Compresso: Pragmatic Main Memory Compression

Esha Choukse
University of Texas At Austin
esha.choukse@utexas.edu

Mattan Erez
University of Texas at Austin
mattan.erez@utexas.edu

Alaa R. Alameldeen
Intel Labs
alaa.r.alameldeen@intel.com

***Abstract*—Today, larger memory capacity and higher memory bandwidth are required for better performance and energy efficiency for many important client and datacenter applications. Hardware memory compression provides a promising direction to achieve this without increasing system cost. Unfortunately, current memory compression solutions face two significant challenges. First, keeping memory compressed requires additional memory accesses, sometimes on the critical path, which can cause performance overheads. Second, they require changing the operating system to take advantage of the increased capacity, and to handle incompressible data, which delays deployment. We propose Compresso, a hardware memory compression architecture that minimizes memory overheads due to compression, with no changes to the OS. We identify new data-movement trade-offs and propose optimizations that reduce additional memory movement to improve system efficiency. We propose a holistic evaluation for compressed systems. *Our results show that Compresso achieves a 1.85x compression for main memory on average, with a 24% speedup over a competitive hardware compressed system for single-core systems and 27% for multi-core systems.* As compared to competitive compressed systems, Compresso not only reduces performance overhead of compression, but also increases performance gain from higher memory capacity.**

I. INTRODUCTION

Memory compression can improve performance and reduce cost for systems with high memory demands, such as those used for machine learning, graph analytics, databases, gaming, and autonomous driving. We present Compresso, the first compressed main-memory architecture that: (1) explicitly optimizes for new trade-offs between compression mechanisms and the additional data movement required for their implementation, and (2) can be used without any modifications to either applications or the operating system.

Compressing data in main memory increases its effective capacity, resulting in fewer accesses to secondary storage, thereby boosting performance. Fewer I/O accesses also improve tail latency [1] and decrease the need to partition tasks across nodes just to reduce I/O accesses [2, 3]. Additionally, transferring compressed cache lines from memory requires fewer bytes, thereby reducing memory bandwidth usage. The saved bytes may be used to prefetch other data [4, 5], or may even directly increase effective bandwidth [6, 7].

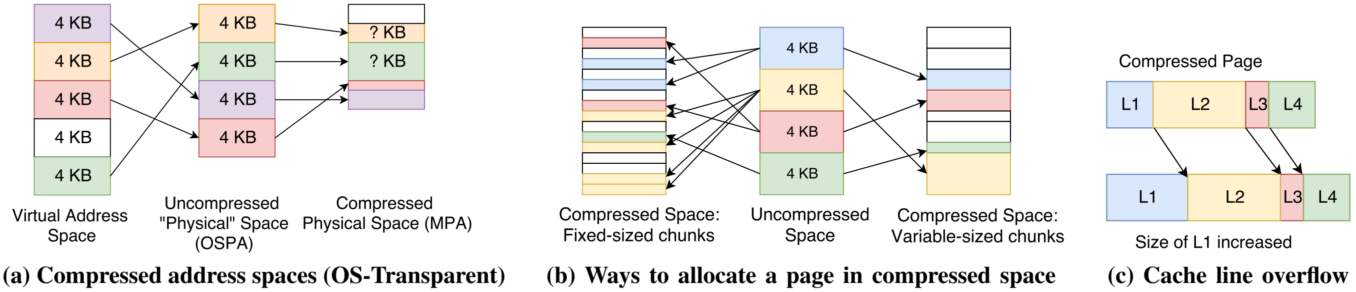
Hardware memory compression solutions like IBM MXT [8], LCP [4], RMC [9], Buri [10], DMC [11], and CMH [12] have been proposed to enable larger capacity, and higher bandwidth, with low overhead. Unfortunately, hardware memory compression faces major challenges that prevent its wide-spread adoption. First, compressed memory management results in additional data movement, which can lead to performance overheads, and has been left widely unacknowledged in previous work. Second, to take advantage of larger memory capacity, hardware memory compression techniques need support from the operating system to dynamically change system memory capacity and address scenarios where memory data is incompressible. Such support limits adoptive potential of the solutions. On the other hand, if compression is implemented without OS support, hardware needs innovative solutions for dealing with incompressible data.

Additional data movement in a compressed system is caused due to metadata accesses for additional translation, change in compressibility of the data, and compression across cache line boundaries in memory. We demonstrate and analyze the magnitude of the data movement challenge, and identify novel trade-offs. We then use allocation, data-packing and prediction based optimizations to alleviate this challenge. Using these optimizations, we propose a new, efficient compressed memory system, Compresso, that provides high compression ratios with low overhead, maximizing performance gain.

In addition, Compresso is OS-transparent and can run any standard modern OS (e.g., Linux), with no OS or software modifications. Unlike prior approaches that sacrifice OS transparency for situations when the system is running out of promised memory [8, 10], Compresso utilizes existing features of modern operating systems to reclaim memory without needing the OS to be compression-aware.

We also propose a novel methodology to evaluate the performance impact of increased effective memory capacity due to compression. The **main contributions** of this paper are:

- We identify, demonstrate, and analyze the data movement-related overheads of main memory compression, and present new trade-offs that need to be considered when designing a main memory compression architecture.
- We propose Compresso, with optimizations to reduce compressed data movement in a hardware compressed memory, while maintaining high compression ratio by



(a) Compressed address spaces (OS-Transparent) **(b) Ways to allocate a page in compressed space** **(c) Cache line overflow**

Fig. 1: Overview of data organization in compressed memory systems.

repacking data at the right time. Compresso exhibits only 15% compressed data movement accesses, as compared to 63% in an enhanced LCP-based competitive baseline (Section 4).

- We propose the first approach where the OS is completely agnostic to main memory compression, and all hardware changes are limited to the memory controller (Section 5).
- We devise a novel methodology for holistic evaluation that takes into account the capacity benefits from compression, in addition to its overheads (Section 6).
- We evaluate Compresso and compare it to an uncompressed memory baseline. When memory is constrained, Compresso increases the effective memory capacity by 85% and achieves a 29% speedup, as opposed to an 11% speedup achieved by the best prior work. When memory is unconstrained, Compresso matches the performance of the uncompressed system, while the best prior work degrades performance by 6%. Overall, Compresso outperforms best prior work by 24% (Section 7).

II. OVERVIEW

In this paper, we assume that the main memory stores compressed data, while the caches store uncompressed data (cache compression is orthogonal to memory compression). In this section, we discuss important design parameters for compressed memory architectures, and Compresso design choices.

A. Compression Algorithm and Granularity

The aim of memory compression is to increase memory capacity and lower bandwidth demand, which requires a sufficiently high compression ratio to make an observable difference. However, since the core uses uncompressed data, the decompression latency lies on the critical path. Several compression algorithms have been proposed and used in this domain: Frequent Pattern Compression (FPC) [13], Lempel-Ziv (LZ) [14], C-Pack [15], Base-Delta-Immediate (BDI) [16], Bit-Plane Compression (BPC) [6] and others [17, 18]. Although LZ results in the highest compression, its dictionary-based approach results in high energy overhead. We chose BPC due to its higher average compression ratio compared to other algorithms. BPC is a context-based compressor that transforms the data using its *Delta-Bitplane-Xor transform* to increase data compressibility, and then encodes the data. Kim et al. [6] describe BPC in the context of memory bandwidth optimization of a GPU. We adapt it for CPU memory-capacity compression by decreasing the compression granularity from 128 bytes to 64 bytes to match the cache lines in CPUs. We also observe that always applying BPC's transform is suboptimal. We add a module that compresses data with and without the transform, in parallel, and chooses the best option. Our optimizations save an average of 13% more memory, compared to baseline BPC. *Compresso uses the modified BPC compression algorithm, achieving 1.85x average compression on a wide range of applications.*

Compression Granularity. A larger compression granularity, i.e., compressing larger blocks of data, allows a higher compression ratio at the cost of higher latency and data movement requirements, due to different access granularities between core and memory. *Compresso uses the compression granularity of 64B.*

B. Address Translation Boundaries

Since different data compress to different sizes, cache lines and pages are variable-sized in compressed memory systems. As shown in Fig. 1a, the memory space available to the OS for allocating to the applications is greater than the actual installed memory. In *any* compressed memory system, there are two levels of translation. *(i) Virtual Address (VA)* to *OS Physical Address (OSPA)*, the traditional virtual-to-physical address translation, occurring before accessing the caches to avoid aliasing problems and *(ii) OSPA to Machine Physical Address (MPA)*, occurring before accessing main memory.

Fixed-size pages and cache lines from the VA need to be translated to variable-sized ones in MPA. OS-transparent compression translates both the page and cache line levels in the OSPA-to-MPA layer in the memory controller. On the other hand, compression-aware OS has variable-sized pages in OSPA space itself, and the memory controller only translates addresses into the variable cache line sizes for the MPA space.

Bus transactions in the system happen in the OSPA space. Since OS-aware solutions have different page sizes as per compressibility, in OSPA, their bus transactions do not follow current architectural specifications. Hence, all hardware, including all peripheral devices and associated drivers, must be modified to accommodate the change. *Compresso, being OS-transparent, has the same, fixed-page size in VA and OSPA spaces, for example, only 4KB pages.* Larger OS page sizes (2MB, 1GB etc.) can be broken into their 4KB building blocks in the MPA.

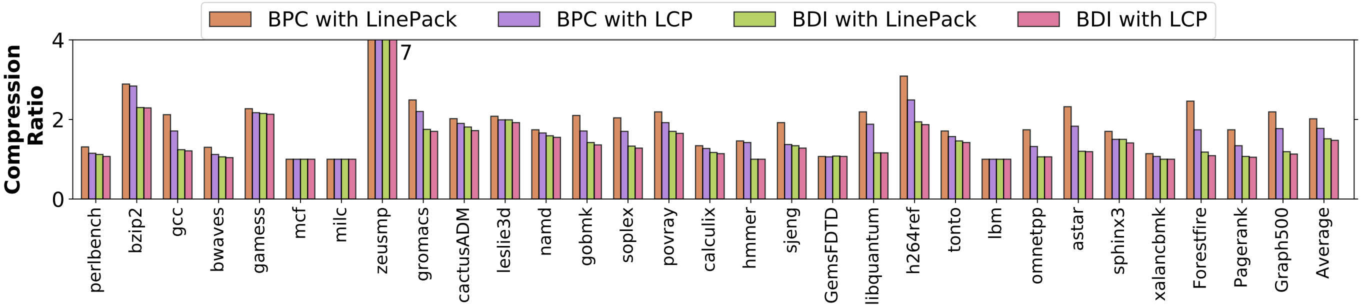


Fig. 2: Compression ratio with BPC and BDI with LCP-Packing vs LinePack.

C. Packing Compressed Cache Lines

Compressed cachelines are variable-sized, thus requiring a scheme for their placement within a page. The tradeoff lies between a high compression ratio and the ease of calculating the offset of the cacheline. There are two main ways of packing the cachelines. *(i)* **LinePack**: Storing the encoded sizes of each cacheline within the page, and calculating the offset by summing the line-sizes before it. For example, we could compress cache lines to 4 allowed sizes, thereby requiring 2 bits of metadata per cacheline. *(ii)* **LCP-Packing**: Linearly Compressed Pages (LCP) [4] proposed a smart way to reduce translation overhead, by compressing all the cache lines within a page to the same target size. Exceptions, i.e., cache lines that do not compress to the target size, are stored uncompressed and handled with explicit pointers in the metadata. Recent work [19] shows that these exceptions can be frequent for some benchmarks.

There are two main motivations for using LCP-packing compared to LinePack. *(i)* Simpler line-offset calculation. However, this calculation is required only when accessing main memory. In our evaluation, we describe a circuit that computes the cacheline offset for LinePack in effectively one additional memory cycle. *(ii)* LCP-packing allows a speculative main memory request in parallel with the metadata request. For exceptions, this leads to extra memory accesses. Additionally, this assumes the TLB to be aware of target line-size per page, which is not feasible in an OS-transparent system.

LCP-packing trades away high compression for simpler translation. Given its lower packing flexibility, it can match LinePack's compression only for those pages that have similar data across most cache lines; but underperforms when data is more variable. We find that due to this, LCP-packing works well with simpler compression algorithms. As shown in Fig. 2, while LCP loses only 2.3% in compression ratio compared to LinePack when used with BDI, it reduces the compression ratio by 13% when used for the more-aggressive BPC. *Compresso uses LinePack with 4 possible cache line sizes.*

D. Page Packing and Allocation

Fig. 1b shows the two ways a system can allocate variable-sized pages. *(i)* Variable-sized chunks. The drawback of this method is that it causes fragmentation and requires sophisticated management. *(ii)* Incremental allocation in fixed-sized chunks. These are trivial to manage, but need more pointers in the metadata.

A smaller base unit allows for better maximum compression. We choose 512B to be the base unit in order to balance metadata with compression. Both schemes have a 512B minimum MPA allocation for non-zero OSPA pages, however, chunk-based packing supports 8 MPA page sizes. For variable-sized chunks, more sizes lead to more fragmentation. Additionally, more page sizes can lead to more compressed data movement as discussed later, in Section IV-A1. Hence, we choose to evaluate the scheme with only 4 variable-sized chunks. We compare *variable-sized chunks* (512B, 1KB, 2KB and 4KB) with *512B fixed-sized chunks. Compresso uses incremental allocation in 512B chunks*, thereby allowing 8 page sizes (512B, 1KB, 1.5KB and so on).

III. COMPRESSO METADATA

In *both* OS-aware or OS-transparent compressed systems, each main-memory access (an LLC fill or writeback) requires OSPA to MPA translation. This translation uses metadata for identifying the offset within a page that corresponds to the OSPA cacheline. In an OS-transparent system, the translation from OSPA to MPA page number also happens using this metadata. To perform translation, Compresso maintains metadata for each OSPA page in a dedicated MPA space. The total number of OSPA pages is fixed and is equal to the memory capacity Compresso advertises to the OS during boot. Compresso uses 64B per OSPA page as metadata, similar to LCP [4], amounting to a storage overhead of 1.6% of the main memory capacity. This metadata is stored in main memory, not exposed to the OS, and has one entry per OSPA page, making the metadata address computation just a bitwise shift and add.

Fig. 3 shows a metadata entry in Compresso. The control section includes the page size and valid, zero, and compressed-page flags. The valid bit is set if the OSPA page has been mapped in MPA; the zero bit is set if the page contains all zeros; and the compressed bit is cleared if the page is uncompressed. Compresso also tracks the free space in each page to dynamically identify opportunities for repacking for better compression.

The second part of metadata entry is the *machine page frame numbers* (MPFNs). Up to 8 MPFNs are stored for pointing to the 512B chunks that comprise a compressed

Control/ Free bytes (2B)	MPFN of upto 8 chunks of 512B (8*4B = 32B)	Encoded Line Sizes (64*2bits = 16B)	Inflated line indices for 17 cachelines (14B)

Fig. 3: Metadata entry organization.

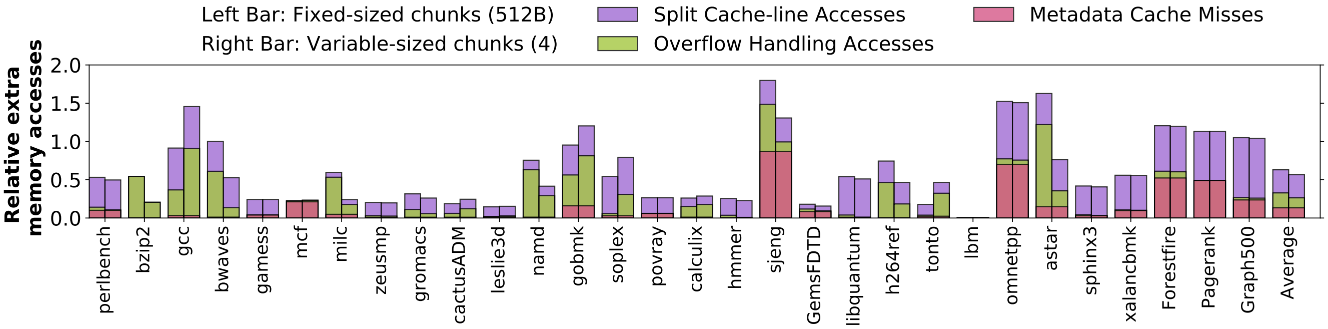


Fig. 4: Additional compression-related data movement memory traffic relative to uncompressed baseline.

OSPA page. Each cacheline can be compressed to one of 4 possible sizes and Compresso needs 16B to store the encoded line sizes (2 bits X 64, 64 cache lines in a 4KB page).

Inflation Room. Writebacks to a cacheline in the memory can lead to overflows when its compressed size grows. In a naive system, such an overflow would lead to multiple memory accesses, as the rest of the page would have to be moved to make more space. Instead, Compresso allows some number of such *inflated cachelines* to be stored uncompressed in the *inflation room* at the end of an MPA page, provided that there is space in that page (Fig. 5a). This is similar to the exception region in LCP, but is used for an entirely different reason—to reduce compression-related data movement, rather than to support a specific packing scheme. The cacheline sequence numbers corresponding to such inflated lines, along with the total number of inflated lines per page, are kept in the metadata. A cache line size (64B) aligned metadata is optimal for reducing extra data accesses. We use the leftover bytes to store 17 inflation pointers (6 bits each), and a counter for the number of inflated cachelines (6 bits).

IV. ADDITIONAL DATA MOVEMENT

Addressing and managing compressed data in the main memory can lead to significant data movement overhead, an average of 63% additional accesses compared to an uncompressed memory, and remains unevaluated in the previous work. The additional memory accesses due to compression can be from three sources: *(i)* **Split-access cache lines**: As the cache lines are variable-sized in a compressed memory system, they might be stored across the regular cache line boundaries in the memory, thereby leading to multiple memory accesses on the critical path (Fig. 5a). *(ii)* **Change in compressibility (overflows)**: As a cache line is written back from the LLC to the memory, its compressibility could have been decreased, meaning that it no longer fits in the space allocated to it. As shown in Fig. 1c, such a *cache line overflow* can lead to movement of the cache lines underneath the changed line. This happens on average 4 times per 1000 instructions. Moreover, if such incompressible data is streamed to a page, as its cache lines grow one by one in size, the page can eventually overflow its current allocation, for example, a 1KB page overflows to the next possible page size, say, 2KB. Such a *page overflow* occurs once every million instructions, and can require the relocation of the complete page, leading to a lot of additional memory reads and writes. *(iii)* **Metadata accesses**: As discussed earlier, every memory access needs an OSPA to MPA translation. Despite caching the relevant metadata, there can be misses, leading to additional memory accesses on the critical path.

Fig. 4 shows additional memory accesses from compression in a competitive baseline (based on LCP), relative to original memory accesses by each benchmark. These additional accesses amount to a high average of 63%, with a maximum of 180%.

A. Data Movement Related Trade-offs

A major contribution of Compresso is identifying new data movement related trade-offs.

1) Possible Sizes for Cache Lines and Pages

There exists a trade-off between compression ratio and the frequency of compressed data movement while choosing the possible number of sizes for pages and cache lines. Here, a possible size indicates one of the permissible sizes for a compressed page or a compressed cache line. Prior work only tries to maximize the compression ratio. More sizes lead to better compression. However, more bins are likely to cause more overflows, and trigger data movement. With respect to cache line packing, using 8 or 4 compressed cache line sizes offers 1.82 or 1.59 average compression ratio, respectively, when coupled with 8 possible page sizes. However, 17.5% more cache line overflows occur with 8 cache line sizes than with 4 cache line sizes. Supporting 8 size bins requires more metadata, due to larger encoded cache-block sizes, and complicates the offset-calculation circuit. *Compresso uses 4 cache line bins.*

A similar tradeoff exists with page size bins. 8 page sizes can achieve an average compression ratio of 1.85, but 4 page sizes only reach 1.59. On the other hand, using 8 page sizes leads to upto 53% more page resizing accesses compared to 4 page sizes. Compresso uses 8 page sizes with incremental fixed-sized chunks of 512B. This choice enables some important data movement optimizations that we discuss later. However, without those data movement optimizations, the better choice would be to have only 4 page sizes.

2) Cache Line Alignment

Split-access cache lines that cross cache line boundaries in memory lead to 30.9% extra memory accesses on average (Fig. 4), and yet have not been discussed in the previous work. In order to avoid such split-accesses, the system can choose to keep holes in the page to make sure that the lines are aligned. This leads to loss in compression (23% loss on average with BPC) and complex calculations for cache line offsets. The other possible solution is to have alignment-friendly cache line sizes, as we discuss later.

B. Data-Movement Optimizations

We now describe the optimizations we use in Compresso in order to alleviate the problem of compressed data movement. Fig. 6 shows how these optimizations reduce data movement. Without the full set of optimizations, our choice of incremental allocation of 512B chunks does not outperform using 4 variable page size allocation. However, with all optimizations, Compresso saves significant data movement compared to other configurations, reducing the average relative extra accesses to 15% compared to 63% in a *competitive compressed memory system* that uses 4 page sizes. We describe and analyze the mechanisms below.

1) Alignment-Friendly Cache Line Sizes

Previous work [4, 9] selected the possible sizes for cache lines as an optimization problem, maximizing the compression ratio. This led them to use the cache line sizes of 0, 22, 44 and 64B. This choice leads to an average of 30.9% of cache lines stored split across cache line boundaries. Accessing any of these cache lines would require an extra memory access on the critical path. We redo this search in order to maintain high compression ratio while decreasing the alignment problem, arriving at cache line sizes of 0, 8, 32 and 64B. The decrease in compression is just 0.25%, while at the same time bringing down the split-access cache lines from 30.9% to 3.2%. This decrease in split-access cache lines is intuitive, given that the cache line boundaries are at 64B. We note that if changes are made to the cache line packing algorithm such that it introduces holes in the pages to avoid split-access cache lines, we can completely avoid extra accesses from misalignment. However, that leads to higher complexity in the line-packing algorithm, as well as the offset calculation of a line in a page. Hence, we skip that optimization. Alignment-friendly cache line sizes bring down the extra accesses from 63% to 36%.

2) Page-Overflow Prediction

One of the major contributors to compression data movement is frequent cache line overflows. This often occurs in scenarios of streaming incompressible data. For example, it is common for applications to initialize their variables with zeros, and then write back incompressible values to them. If a zero page, which offers the maximum compressibility, is being written back with incompressible data, its cache lines will overflow one by one, causing the complete page to overflow multiple times, as it jumps the possible page sizes one by one. In such cases, we use a predictor to keep the page uncompressed to avoid the data movement due to compression. A repacking mechanism later restores the overall compression ratio.

We associate a 2-bit saturating counter with each entry in the metadata cache (Fig. 5b). The counter is incremented when any writeback to the associated page results in a cache line overflow and is decremented upon cache line underflows (i.e., new data being more compressible). Another 3-bit global predictor changes state based on page overflows in the system. We speculatively increase a page's size to the maximum (4KB) when the local as well as global predictors have the higher bit set. Hence, a page is stored uncompressed if it receives multiple streaming cache line overflows during a phase when the overall system is experiencing page overflows.

False negatives of this predictor lose the opportunity to save data-movement, while false positives squander compression without any reduction in data-movement. We incur 22.5% false negatives and 19% false positives on an average. This optimization reduces the remaining extra accesses from 36% to 26%.

3) Dynamic Inflation Room Expansion

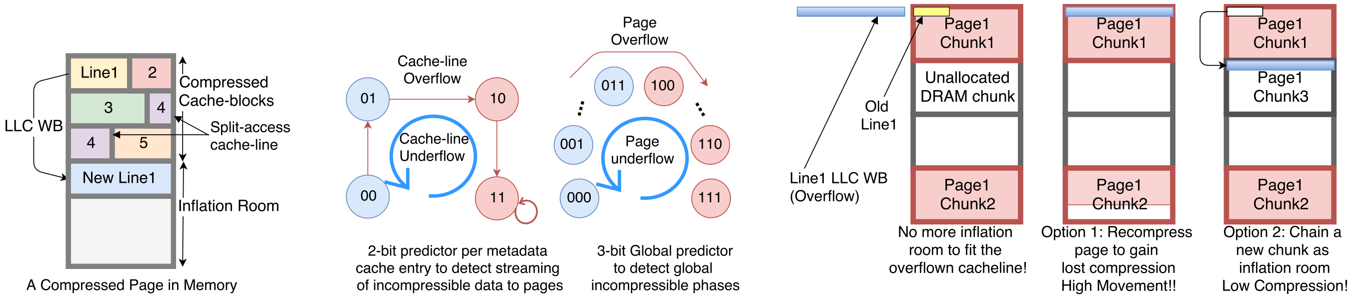
As discussed earlier, we use inflation room to store the cache lines that have overflown their initial compressed size. However, frequently, when a cache line overflows, it cannot be placed in the inflation room because of insufficient space in the MPA page, despite having available inflation pointers in the metadata. As shown in Option 1 in Fig. 5c, prior work would recompress the complete page, involving up to 64 cache line reads and 64 writes. Instead, we go for the optimized Option 2 (Fig. 5c) and allocate an additional 512B chunk, expanding the inflation room, requiring only 1 cache line write. Due to metadata constraints, this can be done only till the page has fewer than 8 allocations of 512B chunks, and fewer than 17 inflated lines. With this, we have reduced the remaining extra accesses from 26% to 19%.

4) Dynamic Page Repacking

Previous work [4, 9] in this field does not consider or evaluate repacking (recompressing) a page while it is in main memory. It is widely assumed that a page only grows in size from its allocation, till it is freed (which then leads to zero storage for OS-aware compression). However, we note that even for OS-aware compressed systems, repacking is important for long running applications. Fig. 7 shows the loss in compression ratio if no repacking is performed (24% storage benefits are squandered on average).

So far we have only focussed on dealing with decreasing compressibility, and not increase in compressibility, or underflows. However, as the cache lines within a page become more compressible, the page should be repacked. Additionally, we have proposed data-movement optimizations that leave compressible pages uncompressed in order to save on data-movement. Such poor packing will squander potentially high compressibility. On the other hand, repacking requires data movement, potentially moving many cache lines within the page. If a page is repacked on each writeback that improves compression ratio, it may lead to significant data movement.

The compression-squandering optimizations mostly affect the entries that are hot in the metadata cache, since they target the pages with streaming incompressible data, that get evicted from LLC with high locality. Using this insight, we choose metadata cache eviction of a page's entry as a trigger


Line1
2
3
4
4
5
New Line1
LLC WB
Compressed Cache-blocks
Split-access cache-line
Inflation Room
A Compressed Page in Memory
Cache-line Overflow
01
10
00
11
Cache-line Underflow
2-bit predictor per metadata cache entry to detect streaming of incompressible data to pages
Page Overflow
011
100
001
110
000
111
Page underflow
3-bit Global predictor to detect global incompressible phases
Line1 LLC WB (Overflow)
Old Line1
Page1 Chunk1
Unallocated DRAM chunk
Page1 Chunk2
No more inflation room to fit the overflown cacheline!
Page1 Chunk1
Page1 Chunk2
Option 1: Recompress page to gain lost compression High Movement!!
Page1 Chunk1
Page1 Chunk3
Page1 Chunk2
Option 2: Chain a new chunk as inflation room Low Compression!

(a) Inflation Room **(b) Page-Overflow Predictor** **(c) Dynamic Inflation Room Expansion**

Fig. 5: Illustrations for data-movement optimizations.

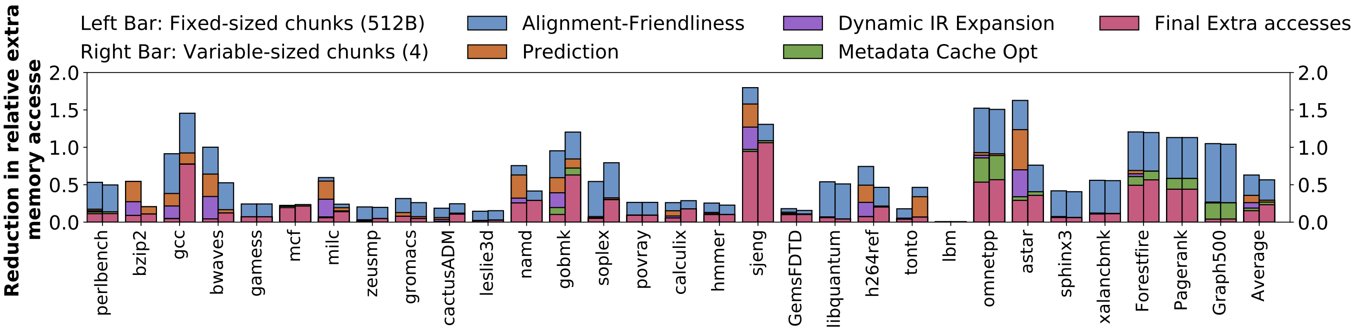


Fig. 6: Reduction in additional compression-related data movement memory traffic as optimizations are applied one by one.

to check for its repacking. *Compresso dynamically tracks the potential free space within each page, as part of its metadata entry.* This is novel to Compresso and important to keep the overheads of repacking low. Repacking is only triggered if enough space is available to positively impact actual effective memory use (i.e., deallocate at least one 512B chunk).

Dynamic Page Repacking reduces the compression squandered from 24% to 2.6%, while amounting to only 1.8% extra accesses. The small memory access overhead of repacking is attributed to a reasonable choice of trigger for repacking.

5) Metadata Cache Optimization

A metadata cache of at least the same size as the second-level TLB should be used in order to keep the common case of a TLB hit fast. For most applications, this is a large-enough size for capturing the metadata of its hot pages. However, some benchmarks, like Forestfire and omnetpp, have high miss rates, as shown in Fig. 4. A multi-core scenario with such applications is particularly problematic.

We identify an optimization that expands metadata cache coverage with low cost. We exploit the fact that all cache lines in an uncompressed OSPA page are exactly 64B, and only cache the first 32B of metadata for such pages (Fig. 3). The saved cache space from these half-sized entries is used for additional metadata entries, at the cost of a small amount of extra logic and tag area. This optimization allows us to double the effective cache size for incompressible data. Fig. 6 shows that this optimization exhibits crucial increase in hit rates for omnetpp, Forestfire, Pagerank and Graph500, despite using the same area as a regular cache. This optimization brings down the remaining extra accesses from 19% to 15%.

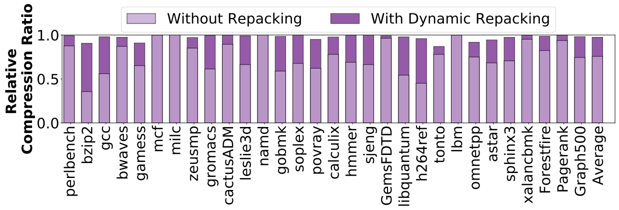


Fig. 7: Loss in compression ratio from no repacking.

Challenge to deal with	OS-Aware	OS-Transparent
Translation from OSPA to MPA	Yes	Yes
Data movement due to size change	Yes	Yes
Metadata access overheads	Yes	Yes
No knowledge of free pages in OSPA	No	Yes
Overcommitment of memory by the OS	No	Yes

Tab. I: OS-aware vs. OS-transparent compression.

Fig. 6 shows, after applying all the data-movement optimizations, the extra accesses are down to 15% on average, of which 3.2% are due to split-access cache lines, 2.1% due to change in compression, and 9.7% due to metadata cache misses. Note that these optimizations are orthogonal and can be applied separately, with the exception of Dynamic Inflation Room Expansion, which needs to be used in combination with a fixed-size chunk-based allocation.

V. OS-TRANSPARENCY CHALLENGES

There are a few challenges of compressed memory that are specific to OS-transparent systems (Tab. I).

A. Keeping Memory Compressed

It is important to keep the memory highly compressed to avoid running out of memory. For this, compression-aware OS banks on using zero bytes for free pages. Even partially OS-aware solutions like IBM MXT [8] require the OS to zero out a page upon freeing it, which breaks OS transparency and has high performance overheads. None of the previous work actively repacks pages for better compression. *Compresso keeps memory compressed by aggressively re-compressing pages to a smaller size when possible, as explained in Section IV-B4.*

B. Running Out Of Main Memory

In a compressed system, the OS is promised an address space larger than what is physically available. In cases where the data in main memory is less compressible than initially assumed, this can lead to an out-of-memory scenario in the MPA space. When this happens, solutions prior to Compresso raise an exception to the OS, thereby breaking the OS transparency and requiring OS modifications. Compresso, on the other hand, maintains complete OS transparency by leveraging existing OS features developed for use by virtual machines. Our challenge is very similar to the over-commitment of memory in a virtualized environment, where the total memory of all running virtual machines exceeds the memory capacity of the host. Any modern OS that can run as a virtual machine, including Linux, Windows, and MacOS, uses *memory ballooning* [20]. Ballooning helps safely overcommit memory in a virtualized environment without substantial performance degradation due to paging on the host (Fig. 8). When the memory pressure increases in one of the virtual machines (e.g., VM0), the hypervisor uses a *ballooning driver*, which is run within each guest, to reclaim memory from another guest VM (e.g., VM1) and allocate it to the memory-hogging virtual machine (VM0). To reclaim space, the ballooning driver "inflates" by demanding pages from the guest OS using system calls. With more memory required for the balloon, the guest uses its regular paging mechanism to provide that space, possibly paging out used pages. This means that only free pages or cold pages are freed up. The page numbers claimed by the balloon driver are communicated to the hypervisor, which then unmaps them in physical memory.

We propose to use the same OS facility to solve the problem of memory pressure resulting from poorly-compressed data. The Compresso driver can inflate and inform the hardware about the freed pages. Hardware then marks these pages as invalid in its metadata, requiring no storage in the MPA space. In case of Linux, the Compresso driver uses the APIs around the core function `__alloc_pages()`. We recently discovered similar OS-transparency ideas in a patent [21] with little detail and no evaluation. Liu et al [22] evaluate the performance overheads of ballooning on Linux. They show that in a 2GB RAM system, 1GB can be reclaimed in under 500ms without fragmentation and 1s for highly fragmented systems.

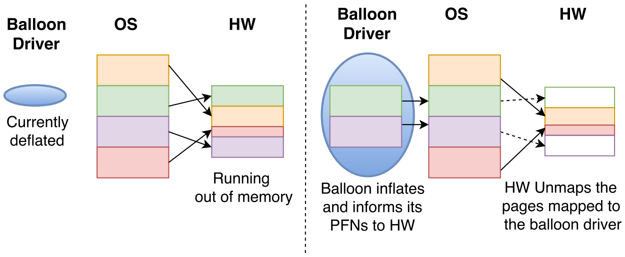

Balloon Driver
OS
HW
Currently deflated
Running out of memory
Balloon Driver
OS
HW
Balloon inflates and informs its PFNs to HW
HW Unmaps the pages mapped to the balloon driver

Fig. 8: Ballooning for OS-transparent handling of out-of-memory.

Memory %	LCP		Compresso		Unconstrained	
	1-Core	4-Core	1-Core	4-Core	1-Core	4-Core
80%*	1.04	1.54	1.15	1.78	1.24	2.1
70%*	**1.11**	**1.97**	**1.29**	**2.33**	**1.39**	**2.51**
60%*	1.28	2.45	1.56	2.81	1.72	3.23

All performance relative to uncompressed constrained memory baseline.
*Not all benchmarks finish execution.

Tab. II: Speedup Results from Memory-capacity impact evaluation with different constrained memory baselines.

VI. METHODOLOGY

A. Novel Dual-Simulation Methodology

In order to evaluate a compressed system in a holistic fashion, it is imperative to understand all its aspects: *(i)* Overheads due to compression latency and additional data movement, *(ii)* Bandwidth impact of compression due to inherent prefetching benefits and zero cache lines, and *(iii)* Impact of memory capacity increase on the application. The first two aspects of a compressed system have been evaluated in most previous work using cycle-based simulations. However, we observe that the memory capacity impact evaluation is missing, without which, a compressed system's evaluation is incomplete.

For a more complete evaluation of the system, we use two performance evaluation mechanisms: *(i) cycle-based simulations* to evaluate the latency overheads and bandwidth benefits of compression; and *(ii) memory-capacity impact runs* to evaluate the performance benefits of reduced OS paging when more effective capacity is made available through compression.

Memory-Capacity Impact Evaluation. Memory capacity impact evaluation requires complete runs of benchmarks, since compression ratio does not generally exhibit a recurring phase behavior. Since complete runs in simulators are highly time-consuming, we run benchmarks on a real HW (Intel Xeon E5 v3 CPU) and OS (Linux: Kernel 3.16). The idea is to change the memory available to the benchmark dynamically according to its real-time compressibility. We use a profiling stage, in which we run the applications and pause them every 200M instructions to dump the memory they have allocated in the physical address space and save vectors of the compression ratio over instruction intervals. We then run the benchmark using *taskset* on a fixed core, and use the hardware counters to calculate the number of retired instructions in that core. This instruction number is used to look up the saved vectors to get the compression ratio and accordingly change the memory budget.

We use the *cgroups* feature of Linux to budget the memory, either statically or dynamically. Static budgeting of the memory replicates a regular system. Dynamically changing the memory available to the system based on the compressibility of the data in the memory allows us to emulate a compressed system. The OS uses the swap area to page out the pages that do not fit in the limited memory provided for the runs.

We evaluate the impact of compression on a system with memory constrained to different fractions of the footprint of the running benchmark or workload (Tab. II). As the system becomes more memory constrained, the benefits of compression generally increase, but at 60% of the footprint, several benchmarks stall because of frequent paging. We illustrate the results with 70% constrained memory in Fig. 10 and Fig. 11. Note that if the system is not memory-capacity constrained, then the cycle-based evaluation alone captures the performance impact of Compresso.

B. CompressPoints for Representative Regions

Cycle-based simulation requires representative regions of a benchmark whose simulation results can be extrapolated for the complete benchmark. SimPoints [24] uses just the basic-block vectors (basic-block execution counts) to characterize the intervals of a benchmark and has been shown to correlate well with pipeline and caching behavior. However, while evaluating a compressed system, representativeness of data is crucial, leading us to use CompressPoints [23] instead.

CompressPoints extends basic-block vectors with memory compression metrics, including the compression ratio, rate of page overflows and underflows, and memory usage within each interval. This leads to much better representativeness of compression ratio. Consider GemsFDTD (Fig. 9), which exhibits remarkably different compressibility between the SimPoint and the CompressPoint. The performance simulation results presented in this paper use 10 CompressPoints of length 200 million instructions each.

C. Accuracy of Virtual Address Translation

Modern operating systems map all zero-initialized pages in the system to a single page frame, marking it as a copy-on-write. Therefore, if the virtual address space of the benchmark is used to take the memory snapshots, the compression ratios can be misleadingly high. In order to have a realistic compression ratio, we follow the approach used in [19] and [23], and use Linux *proc* filesystem. We use the *pagemap* to get the physical addresses and the *kpageflags* file to check whether a page is resident in main memory.

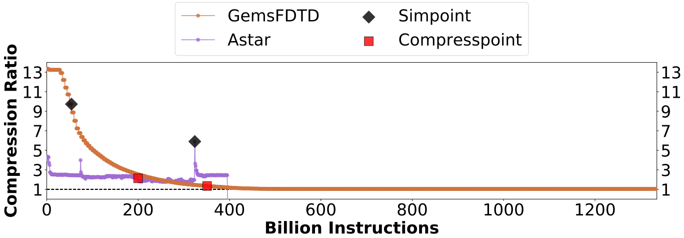


Fig. 9: Difference between the compressibility representativeness of SimPoint and CompressPoint for GemsFDTD and astar [23].

Core	3GHz OOO core × 1-4 Issue Width = 4 ROB = 192 entries 64KB L1D, 512KB L2 16-way 2MB L3 for 1-core, 8MB shared L3 for 4-core 64B cachelines TLB miss overhead = 0
DRAM	8GB DDR4-2666MHz (Capacity varied in Memory capacity evaluation) BL=8, tCL=18, tRCD=18, tRP=18 Decompression/Compression overhead = 12 Metadata cache hit latency = 2 Compression related accesses added to regular RD/WR Queues Compressed line sizes : 0/8/32/64 B Compressed page sizes : Basic Compressed System : 0/512B/1K/2K/4K Compresso : 0/512B/1K/1.5K/2K/2.5K/3K/3.5K/4K

Tab. III: Performance simulation parameters.

D. Simulator and Benchmarks

We use SPECcpu2006, Graph500, Forestfire [25], and Pagerank (centrality) [25] benchmarks. For cycle-based simulation, we extend zsim [26]. For energy estimation, we use McPAT [27] and CACTI [28]. For BPC energy estimate, we use 40nm TSMC standard cells. The evaluation parameters are detailed in Tab. III. We add a 12-cycle latency for compression/decompression with BPC. The model is similar to what is proposed by Kim et al. [6] for BPC in GPGPUs. Extending it to CPUs, we use 8 cycles for buffering the cache line from DDR4, 2 cycles to process 17 bit-planes, and 2 cycles for the concatenation.

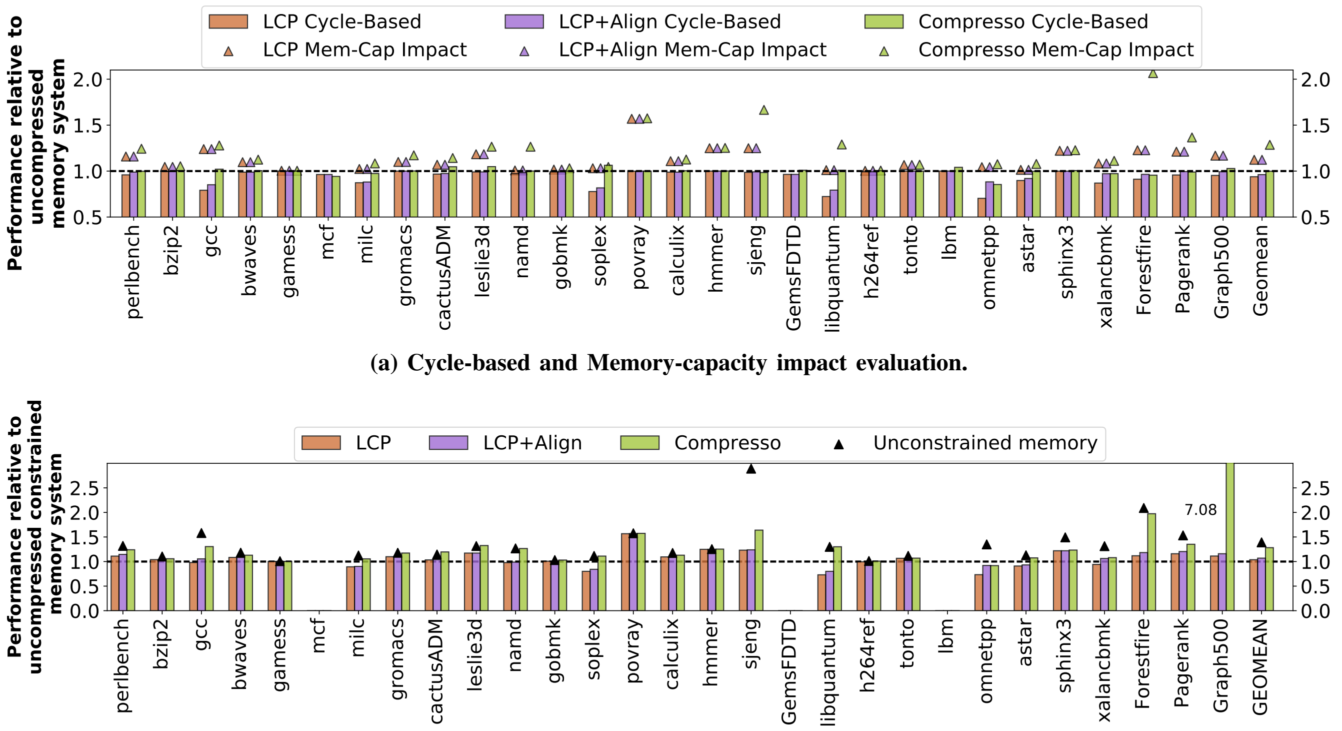
E. Multi-Core Simulations

We divide the benchmarks into groups (high and low) based on their single-core speedup, metadata cache hit rate, and sensitivity to limited memory. The mixes are created by selecting benchmarks such that there is equal representation from each group, as shown in Tab. IV. Mix10 represents a worst case scenario for compression overhead.

We use 10 multi-core CompressPoints, each of 200M instructions found from the methodology described in [23]. We use the syncedFastForward feature of zsim for multicore simulations, which fast-forwards the execution of all the cores till they reach their CompressPoints, and then simulates their run together, such that they are always under contention. For memory capacity impact evaluation, the workload consists of benchmarks rerunning under contention till the longest running benchmark completes execution on the unconstrained memory system. The same workload is then run on the constrained memory system and the percentage progress of each benchmark is measured after the elapsed time equals to the total runtime on the unconstrained memory system. The average progress across the individual benchmarks in the workload is chosen as the metric for comparison.

F. Evaluated Systems and Baselines

In order to get a competitive baseline, we compare Compresso to an optimized version of the system described in the LCP [4] paper. It uses 4 different compressed page sizes with an inflation room, and same size metadata cache



(a) Cycle-based and Memory-capacity impact evaluation.

(b) Overall performance (combining cycle-based and memory-capacity impact evaluations).
Mcf, GemsFDTD, and lbm are excluded, since they stall due to excessive paging if memory is constrained. Further discussion in text.

Fig. 10: Performance results for single-core system

as Compresso. It uses the OS-aware LCP with our optimized BPC compression algorithm. The compression benefits of zero-line accesses and free prefetch are simulated. **This forms the most competitive baseline based on prior work, only lacking Compresso's data movement optimizations.** Another related work, DMC [11], reports high compression ratios, which are achieved by opportunistically changing the granularity of compression – this involves substantial additional data movement. Buri [10] is another compressed memory system optimized for large databases, which also uses LCP. Therefore, LCP with an OS-aware behavior, is the most competitive baseline for Compresso. In addition, we separate the impact of Alignment-friendly cache lines by evaluating an LCP+Align version of LCP-system. We also evaluated Compresso with LCP-packing (instead of LinePack), but since the results were inferior across all benchmarks (average slowdown of 11.8%), we do not report them in the paper.

Overall Performance is calculated as a multiplication of the performance numbers from the cycle-based simulation and the memory-capacity impact evaluation, since the two speedups are mutually independent. This overall performance metric represents our best effort to estimate performance by combining results from the memory-capacity impact and cycle-based simulations. Unfortunately, this metric does not take into account application phase changes and more complex interactions where capacity and latency do not have a uniform impact on performance. The unconstrained memory system provides an upper bound on the performance gains possible with a larger memory capacity.

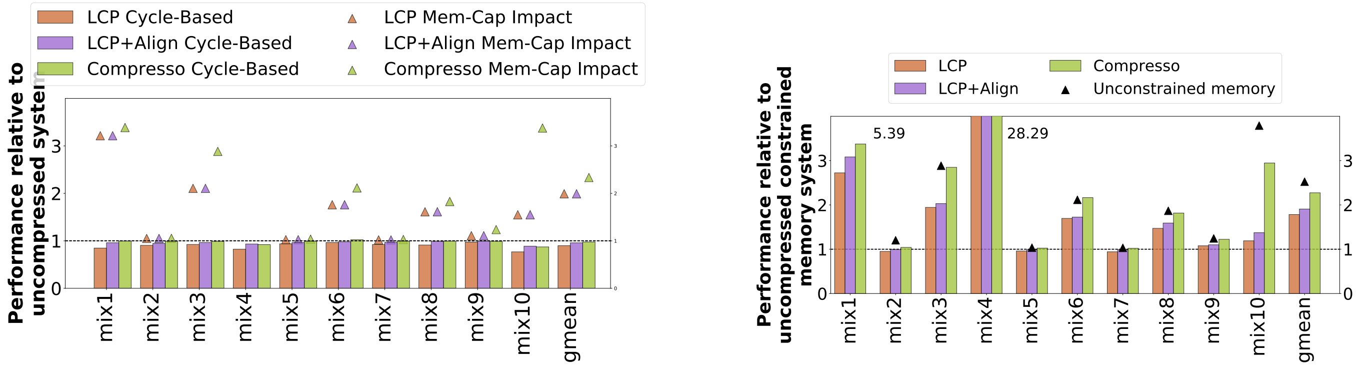
VII. EVALUATION

Fig. 6 shows the impact of each data-movement optimization on the additional memory accesses. In this section, we present the results from performance simulations, and energy-area overheads.

A. Single-Core Performance

Cycle-Based Simulation. Fig. 10a shows the relative performance with respect to an uncompressed system. The geomeans are 0.938 for LCP-system, 0.961 for LCP-system with Alignment-friendly cache lines, and 0.998 for Compresso. These results only show slowdowns due to compression overheads or improvements due to bandwidth benefits, and not the impact of increased memory capacity.

The applications gcc, Graph500, cactusADM, libquantum, leslie3d, and soplex gain more than 2% performance over the baseline uncompressed system. This happens because of two reasons. First, fills (and writebacks) of all-zero cache lines do not require memory access and are handled by accessing (cached) compression metadata alone. Benchmarks like leslie3d and soplex exhibit high zero-line accesses of 43% and 25%, respectively. Among these, soplex, having the highest bandwidth requirements, exhibits an overall relative performance of 6%. Second, a single 64B access to compressed memory returns multiple compressed cache lines and acts as a free prefetch. The libquantum benchmark, which also exhibits high bandwidth demand, requires 12% fewer memory accesses in a compressed system because of this prefetching. Compression



(a) Cycle-based and Memory-capacity impact evaluation results.

(b) Overall performance.

Fig. 11: Performance results for multi-core system

also increases the row buffer locality of libquantum and other high spatial-locality benchmarks. The overall speedup for libquantum with Compresso is 2% above baseline.

LCP-system (baseline), being OS-aware, requires a page fault upon every page overflow. Being OS-transparent, Compresso has a cheaper page overflow handling mechanism in addition to having a lower number of page overflows overall, as a result of the data movement optimizations we implement. This results in the high incremental speedup benefits of Compresso over LCP-system for benchmarks like gcc, cactusADM, libquantum, astar, and soplex. These numbers can be directly explained with Fig. 6. For instance, data movement optimizations bring down the extra memory access of gcc from 91% to 4% and of libquantum from 54% to 1%. Compresso decreases the maximum slowdown due to compression from 31% to 15%.

For benchmarks mcf, omnetpp, Forestfire, and Pagerank, the LCP+Align performs slightly better than Compresso. This is due to the fact that in parallel with a metadata memory access, LCP allows speculative access to the memory based on the assumption that the cache line being accessed is not an exception. This benefits the applications with very high metadata miss rates.

Optimization-Based Performance Breakdown. On average, performance is increased using alignment-friendly cache line sizes by 2.4%, predicting incompressibility by 1.2%, Dynamic Inflation Room Expansion by 1.1% and metadata cache optimization by 0.5%.

Memory Capacity Impact Evaluation. Fig. 10a shows the impact of memory compression applied to a constrained memory system (at 70%), relative to an uncompressed baseline. Gamess, h264ref, and bzip2 do not show much sensitivity to limited memory, as even the uncompressed constrained system achieves the maximum performance of an unconstrained memory system (Fig. 10b). Mcf, GemsFDTD, and lbm are highly sensitive to memory capacity and stall due to frequent paging in a constrained memory environment. Since they are all incompressible, compression does not help either. However, when run as a part of a workload with benchmarks that are compressible, they complete execution, as we show in Section VII-B.

Mix1	mcf, GemsFDTD, libquantum, soplex
Mix2	milc, astar, gamess, tonto
Mix3	Forestfire, lbm, leslie3d, hmmer
Mix4	sjeng, omnetpp, gcc, namd
Mix5	xalancbmk, cactusADM, calculix, sphinx3
Mix6	perlbench, bzip2, gromacs, gobmk
Mix7	bwaves, povray, h264ref, Pagerank
Mix8	mcf, bwaves, Graph500, perlbench
Mix9	Forestfire, povray, gamess, hmmer
Mix10	Forestfire, Pagerank, Graph500, cactusADM

Tab. IV: Workloads for multi-core evaluation

The rest of the benchmarks are either almost-linearly sensitive to the amount of memory they have, or, in some cases, need at least some threshold level of memory to perform well (Graph500, Forestfire, namd). Due to the lower compression benefits of LCP, it exhibits an average relative performance of 1.11 as compared to Compresso's 1.29. The upper bound for performance is approximated with unconstrained memory system, which achieves a relative speedup of 1.39.

Combining the results from the two evaluations, we observe an average relative overall speedup of 1.03 for LCP, 1.06 for LCP+Align and 1.28 for Compresso. *Compresso therefore, outperforms the LCP by 24.2%.*

B. 4-Core Performance Evaluation

Cycle-Based Simulation. Fig. 11a shows the performance overheads of compression in a 4-core system. Mix1 shows good performance with Compresso, despite containing mcf, which is a bad performer in the single-core runs. This is because of the more pronounced bandwidth reduction benefits due to soplex and libquantum in Mix1. A similar pattern can be seen for Mixes 2, 3, and 8. These Mixes also exhibit much higher performance with Compresso than with LCP, due to the data movement optimizations, and low page overflow handling overhead.

The pressure on the metadata cache is higher in a 4-core scenario. The impact can be seen in Mixes 4 and 10. Mix4 contains omnetpp and sjeng, which are bad performers in isolation due to high metadata miss rates. The compression bandwidth benefits from gcc and namd balance this effect a little, but it still results in a low performance of 0.92. Mix10 contains Forestfire, Pagerank, and Graph500, all of which have high metadata miss rates, but also have good compression, resulting in 0.87 speedup (i.e., 13% slowdown). Compresso out-performs

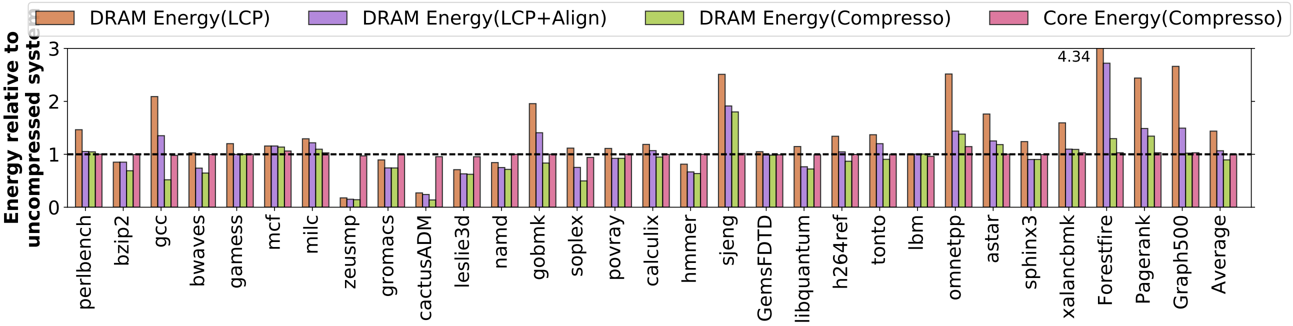


Fig. 12: Energy evaluation of Compresso

LCP across all Mixes. LCP+Align out-performs Compresso for Mixes 4 and 10 by 1.3% and 1.4%, respectively, due to the benefits gained from LCP's parallel speculative memory access. We evaluated these systems with a 96KB metadata cache. If targeting warehouse-scale computing, the metadata cache size could be increased, which could lead to better performance in benchmarks like omnetpp, Forestfire, and Pagerank.

On average, relative speedups are 0.975 for Compresso, 0.9 for LCP, and 0.95 for LCP+Align.

Optimization-Based Performance Breakdown. On average, performance is increased using alignment-friendly cache line sizes by 5.5%, and 1% each by predicting incompressibility, dynamic IR expansion and metadata cache optimization.

4-Core Memory Capacity Impact Evaluation. Fig. 11a shows the performance benefits of compression when applied to a constrained memory system. Note that since not all benchmarks reach their maximum footprint at the same time, it allows a slack memory based on the phases of different benchmarks in a workload. This is the reason why Mixes 2, 5 and 7 are not very sensitive to the memory availability, despite containing xalancbmk, Pagerank, and povray, all of which are very sensitive to limited memory when run in isolation.

Once some benchmarks in the workload have compressed enough to allow the other incompressible benchmarks in the workload to use more memory, the workload performs well and additional memory does not make a difference. This can be seen in the behavior of Mix 4.

The other Mixes exhibit better performance with Compresso, since Compresso frees up more space as compared to LCP. Mix10 again is interesting from this point of view, since it gains more than 100% performance with Compresso over LCP. This is because Forestfire, Pagerank, and Graph500 all gain significant memory capacity when using LinePack over LCP. On average, Compresso achieves 2.33 relative performance over a constrained memory system, as compared to 1.97 achieved by LCP, and reaches very close to the upper bound, unconstrained memory's performance (2.51) (Fig. 11b).

Fig. 11b also shows the overall performance in a multi-core scenario, combining the cycle-based and memory-capacity impact evaluations. The relative overall speedup is 2.27 for Compresso, 1.78 for LCP and, 1.9 for LCP+Align, relative to the constrained memory system. *Overall, in a 4-core setup, Compresso achieves a 27.5% performance over LCP.*

Related Work	OS-transparent	Hardware changes	Compression Granularity	Cacheline packing	Data-movement optimizations
IBM-MXT	Partially	LLC, MC	1KB	N/A	N/A
RMC	No	BST, MC	64B	LinePack	Light
LCP	No	TLBs, MC	64B	LCP	No
Buri	Partially	MC	64B	LCP	No
DMC	Partially	MC	64B or 1KB	LCP or N/A	No
Compresso	Yes	MC	64B	LinePack	Yes

Tab. V: Related work summary

C. Energy Overheads

Major effects of Compresso on system energy are on: *(i)* core's energy usage due to slowdown/speedup; *(ii)* DRAM energy due to change in data movement; and *(iii)* memory controller energy due to the BPC compressor/decompressor and metadata cache. Upon synthesizing BPC with 40nm TSMC standard cells [28] at 800MHz, we find that the active power utilization is 7 mW, which is less than 0.4% of a DRAM channel's active power for a 2GB DDR4-2666 channel. The access energy of an 8-way 96KB metadata cache is 0.08 nJ, which is less than 0.8% of a DRAM read access energy.

Fig. 12 shows that well-compressed benchmarks like zeusmp and cactusADM exhibit DRAM energy benefits due to accesses to zero cache lines that hit in metadata cache. Mcf, sjeng, Forestfire, Pagerank and omnetpp exhibit higher DRAM energy due to extra memory accesses from metadata cache misses, and astar has higher DRAM energy due to compression-related data movement. In comparison to the LCP-system, Compresso achieves 60% more energy savings, and 19% over the LCP+Align system. Compared to an uncompressed system, Compresso reduces DRAM energy by 11% and has equal core energy usage.

Previous work [29] shows that compression may increase bus toggling energy. While a detailed evaluation of bus models is beyond our scope, Seol et al. [30] have demonstrated that the DRAM channel switching energy, for example, comprises only 7% of the DRAM subsystem energy. Further, Ghose et al. [31] conclude that BDI-based compression does not consume more internal DRAM switching energy than a baseline memory system, though an energy-optimized encoding can reduce internal-DRAM switching energy by 24% on average across a set of benchmarks. Given that Compresso reduces compressed data movement by an average of 45% and its better compression ratio decreases paging, the overall energy savings are positive.

D. Area Overhead

The BPC compressor unit with our optimizations synthesized using 40nm TSMC standard cells [32] at 800MHz requires an overall area of 43Kμm^2, corresponding to roughly 61K NAND2 gates. A 96KB metadata (single read/write port) cache has an area of roughly 100Kμm^2 [28]. Although the area overhead is not negligible, the benefits from Compresso justify it.

E. Cache Line Offset Calculation

In Compresso, we find the position of a cache line within a compressed page by looking up the metadata cache and calculating the offset with a custom low-latency arithmetic unit. The bin sizes (0/8/32/64B) are first shifted right by 3 bits to reduce their width. We then need to add up to 63 numbers of values 0/1/4/8. A simple 63-input 4-bit adder requires under 1.5K NAND gates and 38 NAND-gate delays, which can be reduced to 32 gate delays with some optimizations that take possible inputs into account. DDR4-2666MHz allows only ˜30 gate delays in one cycle, but this offset calculation can be partially parallelized with metadata cache lookup, thereby making the overhead only 1 cycle.

VIII. RELATED WORK

Software memory compression has been implemented in commercial systems for the past decade. For example, Linux (since Linux 3.14 and Android 4.4), iOS [33] and Windows [34] keep background applications compressed. Any access to compressed pages generates a compressed-page fault, prompting the OS to decompress the page into a full physical frame. Since the hardware is not involved, hot pages are stored uncompressed, thereby limiting the compression benefits.

Most other prior work in system compression deals with cache compression or bandwidth compression [5]–[7, 13, 16, 18, 35]–[45]. We summarize the prior work in main memory compression in Tab. V.

IBM MXT [8] was mostly non-intrusive, but required a few OS changes. The OS was informed of an out-of-memory condition by the hardware changing watermarks based on compressibility. Furthermore, OS was required to zero out free pages to avoid repacking data in hardware. Since the compression granularity was 1KB, a memory read would result in 1KB worth of data being decompressed and transferred. To address this, a large 32MB L3 cache with a 1KB line size was introduced. These features would significantly hinder the performance and energy efficiency of MXT in today's systems.

Robust Memory Compression [9] was an OS-aware mechanism, proposed to improve performance of MXT. RMC used translation metadata as part of page table data and cached this metadata on chip in the *block size table*. RMC used 4 possible page sizes, and divided a compressed page into 4 subpages, with a hysteresis area at the end of each subpage to store inflated cache lines.

Linearly Compressed Pages [4] was also OS-aware. It addressed the challenge of cache line offset calculation by compressing them all to same size, such that compression still remains high. It used an exception region for larger cachelines in order to achieve higher compression, hence still requiring metadata access for every memory access.

Buri [10] targeted lightweight changes to the OS by managing memory allocation and deallocation in hardware. However, it required the OS to adapt to scenarios with incompressible data, using similar methods as MXT. It used an OS-visible "shadow address" space, similar to OSPA and used the LCP approach to handle computing cache line offsets within a compressed page.

DMC [11] proposed using two kinds of compression : LZ at 1KB granularity for cold pages and LCP with BDI for hot pages. It decides between the 2 compression mechanisms in an OS-transparent fashion, at 32KB boundaries, which can potentially increase the data movement. Additionally, LCP is now applied for 32KB pages instead of 4KB pages, resulting in lower compression.

CMH [12] proposed compression to increase the capacity of a Hybrid Memory Cube (HMC), by having compressed and uncompressed regions per vault, with a floating boundary in between. The compression and allocation granularity is at the cache-block level, thereby not needing any page movement. However, having to maintain strictly separate compressed and uncompressed regions amplifies the problem of cache-block level data movement, causing their scheme to require explicit defragmentation every 100 million cycles.

IX. CONCLUSION

We identify important tradeoffs between compression aggressiveness and unwanted data movement overheads of compression, and optimize on these tradeoffs. This enables Compresso to acheive higher compression benefits while also reducing the performance overhead from this data movement. We present the first main-memory compression architecture that is designed to run an unmodified operating system. We propose a detailed holistic evaluation with high accuracy using the novel methodologies of Memory Capacity Impact Evaluations. Overall, by reducing the data movement overheads while keeping high compression, we make main memory compression pragmatic for adoption by real systems, and show the same with holistic evaluation.

X. ACKNOWLEDGEMENTS

We thank our anonymous reviewers for their valuable feedback and suggestions. This work was funded in part by a gift from Intel Corporation, and supported by the National Science Foundation Grant #1719061. We thank Michael Sullivan, Jungrae Kim, Yongkee Kwon, Tianhao Zheng, Vivek Kozhikkottu, Amin Firoozshahian, and Omid Azizi for their help and feedback during various phases of the work.

REFERENCES

[1] J. Dean and L. A. Barroso, "The Tail at Scale," *Communications of the ACM*, vol. 56, pp. 74–80, 2013.

[2] M. E. Haque, Y. hun Eom, Y. He, S. Elnikety, R. Bianchini, and K. S. McKinley, "Few-to-Many: Incremental Parallelism for Reducing Tail Latency in Interactive Services," in *ASPLOS*, 2015, pp. 161–175.

[3] M. Jeon, Y. He, S. Elnikety, A. L. Cox, and S. Rixner, "Adaptive Parallelism for Web Search," in *Eurosys*, 2013, pp. 155–168.

[4] G. Pekhimenko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, P. Gibbons, M. Kozuch, and T. Mowry, “Linearly compressed pages: a low-complexity, low-latency main memory compression framework,” in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013, pp. 172–184.

[5] R. G. Y. Zhang, “Enabling Partial Cache Line Prefetching Through Data Compression,” in *ICPP*, 2000, pp. 277–285.

[6] J. Kim, M. Sullivan, E. Choukse, and M. Erez, “Bit-Plane Compression: Transforming Data for Better Compression in Many-Core Architectures,” in *Proceedings of the 43rd Annual International Symposium on Computer Architecture*, 2016, pp. 329–340.

[7] J. Yang, Y. Zhang, and R. Gupta, “Frequent value compression in data caches,” in *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2000, pp. 258–265.

[8] R. B. Tremaine, P. A. Franaszek, J. T. Robinson, C. O. Schulz, T. B. Smith, M. Wazlowski, and P. M. Bland, “IBM Memory Expansion Technology (MXT),” in *IBM Journal of Research and Development, vol. 45, No. 2*, 2001, pp. 271–285.

[9] M. Ekman and P. Stenstrom, “A Robust Main-Memory Compression Scheme,” in *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, 2005, pp. 74–85.

[10] J. Zhao, S. Li, J. Chang, J. L. Byrne, L. L. Ramirez, K. Lim, Y. Xie, and P. Faraboschi, “Buri: Scaling big-memory computing with hardware-based memory expansion,” *ACM Trans. Archit. Code Optim.*, vol. 12, no. 3, pp. 31:1–31:24, 2015.

[11] S. Kim, S. Lee, T. Kim, and J. Huh, “Transparent dual memory compression architecture,” in *Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques*, 2017, pp. 206–218.

[12] C. Qian, L. Huang, Q. Yu, Z. Wang, and B. Childers, “CMH: Compression management for improving capacity in the hybrid memory cube,” in *Proceedings of the 15th ACM International Conference on Computing Frontiers*, 2018.

[13] A. R. Alameldeen and D. A. Wood, “Frequent Pattern Compression: A significance-based compression scheme for L2 caches,” Technical Report 1500, Computer Sciences Department, University of Wisconsin-Madison, Tech. Rep., 2004.

[14] J. Ziv and A. Lempel, “A universal algorithm for sequential data compression,” *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977.

[15] X. Chen, L. Yang, R. Dick, L. Shang, and H. Lekatsa, “C-PACK: a high-performance microprocessor cache compression algorithm,” in *IEEE Educational Activities Department vol. 18*, 2010, pp. 1196–1208.

[16] G. Pekhimenko, V. Seshadri, O. Mutlu, P. Gibbons, M. Kozuch, and T. Mowry, “Base-delta-immediate compression: practical data compression for on-chip caches,” in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, 2012, pp. 377–388.

[17] A. Arelakis, F. Dahlgren, and P. Stenstrom, “HyComp: A Hybrid Cache Compression Method for Selection of Data-type-specific Compression Methods,” in *Proceedings of the 48th International Symposium on Microarchitecture*. New York, NY, USA: ACM, 2015, pp. 38–49.

[18] A. Arelakis and P. Stenstrom, “SC2: A Statistical Compression Cache Scheme,” in *Proceeding of the 41st Annual International Symposium on Computer Architecture*. Piscataway, NJ, USA: IEEE Press, 2014, pp. 145–156.

[19] S. Sardashti and D. A. Wood, “Could compression be of general use? evaluating memory compression across domains,” *ACM Trans. Archit. Code Optim.*, vol. 14, no. 4, pp. 44:1–44:24, Dec. 2017.

[20] C. A. Waldspurger, “Memory resource management in VMware ESX server,” in *Proceedings of the 5th symposium on Operating systems design and implementation (OSDI)*, 2002, pp. 181–194.

[21] P. A. Franaszek and D. E. Poff, “Management of Guest OS Memory Compression In Virtualized Systems,” Patent US20 080 307, 2007.

[22] T. Chen, “Introduction to acpi-based memory hot-plug,” in *LinuxCon/CloudOpen Japan*, 2013.

[23] E. Choukse, M. Erez, and A. R. Alameldeen, “CompressPoints: An Evaluation Methodology for Compressed Memory Systems,” in *IEEE Computer Architecture Letters*, vol. PP, no. 99, 2018, pp. 1–1.

[24] E. Perelman, G. Hamerly, M. Biesbrouck, T. Sherwood, and B. Calder, “Using SimPoint for accurate and efficient simulation,” in *Proceedings of the International Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2003, pp. 318–319.

[25] J. Leskovec and R. Sosič, “SNAP: A General-Purpose Network Analysis and Graph-Mining Library,” *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 8, p. 1, 2016.

[26] D. Sanchez and C. Kozyrakis, “ZSim: fast and accurate microarchitectural simulation of thousand-core systems,” in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013, pp. 475–486.

[27] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, “McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures,” in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009, pp. 469–480.

[28] S. J. E. Wilton and N. P. Jouppi, “CACTI: An Enhanced Cache Access and Cycle Time Model,” *IEEE Journal of Solid-State Circuits*, vol. 31, pp. 677–688, 1996.

[29] G. Pekhimenko, E. Bolotin, N. Vijaykumar, O. Mutlu, T. C. Mowry, and S. W. Keckler, “A case for toggle-aware compression for GPU systems,” *International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[30] H. Seol, W. Shin, J. Jang, J. Choi, J. Suh, and L.-S. Kim, “Energy Efficient Data Encoding in DRAM Channels Exploiting Data Value Similarity,” in *Proceedings of the 43rd International Symposium on Computer Architecture*, 2016.

[31] S. Ghose, A. G. Yaglikçi, R. Gupta, D. Lee, K. Kudrolli, W. X. Liu, H. Hassan, K. K. Chang, N. Chatterjee, A. Agrawal, M. O’Connor, and O. Mutlu, “What Your DRAM Power Models Are Not Telling You: Lessons from a Detailed Experimental Study,” in *SIGMETRICS*, 2018.

[32] Taiwan Semiconductor Manufacturing Company, “40nm CMOS Standard Cell Library v120b,” 2009.

[33] “Apple Releases Developer Preview of OS X Mavericks With More Than 200 New Features,” 2013. [Online]. Available: https://www.apple.com/pr/library/2013/06/10Apple-Releases-Developer-Preview-of-OS-X-Mavericks-With-More-Than-200-New-Features.html

[34] “Announcing Windows 10 insider preview,” 2015. [Online]. Available: https://blogs.windows.com/windowsexperience/2015/08/18/announcing-windows-10-insider-preview-build-10525/#36QzdLwDd3Eb45ol.97

[35] E. G. Hallnor and S. K. Reinhardt, “A unified compressed memory hierarchy,” in *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, 2005, pp. 201–212.

[36] A. R. Alameldeen and D. A. Wood, “Adaptive cache compression for high-performance processors,” in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, 2004, pp. 212–.

[37] A. Shafiee, M. Taassori, R. Balasubramonian, and A. Davis, “MemZip: Exploring unconventional benefits from memory compression,” in *International Symposium on High Performance Computer Architecture (HPCA)*, 2014, pp. 638–649.

[38] D. J. Palframan, N. S. Kim, and M. H. Lipasti, “COP: To Compress and Protect Main Memory,” in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*. New York, NY, USA: ACM, 2015, pp. 682–693.

[39] S. Sardashti, A. Seznec, and D. A. Wood, “Yet Another Compressed Cache: A Low-Cost Yet Effective Compressed Cache,” *ACM Trans. Archit. Code Optim.*, pp. 27:1–27:25, 2016.

[40] J. Kim, M. Sullivan, S.-L. Gong, and M. Erez, “Frugal ECC: Efficient and Versatile Memory Error Protection Through Fine-grained Compression,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 12:1–12:12.

[41] “Qualcomm Centriq 2400 Processor,” 2017. [Online]. Available: https://www.qualcomm.com/media/documents/files/qualcomm-centriq-2400-processor.pdf

[42] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay, “Rock: A high-performance Sparc CMT processor,” *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, vol. 29, no. 2, pp. 6–16, March 2009.

[43] R. de Castro, A. Lago, and M. Silva, “Adaptive compressed caching: Design and implementation,” in *Computer Architecture and High Performance Computing, 2003. Proceedings. 15th Symposium on*, 2003.

[44] H. Alam, T. Zhang, M. Erez, and Y. Etsion, “Do-It-Yourself Virtual Memory Translation,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture*. New York, NY, USA: ACM, 2017, pp. 457–468.

[45] P. Franaszek, J. Robinson, and J. Thomas, “Parallel compression with cooperative dictionary construction,” in *Proceedings of the Data Compression Conference*, 1996, pp. 200–209.