Translation-optimized Memory Compression for Capacity

Gagandeep Panwar[†], Muhammad Laghari[†], David Bears[†], Yuqing Liu[†], Chandler Jearls[†§*],
Esha Choukse[‡], Kirk W. Cameron[†], Ali R. Butt[†], Xun Jian[†]
[†]*Virginia Tech* [§]*Apple* [‡]*Microsoft Research*

***Abstract*—The demand for memory is ever increasing. Many prior works have explored hardware memory compression to increase effective memory capacity. However, prior works compress and pack/migrate data at a small - memory block-level - granularity; this introduces an additional block-level translation after the page-level virtual address translation. In general, the smaller the granularity of address translation, the higher the translation overhead. As such, this additional block-level translation exacerbates the well-known address translation problem for large and/or irregular workloads.**

A promising solution is to only save memory from cold (i.e., less recently accessed) pages without saving memory from hot (i.e., more recently accessed) pages (e.g., keep the hot pages uncompressed); this avoids block-level translation overhead for hot pages. However, it still faces two challenges. First, after a compressed cold page becomes hot again, migrating the page to a full 4KB DRAM location still adds another level (albeit page-level, instead of block-level) of translation on top of existing virtual address translation. Second, only compressing cold data require compressing them very aggressively to achieve high overall memory savings; decompressing very aggressively compressed data is very slow (e.g., $> 800ns$ assuming the latest Deflate ASIC in industry).

This paper presents Translation-optimized Memory Compression for Capacity (TMCC) to tackle the two challenges above. To address the first challenge, we propose compressing page table blocks in hardware to opportunistically embed compression translations into them in a software-transparent manner to effectively prefetch compression translations during a page walk, instead of serially fetching them after the walk. To address the second challenge, we perform a large design space exploration across many hardware configurations and diverse workloads to derive and implement in HDL an ASIC Deflate that is specialized for memory; for memory pages, it is 4X as fast as the state-of-the art ASIC Deflate, with little to no sacrifice in compression ratio.

Our evaluations show that for large and/or irregular workloads, TMCC can either improve performance by 14% without sacrificing effective capacity or provide 2.2x the effective capacity without sacrificing performance compared to a state-of-the-art hardware memory compression for capacity.

***Keywords*-memory; hardware memory compression; address translation; memory subsystem; compression ASIC**

I. Introduction

Memory is a costly resource in computing. For example, under many VM instances in AWS (e.g., t2, t3, t3a, t4g), doubling a VM's memory size while keeping the number of vCPUs the same doubles the total hourly cost of the VM (e.g., going from a 0.5GB VM with 1 vCPU to a 1GB VM with 1 vCPU doubles the total hourly cost of the VM). Other large-scale data center operators (e.g., Facebook [1], Microsoft [2], Google [3]) also report that memory makes up a large and rising fraction of total infrastructure cost.

To increase effective memory capacity without increasing actual DRAM cost, many prior works [4], [5], [6], [7], [8], [9], [10] have explored hardware memory compression. Hardware transparently compresses DRAM content on-the-fly with the memory controller evicting/writing back memory blocks to DRAM. To increase effective memory capacity (i.e., to store more values in memory), the memory controller also transparently migrates compressed data closer together to free up space in DRAM for future data. To migrate data, the memory controller takes on several OS features; specifically, memory controller maintains a dynamic, page-table-like, fully-associative, physical address to DRAM address translation table that can map any physical address to any DRAM address; we refer to these new hardware-managed translation entries as *Compression Translation Entries (CTEs)*, as they are similar to OS page table entries (PTEs). Prior works cache CTEs in the memory controller via a dedicated CTE cache, similar to the TLBs dedicated to caching PTEs.

This new dynamic physical-to-DRAM address translation increases the end-to-end latency of memory accesses, however. This translation takes place serially after the existing virtual-to-physical translation produces a physical address; if that physical address incurs an LLC miss and the LLC miss suffers from a CTE miss in the CTE cache, memory controller has to wait for the missing CTE to arrive from DRAM before knowing where in DRAM to fetch the missing data block.

This paper explores and addresses the problem of high address translation overheads that large and/or irregular workloads suffer under hardware memory compression. We note that just like how these workloads suffer from high PTE miss rates, they also suffer from high CTE miss rates. To make the matter worse, prior works migrate memory content at memory block granularity; this requires much more fine-grained address translation than existing virtual-to-physical translation, which typically operates at 4KB page granularity. In general, the finer the coverage of translations, the less cacheable the translations become, and higher the translation miss rate.

A promising solution to tackle the new address translation

[*] All work done at Virginia Tech.

978-1-6654-6272-3/22/$31.00 ©2022 IEEE
DOI 10.1109/MICRO56248.2022.00073

overhead is to let hardware take on an OS-inspired approach: only save memory from cold (i.e., less recently accessed) pages without saving memory from hot (i.e., recently accessed) pages (e.g., keep the hot pages uncompressed), like OS memory compression. Saving memory only from cold, but not hot, pages can mitigate the block-level translation overhead due to saving memory from hot pages in hardware.

Such an OS-inspired hardware memory compression faces two challenges, however. A) After a compressed cold page becomes hot again, migrating the page to a full 4KB DRAM location still adds another level (albeit page-level, instead of block-level) of translation for future accesses to the newly hot page. B) Only compressing cold pages requires very aggressively compressing cold pages to achieve the same total memory savings as prior works' approach of saving memory from all (both cold and hot) pages; decompressing aggressively compressed pages incurs high latency overhead (e.g., $> 800ns$ in IBM's state-of-the-art ASIC Deflate [11]).

This paper presents Translation-optimized Memory Compression for Capacity (TMCC) to enable high performance hardware memory compression for large and/or irregular workloads. TMCC builds on the OS-inspired approach above, but addresses its two key challenges.

To address Challenge A), we make two observations. First, CTE misses typically occur after PTE misses in TLB because CTEs, especially the page-level CTEs under an OS-inspired approach, have similar translation reach as PTEs. Second, we observe page table blocks (PTBs) are highly compressible because adjacent virtual pages often have identical status bits and the most significant bits in physical page numbers are unused. As such, to hide the latency of CTE misses, TMCC transparently compresses each PTB in hardware to free up space in the PTB to embed the CTEs of the 4KB pages (i.e., either data pages or page table pages) that the PTB points to; this enables each page walk to also prefetch the matching CTE required for fetching from DRAM either the end data or the next PTB.

To address Challenge B), we take IBM's state-of-the-art ASIC Deflate design [11], which was designed for both storage and memory, and specialize it for memory. Specifically, we perform a large design space exploration across many dimensions of hardware configurations available under Deflate and across diverse workloads; the end product is an ASIC Deflate specialized for memory that is 4X as fast as the state-of-the-art Deflate when it comes to memory pages.

The contributions of this paper are as follows:

1) We are the first to tackle the address translation problem faced by large and/or irregular workloads when using hardware memory compression to improve effective memory capacity.
2) We identify CTE cache misses mostly follow TLB misses (e.g., for 89% of the time, on average). As such, we propose embedding CTEs into PTBs to enable accurate prefetch of CTEs during the normal page walks after TLB misses.
3) We are the first to specialize ASIC Deflate for memory. Our ASIC Deflate decompresses 4KB memory pages 4X as fast as the best general-purpose Deflate. We publicly release our HDL at *https://github.com/HEAP-Lab-VT/ASIC-DEFLATE-for-memory*.
4) We compare against Compresso [6], a state-of-the-art prior work on hardware memory compression; our evaluations show that for large and/or irregular workloads, TMCC can either improve performance by 14% without sacrificing effective capacity or provide 2.2x the effective capacity without sacrificing performance.

II. Background

Conventional Address Translation: OS typically maps virtual addresses to physical addresses at 4 KB page granularity. OS maintains a page table for each program to map virtual pages to physical pages. CPUs incorporate a per-core translation lookaside buffer (TLB) to cache recently used page table entries (PTEs). A TLB has a limited size (e.g., 2048 entries). A TLB miss triggers the page walk. The page walk performs a sequence of memory accesses to traverse the page table. Each step in a page walk fetches a 64B block of eight PTEs; we call this block a page table block (PTB).

Hardware memory compression: Many prior works [4], [5], [6], [7], [8], [9], [10], [12], [13] have explored hardware memory compression; memory controller (MC) transparently compresses content on-the-fly with evicting/writing back memory blocks to DRAM and transparently decompresses DRAM content on-the-fly for every LLC miss.

Broadly, prior works on hardware memory compression falls under two broad categories. One body of works compress memory values to increase effective memory bandwidth [13], [14], [15], [16], [17], [18], [19], [20]. Compressing memory blocks reduces the number of memory bus cycles required to transfer data to and from memory. Another body of works use compression to increase effective memory capacity by migrating compressed blocks closer to free up a large contiguous space in DRAM for future use.

Intuitively, increasing effective capacity requires more aggressive data migration than compressing memory to increase effective bandwidth. The former carries out fully-associative data migration in DRAM. In comparison, the latter either keeps memory blocks in place after compression [13], [17], [20] or migrates compressed memory blocks to a neighboring location in DRAM [19].

To migrate data transparently in hardware, prior works on increasing effective capacity borrow two OS memory management features and implement them in hardware.

First, prior works borrow from OS' free list; MC maintains a linked-list-based Free List [4], [6] to track free space in DRAM at a coarse (e.g., 256B [4] or 512B [6]) granularity called *chunks*. When detecting that sufficient slack currently exists within the space taken up by a page, prior works

repack the page's content closer together to free up chunk(s) to push to (i.e., track at the top of) the Free List. When a page becomes less compressible and cannot fit in its currently allocated chunks, prior works pop a chunk from (i.e., stop tracking it in) Free List to allocate the chunk to the page.

Second, prior works borrow from OS page tables; MC maintains a dynamic, page-table-like, fully-associative, physical address to DRAM address translation table that can map any physical address to any DRAM address. We refer to these new hardware-managed translation entries as *Compression Translation Entries (CTEs)*, as they are similar to OS page table entries (PTEs). MC stores the CTEs in DRAM as a linear 1-level table. Each CTE (a.k.a, metadata block [6], [4], [8] in prior works) contains individual fields to track the DRAM address of individual 64B blocks within a group of blocks. This is because compression ratio varies across blocks; as such, after saving memory through repacking, different blocks start at irregular-spaced DRAM addresses, instead of regular-spaced DRAM addresses like current systems without hardware compression. Prior works cache these CTEs in a dedicated CTE cache, similar to TLBs dedicated to caching PTEs.

III. Problem

Large workloads (i.e., ones with large memory footprint) are ubiquitous in today's computing landscape [21]; examples include graph analytics, machine learning, and in-memory databases [22]. However, large workloads suffer from high address translation overhead because their PTEs are too numerous to fit in TLBs. Similarly, the PTEs of workloads with irregular access patterns also cache poorly in TLBs. As such, many works [23], [24], [25], [26], [27], [28] have explored how to improve address translation for large and/or irregular workloads in the context of conventional systems without hardware memory compression.

This paper explores the problem of high address translation overheads that large and/or irregular workloads suffer under hardware memory compression for capacity. We note that just like how they suffer from high PTE miss rates in TLBs, they also suffer from high CTE miss rates under hardware memory compression. Making the matter worse, prior works on hardware memory compression translate from physical to DRAM addresses at memory block granularity, instead of page granularity. It is well-known that the finer the translations, the higher the translation miss rate.

Take for example Compresso [6], the state-of-the-art hardware memory compression for capacity. To perform physical-to-DRAM address translation for a 4KB range of physical addresses, Compresso requires a 64B CTE; each CTE records per-block metadata to translate individual blocks within the 4KB range. Overall, each CTE in Compresso costs 8X as much space as a PTE, which is only 8B. Compresso caches CTEs in a 64KB CTE cache in MC; as such, the CTE cache reaches only $64KB/64B = 1K$ pages.

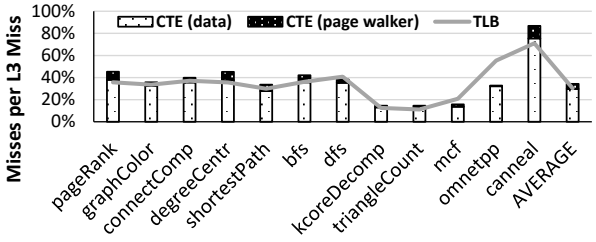

Figure 1: CTE and TLB misses for large and/or irregular workloads, normalized to total L3 misses. Measured in Gem5 by simulating 4 cores with 2K TLB entries, 1MB L2, sharing an 8MB L3 and 64KB CTE cache.

Because TLBs typically have a similar (e.g., 1.5K) number of entries, we expect CTE misses to be similarly frequent as TLB misses.

Figure 1 shows total TLB misses and CTE misses normalized to the number of last-level cache (LLC) *misses*; **we show TLB misses normalized to LLC misses, instead of 1000 instructions, to more closely compare with CTE misses**. Figure 1 includes all workloads used by recent prior works [23], [24] on improving address translation that we know how to run in Gem5 [29]. When evaluating IBM's GraphBIG [30], we use a Facebook-like social media graph dataset (see *datagen-8_5-fb* [31]) and multi-threading. On average across all workloads, CTE miss rate is higher than TLB miss rate (i.e., 34% vs 30%). We were initially surprised by this finding because the CTE misses in Figure 1 only include CTE misses for fulfilling LLC misses. By closer inspection, we find CTEs miss more often because all regular memory requests, including requests for PTBs themselves from the page walker, require accessing CTEs; TLB misses, however, only occur for data (and instruction).

To reduce CTE miss rate, one possible solution is to make the CTE cache bigger. We evaluate CTE hit rate using a 256KB metadata cache. Figure 2 shows an average hit rate of 70.5% for a 256KB metadata cache; this means it still misses $1 - 70.5\% = 29.5\%$ of the time. As such, making CTE cache bigger does not effectively reduce CTE miss rate.

Another possible solution is to use LLC as a victim cache for CTEs evicted from the CTE cache. Figure 2 shows that even when caching in LLC, a high 21% of

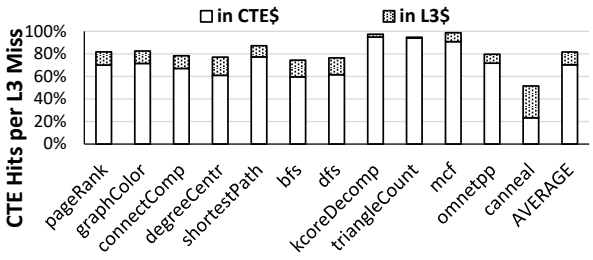

Figure 2: CTE hits normalized to regular LLC misses assuming a 4X CTE cache and LLC as a victim cache for CTEs.

total CTE accesses still go to DRAM. Another problem is that modern server CPUs have a long LLC access time – ~20ns [32], [33] – due to having a distributed network-on-chip architecture. As such, even if a CTE cache miss hits in LLC, the subsequent data or PTB miss that needs the CTE still slows down by 20ns. For example, assuming a DRAM latency of ~35ns, a CTE cache hit can save ~35ns; however, a CTE cache miss that hits in LLC only saves 35ns-20ns=15ns. Making matters worse, a CTE cache miss that also misses in LLC actually increases total memory access latency; when MC realizes that the CTE access misses in LLC 20ns later, it also fetches CTE from DRAM 20ns later compared to not caching CTEs in LLC. Figure 2 shows that CTE accesses that hit or miss in LLC are roughly equal; as such, caching CTE in LLC increases average memory access time. We find in our simulations that caching CTEs in LLC is actually slightly slower than not caching CTEs in LLC. **CTEs are not cached in LLC in the rest of the paper**.

IV. A Promising Solution: Taking an OS-inspired Approach to Hardware

A promising solution to tackle the address translation overhead under hardware memory compression is to let hardware take on yet another OS feature: only save memory from cold (i.e., less recently accessed) pages without saving memory from hot (i.e., recently accessed) pages (e.g., keep the hot pages uncompressed), like OS memory compression. When hardware does not save memory from hot pages, hardware can lay out hot pages' memory blocks regularly either like uncompressed memory or like compressing memory for expanding effective bandwidth (see Section II). For hot pages, which are most critical to performance, doing so helps to avoid the overhead of fine-grained block-level translation.

Specifically, avoiding block-level translation can significantly increase the translation reach of each CTE and, therefore, significantly reduce CTE cache miss rate. Consider for example Compresso; each 64B CTE cacheline only translates for one 4KB physical page due to storing a translation for every block in the page. After switching over to page-level translation like OS, each 64B CTE cacheline can translate for eight pages, like how a PTB translates for eight virtual pages. For the workloads in Figure 1, we find switching from block-level translation to page-level translation eliminates 40% of CTE misses, on average, while simply quadrupling the size of the CTE cache only reduces CTE miss rate by 13% (from 34% down to 29.5%, see Section III). Page-level translation is so effective due to increasing effective CTE cache size by 8X and better exploiting spatial locality (i.e., fetching from DRAM a CTE block that translates at page level equates to fetching eight adjacent CTE blocks that translate at block level).

A. Background on OS Compression

OSes also compress memory [34], [35], [36], [37], [38]. OS does so in many data centers (e.g., Google Cloud [3], IBM Cloud [39], Facebook [1]).

In the eyes of an architect, OS memory compression manages memory as a 2-level exclusive hierarchy: (i) Memory Level 1 (ML1) stores everything uncompressed, (ii) Memory Level 2 (ML2) stores everything compressed. Accesses to ML1 are overhead-free (e.g., incurs no translation overhead). Accesses to a compressed virtual page in ML2 incurs a page fault to wake up OS to pop a free physical page from ML1's free list and migrate the virtual page to the page.

Because ML1 provides no gain in effective capacity, providing significant gain in overall effective capacity requires ML2 to aggressively save memory from the pages ML2 is storing. As such, ML2 uses aggressive page-granularity compression algorithms, such as Deflate [11]. ML2 also keeps many free lists, each tracking sub-physical pages of a different size, to store any compressed virtual page in a practically ideal matching sub-physical page [40], [41].

ML2 gracefully grows and shrinks relative to ML1 with increasing and decreasing memory usage. When everything can fit in memory uncompressed, ML2 shrinks to zero bytes in physical size so ML1 can have every physical page. Specifically, when ML2's free list(s) get large (e.g., due to reducing memory usage), ML2 donates free physical pages from its free list(s) to ML1. OS also grows ML1 free list, when it gets small, by migrating cold virtual pages to ML2. Migrating a virtual page to ML2 shrinks one of ML2's free lists. If a ML2 free list gets empty, ML1 gives cold victim physical pages to ML2 (i.e., track them in ML2 instead of ML1), so that ML2 can compress the virtual pages currently in the victim pages to free space in the victims to grow ML2's free list(s).

B. Taking the OS Approach to Hardware

Hardware memory compression can also be enhanced to manage DRAM as ML1 and ML2 like OS memory compression, with simple adaptations.

One adaption is to simplify CTEs: instead of finely tracking individual memory blocks, track a single 4KB page worth of content collectively at coarse granularity, just like a PTE. Specifically, each CTE now only records the starting DRAM address of an entire page, without recording any individualized tracking for every block in the page.

Another adaptation is to extend prior works' Free Lists to ML1 and ML2. Figure 3a shows a Free List in prior work [6], [4]. Making it work for ML1 involves increasing chunk size to 4KB (see Figure 3b). ML2 requires multiple Free Lists, each tracking free equally-sized sub-chunks. The purpose of each sub-chunk is to store an entire compressed page. Equally-sized sub-chunks can be created fragmentation-free by evenly dividing a group of M interlinked chunks, which we call a super-chunk, into N sub-

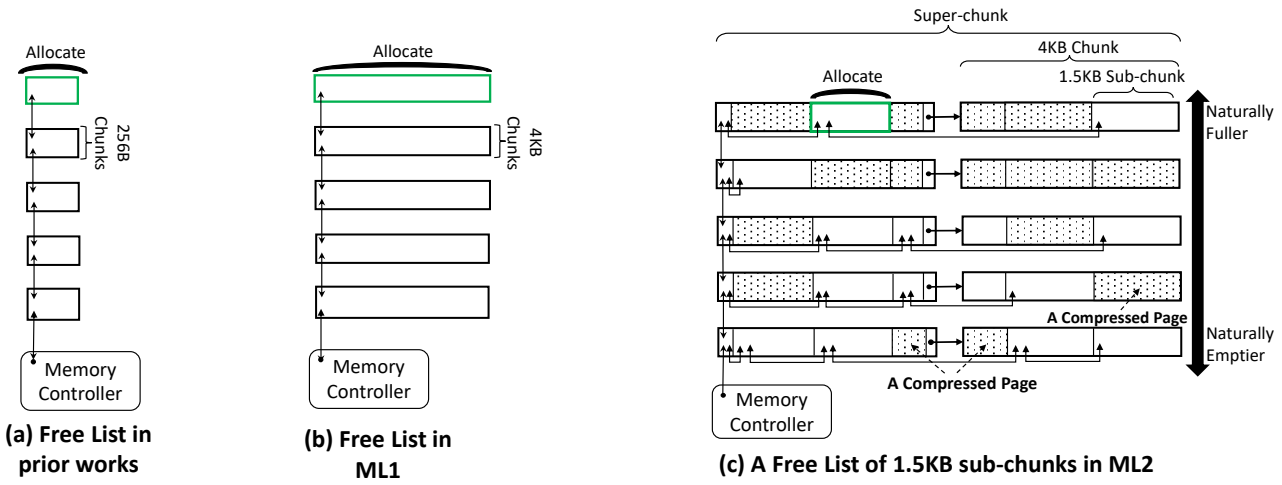

Figure 3: (a) A simplified design of the Hardware Free List in prior work [4]; it stores a pair of pointers in free chunks 'for free' to implement a doubly linked list to track free 256B chunks. (b) ML1 Free List, which tracks free 4KB chunks. (c) An ML2 Free List that tracks 1.5KB free sub-chunks; it tracks all super-chunks containing at least one free 1.5KB sub-chunk.

chunks, where $N > M$ and N,M are chosen to minimize $(4KB \cdot M) mod N$. Figure 3c shows an example ML2 Free List for tracking 1.5KB sub-chunks. When all sub-chunks in a super-chunk becomes free (e.g., the compressed pages they store have all migrated to ML1 over time), the chunks in the super-chunk are returned to ML1's Free List. The super-chunks towards the bottom of an ML2 Free list naturally tend to be emptier than super-chunks towards the top. This is because A) ML2 always allocates sub-chunks from the top of ML2 Free List(s) to handle migration to ML2 and B) ML2 tracks at the top of ML2 Free List(s) super-chunks that transition from having no free sub-chunk to having one free sub-chunk (e.g., after a page migrates to ML1).

Beside adapting prior works' CTEs and Free Lists, a new necessary component is a new doubly linked list to track the recency of pages stored in ML1; we call them the *Recency List*. Besides the list pointers, each Recency List element tracks a page in ML1 by recording the page's PPN (i.e., physical page number). The head and tail of Recency List track the hottest and coldest pages in ML1, respectively. ML1 updates the Recency List for a small (i.e., 1% of) fraction of randomly chosen accesses to ML1; when updating the Recency List for an access to ML1, ML1 moves the accessed page's list element to the hot end of the list. ML1 evicts victims from the cold end of Recency List. In the uncommon case that the victim turns out to be incompressible, ML1 retains the page in ML1; ML1 simply removes the page from the Recency List to avoid uselessly compressing it again. As subsequent writebacks may increase a page's compression ratio, ML1 adds an incompressible page back to the Recency List at 1% probability after a writeback to an incompressible page. ML1 can record whether a page is incompressible via an 'isIncompressible' bit in each CTE.

While ML1 is uncompressed in OS, in hardware, ML1 can also be compressed to increase effective memory bandwidth. One of the many prior memory compression techniques for improving bandwidth (e.g., TMC [19]) can be readily applied to ML1.

C. Challenges of the OS-inspired Approach

Such an OS-inspired hardware compression faces two challenges, however. A) After compressed cold pages are accessed again, migrating them from ML2 to ML1 still adds another level (albeit page-level, instead of block-level) of translation for future accesses to all pages in ML1. B) Only compressing cold pages require very aggressively compressing cold pages to achieve high overall memory saving; decompressing aggressively compressed pages for every access to ML2 is slow. We describe these challenges in detail below.

Performance Challenge under ML1: In OS memory compression, accesses to ML1 incur no overhead. When OS migrates a virtual page from ML2 to a free physical page in ML1, OS directly records the new physical page's PPN in the virtual page's PTE. As such, future accesses to the virtual page in ML1 requires the same amount of translation as a system that turns off memory compression.

However, when hardware memory compression migrates a page from ML2 to ML1 after a program accesses the page, hardware cannot directly update the program's PTE because PTEs are OS-managed. Raising an interrupt to ask OS to update the PTE for hardware would defeat the main purpose of hardware memory compression – avoid the costly page faults under OS memory compression. Instead, hardware tracks the page's new DRAM location through a new layer of translation (i.e., through CTEs). As such, hardware has to use the PPN recorded in the page's PTE to indirectly access a

CTE to obtain the data's DRAM address; this requires a new level of serial page-level translation (see Figure 4b), unlike ML1 accesses under OS compression (see Figure 4a). For the workloads in Figure 1, this added page-level translation still causes 20% of LLC misses to suffer from CTE misses.

As such, how to address the latency overhead due to the page-level translation for ML1 in hardware is a challenge.

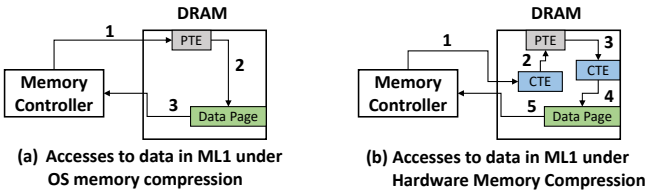

Figure 4: Sequence of memory accesses to ML1.

Performance Challenge under ML2: The key latency bottleneck for ML2 is decompressing aggressively-compressed pages when they are accessed in ML2. OS typically use aggressive page-granularity compression, such as Deflate, to save memory in ML2. For decades, Deflate has been used across many application scenarios (e.g., file systems, network, memory). Due to Deflate's high and robust compression ratio, IBM integrates ASIC Deflate into Power9 and z15 CPUs [11]. This state-of-the-art ASIC Deflate achieves a peak throughput of 15 GB/s for large streams of data [11]. However, it has a setup time (T_0 [11]) of 650-780ns for each new independent input (e.g., a new independent page). This delay can be crippling for small inputs, such as 4KB memory pages. This long delay also limits the bandwidth for reading and writing 4KB compressed pages to only 4 GB/s and 2 GB/s per module, respectively. This amounts to a mere 16% and 8% bandwidth of a DDR4-3200 memory channel. While long latency and low bandwidth is okay for ML2 accesses under OS compression, where overall performance is limited by software overheads, they are inadequate for hardware memory compression.

As such, how to address the high decompression overhead for ML2 accesses in hardware, without significantly sacrificing ML2's compression ratio, is a challenge.

V. Translation-optimized Memory Compression

We propose Translation-optimized Memory Compression for Capacity (TMCC) to enable fast hardware memory compression for large and/or irregular workloads. TMCC builds on the OS-inspired approach in Section IV, but effectively addresses the latency overheads for both ML1 and ML2.

A. Addressing the Translation Overhead Under ML1

To effectively address the problem of long-latency serial translation for accesses to ML1 during CTE misses, TMCC parallelizes the data access with the corresponding CTE access; instead of the conventional approach of waiting for the missing CTE to arrive from DRAM and then use it to calculate the DRAM address to serve the L3 miss, TMCC carries out both DRAM accesses in parallel. This effectively hides the CTE miss latency from the total DRAM access latency for serving an L3 miss request.

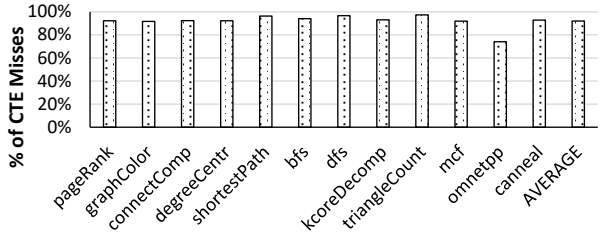

Figure 5: Fraction of total CTE misses that are due to walker and data/instruction accesses immediately after a TLB miss. Each 8B CTE translates for a 4KB page.

To parallelize DRAM accesses for CTE and for the actual L3 miss, we make two enabling observations.

1) Observations: First, we observe CTE misses typically occur immediately after PTE misses. This is also true for the OS-inspired approach to hardware memory compression in Section IV, where each 8B CTE translates for a 4KB page. Similarly, each level N+1 PTE (e.g., L2 PTE) tracks 4KB worth of level N PTEs, while each L1 PTE tracks 4KB of data (or instructions). Due to CTEs and PTEs providing the same translation reach, accesses that cause PTE misses in TLB will likely also cause CTE misses in CTE cache. Figure 5 shows that 89% of all CTE misses for LLC miss requests are due to LLC misses related to a TLB miss (i.e., page walker misses in LLC and/or the subsequent data/instruction miss in LLC).

Second, we observe each page table block or PTB (i.e., a 64B worth of PTEs) is highly compressible because, intuitively, adjacent virtual address ranges often have identical status bits; moreover, many bits in PPN are also identical.

For example, each 8B PTE in x86 consists of 24 status bits that record various permissions and a 40-bit PPN [42]. Figure 6 shows the fraction of L1 page table blocks (i.e., storing L1 PTEs) and L2 page table blocks (i.e., storing L2 PTEs) that have identical status bits across all PTEs within the same PTB; it is 99.94% and 99.3%, on average, for L1 and L2 page table blocks. Meanwhile, many of the most significant bits in the PPN are identical, depending on the actual amount of DRAM currently installed in the system.

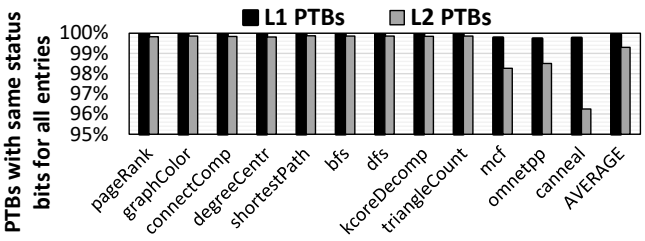

Figure 6: PTBs where status bits are same across all entries.

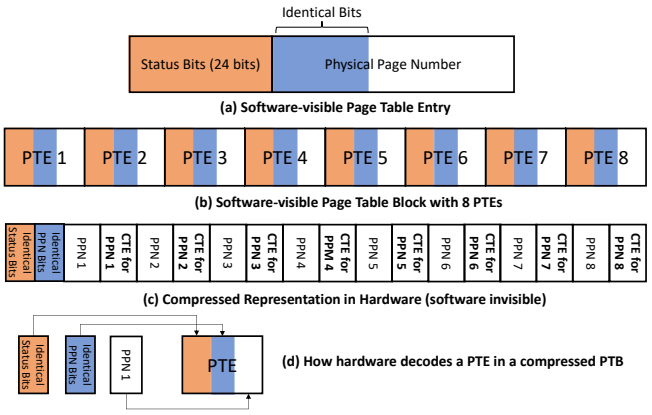

Figure 7: Comparing and contrasting (a) conventional PTE and (b) conventional PTB encoding with (c) our hardware-compressed PTB encoding. 'CTE for PPN1' translates 'PPN1' to a DRAM address. TMCC compresses a PTB only if the status bits across 8 PTEs are identical. To compress a PTB, TMCC records the status bits only once and truncates the leading identical bits in PPNs according to how much memory is installed.

For example, in a machine with 4TB of OS physical pages, the most significant 10 bits of the PPN are almost always identical (e.g., all zeroes or reused as identical extended permission bits by Intel MKTME [43]).

2) Key Idea: Based on our observations, we propose transparently compressing each PTB in hardware to free up space in the PTB to embed the CTEs of the 4KB pages (i.e., either data pages or page table pages) that the PTB points to; this enables each page walk access to also prefetch the matching CTE required either for the next page walk access (i.e., to the next PTB) or for the actual data (or instruction) access after the walk.

Figure 7c shows a compressed PTB. For each PTE in the PTB, TMCC opportunistically stores in the compressed PTB a CTE responsible for translating the PPN that the PTE contains into a DRAM address. As such, as a page walker fetches a PTB, the CTE for the next access (i.e., either the next page walker access or the end data access) becomes available at the same time as the PPN for the next access. Directly having in the PTB the CTE needed for the next access eliminates the need to serially fetch and wait for CTE to arrive from DRAM before knowing the next DRAM address to access.

A practical challenge is that after migrating a page (e.g., from ML1 to ML2 after the page becomes cold), the corresponding CTE embedded in the page's PTB should be updated. However, hardware has no easy way to use the PPN of the migrating page to find/access the page's PTB(s). TMCC addresses this challenge by lazily updating the CTE in the PTB later around when the PTB is naturally accessed by the page walker, instead of updating it at the time of migrating the page. However, this means that for the first page walker access to the PTB after migrating one of the pages that the PTB points to, the corresponding CTE is

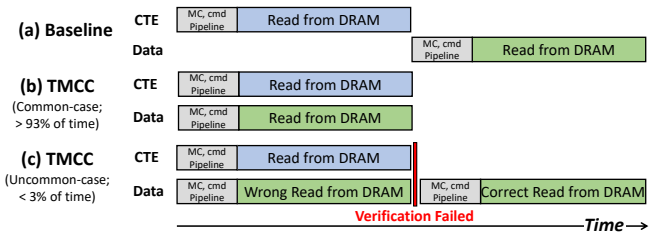

Figure 8: MC's timeline for serving LLC misses that also suffer from CTE cache miss. In the figure, the sum of TMCC common-case and TMCC uncommon-case do not add up to 100% because there is another uncommon-case scenario that the PTB does not currently embed any CTE (as opposed to embedding the right or wrong CTE).

out-of-date. To ensure correctness, TMCC also accesses the correct CTE in DRAM (or in CTE cache) in parallel to verify the correctness of the DRAM access. Figure 8 compares and contrasts how TMCC serves an LLC miss that also misses in CTE cache with the baseline approach. Figure 9 provides an architectural overview of TMCC.

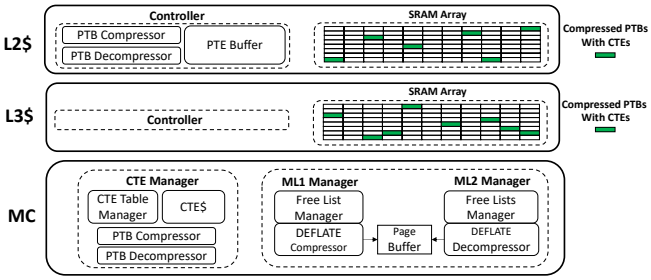

Figure 9: Overview of architectural changes under TMCC.

3) Detailed Actions Following a TLB Miss: After a TLB miss for instruction X, if the page walker accesses L2, L2 buffers into a temporary buffer every CTE within the accessed PTB. We call this temporary buffer the *CTE Buffer*. CTE Buffer inserts each CTE as a new key value pair. The key is the PPN that the PTE records; the value consists of the embedded CTE for the PPN and the physical address of the PTB holding the PTE (See Figure 10).

When L2 receives another page walker access or the end data (or instruction) access for instruction X (L2 need not know the access is actually for instruction X), L2 extracts the PPN from the received request to lookup the CTE Buffer to obtain the CTE for MC to translate the PPN. If the request misses in L2, L2 forwards the request to LLC, as usual, and piggybacks the CTE in the request. If LLC also misses, LLC

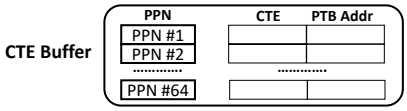

Figure 10: CTE Buffer. PPN is the key for lookup.

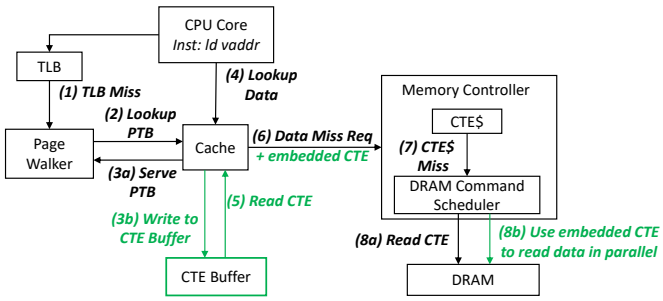

Figure 11: How MC gets the embedded CTE to access data in DRAM in parallel with accessing the actual CTE in DRAM. 'a' and 'b' actions under the same step are in parallel.

forwards the request and the piggybacked CTE to MC.

When receiving a request from LLC, MC first looks up CTE cache (i.e., by extracting the PPN from the request to access CTE cache). If the request hits in CTE cache, MC uses the CTE from the cache to translate the request's physical address to DRAM address to access DRAM.

If the request misses in CTE cache, two cases can occur.

The uncommon case is that the request has no embedded CTE; as such, MC takes the same actions as prior hardware memory compression designs – access CTE in DRAM and then serially access DRAM to service the LLC miss.

The common case is that the request has an embedded CTE; MC uses the CTE to speculatively translate the LLC's request to DRAM address to access DRAM in parallel with accessing the actual CTE in DRAM. Figure 11 shows this common case. When both DRAM accesses complete, MC checks whether the correct CTE from DRAM matches the embedded CTE. In the common case that they match, MC can directly respond to LLC. In the uncommon case that they mismatch, MC uses the correct CTE to translate the LLC request and re-access DRAM (see Figure 8c).

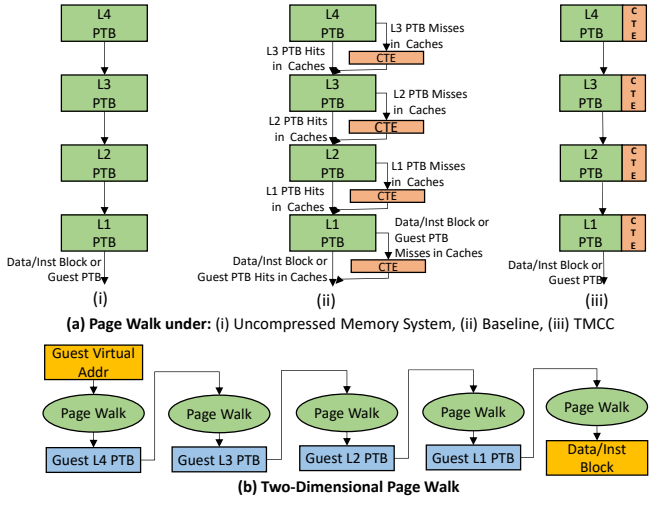

Figure 12: (a) Contrasting page walk under an uncompressed memory system, baseline, and TMCC. (b) 2D page walk for a virtual machine.

Also in both cases, MC piggybacks the correct CTE in the response back to LLC and L2. When receiving a response, L2 extracts the PPN from the response to look up CTE Buffer. On CTE Buffer hit, if the CTE Buffer entry has a mismatching CTE or has no CTE, L2 stores the correct CTE into the entry and uses the PTB physical address that the element records to fetch and update the PTB with the incoming CTE.

Embedding CTEs in PTBs not only reduces the latency to fetch data/instruction from memory at the end of a page walk, but can also reduce the latency to fetch PTB blocks from memory. In other words, embedding CTEs in PTBs can benefit the entire page walk (see (iii) in Figure 12a).

Embedding CTEs in PTBs also benefits 2D page walks for VMs. Each 2D page walk (see Figure 12b) requires multiple regular page walks that only use host PTBs, just like a page walk for a native application. As such, TMCC carries out the same actions during each page walk within a 2D page walk as a regular page walk.

4) Details on Tracking Compressed PTBs: To track which blocks in DRAM are encoded via the compressed PTB encoding (see Figure 7c), each CTE contains a bit vector of 32 bits; each bit tracks whether two adjacent blocks in a page are both currently using the compressed PTB encoding. When one block in a pair of adjacent blocks undergoes an encoding change (i.e., from uncompressed to compressed or vice versa), the MC enacts the same encoding change for the other block when it writes to memory the original block with changed encoding. Figure 13 shows the internal layout of a CTE.

To clarify, compressing memory blocks using our PTB encoding only affects the encoding of individual memory blocks in a page in ML1, *without affecting their DRAM location*; the 32-bit vector only serves to record the format of the blocks in each page in ML1, and not to migrate the blocks. **TMCC does not perform any block-level translation, even for compressed PTBs.**

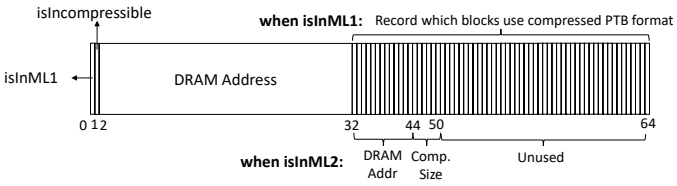

Figure 13: Internal layout of a CTE under TMCC

After fetching from DRAM a memory block encoded in compressed PTB format, MC replies the block back to LLC in compressed format. Under TMCC, the only compressed content on-chip are PTBs (i.e., cachelines accessed by the page walker). Every L2 and L3 cacheline has a new data bit to record whether the cacheline is compressed. Conversely, when L3 writes back a dirty cacheline to MC, MC checks the new data bit to set the CTE's bit vector accordingly.

Apart from MC, L2 also contains PTB decompressor and compressor. When an L1 cache or a page walker requests a block from L2 and the L2 copy is compressed (i.e., the new data bit - see paragraph above - in the copy is set), L2 replies with a decompressed copy; because all software-initiated memory accesses pass through L1 cache, always replying decompressed copy to L1 ensures CTEs embedded in PTBs are invisible to software, such as OS. When L2 receives from L1 a dirty eviction, L2 checks whether the dirty block's value is compressible under the compressed PTB format; if so and if the L2 copy is currently compressed, L2 copies into the incoming dirty block any embedded CTEs held in the stale L2 copy (note that L2 is inclusive of L1) to seek to preserve the embedded CTEs when OS modifies a PTB (e.g., to remap a virtual page elsewhere). Lastly, when receiving an uncompressed block from L3, if the requester is the page walker, L2 compresses the block before caching it; this is how TMCC initially compresses PTBs in a PTB page when OS creates the PTB page or migrates the PTB page to a new address.

5) When TMCC Cannot Embed in PTBs: Compression can only free up limited space in each PTB. As such, TMCC only embeds into PTBs truncated CTEs, with only enough bits to identify a 4KB DRAM address range within an MC's DRAM. Assuming each MC manages up to 1TB of DRAM, each truncated CTE is only $log_2(1TB/4KB) = 28$ bits. Assuming the TMCC enables up to 4X physical pages in the OS, TMCC can embed 8 CTEs in the PTB under this configuration (i.e., for all 8 PTEs).

In bigger machines with bigger PPNs, however, each compressed PTB cannot fit eight CTEs. We calculate that for systems with 4TB and 16TB of DRAM, each compressed PTB can only fit seven and six CTEs respectively.

6) Overheads: Decompressing PTBs take ≤ 1 cycle; it only needs wiring to concatenate plaintext (see Figure 7). Each CTE Buffer has 64 entries; it requires a total of ~1KB.

By always migrating memory content at page granularity, instead of block granularity, TMCC reduces the size of each page's CTE from 64B to 8B. Assuming an OS that boots up with 4X OS physical memory as DRAM size, total size of all CTEs in DRAM reduces from 6.25% to only 0.78%.

By taking on an OS-inspired approach, TMCC requires a Recency List in ML1 (see Section IV-B). Unlike Free Lists, which can store linked list pointers for free in free chunks and sub-chunks, storing ML1's Recency List's pointers takes up memory. Recency List uses 0.4% of DRAM. An OS-inspired approach also requires tracking incompressible pages in ML1 to prevent ML1 from repeatedly trying to evict the same incompressible pages to ML2.

B. Addressing Long Decompression Latency for ML2

While Deflate is effective and robust, it is slow. The state-of-the-art ASIC Deflate from IBM [11] takes 1100ns to decompress a 4KB page. Deflate is slow because it serially combines two aggressive algorithms - LZ and Huffman.

We note the state-of-the-art ASIC Deflate from IBM is a general-purpose design targeting both storage and memory. Intuitively, there can be much room for improvement when specializing Deflate just for memory. In addition, while a general-purpose ASIC Deflate designed also for storage has to strictly abide by the Deflate standard to provide compatibility across systems, an ASIC Deflate specialized for memory does not; memory values are locally produced and consumed. Unshackling from the constraints of the standard allows more room for specialization and optimization.

To specialize ASIC Deflate for memory, we first implement Deflate in HDL to identify performance bottlenecks. We then perform large design space exploration in HDL to address the bottlenecks. To explore the large design space, we make our HDL highly parameterizable by using the Chisel design language; the tunable parameters include how many characters to encode and decode per cycle, LZ sliding window size (i.e., CAM size), the number of characters in the Huffman tree, the maximum depth of the tree, sample size for frequency counting, etc. We also use a wide range of diverse workloads spanning three C/C++ benchmark suites and three Java benchmark suites to evaluate the impact on compression ratio due to the hardware design choices.

Our ASIC Deflate specialized for memory decompresses each 4KB page in $\sim 1/4^{th}$ the time as the state-of-the-art ASIC Deflate from IBM [11], while providing similar compression ratio. We test our ASIC Deflate via RTL simulations on 50 million 4KB memory pages. We publicly release the HDL for our memory-specialized ASIC Deflate at https://github.com/HEAP-lab-VT/ASIC-DEFLATE-for-memory.

1) Local Optimizations to Huffman: We implement Huffman in Chisel from the ground up. In the process, we identify the tree construction for compression and tree reconstruction for decompression as the key performance bottleneck for Huffman; this is especially true when using the canonical Huffman tree format, which compresses the tree itself. Making matters worse, the Deflate standard (RFC 1951) specifies building two canonical Huffman trees from LZ output (i.e., one for literals and one for LZ match offsets), performing runlength encoding on the two trees, and compressing the two with a third and final Huffman tree. We also confirm through the IBM authors that the high setup time of IBM's ASIC Deflate is primarily due to building and restoring the Huffman trees.

To avoid this high latency, our solution is two-fold: use a reduced Huffman tree and store it uncompressed.

Our reduced Huffman tree reduces the latency and area required to build and traverse the tree by only compressing the most common input characters and leaving the remaining input characters uncompressed. We find that for non-zero memory pages, the tree can be reduced to just 16 codes instead of the usual 286 as specified in RFC 1951 at the

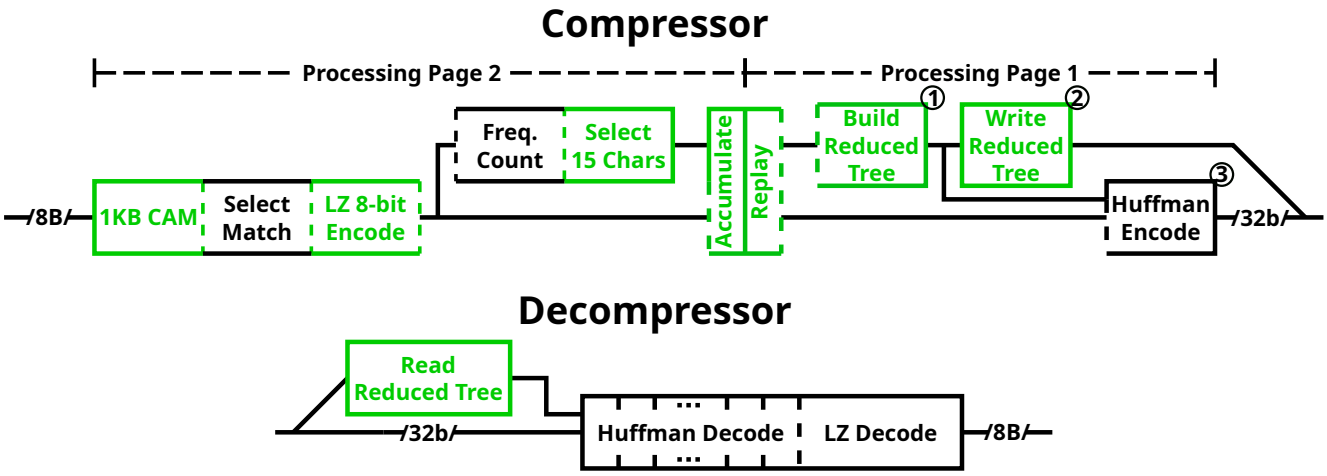

Figure 14: High-level architecture of our Deflate compressor and decompressor. Modules highlighted in green are substantially modified or new compared to IBM's design. Each vertical dashed line separates two stages that are pipelined w.r.t one another; each vertical solid line separates two modules that run serially one after the other (i.e., the earlier module generates and buffers all of its outputs before passing them to the next module).

cost of only sacrificing 1% compression ratio, on average. 15 out of the 16 codes are for the 15 hottest characters (i.e., byte-sized values) in each 4KB page going into the Huffman compressor; the one remaining code in the reduced tree is for an escape character to encode the other 200+ characters missing in the tree. Our Huffman encodes each character missing in the tree as the escape code (i.e., the Huffman code for our escape character) plus the missing character.

We further reduce decompression latency by storing the Huffman tree uncompressed. Due to having only 16 codes, our reduced tree is much smaller than the tree under the standard Deflate; this eliminates the need to aggressively compress the tree like the standard Deflate. As such, our compressor outputs the tree in a plain format, instead of the canonical Huffman tree format or any other compressed format, so that when a compressed page is accessed later, the Huffman decompressor can directly use the tree without having to first slowly decompress the tree.

However, prefixing each character not in the reduced tree with an escape code can sometimes make Huffman output bigger than the input (e.g., when most bytes in an input page are characters missing in the tree). This problem can be addressed by dynamically skipping our Huffman for pages that would become bigger after going through our Huffman. Empirically, we find that dynamically skipping our Huffman can improve compression ratio by 5%, on geometric mean. We are implementing this dynamic feature in hardware at the time of this writing.

2) Local Optimizations to LZ: One issue with specializing Deflate ASIC for memory is that the ASIC cannot be used for anything else (e.g., storage, network), unlike a general-purpose Deflate ASIC. As such, the specialized ASIC should use as little area as possible. We find that LZ takes up most of the area in our ASIC Deflate. When we implement IBM's Deflate LZ and reduces LZ's CAM size from 32KB to 4KB to match memory page size, LZ still takes up 0.24 mm^2 for the compressor and 0.09 mm^2 for the decompressor under the 7nm technology node.

We identify that the CAM for performing sliding-window pattern match as the main contributor to this area. As such, we explore the area and compression ratio of the LZ module under different CAM sizes ranging from 256B to 4KB. We find using a 1KB CAM reduces the LZ compressor to 0.060 mm^2 and the LZ decompressor to 0.022 mm^2, while reducing the compression ratio of non-zero memory pages by only 1.6%. However, smaller (e.g., 512B, 256B) CAMs degrade the compression ratio much more severely.

RFC 1951 specifies a 286-character alphabet for LZ outputs. Since 286 is not a power of two, such an alphabet is not space-efficient; as a result, LZ outputs 9-bit characters for 8-bit character inputs. This poses no problem for standard Deflate because this inefficient alphabet is only used internally; LZ outputs are re-encoded using full Huffman trees and, therefore, do not appear at the final Deflate output and do not compromise the final compression ratio. Due to using a reduced Huffman tree, however, our Deflate can directly output LZ's outputs (e.g., after prefixing them with our escape code). As such, our LZ outputs use a space-efficient $2^8 = 256$-symbol alphabet, like how LZ is used today when it is standalone (i.e., outside of Deflate).

3) Across-Deflate Optimization: Huffman must count the frequencies of the characters in an LZ-compressed page to generate a Huffman tree before using the tree to compress the individual characters in the LZ-compressed page. Precisely counting the frequencies of the characters in an LZ-compressed page requires analyzing the entire page; as such, Huffman compression can only begin after LZ has compressed the entire page, instead of working concurrently with LZ. Having only LZ or Huffman busy, but not both at the same time, can significantly hurt throughput.

To increase throughput, IBM's design resorts to approximate frequency counting (a.k.a 1.1 Pass in [11]) by only analyzing a small portion of LZ output (i.e., a 32KB segment) at the start of a much bigger LZ output (i.e., a 256KB output). This allows Huffman to operate mostly concurrently with LZ, except for just when LZ is outputting the first 32KB out of the 256KB. Meanwhile, the 32KB of LZ output is still big enough to accurately represent the frequency distributions of the overall 256KB of LZ output.

To maximize throughput without reducing compression ratio, our Deflate operates both LZ and Huffman concurrently by using them to process two independent memory pages (see "Page 1" and "Page 2" in Figure 14). This requires adding a buffer (see "Accumulate" in Figure 14) to buffer the entire LZ output; in comparison, IBM's 1.1 Pass only buffers a small fraction of LZ output (e.g., the first 32KB of the 256KB LZ output). However, as memory pages are smaller than files, our buffering overhead is also small. We note that 1.1 Pass is still better along some dimensions (e.g., area); as such, the HDL we release also supports it as a tunable parameter. But we disable it by default as it significantly reduces compression ratio for 4KB pages.

4) Additional Details per Module: This section presents more details for each module in our ASIC Deflate, in the order that they appear in Figure 14.

The first three pipeline stages in the compressor perform LZ compression. The first stage, *1KB CAM*, identifies matches between the most recent 1KB of history and the input characters to LZ in the current cycle. This matching relies on a sliding-window CAM based on IBM's near-history CAM [11]. The match result passes on to the *Select Match* stage; to simplify hardware design, our *Select Match* uses a greedy algorithm to select matches to encode, instead of the "lazy matching" described in RFC 1951. The third stage – LZ 8-bit Encode – encodes the matches and literals using an alphabet with 256 characters.

In our current design, the three stages above can sometimes stall due to pipeline hazards, depending on the length of the matched sequences. As a result, our Deflate only takes in 8 characters/bytes per cycle, just like the IBM design. Taking in more characters per cycle worsens the pipeline hazards and yields diminishing return in performance.

Huffman compression starts with *Frequency Count*; this pipeline stage reads LZ output to calculate the frequency of each 8-bit character in an LZ-compressed page. The next pipeline stage - *Select 15 Characters* - identifies the 15 hottest characters across the entire LZ-compressed page.

Accumulate and *Replay* work together to enable LZ and Huffman to work concurrently on separate pages. *Accumulate* buffers the output of *Select 15 Characters* and *LZ 8-bit Encode* and waits for the Huffman modules after *Accumulate* to finish processing their current page; then, *Accumulate* logically transfers its content to the *Replay* module to replay the buffered values to the later Huffman modules.

Build Reduced Tree then builds a Huffman tree with 16 leaves in the usual way – by repeatedly combining the two nodes with the lowest frequency. To limit the depth of the tree, when a pair of sibling nodes would exceed a tunable depth threshold, *Build Reduced Tree* discards the less-frequent sibling and promotes the other to keep the tree depth below the threshold. *Build Reduced Tree* never discards the escape code. Generating the tree takes up to 32 cycles. *Write Reduced Tree* then takes up to 16 cycles to write the tree to output using an uncompressed format (see Section V-B1). *Huffman Encode* then compresses the LZ-compressed bytes and outputs the Huffman codes at up to 32-bits per cycle.

Our decompressor begins with *Read Reduced Tree*, which takes 16 cycles to read in the Huffman tree and sets up the registers in Huffman Decompress; this is a significant improvement over IBM's design, which takes $> 500ns$ to reconstruct the tree. Next, *Huffman Decompress* decodes up to 8 input codes or 32 input bits per cycle, whichever is smaller, via a multi-stage pipelined decoder based on IBM's design [11] [44]. The last pipeline stage, *LZ Decompress*, outputs up to 8B of plaintext per cycle.

5) ASIC Deflate Performance: We synthesize our memory-specialized ASIC Deflate on a 7nm ASAP technology node [45] at 0.7V using Synopsys Design Compiler [46]; our Deflate runs at 2.5 GHz with a total area of 0.13 mm^2 (see Table I). We use Verilator [47], an industry-standard high-speed RTL simulator, to measure the full-page latency, half-page latency, and throughput; Table II shows the results. The total throughput of one Deflate module (both compressor and decompressor) is 32.0 GB/s; this exceeds the channel bandwidth of DDR4-3200 (i.e., 25.6 GB/s).

Module	Area	Power
LZ Decompressor	0.022 mm^2	100 mW
LZ Compressor	0.060 mm^2	160 mW
Huffman Decompressor	0.014 mm^2	27 mW
Huffman Compressor	0.034 mm^2	160 mW
Complete Unit	0.13 mm^2	447 mW

Table I: Synthesis results for our ASIC Deflate.

Module	Latency	½-page Latency	Throughput
Our Decompressor	277 ns	140 ns	14.8 GB/s
Our Compressor	662 ns	N/A	17.2 GB/s
IBM Decompressor	1100 ns	878 ns	3.7 GB/s
IBM Compressor	1050 ns	N/A	3.9 GB/s

Table II: Deflate performance for 4KB memory pages.

To compare against IBM's design, we use the formula in [11] to analytically calculate the performance of IBM's design. For 4KB memory pages, our memory-specialized ASIC Deflate outperforms IBM's Deflate in every performance metric by several times. Notably, our half-page decompression latency – the average time to decompress a needed block in a page to satisfy an L3 miss – is 6X as fast.

To measure the compression ratio of our design, we examine programs with $> 200MB$ memory footprint from

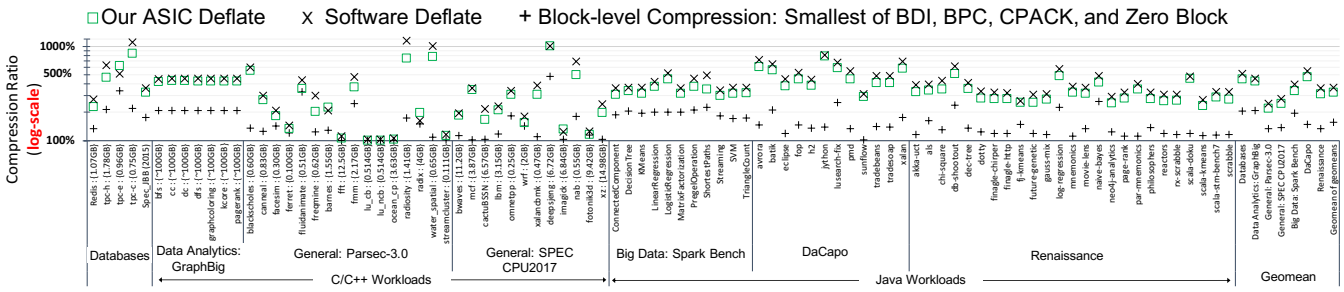

Figure 15: Compression ratio under an aggressive block-level compression, our ASIC Deflate, and software Deflate (gzip).

three C/C++ benchmark suites [30], [48], [49] and three Java benchmark suites [50], [51], [52]. For each program, we take 10 memory dumps equally spaced across its lifetime and deleted all all-zero pages from each dump; note that all-zero pages in a memory dump are typically untouched or deduplicable virtual pages. We calculate the compression ratio of a workload as the maximum size across its 10 uncompressed dumps divided by the maximum compressed size across its 10 dumps.

Figure 15 shows our measurements. Across all benchmarks, our ASIC Deflate specialized for memory achieves a geomean of 3.4x compression, which is only 12% lower than GZIP. Note IBM's ASIC Deflate also has a 11% lower compression ratio than GZIP [11]. Dynamic skipping of Huffman (see Section V-B1) can increase the compression ratio to 3.6x, which is within 7% of GZIP.

As another reference for comparison, Figure 15 also shows the compression ratio of the memory dumps under block-level compression. We model a 64B-block-level compression that chooses the smallest output between BPC [12], BDI [53], Cpack [54], and Zero Block; across the same benchmark suites, the geomean compression is only 1.51x.

VI. Simulation Methodology

We evaluate TMCC's performance under cycle-accurate simulators. We use Gem5 [29] and Ramulator [55] to simulate CPU and DRAM, respectively. Table III lists the simulated system's parameters. We evaluate workloads used by recent prior works [23], [24] on virtual address translation in conventional systems. We simulate all such workloads that we could run in Gem5. They span IBM's GraphBIG [30], SPEC CPU2017 [56], and PARSEC 3.0 [48]. All workloads are multi-threaded except *mcf* and *omnetpp* which are single-threaded. For single-threaded workloads, we evaluate four instances of the same benchmark. Figure 16 shows the memory intensiveness of the benchmarks.

We fast forward each benchmark deep into the region of interest using Gem5's KVM mode in native execution speed; each benchmark reaches at least 95% of its maximum memory footprint. Then, we fetch all of the benchmark's memory values to place, compress, and pack them into

CPU	4 cores, 2.8GHz, 4-wide OoO, 2048 TLB entries
Caches	Size: 64KB L1d$+L1i$, 256KB L2$ inclusive, 8MB L3$ exclusive, 1 KB page walk cache per core (similar to [23]) L1$: 3 cycles, L2$: +11 cycles, L3$: +50 cycles
Prefetchers	Next-line with automatic turn-off: L1$, L2$ Stride: L1$ (degree 2), L2$ (degree 4)
DRAM	DDR4-3200, 1-channel, 8-ranks, MC to Cache NoC latency: 18ns FR-FCFS scheduling policy with row access cap of 4 XOR-based mapping function like Intel Skylake [57] tCL: 13.75ns, tRCD: 13.75ns, tRP: 13.75ns
CTE$	TMCC: 64KB, 32KB reach per 64B CTE block Compresso: 128KB, 4KB reach per 64B CTE block

Table III: Simulated Microarchitecture parameters.

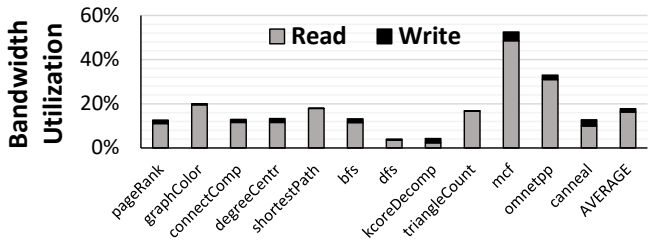

Figure 16: Memory access characterization for the evaluated benchmarks under no hardware memory compression.

available memory. Next, we use at least one second of atomic simulation to warm up ML1, ML2, and embedded CTEs in compressed PTBs. Subsequently, we warm up the branch predictor and prefetchers using 10ms of detailed simulation (without any compression-related performance overheads). Finally, we use 20ms of detailed simulation to evaluate performance. We use store instructions/cycle to evaluate performance.

We simulate one-level TLBs; Gem5 lacks two-level TLB for x86. To keep TLB hit rate consistent with real systems, we increase the number of entries in L1 TLB to 2048, which is similar to the total number of TLB entries AMD's Zen 3 [58]. This ensures a similar TLB hit rate between simulations and the real world; this is essential as TMCC optimizes for memory accesses following TLB misses.

Modeling Details for TMCC's Page-level Accesses: To prevent the faster block-level ML1 accesses from suffering

long queuing delays due to the bandwidth-intensive page-level accesses to ML2, we simulate the FR-FCFS-Capped command scheduling policy; prior works use capped policies to improve fairness. TMCC also carefully issues the 64B read and/or write requests to carry out page-level accesses so that these requests only consume at most 10 slots in MC's read/write queue at a time. To prevent the many writes due to page migration from blocking reads to the channel, TMCC only targetedly puts into write mode the rank accessed by page write, without putting the entire channel into write mode; prior works (e.g., Nonblocking Writes [59]) also only put individual ranks in a channel into write mode.

When reading from ML2, MC responds to LLC as soon as MC decompresses the requested block. In the background, MC migrates the decompressed page to ML1. This background migration is similar to the background repacking in prior works [6]. We model a 32KB buffer (i.e., eight 4KB entries) in MC to buffer data for the transfer. Accesses to ML2 are stalled when all eight entries are full.

When ML1 Free List has < 4000 chunks, ML1 grows the list by continuously evicting cold pages to ML2. The resultant ML1-to-ML2 page migrations have lower priority than LLC accesses to ML2, which trigger ML2-to-ML1 page migration; ML1 pauses eviction when LLC accesses to ML2 are outstanding or pending. But their priorities are flipped while ML1 Free List has < 3000 chunks.

VII. Simulation Results

Figure 17 shows TMCC's performance normalized to Compresso [6], a recent prior work on hardware memory compression for capacity. In this comparison, TMCC saves the same amount of DRAM as Compresso for each workload. On average across all workloads, TMCC improves performance by 14%. The improvement is highest for $shortestPath$ and $canneal$; they have high memory access rate (see Figure 16) and high CTE cache miss rate (see Figure 2). The improvement is the lowest for $kcore$ and $triangleCount$; they have low CTE cache miss rate (see Figure 2).

TMCC's performance improvement primarily comes from hiding memory latency overhead due to address translation. Figure 18 shows the average L3 miss latency of a system with: (i) No Compression, (ii) Compresso and (iii) TMCC at

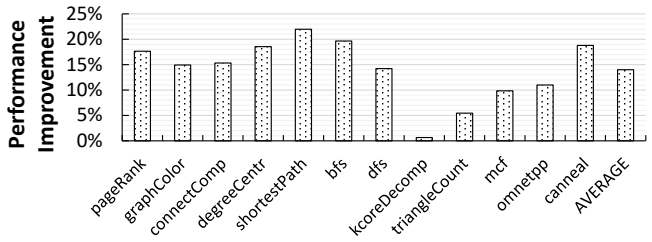

Figure 17: Performance normalized to Compresso when saving the same amount of memory as Compresso.

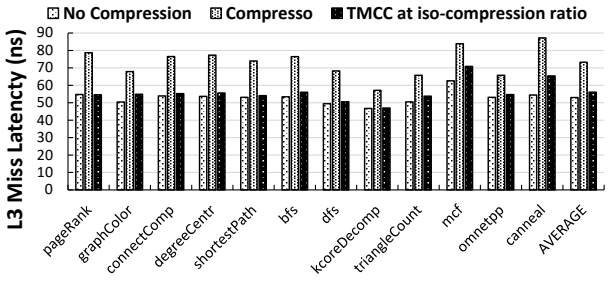

Figure 18: L3 miss latency under different systems.

iso-compression ratio as Compresso. For a system with no compression, L3 miss latency is 53ns; this includes all sources of L3 miss latency (e.g., NoC latency between MC and LLC tile), not just DRAM latency. Under TMCC, the average miss latency is very close to no compression – only 56.4ns. For Compresso, the average L3 miss latency is considerably higher – 73.9ns; the ~20ns longer memory access latency comes from accessing data serially after accessing CTEs for each CTE cache miss.

TMCC's latency benefit over Compresso are primarily due to fetching from DRAM normal memory blocks and their CTEs in parallel. As described in Section V-A3, TMCC accesses CTE and normal blocks in DRAM in parallel to verify the normal blocks speculatively fetched using CTEs embedded in CTEs. On average, 22% of LLC misses that hit in ML1 are satisfied by fetching normal blocks and CTEs from DRAM in parallel (see Figure 19). However, 22% is only a minority of LLC misses. This is because MC always caches the CTE after it arrives from DRAM. Because TMCC obtains embedded CTEs from compressed PTBs, which are only accessed during page walks, TMCC cannot use embedded CTEs to speed up address translation for LLC misses that are not preceded by page walks. As such, caching a CTE after fetching it from DRAM (e.g., after accessing the CTE in DRAM in parallel with normal data for verification after a TLB miss to a page) speeds up address translation for LLC misses that hit in TLB (e.g., later accesses to the same page).

Some of the latency benefit also comes from reducing how frequently TMCC accesses DRAM to fetch CTEs compared to Compresso. Like prior works, TMCC only fetches CTEs from DRAM when they miss in CTE cache; fetching CTEs from DRAM in parallel with normal blocks for verification

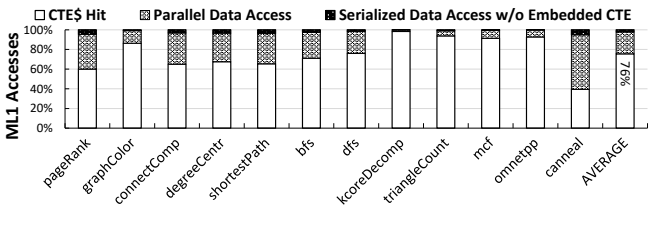

Figure 19: Distribution of ML1 read accesses.

Benchmark	DRAM Usage: GB			Compression Ratio		Normalized Comp. Ratio
	Col A (Uncomp.)	Col B (Compresso)	Col C (TMCC)	Col D (Compresso)	Col E (TMCC)	Col F
pageRank	106	82.2	35.3	1.29	3.00	2.33
graphCol	106	82.5	35.3	1.28	3.00	2.33
connComp	105	83.2	35.0	1.26	3.00	2.38
degCentr	105	82.9	35.0	1.27	3.00	2.37
shortestPath	105	82.8	35.0	1.27	3.00	2.37
bfs	105	82.7	35.0	1.27	3.00	2.36
dfs	105	81.4	35.0	1.29	3.00	2.33
kcore	105	83.7	35.0	1.25	3.00	2.39
triCount	108	83.1	36.0	1.30	3.00	2.31
mcf	15.0	13.9	6.00	1.08	2.50	2.32
omnetpp	1.00	0.63	0.40	1.60	2.50	1.58
canneal	1.10	0.95	0.73	1.15	1.50	1.30
					Average	2.2

Table IV: Compression ratio normalized to Compresso when TMCC offers the same performance as Compresso. Col A shows each workload's original memory footprint. Col B shows how much DRAM each workload uses under Compresso. Col C shows how much DRAM each workload uses under TMCC when TMCC's performance reduces down to Compresso's.

when the CTEs are already cached is unnecessary and avoided by our design in Section V-A3. As such, TMCC's DRAM access rate for CTEs (i.e., number of CTE fetches from DRAM per LLC miss) equals TMCC's CTE miss rate. Because TMCC's CTE miss rate is $1-76\%$ (76% is TMCC's average CTE hit rate, see Figure 19), TMCC's DRAM access rate for CTE is 24%. Compresso has a much higher – 34% – DRAM access rate for CTEs as block-level compression reduces CTE reach compared to page-level compression.

Sensitivity Analysis – Saving more Memory: TMCC builds on an OS-inspired approach; as such, TMCC also inherits the following behavior from OS memory compression: as a system's memory usage increases (e.g., due to having more background/context-switched processes in memory), more pages are migrated to ML2 and more DRAM will be saved by ML2's compression (see Section IV-A). As such, having high memory usage in a system can naturally trigger TMCC to save more DRAM than Compresso, especially since TMCC uses page-level Deflate, instead of block-level compression as does Compresso; in our evaluation, Compresso uses the block-level compression in Figure 15.

But, of course, saving more DRAM also comes at the cost of performance, as more pages will be stored in ML2 compressed. To fairly evaluate TMCC's memory savings over Compresso, we evaluate the performance of TMCC at various higher memory savings to identify operating points where TMCC can still provide the same (i.e., $>$ 99% of the) performance as Compresso. Table IV shows for each benchmark TMCC's compression ratio normalized to Compresso when operating at the same, instead of higher, performance as Compresso. It is 2.2x on average.

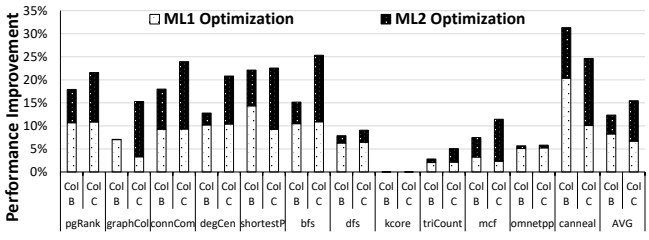

Figure 20: Improvement over barebone OS-inspired hardware compression under the two DRAM usage scenarios in Table IV Columns B and C.

Figure 20 shows improvement over the bare-bone OS-inspired hardware compression in Section IV. When both designs are saving the same **small** amount of DRAM (i.e., each workload's DRAM usage matches Column B in Table IV), TMCC improves performance by 12.5%. Figure 20 shows the split of benefit due to TMCC's ML1 and ML2 optimizations; they improve performance by 8.25% and 4.25%, respectively.

When both designs are aggressively saving the same amount of DRAM (i.e., each workload's DRAM usage matches Column C in Table IV), TMCC improves performance by 15.4% over the bare-bone OS-inspired hardware compression. In this scenario, the performance benefit due to TMCC's ML2 optimization surpasses the benefit from ML1 optimization. When aggressively saving DRAM, accesses to ML2 become more frequent (see Figure 21); higher ML2 access rate increases the impact of ML2 optimizations, while diminishing the impact of ML1 optimizations.

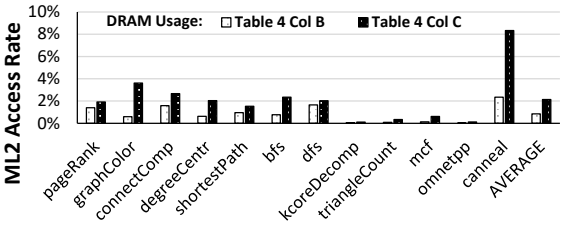

Figure 21: ML2 Accesses normalized to total LLC misses and writebacks under the two DRAM usages in Table IV Columns B and C.

Sensitivity Analysis – Smaller Workloads: We also evaluate TMCC with smaller workloads - remaining PARSEC 3.0 benchmarks and RocksDB using 1GB Twitter dataset. When saving the same amount of DRAM from these workloads as Compresso, TMCC can provide a maximum performance benefit of 5% (for $RocksDB$) and loses a maximum performance of 0.1% (for $freqmine$) compared to Compresso. The average performance is within 1% of Compresso. TMCC provides no meaningful performance benefit for these workloads because they are small and have regular access patterns. However, even for these workloads, TMCC can still provide benefits; our evaluation shows TMCC can provide 1.7X compression ratio on average as Compresso, while still providing the same (i.e., $>$ 99% of the) performance as Compresso for every workload. The maximum is 3.1x, for $blackscholes$.

VIII. Discussion

Huge Pages: TMCC's ML1 optimization is ineffective for huge pages. Each PTB for huge pages covers eight huge pages or $8{\cdot}2MB = 16MB$. This equates to $16MB/4KB =$

$4K$ regular 4KB pages; 4K CTEs is too numerous to fit in a PTB. However, TMCC's ML2 optimization still applies. Compared to prior works on hardware memory compression, TMCC still improves CTE cache hit rate due to using page-level, instead of block-level, translation. When repeating our evaluation under huge pages across the workloads in Figure 17, we find that compared to Compresso, TMCC can either improve average performance by 6%, while saving the same amount of memory, or provide 1.8X the effective memory capacity, while providing the same performance.

Memory Interleaving: Some CPUs not only interleave adjacent physical address ranges across the many memory channels in a memory controller, but also interleave adjacent physical address ranges across multiple memory controllers (MCs) to help balance bandwidth utilization. The granularity of this inter-MC memory interleaving can vary across CPUs, BIOS settings, and installed DIMM count/positions.

As TMCC resides in MC, interleaving memory across multiple MCs at sub-page (i.e., <4KB) granularity can interfere with TMCC's page-level compression. Therefore, TMCC requires address mapping to only interleave memory across memory controllers at ≥4KB granularity, instead of sub-page granularity.

We evaluate the performance impact of two TMCC-compatible interleaving policies on bandwidth-intensive benchmarks from a prior work on improving memory bandwidth [60]. We choose these benchmarks because high bandwidth usage magnifies performance differences across different interleaving policies. We simulate a system with 16 cores and two MCs with two channels per MC. The baseline interleaving policy performs sub-page interleaving at 512B granularity across MCs and at 256B granularity across the channels within each MC.

Figure 22 compares sub-page interleaving only for channels within each MC (i.e., MCs are interleaved at 4KB and constituent channels are interleaved at 256B) against the baseline interleaving. The average performance is within 1%. The maximum degradation is $< 5\%$. However, using coarser interleaving improves row buffer locality and hit rate and, therefore, provides a maximum performance improvement of 10%. For sensitivity analysis, Figure 22 also compares always interleaving pages across channels (i.e., no sub-page interleaving across channels) to the baseline interleaving; the performance degradation is more pronounced (e.g., 5%, 10% and 11% for $linpack$, sp_D, and $hpcg$, respectively).

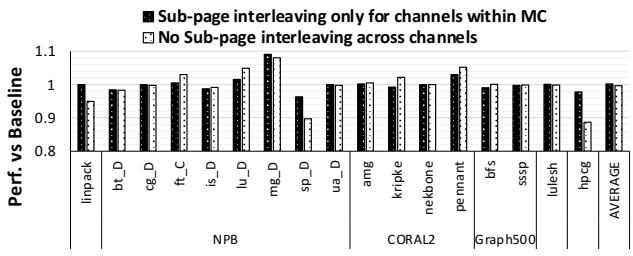

Figure 22: Performance of two TMCC-compatible interleaving policies normalized to the baseline of using sub-page interleaving across MCs.

Our results match that of a prior OS work [61]; this work turns off hardware-level sub-page interleaving across MCs and, instead, interleaves pages across MCs by modifying the memory allocator to map adjacent 4KB virtual pages to different MCs. Only 1% real-system performance difference is reported; this includes the overheads of the OS changes.

IX. Related Work

A prior work – LCP [5] – relies on OS support to embed some CTE information into PTEs; OS manages new compressed pages of different sizes (e.g., 2KB, 1KB) and records the compressed page size in PTEs. LCP uses the embedded compressed size of a page to predict the page's data blocks' DRAM locations to speculatively access data in DRAM in parallel with accessing the CTE in DRAM. Beside changing OS, the speculative parallel accesses are often wrong (e.g., as often as ~50% of the time for many workloads, see Figure 16 in [5]) because using compressed page size alone to accurately predict per-block DRAM location is difficult. When a page changes between different preset sizes due to fluctuation in compression ratio, hardware also raises interrupt to tell OS to update the page size recorded in the PTE. A later work shows these interrupts are costly [6].

Unlike LCP, TMCC embeds CTEs into PTBs software-transparently by compressing PTBs in hardware. By migrating memory content only at the page granularity, TMCC keeps CTEs small enough to fit them in PTBs to enable highly accurate speculative parallel accesses to DRAM.

X. Conclusion

This paper is the first to explore the address translation problem for large and/or irregular workloads under hardware memory compression for capacity. TMCC builds on an OS-inspired approach by addressing its latency overheads while accessing both hot pages and cold pages. For hot pages, TMCC hides the latency overhead of physical to DRAM address translation by compressing PTBs to free space in them to embed CTEs. For cold pages, we specialize ASIC Deflate for memory to reduce decompression latency by 4X compared to IBM's state-of-the art ASIC Deflate. Our evaluations show that for large and/or irregular workloads, TMCC can either improve performance by 14% without sacrificing effective capacity or provide 2.2x the effective memory capacity without sacrificing performance, when compared to state-of-the art hardware memory compression.

Acknowledgment

We thank National Science Foundation (NSF) for supporting this work under grants 1942590 and 1919113. We also thank Advanced Research Computing (ARC) at Virginia Tech for providing computational resources.

REFERENCES

[1] J. Weiner, N. Agarwal, D. Schatzberg, L. Yang, H. Wang, B. Sanouillet, B. Sharma, T. Heo, M. Jain, C. Tang, and D. Skarlatos, "Tmo: Transparent memory offloading in datacenters," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 609–621. [Online]. Available: https://doi.org/10.1145/3503222.3507731

[2] A. Fuerst, S. Novaković, I. n. Goiri, G. I. Chaudhry, P. Sharma, K. Arya, K. Broas, E. Bak, M. Iyigun, and R. Bianchini, "Memory-harvesting vms in cloud platforms," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 583–594. [Online]. Available: https://doi.org/10.1145/3503222.3507725

[3] A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid, G. Thelen, K. A. Yurtsever, Y. Zhao, and P. Ranganathan, "Software-defined far memory in warehouse-scale computers," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 317–330. [Online]. Available: https://doi.org/10.1145/3297858.3304053

[4] R. Tremaine, T. Smith, M. Wazlowski, D. Har, K.-K. Mak, and S. Arramreddy, "Pinnacle: Ibm mxt in a memory controller chip," *IEEE Micro*, vol. 21, no. 2, pp. 56–68, 2001.

[5] G. Pekhimnko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Linearly compressed pages: A low-complexity, low-latency main memory compression framework," in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013, pp. 172–184.

[6] E. Choukse, M. Erez, and A. R. Alameldeen, "Compresso: Pragmatic main memory compression," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-51. IEEE Press, 2018, p. 546–558. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00051

[7] M. Ekman and P. Stenstrom, "A robust main-memory compression scheme," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, ser. ISCA '05. USA: IEEE Computer Society, 2005, p. 74–85. [Online]. Available: https://doi.org/10.1109/ISCA.2005.6

[8] 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, oct 2015. [Online]. Available: https://doi.org/10.1145/2808233

[9] 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*, ser. CF '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 121–128. [Online]. Available: https://doi.org/10.1145/3203217.3203235

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

[11] B. Abali, B. Blaner, J. Reilly, M. Klein, A. Mishra, C. B. Agricola, B. Sendir, A. Buyuktosunoglu, C. Jacobi, W. J. Starke, H. Myneni, and C. Wang, "Data compression accelerator on ibm power9 and z15 processors," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ser. ISCA '20. IEEE Press, 2020, p. 1–14. [Online]. Available: https://doi.org/10.1109/ISCA45697.2020.00012

[12] J. Kim, M. Sullivan, E. Choukse, and M. Erez, "Bit-plane compression: Transforming data for better compression in many-core architectures," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, June 2016, pp. 329–340.

[13] A. Shafiee, M. Taassori, R. Balasubramonian, and A. Davis, "Memzip: Exploring unconventional benefits from memory compression," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014, pp. 638–649.

[14] G. Pekhimenko, E. Bolotin, N. Vijaykumar, O. Mutlu, T. C. Mowry, and S. W. Keckler, "A case for toggle-aware compression for gpu systems," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 188–200.

[15] V. Young, S. Kariyappa, and M. K. Qureshi, "CRAM: efficient hardware-based memory compression for bandwidth enhancement," *CoRR*, vol. abs/1807.07685, 2018. [Online]. Available: http://arxiv.org/abs/1807.07685

[16] A. R. Alameldeen and D. A. Wood, "Interactions between compression and prefetching in chip multiprocessors," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007, pp. 228–239.

[17] V. Sathish, M. J. Schulte, and N. S. Kim, "Lossless and lossy memory i/o link compression for improving performance of gpgpu workloads," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 325–334. [Online]. Available: https://doi.org/10.1145/2370816.2370864

[18] N. Vijaykumar, G. Pekhimenko, A. Jog, A. Bhowmick, R. Ausavarungnirun, C. Das, M. Kandemir, T. C. Mowry, and O. Mutlu, "A case for core-assisted bottleneck acceleration in gpus: Enabling flexible data compression with assist warps," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 41–53. [Online]. Available: https://doi.org/10.1145/2749469.2750399

[19] M. K. Q. Vinson Young, Sanjay Kariyappa, "Enabling transparent memory-compression for commodity memory systems," in *2019 IEEE 25st International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2010.

[20] S. Hong, P. J. Nair, B. Abali, A. Buyuktosunoglu, K.-H. Kim, and M. B. Healy, "Attaché: Towards ideal memory compression by mitigating metadata bandwidth overheads," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-51. IEEE Press, 2018, p. 326–338. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00034

[21] A. Davoudian and M. Liu, "Big data systems: A software engineering perspective," *ACM Comput. Surv.*, vol. 53, no. 5, sep 2020. [Online]. Available: https://doi.org/10.1145/3408314

[22] V. Karakostas, O. S. Unsal, M. Nemirovsky, A. Cristal, and M. Swift, "Performance analysis of the memory management unit under scale-out workloads," in *2014 IEEE International Symposium on Workload Characterization (IISWC)*, 2014, pp. 1–12.

[23] A. Margaritov, D. Ustiugov, E. Bugnion, and B. Grot, "Prefetched address translation," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 1023–1036. [Online]. Available: https://doi.org/10.1145/3352460.3358294

[24] C. H. Park, I. Vougioukas, A. Sandberg, and D. Black-Schaffer, *Every Walk's a Hit: Making Page Walks Single-Access Cache Hits*. New York, NY, USA: Association for Computing Machinery, 2022, p. 128–141. [Online]. Available: https://doi.org/10.1145/3503222.3507718

[25] D. Skarlatos, A. Kokolis, T. Xu, and J. Torrellas, *Elastic Cuckoo Page Tables: Rethinking Virtual Memory Translation for Parallelism*. New York, NY, USA: Association for Computing Machinery, 2020, p. 1093–1108. [Online]. Available: https://doi.org/10.1145/3373376.3378493

[26] S. Ainsworth and T. M. Jones, "Compendia: Reducing virtual-memory costs via selective densification," in *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management*, ser. ISMM 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 52–65. [Online]. Available: https://doi.org/10.1145/3459898.3463902

[27] C. H. Park, S. Cha, B. Kim, Y. Kwon, D. Black-Schaffer, and J. Huh, "Perforated page: Supporting fragmented memory allocation for large pages," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 913–925.

[28] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "Colt: Coalesced large-reach tlbs," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 258–269.

[29] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[30] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C.-Y. Lin, "Graphbig: Understanding graph computing in the context of industrial solutions," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2807591.2807626

[31] LDBC Graphalytics, "Datasets," Last accessed on Aug 13, 2022. [Online]. Available: https://graphalytics.org/datasets

[32] "Skylake (server) - microarchitectures - intel," Last accessed on Aug 13, 2022. [Online]. Available: https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server)

[33] Joe Chang - qpdma.com, "Memory and cache latency comparisons," Last accessed on Aug 13, 2022. [Online]. Available: http://www.qdpma.com/ServerSystems/ServerSystems_2017.html

[34] Chris Hoffman, "What is memory compression in windows 10?" *howtogeek.com*, 2017. [Online]. Available: https://www.howtogeek.com/319933/what-is-memory-compression-in-windows-10/

[35] Android Developers Documentation, "Memory allocation among processes," Last accessed on Aug 13, 2022. [Online]. Available: https://developer.android.com/topic/performance/memory-management#low_memory_management

[36] N. Gupta, "zram: Compressed ram based block devices," Last accessed on Aug 13, 2022. [Online]. Available: https://www.kernel.org/doc/Documentation/blockdev/zram.txt

[37] IBM Knowledge Center, "Active memory expansion (ame)," February 2016. [Online]. Available: https://www.ibm.com/docs/en/aix/7.2?topic=management-active-memory-expansion-ame

[38] Topher Kessler, "Memory compression brings ram doubler to os x mavericks," *lifewire.com*, 2020. [Online]. Available: https://www.lifewire.com/understanding-compressed-memory-os-x-2260327

[39] IBM Redbooks, *Cloud Security Guidelines for IBM Power Systems*. Vervante, 2015. [Online]. Available: https://www.redbooks.ibm.com/redbooks/pdfs/sg248242.pdf

[40] kernel.org, "zsmalloc," *The Linux Kernel documentation*, last accessed on Aug 13, 2022. [Online]. Available: https://www.kernel.org/doc/html/v4.19/vm/zsmalloc.html

[41] Jonathan Corbet, "The zsmalloc allocator," *LWN.net*, Last accessed on Aug 13, 2022. [Online]. Available: https://lwn.net/Articles/477067/

[42] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4*. Intel, Apr 2022, Volume 3, Chapter 4 - Paging, Figure 4-11: Formats of CR3 and Paging-Structure Entries with 4-Level Paging and 5-Level Paging. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html#combined

[43] K. Huang, "Protect data of virtual machines with mktme on kvm," p. 8, Last accessed on Aug 13, 2022. [Online]. Available: https://www.linux-kvm.org/images/d/d7/Mktme_kvm_forum_2018.pdf

[44] K. B. Agarwal, H. P. Hofstee, D. A. Jamsek, and A. K. Martin, "High bandwidth decompression of variable length encoded data streams," US Patent 20 130 148 745A1, 2014. [Online]. Available: https://patents.google.com/patent/US20130147644

[45] L. T. Clark, V. Vashishtha, L. Shifren, A. Gujja, S. Sinha, B. Cline, C. Ramamurthy, and G. Yeric, "Asap7: A 7-nm finfet predictive process design kit," *Microelectronics Journal*, vol. 53, pp. 105–115, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S002626921630026X

[46] "Synopsys Design Compiler Ultra," Last accessed on Aug 27, 2022. [Online]. Available: https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html

[47] "Verilator," Last accessed on Aug 27, 2022. [Online]. Available: https://www.veripool.org/verilator

[48] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 72–81. [Online]. Available: https://doi.org/10.1145/1454115.1454128

[49] J. Bucek, K.-D. Lange, and J. v. Kistowski, "Spec cpu2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 41–42. [Online]. Available: https://doi.org/10.1145/3185768.3185771

[50] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura, "Sparkbench: A comprehensive benchmarking suite for in memory data analytic platform spark," in *Proceedings of the 12th ACM International Conference on Computing Frontiers*, ser. CF '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2742854.2747283

[51] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. New York, NY, USA: ACM Press, Oct. 2006, pp. 169–190.

[52] A. Prokopec, A. Rosà, D. Leopoldseder, G. Duboscq, P. Tůma, M. Studener, L. Bulej, Y. Zheng, A. Villazón, D. Simon, T. Würthinger, and W. Binder, "Renaissance: A modern benchmark suite for parallel applications on the jvm," in *Proceedings Companion of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, ser. SPLASH Companion 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 11–12. [Online]. Available: https://doi.org/10.1145/3359061.3362778

[53] G. Pekhimenko, V. Seshadri, O. Mutlu, M. A. Kozuch, P. B. Gibbons, and T. C. Mowry, "Base-delta-immediate compression: Practical data compression for on-chip caches," in *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2012, pp. 377–388.

[54] X. Chen, L. Yang, R. P. Dick, L. Shang, and H. Lekatsas, "C-pack: A high-performance microprocessor cache compression algorithm," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18, no. 8, pp. 1196–1208, Aug 2010.

[55] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Comput. Archit. Lett.*, vol. 15, no. 1, pp. 45–49, Jan. 2016. [Online]. Available: https://doi.org/10.1109/LCA.2015.2414456

[56] Standard Performance Evaluation Corporation, "Spec cpu2017," https://www.spec.org/cpu2017/.

[57] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks," in *Proceedings of USENIX Security'16*, 2016.

[58] "Zen 3 - microarchitectures - amd," Last accessed on Aug 13, 2022. [Online]. Available: https://en.wikichip.org/wiki/amd/microarchitectures/zen_3

[59] G. Panwar, D. Zhang, Y. Pang, M. Dahshan, N. DeBardeleben, B. Ravindran, and X. Jian, "Quantifying memory underutilization in hpc systems and using it to improve performance via architecture support," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 821–835. [Online]. Available: https://doi.org/10.1145/3352460.3358267

[60] D. Zhang, G. Panwar, J. B. Kotra, N. DeBardeleben, S. Blanchard, and X. Jian, "Quantifying server memory frequency margin and using it to improve performance in hpc systems," in *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ser. ISCA '21. IEEE Press, 2021, p. 748–761. [Online]. Available: https://doi.org/10.1109/ISCA52012.2021.00064

[61] K. Tovletoglou, L. Mukhanov, D. S. Nikolopoulos, and G. Karakonstantis, *HaRMony: Heterogeneous-Reliability Memory and QoS-Aware Energy Management on Virtualized Servers*. New York, NY, USA: Association for Computing Machinery, 2020, p. 575–590. [Online]. Available: https://doi.org/10.1145/3373376.3378489

[62] R. Achermann, A. Panwar, A. Bhattacharjee, T. Roscoe, and J. Gandhi, *Mitosis: Transparently Self-Replicating Page-Tables for Large-Memory Machines*. New York, NY, USA: Association for Computing Machinery, 2020, p. 283–300. [Online]. Available: https://doi.org/10.1145/3373376.3378468

APPENDIX

A. Abstract

Our artifact demonstrates the following:

- Motivational data in Figures 1 and 5 through Gem5 simulations.
- Observation in Figure 6 through page table dumps.
- Performance results in Figures 17, 18, 19 and 21 through Gem5 simulations.
- Compression ratio of few benchmarks in Figure 15 through memory dumps.
- Functional verification of our ASIC Deflate with RTL simulation.

Due to high memory and disk requirements of our artifacts, we provide the evaluators access to a system with 28-cores, 256 GB RAM and 16 TB of disk space. The system contains: (i) Our Gem5 model with fast forwarded Gem5 checkpoints and submission/processing script(s), (ii) Page Table Dumps and processing script(s), (iii) Memory dumps and script(s) for compression ratio measurement, (iv) Script(s) to simulate RTL and verify functional correctness of our ASIC. We could not make the Gem5 model in our artifact publicly available due to high storage requirement for Gem5 checkpoints. However, the source code for our ASIC Deflate is publicly available at: *https://github.com/HEAP-Lab-VT/ASIC-DEFLATE-for-memory*.

B. Artifact check-list (meta-information)

- **Algorithm:** TMCC opportunistically embeds compression translation entries in PTBs. This allows it to prefetch compression translation entries during a page work, instead of serially fetching them after a page walk. Moreover, TMCC's Deflate ASIC specialized for memory allows 4X faster compression than the state-of-the-art.
- **Program:** Gem5 performance model, RTL for ASIC Deflate
- **Compilation:** gcc-7.5.0, g++-7.5.0, Python2, Java 8+, Verilator 4.220, Gradle 7.4.2
- **Binary:** Different binaries for different experiments.
- **Data set:** Fast-forwarded Gem5 checkpoints, page table dumps, memory dumps.
- **Run-time environment:** Ubuntu 20.04.4 LTS.
- **Hardware:** Intel Xeon W-3175X (28 cores), 256 GB RAM, 16 TB disk.
- **Execution:** Different scripts for different experiments.
- **Metrics:** Different metrics for different figures.
- **Output:** Simulation stats dumps for Gem5 simulations; Terminal output for other experiments.
- **Experiments:** Gem5 simulations, Page table compressibility measurement, Compression ratio measurement, RTL functional verification.
- **How much disk space required (approximately)?:** 2 TB.
- **How much time is needed to prepare workflow (approximately)?:** 20 minutes.
- **How much time is needed to complete experiments (approximately)?:** 36 hours + 15 minutes + 1 hour + 3 hours.
- **Publicly available?:** No.

C. Description

1) How to access: We gave the evaluators SSH access to our own system. Each evaluator was provided their own user account and login credentials.

2) Hardware dependencies: A multi-core system with at least 256 GB RAM and 16 TB of disk space.

3) Software dependencies: (i) Gem5, (ii) Python2 virtual environment with prerequisite packages for Gem5, (iii) Java 8+, Verilator 4.220 and Gradle 7.4.2 for functional verification of ASIC Deflate.

4) Data sets:

- **Fast-forwarded Gem5 checkpoints:** For Gem5 simulations.
- **Page Table dumps:** To determine the fraction of compressibile PTBs (See Figure 6).
- **Memory dumps:** To measure compression ratios in Figure 15.
- **More memory dumps:** To verify functional correctness of our Deflate ASIC. To reduce experiment run time, we use a set of smaller memory dumps.

5) Models: Performance model in Gem5 for systems with: (i) No Compression, (ii) Compresso [6], (iii) TMCC.

D. Installation

- **Python2 virtual environment with all dependencies:** Present in home directory for every user. Simply execute: `source ~/envPy27/bin/activate` to use it.
- **Gem5:** Present for every user at `~/source_gem5/work`. Compile Gem5 using `compilegem5fast` command.
- **Processing script for page table dumps:** Present for every user at `~/page_table_status_bits_all_same`.
- **Processing script to compute compression ratio of memory dumps:** Present for every user at `~/compression_ratio`. The C code is present in the same folder and already compiled. If not, then it can be compiled using `./make_script.sh`.
- **Processing script for ASIC Deflate verification:** Present for every user at `~/rtl_function_verif/`.

E. Experiment workflow

We recommend the evaluators to use a terminal multiplexer such as *tmux* to run experiments. *tmux* ensures spawned processes still run after SSH connection drops. For more details, please visit this link: https://tmuxcheatsheet.com/. Home directory of every user has four folders for four different experiments. The folders are: `source_gem5`, `page_table_status_bits_all_same`, `compression_ratio` and `rtl_function_verif`. **All experiments must be run independently due to resource limitations. The second experiment is an exception and can be run simultaneously with any of the experiments.**

- **Gem5 simulations:** Execute `./run_script.sh` in `~/source_gem5/work`. After the simulations finish, execute `./run_process.sh` in

`~/source_gem5` to process the output. The simulations take 36 hours to finish.
- **Process page table dumps:** Run `./run_process.sh` in `~/page_table_status_bits_all_same`. The script should finish in 15 minutes.
- **Compute compression ratio of memory dumps:** Run `./run_script.sh` in `~/compression_ratio`. The script should take about 1 hour to finish.
- **ASIC Deflate verification:** Change directory into `~/rtl_function_verif` and execute `./run_script.sh`. The simulations take approximately 3 hours to finish. The script prints the results on to the terminal. To re-run, first execute: `gradle clean -p hardware-compressor`.

F. Evaluation and expected results

- **Gem5 simulations:** Printed data (especially averages) should be close to the data in Figures 1, 5, 17, 18, 19 and 21. Some variation for a minority of individual cases is expected. All simulations should finish successfully. We request the evaluators to contact us if they see `NaN` or negative numbers as output as this indicates simulation failure or some other problem.
- **Process page table dumps:** Printed data should be very close to the data in Figure 6. It is also possible to visually inspect page table dumps by opening them in a text editor.
- **Compute compression ratio of memory dumps:** The compression ratio output for benchmarks should be similar to compression ratio data points in Figure 15.
- **ASIC Deflate verification:** In the printed results for every memory dump, `failed (pages)` should read `0`. This means that the output after compression and decompression is same as the original for *every* non-zero 4KB page in memory dumps. Above these results, Gradle should also report `BUILD SUCCESSFUL` in green.

G. Notes

We also provide a `README.txt` in each experiment folder for quick reference. For Gem5 and page table dump experiments, they provide code overview.

H. Methodology

Experiment methodology:

- **Gem5 simulations:** See Section VI.
- **Page table dumps:** We pause benchmarks in the region of interest. Then we use the page table dump tool in public artifact of [62]. We also provide the evaluators the resources to take fresh page table dumps.
- **Compute compression ratio of memory dumps:** We use Linux's *gcore* tool to take memory dump of a program. Subsequently, we process the memory dump to compute the compression ratio. Note that we discard all-zero pages while computing compression ratios.
- **ASIC Deflate verification:** We use Verilator to run RTL simulations of the compressor and decompressor. We verify that each non-zero 4 KB page in the memory dumps are same as original after compression and decompression.

Artifact submission, reviewing and badging methodology:

- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html
- https://www.acm.org/publications/policies/artifact-review-badging