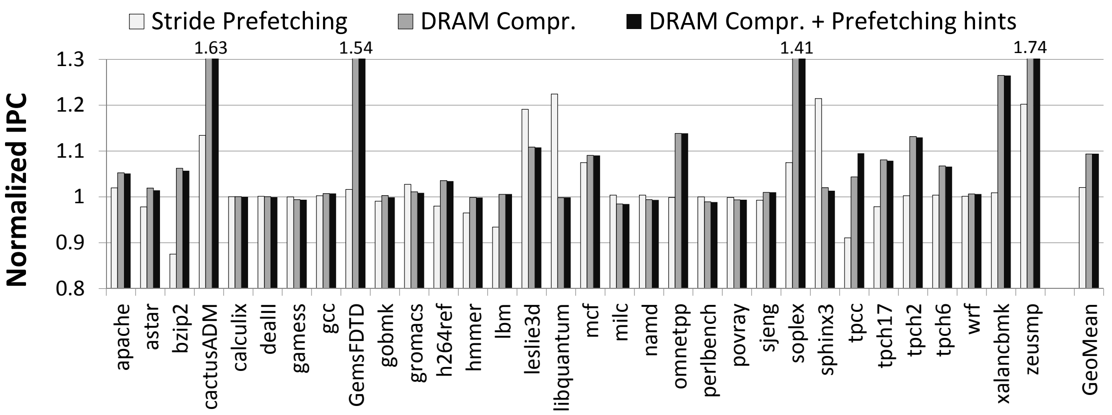


Figure 13: Performance comparison with stride prefetching, and effect of using prefetcher hints with the LCP-framework.

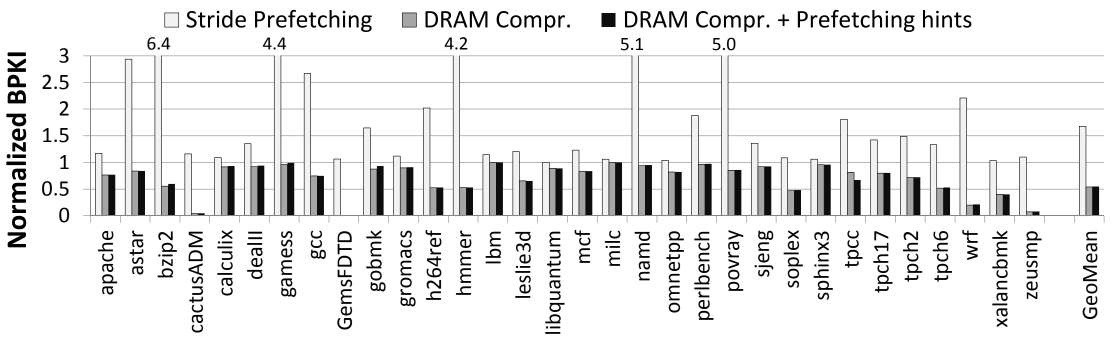


Figure 14: Bandwidth comparison with stride prefetching.

8.6 Effect on GPU Systems

To show the general applicability of DRAM compression for different architectures, we perform a preliminary experiment to analyze the effect of main memory compression on memory bandwidth reduction for a GPU architecture (AMD Evergreen ISA). Figure 15 shows the memory bandwidth reduction with three compression schemes: 1) Frequent Pattern Compression, 2) Base-Delta-Immediate Compression, and 3) Base-Delta-Immediate-rotate Compression (described in Section 4.7.2). As the figure shows, all three mechanisms significantly reduce the bandwidth requirements of most GPU applications, with BDI-rotate showing the best results (48% on average). We conclude that our proposal is effective for GPU systems, and can enable significant performance and energy-efficiency benefits due to this reduction in main memory bandwidth especially in memory-bandwidth-bound GPU applications.

9 Conclusion

With increasing number of cores integrated on a single chip and increasing memory footprint of individual applications, there is an increasing demand for main memory capacity. Data compression is a promising technique to increase the effective main memory capacity without significantly increasing cost and power consumption. As we described in this paper, the primary challenge in incorporating compression in main memory is to device 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.

In this work, 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 ratio). We show that any

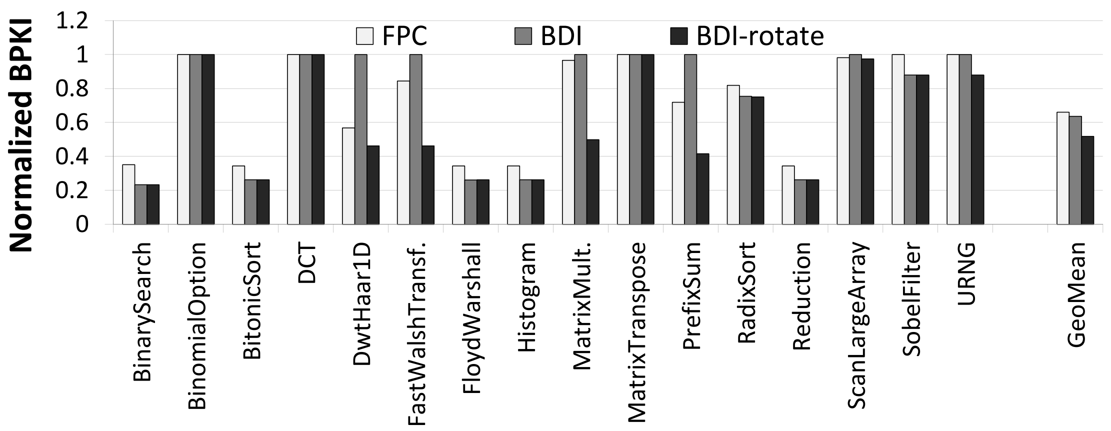


Figure 15: Bandwidth Reduction in GPUs.

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 (Frequent Pattern Compression and Base-Delta-Immediate Compression) and show that it can significantly increase effective memory capacity (by 69%). We show that storing compressed data in main memory can also enable the memory controller to reduce memory bandwidth consumption of both CPU and GPU systems (by 46% and 48%, respectively), leading to significant performance improvements on a wide variety of single-core and multi-core systems with different cache sizes. Finally, we illustrate that our framework integrates well with compression in last-level caches. Based on our results, we conclude that the proposed LCP-based framework provides an effective way of building low-complexity and low-latency designs for compressing main memory.

Acknowledgments

We acknowledge members of the SAFARI and LBA groups for their feedback and for the stimulating research environment they provide. We acknowledge the generous support of AMD, Intel, Oracle, and Samsung. This research was partially supported by NSF (CCF-0953246, CCF-1212962), GSRC, Intel University Research Office, Intel Science and Technology Center for Cloud Computing and NSERC PGS-D.

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] M. Farrens and A. Park. Dynamic Base Register Caching: A Technique for Reducing Address Bus Width. In *ISCA*, 1991.

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

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

[12] S. Iacobovici, L. Spracklen, S. Kadambi, Y. Chou, and S. G. Abraham. Effective stream-based and execution-based data prefetching. ICS '04, 2004.

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

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

[15] 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.

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

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

[18] A. Snavely and D. M. Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor. *ASPLOS-9*, 2000.

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

[20] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers. In *HPCA-13*, pages 63–74, 2007.

[21] M. Thuresson, L. Spracklen, and P. Stenstrom. Memory-Link Compression Schemes: A Value Locality Perspective. *IEEE Trans. Comput.*, 57(7), July 2008.

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

[23] 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.

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

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

[26] J. Yang, R. Gupta, and C. Zhang. Frequent value encoding for low power data buses. *ACM Trans. Des. Autom. Electron. Syst.*, 9(3), 2004.

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

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

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