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

Gennady Pekhimenko[†] Vivek Seshadri[†] Yoongu Kim[†] Hongyi Xin[†]
Onur Mutlu[†] Michael A. Kozuch[⋆] Phillip B. Gibbons[⋆] Todd C. Mowry[†]
[†]Carnegie Mellon University [⋆]Intel Labs Pittsburgh

Abstract

Data compression is a promising approach for meeting the increasing memory capacity demands expected in future systems. Unfortunately, existing compression algorithms do not translate well when directly applied to main memory because they require the memory controller to perform non-trivial computations to locate a cache line within a compressed memory page, thereby increasing access latency and degrading system performance. Prior proposals for addressing this performance degradation problem are either costly or energy inefficient.

*By leveraging the key insight that all cache lines within a page should be compressed to the same size, this paper proposes a new approach to main memory compression—*Linearly Compressed Pages *(LCP)—that avoids the performance degradation problem without requiring costly or energy-inefficient hardware. We show that any compression algorithm can be adapted to fit the requirements of LCP, and we specifically adapt two previously proposed compression algorithms to LCP: Frequent Pattern Compression and Base-Delta-Immediate Compression.*

Evaluations using benchmarks from SPEC CPU2006 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 (34% on average), and improves overall performance (6.1%/13.9%/10.7% for single-/two-/four-core workloads on average) compared to a baseline system that does not employ main memory compression. LCP also decreases energy consumed by the main memory subsystem (9.5% on average over the best prior mechanism).

1. Introduction

Main memory (which is commonly implemented using DRAM technology) is a critical resource in modern systems. To avoid the devastating performance loss resulting from frequent page faults, main memory capacity must be sufficiently provisioned to prevent the target application's working set from overflowing into the orders-of-magnitude-slower backing store (e.g., hard disk or flash storage).

Unfortunately, this minimum main memory capacity is expected to increase rapidly in future systems due to two major trends: (i) applications are generally becoming more data-intensive with increasing working sets, and (ii) with more cores integrated onto single chips, more applications are running concurrently on the system, thereby increasing the aggregate working set size. Simply scaling up main memory capacity at a commensurate rate is unattractive for two reasons: (i) DRAM already constitutes a significant portion of the system's cost and power budget [18], and (ii) for signal integrity reasons, today's high frequency memory channels prevent many DRAM modules from being connected to the same channel [16], effectively limiting the maximum amount of DRAM in a system unless one resorts to expensive off-chip signaling buffers.

If its potential could be realized in practice, *data compression* would be a very attractive approach to effectively increase main memory capacity without requiring significant increases in cost or power, since a compressed piece of data can be stored in a smaller amount of physical memory. Further, such compression could be hidden from application (and most system[1]) software by materializing the uncompressed data as it is brought into the processor cache. Building upon the observation that there is significant redundancy in in-memory data, previous work has proposed a variety of techniques for compressing data in caches [2, 3, 5, 11, 24, 35, 37] and in main memory [1, 6, 7, 9, 33].

1.1. Shortcomings of Prior Approaches

A key stumbling block with making data compression practical is that *decompression* lies on the critical path of accessing any compressed data. Sophisticated compression algorithms, such as Lempel-Ziv and Huffman encoding [12, 38], typically achieve high compression ratios at the expense of large decompression latencies that can significantly degrade performance. To counter this problem, prior work [3, 24, 37] on cache compression proposed specialized compression algorithms that exploit some regular patterns present in in-memory data, and showed that such specialized algorithms have reasonable compression ratios compared to more complex algorithms while incurring much lower decompression latencies.

While promising, applying compression algorithms, sophisticated or simpler, to compress data stored in main memory requires first overcoming the following three challenges. First, *main memory compression complicates memory management*, because the operating system has to map fixed-size virtual pages to variable-size physical pages. Second, because modern processors employ on-chip caches with tags derived from the physical address to avoid aliasing between different cache lines (as physical addresses are unique, while virtual addresses are not), *the cache tagging logic needs to be modified* in light of memory compression to take the main memory address computation off the critical path of latency-critical L1 cache accesses. Third, in contrast with normal virtual-to-physical address translation, the physical page offset of a cache line is often different from the corresponding virtual page offset, since compressed physical cache lines are smaller than their corresponding vir-

[1]We assume that main memory compression is made visible to the memory management functions of the operating system (OS). In Section 2.3, we discuss the drawbacks of a design that makes main memory compression mostly transparent to the OS [1].

tual cache lines. In fact, the location of a compressed cache line in a physical page in main memory depends upon the sizes of the compressed cache lines that come before it in that same physical page. As a result, accessing a cache line within a compressed page in main memory *requires an additional layer of address computation to compute the location of the cache line in main memory* (which we will call the *main memory address*). This additional *main memory address computation* not only adds complexity and cost to the system, but it can also increase the latency of accessing main memory (e.g., it requires up to 22 integer addition operations in one prior design for main memory compression [9]), which in turn can degrade system performance.

While simple solutions exist for these first two challenges (as we describe later in Section 4), prior attempts to mitigate the performance degradation of the third challenge are either costly or inefficient [1, 9]. One approach (IBM MXT [1]) aims to reduce the number of main memory accesses, the causes of long-latency main memory address computations, by adding a large (32MB) uncompressed cache managed at the granularity at which blocks are compressed (1kB). If locality is present in the program, this approach can avoid the latency penalty of main memory address computations to access compressed data. Unfortunately, its benefit comes at a significant additional area and energy cost, and the approach is ineffective for accesses that miss in the large cache. A second approach [9] aims to hide the latency of main memory address computations by speculatively computing the main memory address of *every* last-level cache request in parallel with the cache access (i.e., before it is known whether or not the request needs to access main memory). While this approach can effectively reduce the performance impact of main memory address computation, it wastes a significant amount of energy (as we show in Section 7.3) because many accesses to the last-level cache do not result in an access to main memory.

1.2. Our Approach: Linearly Compressed Pages (LCP)

We aim to build a main memory compression framework that neither incurs the latency penalty for memory accesses nor requires power-inefficient hardware. Our goals are: (i) having low complexity and low latency (especially when performing memory address computation for a cache line within a compressed page), (ii) being compatible with compression employed in on-chip caches (thereby minimizing the number of compressions/decompressions performed), and (iii) supporting compression algorithms with high compression ratios.

To this end, we propose a new approach to compress pages, which we call *Linearly Compressed Pages* (LCP). The key idea of LCP is to compress all of the cache lines within a given page to the same size. Doing so simplifies the computation of the physical address of the cache line, since the page offset is simply the product of the index of the cache line and the compressed cache line size. Based on this idea, a target compressed cache line size is determined for each page. Cache lines that cannot be compressed to the target size for its page are called *exceptions*. All exceptions, along with the metadata required to locate them, are stored separately in the same compressed page. If a page requires more space in compressed form than in uncompressed form, then this page is not compressed. The page table indicates what form the page is stored in.

The LCP framework can be used with any compression algorithm. We adapt two previously proposed compression algorithms (Frequent Pattern Compression (FPC) [2] and Base-Delta-Immediate Compression (BDI) [24]) 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.

Note that, throughout this paper, we assume that compressed cache lines are decompressed before being placed in the processor caches. LCP may be combined with compressed cache designs by storing compressed lines in the higher-level caches (as in [2, 24]), but the techniques are largely orthogonal, and for clarity, we present a LCP design where only main memory is compressed.

An additional, potential benefit of compressing data in main memory, which has not been fully explored by prior work on main memory compression, is *memory bandwidth reduction*. When data are 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. Prior works on bandwidth compression [25, 30, 34] assumed efficient variable-length off-chip data transfers that are hard to achieve with general-purpose DRAM (e.g., DDR3 [22]). 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.

In summary, this paper makes the following contributions:

- We propose a new main memory compression framework–*Linearly Compressed Pages* (LCP) – that solves the problem of efficiently computing the physical address of a compressed cache line in main memory with much lower cost and complexity than prior proposals. We also demonstrate that *any* compression algorithm can be adapted to fit the requirements of LCP.
- We evaluate our design with two state-of-the-art compression algorithms (FPC [2] and BDI [24]), and observe 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 observe that it can considerably reduce memory bandwidth consumption (34% on average), and improve overall performance by 6.1%/13.9%/10.7% for single-/two-/four-core workloads, relative to a system without main memory compression. LCP also decreases energy consumed by the main memory subsystem (9.5% on average over the best prior mechanism).

2. Background on Main Memory Compression

Data compression is widely used in storage structures to increase the effective capacity and bandwidth without increasing the system cost and power consumption significantly. One primary downside of 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 [38] significantly degrades performance due to their high decompression latencies. Thus, prior works on compression of in-memory data have proposed simpler algorithms with low decompression latencies and reasonably high compression ratios, as discussed next.

2.1. Compressing In-Memory Data

Several studies [2, 3, 24, 37] have shown that in-memory data have exploitable patterns that allow for simpler compression techniques. Frequent value compression (FVC) [37] is based on the observation that an application's working set is often dominated by a small set of values. FVC 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. Base-Delta-Immediate (BDI) compression [24] observes that in many cases words co-located in memory have small differences in their values. BDI compression encodes a cache line as a base-value and an array of differences that represent the deviation either from the base-value or from zero (for small values) for each word. These three low-latency compression algorithms have been proposed for on-chip caches, but can be adapted for use as part of the main memory compression framework proposed in this paper.

2.2. Challenges in Main Memory Compression

LCP leverages the fixed-size memory pages of modern systems for the basic units of compression. However, three challenges arise from the fact that different pages (and cache lines within a page) compress to different sizes depending on the compressibility of the data.

Challenge 1: Main Memory Page Mapping. Irregular page sizes in main memory complicate the memory management module of the operating system for two reasons (as shown in Figure 1). First, the operating system needs to allow mappings between the fixed-size virtual pages presented to software and the variable-size physical pages stored in main memory. Second, the operating system must implement mechanisms to efficiently handle fragmentation in main memory.

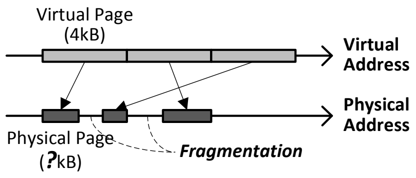


Figure 1: Main Memory Page Mapping Challenge

Challenge 2: Physical Address Tag Computation. On-chip caches (including L1 caches) typically employ tags derived from the physical address of the cache line to avoid aliasing, and in such systems, every cache access requires the physical address of the corresponding cache line to be computed. Hence, because the main memory addresses of the compressed cache lines differ from the nominal physical addresses of those lines, care must be taken that the computation of cache line tags does not lengthen the critical path of latency-critical L1 cache accesses.

Challenge 3: Cache Line Address Computation. 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) must explicitly compute the location of a cache line within a compressed main memory page before accessing it (Figure 2), e.g., as in [9]. This computation not only increases complexity, but can also lengthen the critical path of accessing the cache line from both the main memory and the physically addressed cache. Note that systems that do not employ main memory compression do not suffer from this problem because 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.

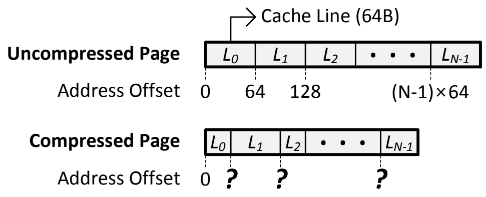


Figure 2: Cache Line Address Computation Challenge

As will be seen shortly, while prior research efforts have considered subsets of these challenges, this paper is the first design that provides a holistic solution to all three challenges, particularly Challenge 3, with low latency and low complexity (both hardware and software complexity).

2.3. Prior Work on Memory Compression

Of the many prior works on using compression for main memory [1, 6, 7, 9, 17, 25, 33], two in particular are the most closely related to the design proposed in this paper, because both of them are mostly hardware designs. We describe these two designs along with their shortcomings.

Tremaine et al. [32] proposed a memory controller design, Pinnacle, based on IBM Memory Extension Technology (MXT) [1] that employed Lempel-Ziv compression [38] to manage main memory. To address the three challenges described above, 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 3). However, there are several drawbacks to this technique: i) such a large cache adds significant cost to the memory controller, ii) the approach requires the main memory address computation logic to be present and used when an access misses in the 32MB cache, and iii) 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 1. On-chip caches are tagged using the real address (instead of the physical address, which is dependent on compressibility), which effectively solves Challenge 2. 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 [9] 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, 3], which has a much smaller decompression latency (5 cycles) than the Lempel-Ziv compression in Pinnacle (64 cycles). To avoid the long latency of a cache line's main memory address computation (Challenge 3), 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. Even though BST has the same number of entries as the translation lookaside buffer (TLB), its size is at least twice that of the TLB. This adds to the complexity and power consumption of the system significantly. To address Challenge 1, 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 [9] do not address Challenge 2.

In summary, prior works on hardware-based main memory compression mitigate the performance degradation due to the main memory address computation problem (Challenge 3) 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 [9].

3. Linearly Compressed Pages

In this section, we provide the basic idea and a brief overview of our proposal, Linearly Compressed Pages (LCP), which overcomes the above-mentioned shortcomings of prior proposals. Further details will follow in Section 4.

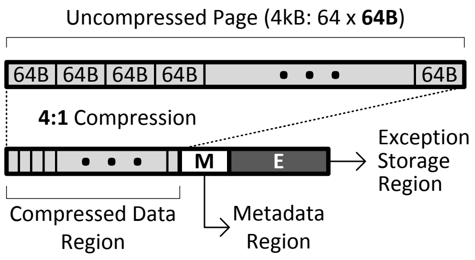


Figure 3: Organization of a Linearly Compressed Page

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 perform this complex calculation (or cache the mapping in a large energy-inefficient structure) in order to access the line.

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 given page* (alleviating the complex and long-latency main memory address calculation problem that arises due to variable-size cache lines), and yet still enable a page to be compressed even if not all cache lines within the page can be compressed to that fixed size (enabling high compression ratios).

Because all the cache lines within a given page are compressed to the same size, 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 $16 \times 2 = 32$, if the line is compressed.

Figure 3 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 is 16B. As shown in the figure, the LCP contains three distinct regions.

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 incompressible cache line as an "exception". The second region, *metadata*, contains all the 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 enables the exception storage to contain unused space. In other words, not all entries in the exception storage may store valid exceptions. As we will describe in Section 4, this enables the memory controller to use the unused 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. LCP Operation

Our LCP-based main memory compression framework consists of components that handle three key issues: (i) page compression (and recompression), (ii) cache line reads from main memory, and (iii) cache line writebacks into main memory. Figure 4 shows the high-level design and operation.

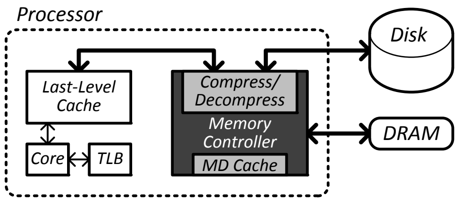


Figure 4: Memory request flow

Page Compression. When a page is accessed for the first time from disk, 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 i) whether the page is compressed, and if so, ii) the compression scheme used to compress the page (details in Section 4.1).

Cache Line Read. Later, when the memory controller receives a *read request* for a cache line within an LCP, it must find and decompress the data. Multiple design solutions are possible to perform this task efficiently. 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 whether the cache line is stored in the compressed format. Second, depending on the result, the controller accesses the cache line either in the compressed format from the *compressed data region* or uncompressed from the *exception storage*.

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 (MD) cache* to the memory controller that caches the metadata of the recently accessed LCPs—the controller avoids the first main memory access to the metadata in cases when the metadata is present in the MD 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 format and *first* accesses the data corresponding to the cache line from the compressed data region. The controller then *overlaps* the latency of the cache line decompression 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 the compressed format), this approach significantly reduces the latency of serving the read request. In the case of a misspeculation (uncommon case), the memory controller issues another request to retrieve the cache line from the exception storage.

Cache Line Writeback. If the memory controller receives a request for a cache line *writeback*, it then 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 incompressible), there are four different possibilities: the cache line (1) was compressed and stays compressed, (2) was uncompressed and stays uncompressed, (3) was uncompressed but becomes compressed, and (4) was compressed but becomes uncompressed. 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. In case 3, the memory controller frees the exception storage 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 enough space in the exception storage region to store the uncompressed cache line. If so, it stores the cache line in an available slot in the region. If there are no free exception storage slots in the exception storage region of the page, the memory controller traps to the operating system, which migrates the page to a new location (which can also involve page recompression). In both cases 3 and 4, the memory controller appropriately modifies the LCP metadata corresponding to the cache line.

Note that in the case of an LLC writeback to main memory (and assuming that TLB information is not available), the cache tag entry is augmented with similar bits to represent page encoding. Cache compression mechanisms, e.g., FPC [2] and BDI [24], already have the corresponding bits for encoding, so that the tag size overhead is minimal when main memory compression is used together with cache compression.

4. Detailed Design

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 of the challenges.

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 (Figure 5). In addition to the information already maintained in the page table entries (such as the base address for a corresponding physical page, `p-base`), each virtual page in the system is associated with the following pieces of metadata: (i) `c-bit`, a bit that indicates if the page is mapped to a compressed physical page (LCP), (ii) `c-type`, a field that indicates the compression scheme used to compress the page, (iii) `c-size`, a field that indicates the size of the LCP, and (iv) `c-base`, a `p-base` extension that enables LCPs to start at an address not aligned with the virtual page size. The number of bits required to store `c-type`, `c-size` and `c-base` depends on the exact implementation of the framework. In our current implementation, we assume 3 bits for `c-type`, 2 bits for `c-size` (4 possible page sizes: 512B, 1kB, 2kB, 4kB), and 3 bits for `c-base` (at most eight 512B compressed pages can fit into 4kB uncompressed slot). Note that existing systems, e.g., [14], usually have enough unused bits (up to 15 bits in Intel X64 systems [14]) in their PTE entries that can be utilized by LCP without increasing the PTE size.

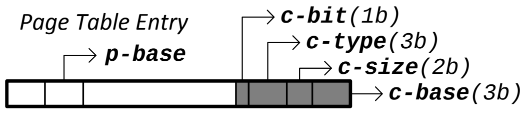


Figure 5: Page table entry extension.

When a virtual page is compressed (the `c-bit` is set), all the compressible cache lines within the page are compressed to the same size, say $\mathcal{C}^*$. The value of $\mathcal{C}^*$ is uniquely determined by the compression scheme used to compress the page, i.e., the `c-type` (Section 4.7 discusses determining the `c-type` for a page). We next describe the LCP organization in more detail.

4.2. LCP Organization

We will discuss each of an LCP's three regions in turn. We begin by define the following symbols: $\mathcal{V}$ is the virtual page size of the system (e.g., 4kB);[2] $\mathcal{C}$ is the uncompressed cache line size (e.g., 64B); $n = \frac{\mathcal{V}}{\mathcal{C}}$ is the number of cache lines per virtual page (e.g., 64); and $\mathcal{M}$ is the size of LCP's metadata region. In addition, on a per-page basis, we define $\mathcal{P}$ to be the compressed physical page size; $\mathcal{C}^*$ to be the compressed cache line size; and n_{avail} to be the number of slots available for exceptions.

4.2.1. Compressed Data Region. The compressed data region is a contiguous array of n slots each of size $\mathcal{C}^*$. 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 $n\mathcal{C}^*$. This organization simplifies the computation required to determine the main memory address for the compressed slot corresponding to a cache line. More specifically, the address of the compressed slot for the i^{th} cache line can be computed as $\texttt{p-base} + 512\texttt{c-base} + i\mathcal{C}^*$, where the first two terms correspond to the start of the LCP and the second indicates the offset within the LCP of the ith compressed slot (see Figure 6). Thus, computing the main memory address of a compressed cache line requires one multiplication and two additions independent of i (fixed latency). This computation requires a lower latency and simpler hardware than prior approaches (e.g., up to 22 additions in the design proposed in [9]), thereby efficiently addressing Challenge 3 (cache line address computation).

4.2.2. Metadata Region. The metadata region of an LCP contains two parts (Figure 7). The first part stores two pieces of information for each cache line in the virtual page: (i) a bit indicating if the cache line is incompressible, i.e., whether the cache line is an *exception*, `e-bit`, and (ii) 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 (`v-bit`) vector to track the state of the slots in the exception storage. If a `v-bit` is set, it indicates that the corresponding slot in the exception storage is used by some uncompressed cache line within the page.

[2] LCP also works with larger pages sizes, e.g., 4MB or 1GB, by scaling the corresponding sizes of the metadata and compressed data regions.

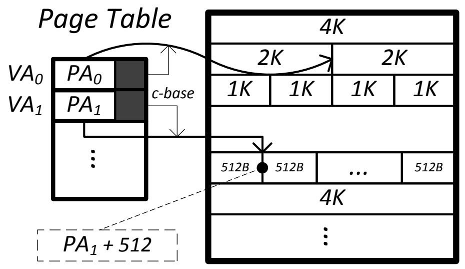


Figure 6: Physical memory layout with LCP framework.

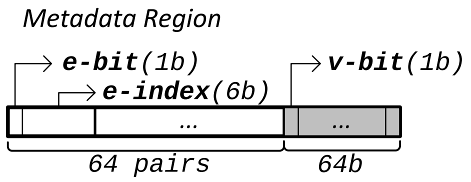


Figure 7: Metadata region, when $n = 64$.

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 ($\mathcal{C}$), we will need at most $1 + \lceil \log_2 n \rceil$ 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 ($\mathcal{M}$) is given by $\mathcal{M} = n(1 + \lceil \log_2 n \rceil) + n$ bits. Since n is fixed for the entire system, the size of the metadata region ($\mathcal{M}$) is the same for all compressed pages (64B in our implementation).

4.2.3. Exception Storage Region. The third region, the exception storage, is the place where all incompressible cache lines of the page are stored. If a cache line is present in the location `e-index` in the exception storage, its main memory address can be computed as: $\texttt{p-base} + 512\texttt{c-base} + n\mathcal{C}^* + \mathcal{M} + \texttt{e-index}\mathcal{C}$. The number of slots available in the exception storage (n_{avail}) is dictated by the size of the compressed physical page allocated by the operating system for the corresponding LCP. The following equation expresses the relation between the physical page size ($\mathcal{P}$), the compressed cache line size ($\mathcal{C}^*$) that is determined by `c-type`, and the number of available slots in the exception storage (n_{avail}):

$$n_{avail} = \lfloor (\mathcal{P} - (n\mathcal{C}^* + \mathcal{M}))/\mathcal{C} \rfloor \quad (1)$$

As mentioned before, the metadata region contains a bit vector that 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 ($\mathcal{P}$) allocated for an LCP and hence, the number of available exception slots (n_{avail}).

4.3. Operating System Memory Management

The first challenge related to main memory compression is to provide operating system 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 LCPs of arbitrary sizes would require the OS to keep track of main memory at a very fine granularity. It could also lead to fragmentation across the entire main memory at a fine granularity. As a result, the OS would need to maintain large amounts of metadata to maintain the locations of individual pages and the free space, which would also lead 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 (a similar approach was proposed to address the memory allocation problem [4]). 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 Section 4.6), the OS chooses the smallest available physical page size that fits 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 1.

4.4. Changes to the Cache Tagging Logic

As mentioned in Section 2.2, modern systems employ physically-tagged caches to avoid aliasing 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 L1 cache access (Challenge 2). 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 aliasing problems without requiring the exact main memory address of a cache line to be computed. The additional index bits are stored within the cacheline tag.

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 hardware structures to be added to the memory controller: (i) a small metadata cache to accelerate main memory lookups in LCP, and (ii) compression/decompression hardware to perform the compression and decompression of cache lines.

4.5.1. Metadata Cache. As described in Section 3.2, a small metadata cache in the memory controller enables our approach, in the common case, to retrieve a compressed cache block in a single main memory access. 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 find that a small 512-entry metadata cache (32kB [3]) can service 88% of the metadata accesses on average across all our workloads. Some applications have significantly lower hit rate, especially sjeng and astar. Analysis of the source code of these applications reveals that the memory 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 (the former requires multiple memory accesses), the low MD cache hit rate does not lead to significant performance degradation for these applications.

We expect the MD cache power to be much lower than the power consumed by other on-chip structures (e.g., L1 caches), because the MD cache is accessed much less frequently (hits in any on-chip cache do not lead to an access in the MD cache).

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) [24]. 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, when a cache line is written back to main memory, it is possible for the cache line to switch from being compressible to being incompressible. 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: (i) the LCP requires a physical page size which is not larger than the uncompressed virtual page size (type-1 page overflow), and (ii) 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. While in transition, the page is locked, so any memory memory request to this page is delayed. In our evaluations, we stall the application for 20,000 cycles [4] when a type-1 overflow occurs; we also find that (on average) type-1 overflow happens less than once per every two million instructions. We vary this latency between 10,000 – 100,000 cycles and observe that the benefits of our framework (e.g., bandwidth compression) far outweigh the overhead due to type-1 overflows.

In a type-2 page overflow, the size of the LCP exceeds the uncompressed virtual page size. Therefore, the OS attempts

[3]We evaluate the sensitivity of performance to MD cache size and find that 32kB is the smallest size that allows our approach to avoid most of the performance loss due to additional metadata accesses.

[4] To fetch a 4KB page, we need to access 64 cache lines (64 bytes each). In the worst case, this will lead to 64 accesses to main memory, most of which are likely to be row-buffer hits. Since a row-buffer hit takes 7.5ns, the total time to fetch the page is around 500ns. On the other hand, the latency penalty of two context-switches (into the OS and out of the OS) is around 4us, assuming each context-switch takes about 2us [19]. Overall, a type-1 overflow takes around 4.5us. For a 4Ghz processor, this is less than 20,000 cycles.

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 more work from the OS than the type-1 overflow. However, we expect page overflows of type-2 to occur rarely. In fact, we never observed a type-2 overflow in our evaluations.

4.6.1. Avoiding Recursive Page Faults. There are two types of pages that require special consideration: (i) pages that keep OS internal data structures, e.g., pages containing information required to handle page faults, and (ii) shared data pages that have more than one page table entry (PTE) mapping to the same physical page. Compressing pages of the first type can potentially lead to recursive page fault handling. The problem can be avoided if the OS sets a special *do not compress* bit, e.g., as a part of the page compression encoding, so that the memory controller does not compress these pages. The second type of pages (shared pages) require consistency across multiple page table entries, such that when one PTE's compression information changes, the second entry is updated as well. There are two possible solutions to this problem. First, as with the first type of pages, these pages can be marked as *do not compress*. Second, the OS can maintain consistency of the shared PTEs by performing multiple synchronous PTE updates (with accompanying TLB shootdowns). While the second solution can potentially lead to better average compressibility, the first solution (used in our implementation) is simpler and requires minimal changes inside the OS.

Another situation that can potentially lead to a recursive exception is the eviction of dirty cache lines from the LLC to DRAM due to some page overflow handling that leads to another overflow. In order to solve this problem, we assume that the memory controller has a small dedicated portion of the main memory that is used as a scratchpad to store cache lines needed to perform page overflow handling. Dirty cache lines that are evicted from LLC to DRAM due to OS overflow handling are stored in this buffer space. The OS is responsible to minimize the memory footprint of the overflow handler. Note that this situation is expected to be very rare in practice, because even a single overflow is infrequent.

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 how to adapt the two compression algorithms used in our evaluation.

4.7.1. Adapting a Compression Algorithm to Fit LCP. Every compression scheme is associated with a compression function, f_c, and a decompression function, f_d. 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 computes the total compressed page size (compressed data + metadata + exceptions, using the formulas from Section 4.2) for each of fixed set of target compressed cache line sizes and selects a target compressed cache line size C^* that minimizes the overall LCP size. 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.

To decompress a compressed cache line of the page, the memory controller reads the fixed-target-sized compressed data and feeds it to the hardware implementation of function f_d. Note the size of the uncompressed cache line (C) is already known to the memory controller.

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 and latency of the design are minimized. For this reason, we adapt two state-of-the-art compression algorithms, specifically designed for such design points in the context of compressing in-cache data, to fit our LCP framework: (i) Frequent Pattern Compression [2], and (ii) Base-Delta-Immediate Compression [24].

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 time to perform the compression search procedure described in Section 4.7.1, we limit the search to four different target cache line sizes: 16B, 21B, 32B and 44B (similar to the fixed sizes used in [9]).

Base-Delta-Immediate (BDI) Compression 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* [24]. 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 either the base value or from zero. The size of the final compressed cache line depends only on the size of the base and the size of the Δs. To employ BDI within our framework, the memory controller attempts to compress a page with different versions of the Base-Delta encoding as described by Pekhimenko *et al.* [24] and then chooses the combination that minimizes the final compressed page size (according to the search procedure in Section 4.7.1).

5. LCP Optimizations

In this section, we propose two simple optimizations to our proposed LCP-based framework: (i) memory bandwidth reduction via compressed cache lines, and (ii) exploiting zero pages and cache lines for higher bandwidth utilization.

5.1. 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.[5] 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 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, some of the cache lines that are retrieved in this manner may not 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 then 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 [13]. 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 7.4 shows the effect of this approach on both performance and bandwidth consumption.

Note that prior works [10, 25, 30, 34] assumed that when a cache line is compressed, only the compressed amount of data can be transferred over the DRAM bus, thereby freeing the bus for the future accesses. Unfortunately, modern DRAM chips are optimized for full cache block accesses [36], so they need to be modified to support such smaller granularity transfers. Our proposal does not require modifications to DRAM itself or the use of specialized DRAM such as GDDR3 [15].

5.2. Zero Pages and Zero Cache Lines

Prior work [2, 8, 9, 24, 35] observed that in-memory data contains significant number of zeros at different granularity: all-zero pages and all-zero cache lines. Because 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 (e.g., `c-type` of 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 LLC or DRAM access by simply zeroing out the allocated cache line in the L1-cache. Second, to compress all-zero cache lines more efficiently, we can 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.

[5]Prior works [10, 25, 30, 34] looked at the possibility of using compression for bandwidth reduction between the memory controller and DRAM. While significant reduction in bandwidth consumption is reported, prior works achieve this reduction either at the cost of increased memory access latency [10, 30, 34], as they have to both compress and decompress a cache line for every request, or based on a specialized main memory design [25], e.g., GDDR3 [15].

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	2 GB, 4 Banks, 8 kB row buffers, 1 memory controller, DDR3-1066 [22]
LCP Design	Type-1 Overflow Penalty: 20,000 cycles

Table 1: Major Parameters of the Simulated Systems.

Using this 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. Methodology

Our evaluations use an in-house, event-driven 32-bit x86 simulator whose front-end is based on Simics [21]. All configurations have private L1 caches and shared L2 caches. Major simulation parameters are provided in Table 1. We use benchmarks from the SPEC CPU2006 suite [27], four TPC-H/TPC-C queries [31], and an Apache web server. All results are collected by running a representative portion (based on PinPoints [23]) of the benchmarks for 1 billion instructions. We build our energy model based on McPat [20], CACTI [29], C-Pack [5], and Synopsys Design Compiler with 65nm library (to evaluate the energy of compression/decompression with BDI and address calculation in RMC [9]).

Metrics. We measure performance of our benchmarks using IPC (instruction per cycle) and effective compression ratio (effective DRAM size increase, e.g., a compression ratio of 1.5 for 2GB DRAM means that a compression scheme employed with this DRAM achieves the size benefits of that of a 3GB DRAM). For multi-programmed workloads we use the weighted speedup [26] performance metric: $(\sum_i \frac{IPC_i^{shared}}{IPC_i^{alone}})$. For bandwidth consumption we use BPKI (bytes transferred over bus per thousand instructions [28]).

Parameters of the Evaluated Schemes. As reported in the respective previous works, we used a decompression latency of 5 cycles for FPC [3] and 1 cycle for BDI [24].

7. Results

In our experiments for both single-core and multi-core systems, we compare five different designs that employ different main memory compression strategies (frameworks) and different compression algorithms: (i) *Baseline* system with no compression, (ii) robust main memory compression (*RMC-FPC*) [9], (iii) and (iv) LCP framework with both FPC and BDI compression algorithms (*LCP-FPC* and *LCP-BDI*), and (v) *MXT* [1]. Note that it is fundamentally possible to build the RMC-BDI design as well, but we found that it leads to either low energy efficiency (due to increase in BST metadata table entry size with many more encodings in BDI) or low compression ratio (when the number of encodings is artificially decreased). Hence, for brevity, we exclude this potential design from our experiments.

In addition, we evaluate two hypothetical designs: Zero

Name	Framework	Compression Algorithm
Baseline	None	None
RMC-FPC	RMC [9]	FPC [2]
LCP-FPC	LCP	FPC [2]
LCP-BDI	LCP	BDI [24]
MXT	MXT [1]	Lempel-Ziv [38]
ZPC	None	Zero Page Compression
LZ	None	Lempel-Ziv [38]

Table 2: List of evaluated designs.

Page Compression (*ZPC*) and Lempel-Ziv (*LZ* [6]) to show some practical boundaries on main memory compression. Table 2 summarizes all the designs.

7.1. Effect on DRAM Capacity

Figure 8 compares the compression ratio of all the designs described in Table 2. We draw two major conclusions. 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-BDI provides a reasonably high compression ratio (1.62 on average), outperforming RMC-FPC (1.59), and LCP-FPC (1.52). (Note that LCP can be also used with both BDI and FPC algorithms together, and the average compression ratio in this case is as high as 1.69.)

Second, while the average compression ratio of ZPC 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 at the TLB-entry level. We conclude that our LCP framework achieves the goal of high compression ratio.

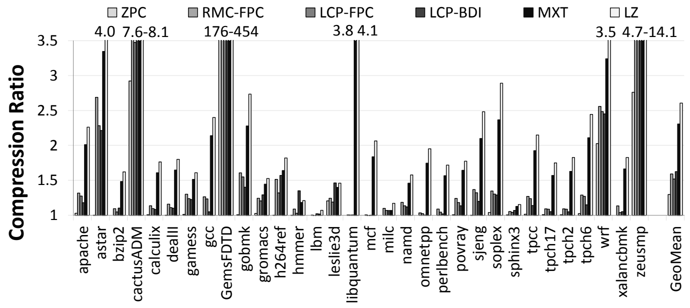


Figure 8: Main memory compression ratio.

7.2. Effect on Performance

Main memory compression can improve performance in two major ways: (i) reduced memory bandwidth requirements can enable less contention on the main memory bus, which is an increasingly important bottleneck in systems, and (ii) reduced memory footprint can reduce long-latency disk accesses. We evaluate the performance improvement due to memory bandwidth reduction (including our optimizations for compressing zero values described in Section 5.2) in Sections 7.2.1 and 7.2.2. Evaluations using our LCP-based frameworks 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 page faults in Section 7.2.3.

7.2.1. Single-Core Results. Figure 9 shows the performance of single-core workloads using three key evaluated designs (RMC-FPC, LCP-FPC, and LCP-BDI) normalized to the *Baseline*. We draw two major conclusions from the figure.

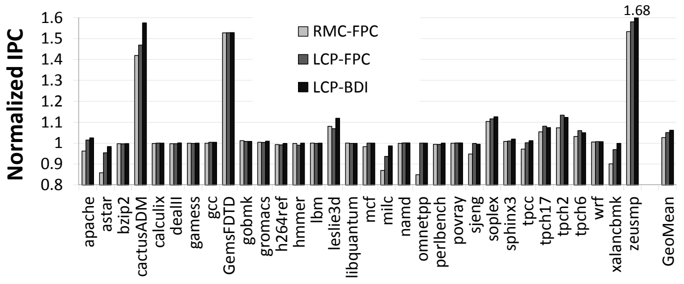


Figure 9: Performance comparison (IPC) of different compressed designs for single-core.

First, compared against an uncompressed system (*Baseline*), the LCP-based designs (LCP-BDI and LCP-FPC) improve performance by 6.1% / 5.2% and also outperform the compression design by Ekman and Stenstrom [9] (RMC-FPC [7]). We conclude that our LCP framework is effective in improving performance by compressing main memory.

Second, LCP-FPC design outperforms RMC-FPC (on average) despite having slightly lower compression ratio. This is mostly due to a lower overhead when accessing metadata information (RMC-FPC needs two memory accesses to *different* main memory pages in the case of a BST table miss, while LCP-based framework performs two accesses to the same main memory page that can be pipelined). This is especially noticeable in several applications, e.g., astar, milc, and xalancbmk that have low metadata table (BST) hit rates. We conclude that our LCP framework is more effective in improving performance than RMC [9].

7.2.2. Multi-Core Results. When the system has a single core, the memory bandwidth pressure may not be large enough to take full advantage of the bandwidth benefits of main memory 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 the bandwidth benefits of memory compression is indeed more critical for multi-core workloads. Using our LCP-based design, LCP-BDI, the average performance improvement (normalized to the performance of the *Baseline* system without compression) is 13.9% for 2-core workloads and 10.7% for 4-core workloads. We summarize our multi-core performance results in Figure 10a.

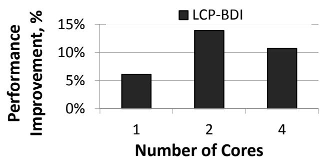
We also vary the last-level cache size (1MB – 16MB) for both single core and multi-core systems across all evaluated

[6]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.

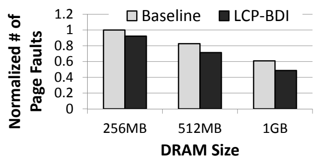
[7]Note that in order to provide a fair comparison, we enhanced the RMC-FPC approach with the same optimizations we did for LCP, e.g., bandwidth compression. The original RMC-FPC design reported an average degradation in performance [9].

workloads. We find that LCP-based designs outperform the *Baseline* across all evaluated systems (average performance improvement for single-core varies from 5.1% to 13.4%), even when the L2 cache size of the system is as large as 16MB.

7.2.3. Effect on the Number of Page Faults. Modern systems are usually designed such that concurrently running applications have enough main memory to avoid most of the potential capacity page faults. At the same time, if the applications' total working set exceeds the main memory capacity, the increased number of page faults can significantly affect performance. To study the effect of LCP-based framework (LCP-BDI) on the number of page faults, we evaluate twenty randomly generated 16-core multiprogrammed mixes of applications from our benchmark set. We also vary the main memory capacity from 256MB to 1GB (larger memories usually lead to almost no page faults for these workload simulations). Our results (Figure 10b) show that the LCP-based framework (LCP-BDI) can decrease the number of page faults by as high as 25% on average (for 1GB DRAM) when compared with the *Baseline* design with no compression. We conclude that the LCP-based framework can significantly decrease the number of page faults, and hence improve system performance beyond the benefits it provides due to reduced bandwidth.



(a) Average performance improvement (weighted speedup).



(b) Number of page faults (normalized to *Baseline* with 256MB).

Figure 10: Performance (with 2 GB DRAM) and number of page faults (varying DRAM size) using LCP-BDI.

7.3. Effect on Bus Bandwidth and Memory Subsystem Energy

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

Figure 11 shows the reduction in main memory bandwidth between LLC and DRAM (in terms of bytes per kilo-instruction, normalized to the *Baseline* system with no compression) using different compression designs. The key observation we make from this figure is that there is a strong correlation between bandwidth compression and performance improvement (Figure 9). Applications that show a significant reduction in bandwidth consumption (e.g., GemsFDTD, 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.

Figure 12 shows the reduction in memory subsystem energy of three systems that employ main memory compression—RMC-FPC, LCP-FPC, and LCP-BDI—normalized to the energy of *Baseline*. The memory subsystem energy includes the static and dynamic energy consumed by caches, TLBs, memory transfers, and DRAM, plus the energy of additionally used components: BST, MD cache, address calculation, compressor/decompressor units. Two observations are in order.

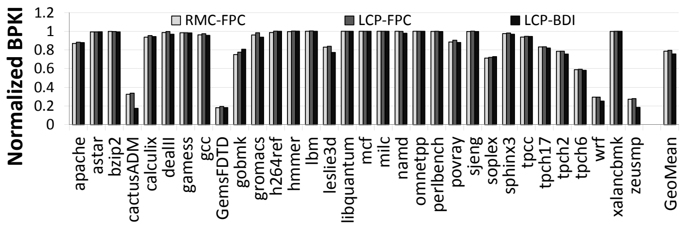


Figure 11: Effect of different main memory compression schemes on memory bandwidth.

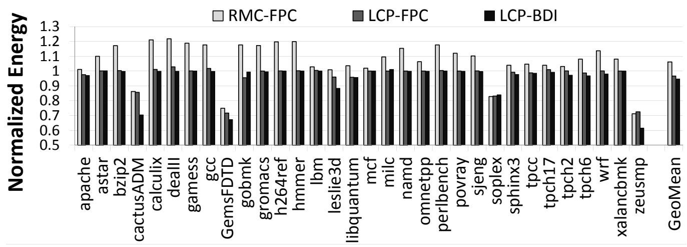


Figure 12: Effect of different main memory compression schemes on memory subsystem energy.

First, our LCP-based designs (LCP-BDI and LCP-FPC) improve the memory subsystem energy by 5.2% / 3.4% on average over the *Baseline* design with no compression, and by 11.3% / 9.5% over the state-of-the-art design (RMC-FPC) based on [9]. This is especially noticeable for bandwidth-limited applications, e.g., zeusmp and cactusADM. We conclude that our framework for main memory compression enables significant energy savings, mostly due to the decrease in bandwidth consumption.

Second, RMC-FPC consumes significantly more energy than *Baseline* (6.1% more energy on average, and as high as 21.7% for dealII). The primary reason for this energy consumption increase is the physical address calculation that RMC-FPC speculatively performs on *every* L1 cache miss (to avoid increasing the memory latency due to complex address calculations). The second reason is the frequent (every L1 miss) accesses to the BST table (described in Section 2) that holds the information to perform address calculation.

Note that other factors, e.g., compression/decompression energy overheads or different compression ratios, are not the reasons for this energy consumption increase. LCP-FPC uses the same compression algorithm as RMC-FPC (and even has a slightly lower compression ratio), but does not increase energy consumption (in fact, LCP-FPC improves the energy consumption due to its decrease in bandwidth consumption). We conclude that our LCP-based framework is a more energy-efficient main memory compression framework than previously proposed designs such as RMC-FPC [9].

7.4. 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. Because this benefit resembles that of prefetching cache lines into the LLC, we compare our LCP-based design to a system that employs a stride prefetcher implemented as described in [13]. Figures 13 and 14 compare the performance and bandwidth consumption of three systems: (i) one that employs stride prefetching, (ii) one that employs LCP-BDI, and (iii) one that employs LCP-BDI along with hints from a prefetcher to avoid cache pollution due to bandwidth compression (Section 5.1). Two conclusions are in order.

First, our LCP-based designs (second and third bars) are competitive with the more general 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, but we observe the benefits of this mechanism on multi-core systems with shared caches (up to 5% performance improvement for some two-core workload mixes).

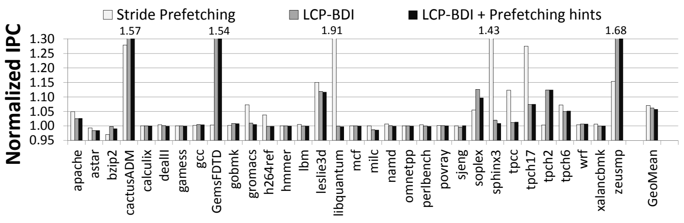


Figure 13: Performance comparison with stride prefetching, and using prefetcher hints with the LCP-framework.

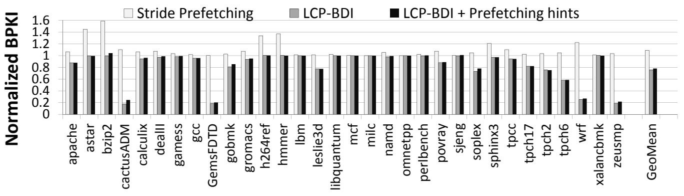


Figure 14: Bandwidth comparison with stride prefetching.

8. Conclusion

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 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.

In this work, we proposed 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 showed that any compression algorithm can be adapted to fit the requirements of our LCP-based framework.

We evaluated the LCP-based framework using two state-of-the-art compression algorithms (Frequent Pattern Compression and Base-Delta-Immediate Compression) and showed that it can significantly increase effective memory capacity (by 69%) and reduce page fault rate (by 27%). We showed that storing compressed data in main memory can also enable the memory controller to reduce memory bandwidth consumption (by 34%), leading to significant performance and energy improvements on a wide variety of single-core and multi-core systems with different cache sizes. Based on our results, we conclude that the proposed LCP-based framework provides an effective approach for designing low-complexity and low-latency compressed main memory.

References

[1] B. Abali *et al.*, "Memory Expansion Technology (MXT): Software Support and Performance," *IBM J. Res. Dev.*, 2001.
[2] A. R. Alameldeen and D. A. Wood, "Adaptive Cache Compression for High-Performance Processors," in *ISCA-31*, 2004.
[3] ——, "Frequent Pattern Compression: A Significance-Based Compression Scheme for L2 Caches," *Tech. Rep.*, 2004.
[4] E. D. Berger, "Memory management for high-performance applications," Ph.D. dissertation, 2002.
[5] X. Chen *et al.*, "C-pack: A high-performance microprocessor cache compression algorithm," in *VLSI Systems, IEEE Transactions on*, 2010.
[6] R. S. de Castro, A. P. do Lago, and D. Da Silva, "Adaptive Compressed Caching: Design and Implementation," in *SBAC-PAD*, 2003.
[7] F. Douglis, "The Compression Cache: Using On-line Compression to Extend Physical Memory," in *Winter USENIX Conference*, 1993.
[8] J. Dusser *et al.*, "Zero-Content Augmented Caches," in *ICS*, 2009.
[9] M. Ekman and P. Stenstrom, "A Robust Main-Memory Compression Scheme," in *ISCA-32*, 2005.
[10] M. Farrens and A. Park, "Dynamic Base Register Caching: A Technique for Reducing Address Bus Width," in *ISCA*, 1991.
[11] E. G. Hallnor and S. K. Reinhardt, "A Unified Compressed Memory Hierarchy," in *HPCA-11*, 2005.
[12] D. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," *IRE*, 1952.
[13] S. Iacobovici *et al.*, "Effective stream-based and execution-based data prefetching," in *ICS*, 2004.
[14] *Intel 64 and IA-32 Architectures Software Developer's Manual*, Intel Corporation, 2013.
[15] JEDEC, "GDDR3 Specific SGRAM Functions, JESD21-C," 2012.
[16] U. Kang *et al.*, "8Gb 3D DDR3 DRAM Using Through-Silicon-Via Technology," in *ISSCC*, 2009.
[17] S. F. Kaplan, "Compressed caching and modern virtual memory simulation," Ph.D. dissertation, 1999.
[18] C. Lefurgy *et al.*, "Energy management for commercial servers," in *IEEE Computer*, 2003.
[19] C. Li, C. Ding, and K. Shen, "Quantifying the Cost of Context Switch," in *ExpCS*, 2007.
[20] S. Li *et al.*, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," *MICRO-42*, 2009.
[21] P. S. Magnusson *et al.*, "Simics: A Full System Simulation Platform," *Computer*, 2002.
[22] Micron, "2Gb: x4, x8, x16, DDR3 SDRAM," 2012.
[23] H. Patil *et al.*, "Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation," *MICRO-37*, 2004.
[24] G. Pekhimenko *et al.*, "Base-Delta-Immediate Compression: A Practical Data Compression Mechanism for On-Chip Caches," in *PACT*, 2012.
[25] 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.
[26] A. Snavely and D. M. Tullsen, "Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor," *ASPLOS-9*, 2000.
[27] SPEC CPU2006 Benchmarks, "http://www.spec.org/."

[28] S. Srinath *et al.*, “Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers,” in *HPCA-13*, 2007.

[29] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, “CACTI 5.1,” HP Laboratories, Tech. Rep. HPL-2008-20, 2008.

[30] M. Thuresson *et al.*, “Memory-Link Compression Schemes: A Value Locality Perspective,” *Trans. Comput.*, 2008.

[31] Transaction Processing Performance Council, “http://www.tpc.org/.”

[32] R. B. Tremaine *et al.*, “Pinnacle: IBM MXT in a Memory Controller Chip,” *IEEE Micro*, 2001.

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

[34] J. Yang, R. Gupta, and C. Zhang, “Frequent value encoding for low power data buses,” *ACM Trans. Des. Autom. Electron. Syst.*, 2004.

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

[36] D. H. Yoon, M. K. Jeong, M. Sullivan, and M. Erez, “The Dynamic Granularity Memory System,” in *ISCA*, 2012.

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

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