Transparent Dual Memory Compression Architecture

Seikwon Kim*†, Seonyoung Lee*, Taehoon Kim*, Jaehyuk Huh*
* *School of Computing, KAIST*
† *S/W Center, Samsung Electronics Co., Ltd.*
{*seikwon, sylee, thkim, jhuh*}*@calab.kaist.ac.kr*

***Abstract*—The increasing memory requirements of big data applications have been driving the precipitous growth of memory capacity in server systems. To maximize the efficiency of external memory, HW-based memory compression techniques have been proposed to increase effective memory capacity. Although such memory compression techniques can improve the memory efficiency significantly, a critical trade-off exists in the HW-based compression techniques. As the memory blocks need to be decompressed as quickly as possible to serve cache misses, latency-optimized techniques apply compression at the cacheline granularity, achieving the decompression latency of less than a few cycles. However, such latency-optimized techniques can lose the potential high compression ratios of capacity-optimized techniques, which compress larger memory blocks with longer latency algorithms.**

Considering the fundamental trade-off in the memory compression, this paper proposes a transparent dual memory compression (DMC) architecture, which selectively uses two compression algorithms with distinct latency and compression characteristics. Exploiting the locality of memory accesses, the proposed architecture compresses less frequently accessed blocks with a capacity-optimized compression algorithm, while keeping recently accessed blocks compressed with a latency-optimized one. Furthermore, instead of relying on the support from the virtual memory system to locate compressed memory blocks, the study advocates a HW-based translation between the uncompressed address space and compressed physical space. This OS-transparent approach eliminates conflicts between compression efficiency and large page support adopted to reduce TLB misses. The proposed compression architecture is applied to the Hybrid Memory Cube (HMC) with a logic layer under the stacked DRAMs. The experimental results show that the proposed compression architecture provides 54% higher compression ratio than the state-of-the-art latency-optimized technique, with no performance degradation over the baseline system without compression.

***Keywords*-memory compression, dual compression technique, OS transparency, locality awareness**

I. Introduction

With ever increasing memory capacity requirements from data serving and analytic applications, the capacity of external DRAM memory has become a critical resource, accounting for a significant portion of the total cost in server systems. To mitigate the increasing capacity pressure on the memory, HW-based memory compression techniques have been proposed to dynamically reduce memory usage [1], [2], [3], [4]. The contents of the external memory are compressed during the page initialization or writebacks from on-chip caches, while the data are decompressed to serve cache misses. Such memory compression techniques exploit the common characteristics of data contents, such as repeating patterns, low value variances, or zero contents. However, one critical constraint of the memory compression techniques is that the memory decompression is on the critical path of execution, as they can potentially increase memory access latency.

To reduce the decompression latency, several prior techniques limit the unit of compression to the cacheline size, and use fast decompression algorithms based on frequent patterns or low value variance within a cacheline. With the latency-optimized techniques, a cacheline-sized data can be decompressed in less than a few cycles [5], [6], [7], [8]. However, there is a fundamental trade-off in memory compression techniques between their decompression latency and compression ratios. The compression ratios of the cacheline-unit techniques are commonly much lower than long latency techniques, such as the Lempel-Ziv (LZ) compression algorithm with a larger compression unit size. For example, the IBM Memory Expansion Technology (MXT) [2] architecture can provide much higher compression ratios with the LZ77 [9] compression algorithm, a variant of the LZ algorithm with the history size of 1KB memory block. However, MXT relies on a very large last-level cache to mitigate the LLC miss latency increase of 64 cycles for decompression.

Another difficult aspect of memory compression is how to locate compressed memory pages efficiently. There have been two approaches for locating compressed data. The first approach, *OS-managed mapping*, relies on the conventional virtual memory system. OS is aware of the compression status of memory pages, and the virtual-to-physical mapping with page tables locates the compressed pages [10], [1]. However, such an OS-managed design causes a conflict against the large page support in virtual memory systems. The support for large pages to reduce TLB misses has become common as exemplified by 2MB and 1GB page sizes in x86 architectures. In the OS-managed mapping, a large page cannot be effectively compressed, if a small portion of the large page is frequently accessed. An alternative approach to the OS-managed design is to add an extra translation layer between the last-level cache (LLC) and external memory [2]. In the *OS-transparent mapping*,

the operating system can only manage the uncompressed logical memory space, and the hardware layer maps the uncompressed space to the compressed real memory space. This OS-transparent approach can eliminate the conflict between compression and large pages of the OS-managed one, but it adds an extra latency for translation.

Considering the aforementioned two aspects of memory compression techniques, this paper proposes an efficient *transparent dual memory compression architecture (DMC)*. To address the trade-off between compression ratios and decompression latency, the proposed architecture uses two compression techniques exploiting the locality in memory accesses. Our application analysis shows that only a fraction of the application data are frequently accessed for a certain time period. By exploiting the localized access patterns, only the recently accessed regions of memory are compressed by a low latency compression technique, while the rest of memory regions are compressed with a slow but high compression ratio technique.

To support automatic dual compression, and to address the conflict between large pages and compression unit sizes, the proposed architecture employs an OS-transparent mapping, while improving the translation latency and efficiency significantly from the prior MXT. With our transparent approach, the operating system only needs to consider how much memory is allocated or free in the uncompressed space. The actual compression and mapping to physical memory are handled by the hardware component. The hardware layer periodically compresses inactive blocks with a high compression ratio algorithm. To reduce the translation overhead, the architecture uses a TLB-like translation caching which improves the caching efficiency and translation miss latency with our optimizations.

In this paper, we apply the proposed dual memory compression technique to the Hybrid Memory Cube (HMC) model, which has a logic layer along with the 3D stacked DRAM layers. The HW-based compression modules are added to the logic layer, and each HMC module can compress the memory contents without any intervention from the operating system. The operating system only needs to adapt to dynamic memory size changes in the uncompressed memory space. The main contributions of the paper are as follows:

- This paper identifies the fundamental trade-off of decompression latency and compression ratios in HW memory compression techniques. Based on the observation, it proposes a novel dynamic dual memory compression method.
- The paper investigates the optimizations of memory-side address space decoupling mechanism to hide the complexity of HW memory compression from the operating system. The optimizations minimize the overhead of the OS-transparency support, providing fast address translation between the uncompressed and compressed address spaces in each memory module.
- The paper investigates a low-overhead hierarchical selection mechanism for compression type. It allows the capacity-optimized compression to be used as much as possible, while the latency overheads are minimized.

The experimental results based on a detailed timing simulation show that the proposed dual compression scheme can provide 2.05 average compression ratio for a multi-core, more than 50% higher than that with the best latency-optimized technique, while the performance degradation for memory decompression is eliminated.

The rest of the paper is organized as follows. Section II discusses the trade-offs in memory compression techniques, and presents the prior work. Section III analyzes the locality of accesses and the motivation for OS-transparency. Section IV and Section V describe the proposed transparent dual compression architecture and design optimizations. Section VI presents the experimental results and Section VII concludes the paper.

II. Background

A. Memory Compression Trade-offs

A critical trade-off in HW-based memory compression techniques exists between their decompression latency and compression ratio. Latency-optimized compression techniques have been derived from cache compression algorithms, compressing data at the cacheline granularity. On the other hand, capacity-optimized approaches attempt to find repeating patterns with a much larger unit of data than the typical 32B or 64B cacheline size.

Capacity-optimized Compression: One of the most representative capacity-optimized techniques is LZSS [11], a derivative of LZ77 [9] compression algorithm. The dictionary-based approach was implemented in the MXT architecture as a pure HW-based memory compression mechanism [2], [3]. The MXT implementation uses 1KB as the unit of compression, and the decompression of a 1KB block takes 64 cycles with four decompression units, with each unit processing 256B using a shared dictionary. In MXT, a large off-chip last-level cache (LLC) with the 1KB cacheline size stores decompressed data.

As the dictionary-based algorithm attempts to find the longest match of the data occurred in the past part of the input, its compression ratios are sensitive to the block (or history) size. As shown in Figure 1, a 64B block size has a much lower compression ratio than 512B or larger block sizes in many applications including SPEC CPU, database workloads (TPC-H) and parallel workloads (NPB). Although the compression granularity must be large enough to provide high compression ratios with LZSS, the block size cannot be arbitrarily increased, since decompression latency should be curtailed, along with the limitation on the hardware component cost. As 1KB block size provides a good compression ratio, and the feasibility of HW implementation of

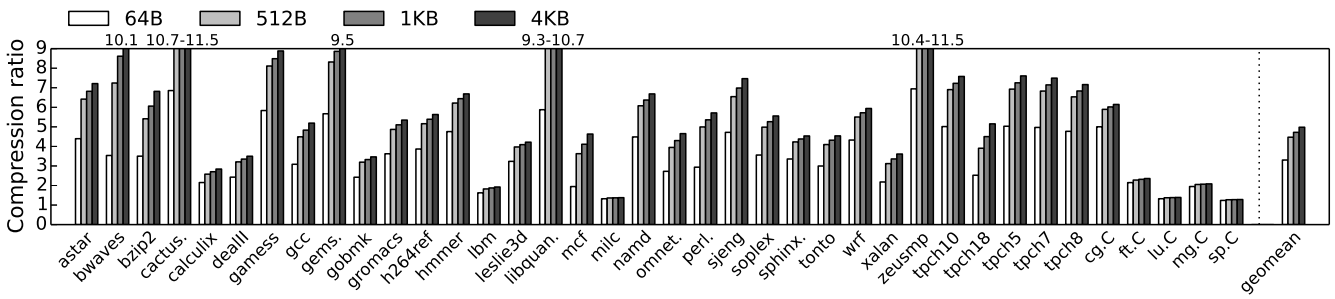

Figure 1: LZSS compression ratios with 64B, 512B, 1KB, and 4KB memory blocks

the 1KB block size for LZSS has been proved by the MXT architecture, this paper will use a variant of LZ with 1KB block as its capacity-optimized algorithm.

Latency-Optimized Compression: Latency-optimized compression techniques compress memory contents only at the cacheline granularity, narrowing the scope of data compressed and decompressed. Some of these algorithms are originated from the cache compression techniques, and applied to the memory compression techniques for their low decompression latency. Frequent Pattern Compression (FPC) targets for word pattern similarity [7]. FPC stores seven common word patterns accompanied with 3-bit prefix, replacing actual words with matching pattern prefixes. The latency to decompress the encoded cacheline is as low as 5 cycles.

Base-Delta-Immediate (BΔI) compression relies on value locality where for many cases, words in a cacheline have a narrow range of value differences [5]. Hence, BΔI splits the cacheline into multiple words and searches for the delta (Δ) distance between the first split word or zero value, called an immediate value, and the rest of the split words. The size of compressed cacheline would be the sum of the size of base and size of all the Δvalues. The decompression latency is optimized to 1 cycle, as it just adds a set of Δvalues to the base or sets zero values in parallel.

B. Locating Compressed Data

OS-managed Mapping: The first approach to locate the compressed data is to rely on the conventional virtual memory system. To use the virtual address translation mechanism for compressed data, the operating system is fully aware of the compression status and size of each memory page at the page granularity. The page table entry has the starting address of the compressed page, and the compression status must also be encoded either in the page table or a separate table to identify the location of requested cacheline within a page.

Ekman and Stenstrom proposed a memory compression architecture using the FPC algorithm [1]. In the architecture using the TLB-based translation, the starting address of a page is identified with the virtual address translation. However, since they allow cachelines within a page to be compressed to different sizes, the compressed size of all cachelines in a page must be known to locate a cacheline address. For all the cachelines in a page, their sizes are encoded in the Block Size Table (BST), and the BST entries are cached in the on-chip BST cache for fast accesses.

Unlike the variable-sized compressed blocks in a page used by Ekman and Stenstrom, LCP (Linearly Compress Pages) enforces all blocks in a page must be compressed to the same size [10]. By the enforced uniform size within a page, the extra compressed block size information is not necessary, but a simple encoding for compressed size of each page is enough to locate a cacheline within a page. A downside of the uniform size enforcement is the potential reduction of compression ratio, since the compression ratio of all the blocks in a page is determined by the one with the largest size. However, in common cases, such uniform size enforcement does not increase the compressed data size significantly due to a low variance within a page. Blocks can be compressed with any cacheline-unit compression schemes, and the study demonstrated LCP with both FPC and BΔI. The proposed technique in this paper adopts the LCP with BΔI (LCP-BΔI) as its latency-optimized compression technique, while it uses OS-transparent mapping, instead of OS-managed mapping.

OS-Transparent Mapping: An alternative approach for locating compressed data is to separate the OS-visible uncompressed address space from the compressed physical space by a HW-based mechanism. The operating system allocates and deallocates pages only at the uncompressed space, and a virtual page is mapped only to the uncompressed space. All caches are addressed with uncompressed space addresses, and the page tables and TLB address translation must also generate uncompressed addresses. OS transparent approach does not require intervention from the operating system, including TLB shootdowns for compression changes. However, once an LLC miss occurs, and the external memory is accessed, an extra layer of translation maps the uncompressed space to the compressed physical space to fetch the compressed block from the memory.

The MXT architecture uses an OS-transparent mapping

mechanism. It uses a slow but capacity-optimized LZSS algorithms at 1KB granularity. To mitigate the long decompression latency of 64 cycles, it has a large 32MB LLC, containing uncompressed data with the 1KB cacheline size. A translation table, called *sector table*, is stored at the fixed location in the memory, and each entry has the mapping information for a 1KB uncompressed memory. In MXT, the compressed physical memory space is managed at 256B sector unit. An 1KB memory can be compressed to less than 121 bits or 1∼4 sectors (256B to 1KB). If a block is compressed to a small size less than 121 bits, they are directly stored in the translation table. Otherwise, the compressed data size is always a multiple of 256B sector size. Once an LLC miss occurs, the translation component accesses the sector table in the memory to find the actual compressed location for the requested data, and the compressed data is fetched from the memory. Similar to MXT, Buri adopts an extra OS-transparent address translation layer [4]. Buri divides the physical address space into a metadata and data section. The address translation table between the OS-visible uncompressed address space to the compressed physical address space is stored in the metadata section. The metadata section for address translation, however, solely exists in DRAM. With this design, DRAM must be accessed twice to respond to cache misses. Unlike MXT, Buri adopts latency-optimized compression techniques such as BDI and FVC [5], [8], [12].

C. Other Related Work

1) Memory Compression: BPC is one of the state-of-the-art compression techniques for GPU workloads. BPC mainly focuses on preprocessing steps for cacheline compression to search more value locality in GPU workloads, especially on arrays [6]. BPC first divides 128B GPU cacheline into 32×32-bit blocks. With these blocks, BPC computes delta(Δ) distance between blocks. Given the Δvalues, BPC transforms the Δvalue array using matrix transpose-like transformation to yield more consecutive zeroes between blocks. Since there are consecutive zeroes with transformed cacheline, BPC applies Runlength-Encoding or FPC.

MemZip, COP, and Frugal ECC focuses on storing ECC in the compressed area for more reliability instead of increasing more capacity [13], [14], [15].

2) Cache Compression: SC^2 exploits statistical value redundancy in caches [16]. SC^2 uses Huffman-based statistical compression. During the sampling phase of SC^2, Huffman code is generated to build dynamic dictionary from the whole cache content [17]. With the Huffmancode, cachelines are compressed with Huffman encoded strings. C-Pack encodes with both pattern matching and dictionary matching [18]. If the words match with the pattern, C-Pack produces the corresponding pattern code. Multiple cache compression algorithms target the zero value, since zero value is the most commonly observed value in memory [19], [8]. They utilize zero as a special value to compress the data [1], [20], [21].

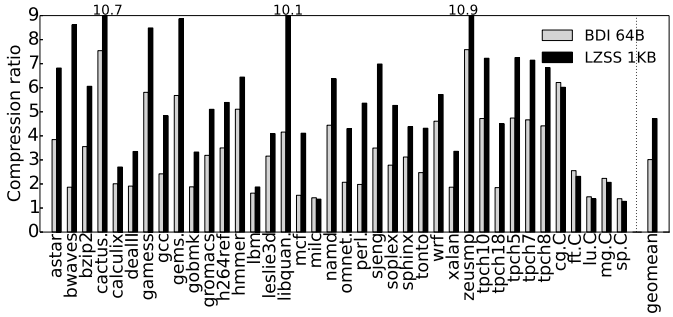

Figure 2: Compression ratio comparison between BΔI and LZSS

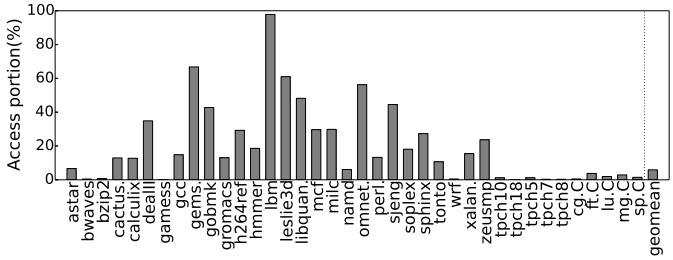

Figure 3: The percentage of 1KB memory blocks accessed during 500M instructions

III. Motivation

A. Latency vs. Capacity-optimized Compression

Capacity-optimized compression techniques provide much higher compression ratios than latency-optimized ones. Figure 2 presents the compression ratio with the latency and capacity-optimized compression techniques, using BΔI or LZSS. BΔI is compressed with 64B granularity and LZSS is compressed in 1KB granularity. The rest of system and workload configurations are presented in Section VI-A. As shown in the Figure 2, LZSS can provide on average 56.8% higher compression ratio than BΔI. However, the decompression latency of the LZSS is as high as 64 cycles in the MXT implementation, compared to 1 cycle for BΔI.

B. Active Memory Regions

Although the latency-optimized compression algorithms can support a single cycle decompression latency, such a low latency is not critical for rarely accessed memory regions. To quantify the portion of memory accessed to serve LLC misses and writebacks for a certain period of time, we first measure how many 1KB pages are accessed during 500M instruction execution. In this trace-based analysis, a 2MB LLC exists and memory is accessed only for LLC misses or writebacks. The trace-based simulation processes 2 billion

instructions, and the results are the average of 4 periods of 500M instructions for each period. Figure 3 presents the percentage of the accessed 1KB memory blocks over the total memory footprint of applications. The total memory footprint includes only the memory pages touched at least once since the applications start.

As shown in the Figure 3, across the benchmark applications, only 5.9% of the memory footprint is accessed for 500M instruction execution. Although several applications such as `omnetpp`, `lbm`, `leslie3d`, and `gemsFDTD` have more 50% of memory footprint accessed during the period, the rest of applications exhibit strong spatially skewed access patterns.

These results show that only a small portion of the application memory blocks are accessed during a relatively long period of application execution. Those frequently accessed memory blocks can be compressed with a latency-optimized compression algorithm to minimize their performance impact. However, the majority of memory footprints are infrequently accessed, and they can be compressed with the capacity-optimized algorithm. Due to the lack of accesses to the memory regions, the negative performance impact of these long decompression latency is low. Exploiting these skewed memory access patterns, the proposed dual compression technique uses a latency-optimized compression and capacity-optimized techniques selectively for memory pages with different access patterns.

C. Software Managed Compression

An alternative mechanism to HW memory compression is the software-managed memory compression. The general idea behind SW-based memory compression is that the swap-out pages can be compressed as part of the memory, instead of the slow storage. Therefore, the memory is divided into the uncompressed and compressed memory pools. The OS compresses swap-out pages to save memory capacity and copies the page to the compressed memory pool. IBM Active Memory Expansion(AME) and Linux Zswap compression scheme are such examples [22], [23].

However, there are several limitations in the SW-managed approaches. First, the cost for accessing the compressed memory page is very high, as it causes a costly page fault, and SW decompression of an entire page. A page fault handling alone without the decompression latency can take up to 2 µs for interrupt, and additional 4 µs for the TLB shoot down on a 4-core platform running the Linux OS [24]. Second, the compression is done only at the page granularity, and thus using large pages can reduce the compression efficiency. With 2MB large page, only a small portion of the larges can be accessed for a time period. In this case, the entire 2MB page cannot be compressed without decomposing the 2MB page into smaller ones. Third, the compression and decompression consume computing capability of the CPUs, and thus compression process is conducted only while CPUs are relatively idle. When all CPUs are very active, the memory compression becomes less effective. According to AME, 7% of additional CPU usage is estimated to gain 21% of additional memory space and 21% of additional CPU cycles to gain 51% of additional memory space [22]. Due to the aforementioned limitations, the pure SW-based approach can compress memory pages conservatively, to avoid additional costs. Unlike the SW-based approach, the proposed temporally and spatially fine-grained HW-based approach will mostly eliminate the overheads while achieving very high compression ratios.

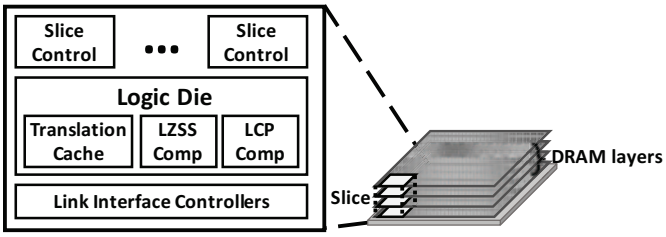

Figure 4: Architecture overview on HMC

IV. Dual Memory Compression

This section describes the proposed *Transparent Dual Memory Compression (DMC)* scheme which achieves both high compression ratio and low latency. The proposed architecture uses the LCP-BΔI compression for recently accessed data, but the rest of the application memory blocks are compressed with a variant of the LZSS compression. The architecture employs an OS-transparent approach with an extra translation layer.

A. Overall Architecture

To provide the transparent compression, the architecture separates the uncompressed address space from the compressed physical address space. The operating system manages memory pages only at the uncompressed space. All caches are indexed and tagged with uncompressed addresses, and cache coherence is also conducted with the uncompressed address space.

Although the proposed compression architecture can be implemented on conventional DRAM by adding extra modules to the CPU side along with memory controllers, this paper applies the design for a memory module with a logic component, such as Hybrid Memory Cube (HMC) [25], [26]. In HMC, the memory dies and logic die are stacked, and the logic layer has the controllers and interconnection routers. In our design for HMC, the extra components for the compression architecture are added to the logic layer. With the HMC-based design, the compression logic needs to manage the physical memory of the particular HMC module only. Figure 4 presents the compression components added to the HMC module.

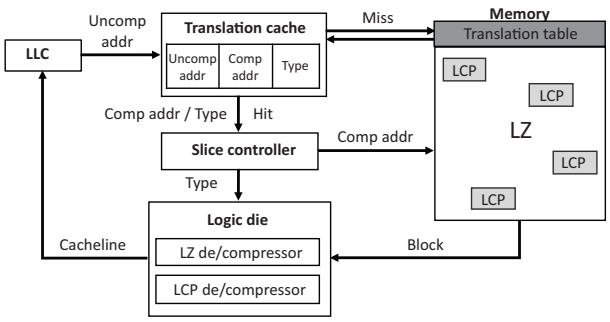

Figure 5: LLC miss handling with dual memory compression

The transparent DMC architecture consists of two compress/decompress units for the LCP and LZ techniques, translation unit, and free space management unit. We use the LCP (Linearly Compressed Pages) scheme with the BΔI compression algorithm [5]. For LZ, we use a variant of LZ, similar to the one implemented in the IBM Memory Extension Technology (MXT) [2]. In the rest of the paper, two compression mechanisms are referred to `DMC-LCP` and `DMC-LZ`. For a memory access request either from an LLC miss or writeback, the uncompressed address is translated to the compressed physical address to find the location of the compressed block in the memory. For the translation support of an HMC module, a translation table at 1KB granularity is created in the memory during the system initialization. To reduce accesses to translation table in the memory, a TLB-like translation cache absorbs the majority of translation requests.

After the translation process is completed, the requested compressed block is fetched and decompressed either by `DMC-LCP` or `DMC-LZ`, based on the compression type. Once a memory block is accessed, the memory block is stored in LCP compression, since it is likely to be accessed in near future. Periodically, the access status of memory regions are checked, and if they are not accessed within the period, they are compressed with `DMC-LZ` for better compression ratio. Figure 5 presents the LLC miss handling with the proposed architecture.

Note that *block* in the rest of paper is the unit of compression/decompression in the proposed architecture. With the slow but capacity-optimized `DMC-LZ` scheme, a 1KB uncompressed block is compressed and decompressed. With the fast `DMC-LCP` scheme, the block size is the cacheline size (64B). For the memory blocks compressed with `DMC-LZ`, a 1KB uncompressed block is mapped to the physical memory block of 0 to 1KB size, depending on its compression ratio. However, for the memory blocks compressed with `DMC-LCP`, a group of blocks constitute a *region*. The region is the mapping unit between the uncompressed and compressed spaces for memory blocks compressed with `DMC-LCP`. Within a region, all the blocks must be compressed with the same compression ratio, which is set to the worst compression ratio of blocks within a region. Such a limit is required to find the block address within a region quickly without any extra fine-grained mapping information.

B. Translation and Memory Management

For the mapping between the uncompressed and compressed spaces, a translation table is stored in the fixed part of the HMC memory. Since the translation table is created at the system boot time, they are created in a contiguous memory region. Unlike virtual memory supports with a much larger virtual address space for each process than the physical space, there is only one uncompressed memory space per HMC module, and its size is limited to a certain multiple of the actual physical space.

Considering these simple requirements for the translation table, a simple flat contiguous translation table is efficient, unlike multi-level tree tables required for conventional virtual memory supports. With the simple flat table design, a translation cache miss will require only one memory access to fetch the corresponding entry.

The capacity overhead of the in-memory translation table is small. For the 4GB HMC module, the 2:1, 4:1, 8:1 ratios of uncompressed and compressed spaces require 0.8%, 1.6%, and 3.1% of the total memory space for the translation table. The uncompressed and compressed space ratios are not fixed, as each application has a different compression ratio. As the compression ratio becomes higher, the translation table grows. For example, when the compression rate becomes 8:1, it requires 3.1% of the physical memory for the translation table, but the available memory size becomes 8 times larger than the uncompressed system. Considering the saving of memory compression, the table overhead is negligible. Since the translation table must reside in the fixed contiguous memory region for fast accesses, increasing the table size for high compression ratio may require migration of existing memory pages to different locations. However, Such table size changes are rare, and the migration can be minimized by avoiding allocating the reserved region for normal pages, unless the memory is almost exhausted.

For `DMC-LZ`, the physical memory space is managed at 64B, 256B, 512B, 768B, and 1KB container units, and a 1KB uncompressed block can be mapped to one of the container sizes. For `DMC-LCP` with the 32KB region size, container units are managed at 2KB, 4KB, 8KB, 16KB and 32KB granularity. To simplify physical memory management, the compressed block is mapped to only one of these five sizes. A special case is that when the memory block content is all zeros, the zero status is directly encoded in the translation table entry without taking up any extra memory. Checking the zero content status is part of the LCP implementation, and thus it does not require any extra logic, once LCP is added.

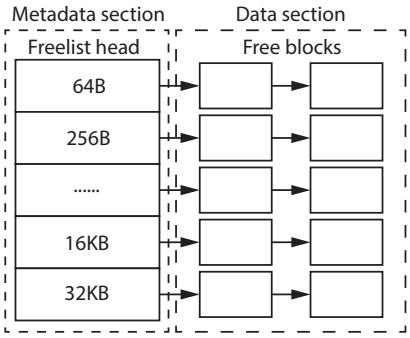

Figure 6: Free block management

Figure 6 represents how memory blocks are allocated. DMC uses a free list to allocate memory. The free list header of a size class points to an unallocated container, connected in a linked list. Free list headers of all size classes are located in the metadata section. If a new block needs to be allocated, the compressed container size is rounded to one of 10 allocatable size classes.

OS Changes: The proposed transparent technique does not require significant changes to the operating system. From the perspective of operating system, the only required modification is to be aware of possible changes of uncompressed memory capacity for each HMC module. Since the current operating systems support hot-plugins of the memory modules, already adapting to memory capacity changes dynamically, extra changes to the operating system are minimal.

Page Initialization: One of the advantages of the proposed transparent approach is the straightforward page initialization. Since the compression is conducted at each memory module, the operating system is not involved for memory compression, during the memory allocation and initialization. When pages are allocated for a new process or a new page is added, the OS only considers the page allocation in the uncompressed space. The allocated pages can be compressed either in DMC-LCP or DMC-LZ, but their status is hidden from the OS. Even for DMA operations of writing data from storage and networks, the DMA address is in the uncompressed space, and DMAed data will be compressed without SW intervention.

C. Enhancing Address Translation

To provide the OS-transparent dual memory compression, the efficiency of translation mechanism is critical. Although the current LLC can absorb a large working set of applications, memory-intensive workloads may suffer from the increased latency for handling the translation cache misses. To improve the translation efficiency of the limited number of the translation cache, we employ an optimization based on the compression characteristics of memory contents.

We first investigate which compression type causes the majority of translation cache misses. Figure 7 presents the portions of `DMC-LZ` and `DMC-LCP` for translation cache hits and misses. Each translation cache entry covers 1KB of uncompressed space for the results. In the figure, the majority of translation cache accesses, both hits and misses, are directed to the `DMC-LCP` compressed memory blocks. It is due to much more frequent accesses to the `DMC-LCP` blocks than those to the `DMC-LZ` blocks.

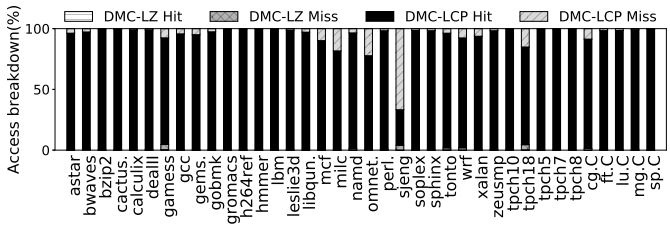

Figure 7: Translation cache access decomposition (1KB coverage for each entry)

Size	1KB	32KB	64KB	128KB	256KB
GeoMean	3.73	3.11	3.02	2.91	2.82

Table I: Compression ratio of LZ with 1KB block: Translation unit is varied from 1KB to 256KB

Although increasing the coverage of each entry can improve translation hits, it can potentially reduce the compression ratio for `DMC-LCP`. For `DMC-LCP`, the translation coverage of each entry is aligned to the region size, as each region is located in a contiguous chunk of memory. However, increasing the translation unit, and thus region size, can reduce the compression ratio, as the compression ratio is set by the worst block compression ratio within a region. Our empirical analysis shows that the 32KB region size of `DMC-LCP` does not reduce the compression ratio significantly, while increasing the coverage 32 times compared to the 1KB region size.

The block size cannot be increased with `DMC-LZ` as the compression unit is the whole block, and increasing the size will increase the decompress latency linearly. With `DMC-LZ`, it is possible to increase the translation granularity while maintaining the block size to 1KB. However, all the blocks within a translation granularity must be compressed to the same size, similar to the restriction of the compressed cache-line size in LCP. Table I shows the average compression ratio, when `DMC-LZ` uses translation granularities up to 256KB, while keeping the block size to 1KB. As shown in the table, there is a significant loss of compression ratio, when the translation granularity is increased beyond 1KB.

Considering the low compression ratio drop with a larger region size of 32KB for `DMC-LCP`, and the latency restriction of `DMC-LZ`, we use a dual granularity translation cache. The `DMC-LCP` block uses a large 32KB region as

the translation unit to reduce the translation cache misses with only a minor compression rate reduction. However, the DMC−LZ memory blocks use a small 1KB block size as the translation unit to maximize the compression ratio and to curtail the decompression latency.

D. Transcompression

Periodic DMC-LZ Compression: The proposed dual compression scheme is based on the access locality of memory pages, and thus, the key design aspect is to determine when to compress a memory block with the slow DMC−LZ compression. Our decision mechanism is based on the periodic checking of the access status of memory blocks.

For efficient access status checking, we use a hierarchical approach, with 512KB *super-region* consisting of 16 32KB *regions*. The access status is recorded and checked at the 512KB super-region granularity, and the compression type is determined at the region granularity. Therefore, for all memory blocks within a region is comparessed with the same compression algorithm.

If a super-region is not accessed during an epoch, all the 16 regions in the super-region are considered to be transcompressed with DMC−LZ in the background. If an idle region is not already compressed in the capacity optimized DMC−LZ, the 32KB data in the uncompressed space are compressed with DMC−LZ at 1KB block granularity. With the super-region unit of access status recording, checking only 8,192 bits per epoch is required for the 4GB memory to find idle regions.

DMC−LCP requires 1 cycle for 64 byte decompression and DMC−LZ requires 16 cycles to compress 64 byte of data with four LZ compression modules. If the process is pipelined with the LCP decompressor and LZ compressor, 260 cycles are required to transcompress 1KB data and 8,320 cycles for a 32KB region.

On-demand DMC-LCP Compression: The region unit is also used for the transcompression from DMC−LZ to DMC−LCP, enforcing the compression type at the same region granularity. If an access to a DMC−LZ compressed region occurs, it will transcompress the entire 32KB region to the DMC−LCP compressed blocks. Note that the 32 1KB DMC−LZ compressed blocks within a region can be scattered in the memory. All the LZ compressed blocks are read and decompressed. After the 32KB uncompressed data become completely buffered, the 32KB region data are re-compressed with DMC−LCP to a contigous chunk of DMC−LCP compressed data, since the region is the translation unit (mapping unit) for DMC−LCP compressed blocks. When spatial locality exists in the application, it reduces accesses DMC−LZ blocks, by pre-converting nearby blocks to low latency DMC−LCP blocks.

Note that the region is just the unit of enforcing a compression type. DMC−LZ still compresses and decompresses at the block granularity (1KB). DMC−LCP compresses at the cacheline granularity, while maintaining the same compressed size for all the cachelines within a region.

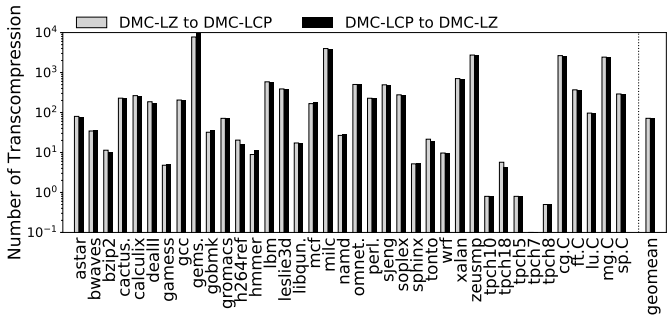

Figure 8: Number of transcompressions between DMC−LCP and DMC−LZ during 50M cycle epoch

V. DMC Optimizations

A. Limiting the Number of Transcompressions

At the end of each epoch, the unaccessed part of DMC−LCP regions must be transcompressed to DMC−LZ in the background for better compression ratio, which consumes the memory bandwidth and energy.

Figure 8 presents the number of transcompressions from DMC−LZ to DMC−LCP and from DMC−LCP to DMC−LZ respectively during 50 million cycle epoch at 32KB sub-region granularity. The average numbers of transcompressions are small for both directions; 71.6 (2.24MB) transcompressions for DMC−LZ to DMC-LCP, and 73.12 (2.29MB) transcompressions for DMC−LCP to DMC−LZ. Assuming 320GB/s of HMC bandwidth [27] and a 4Ghz core, 4GB can be handled within 50 million cycles.

Some workloads such as gemsFDTD, milc, zeusmp, cg.C, and mg.C exhibit a large number of transcompression occurrences. To avoid extensive bandwidth consumption and energy consumption from corresponding workloads, DMC limits the number of transcompressions. The amount of transcompressions can be reduced by curtailing the number of transcompressions for each epoch. The largest number of DMC−LCP to DMC−LZ sub-region transcompressions among workloads is gemsFDTD. As the size of a DMC sub-region is 32KB, the total size of transcompression in gemsFDTD is about 242MB per 50 million cycle epoch. Although this workload does not exceeds the memory bandwidth of HMC, limiting the number of transcompressions to reasonable size only affects a few workloads, while saving energy and bandwidth significantly in such cases. In the evaluation, we limits the transcompression per epoch to 2,400 per 50 million cycles.

B. Optimization on Zero Blocks

Zero values are the most frequently observed value in memory [1], [19], [28], [29], [20]. To utilize the zero value behavior, a number of compression schemes proposed to

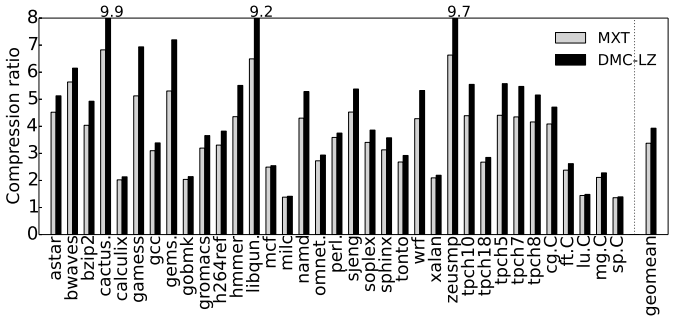

Figure 9: Compression ratios of MXT and DMC-LZ

treat zero as a special value [30], [31], [21]. LCP-BΔI identifies zero blocks and encodes the zero status with a bit information in the translation table entry. To further optimize the LZ compression, we adds the zero content support to the LZ-compressed blocks too. Figure 9 presents the compression ratio changes by adding the support for zero pages. As shown in the figure, adding the zero content support provides 14.3% higher compression ratio than MXT. (Original MXT: 3.37 to DMC-LZ: 3.93)

C. Detecting Excessive Performance Degradation

Memory access patterns without any locality may not be friendly to the DMC scheme either with high translation cache miss rates or frequent accesses to LZ-compressed blocks. If performance overheads caused by DMC exceeds a certain threshold, the OS or administrator can disable the DMC compression entirely. To help detecting the worst case scenario, DMC provides an estimation method for the performance degradation by the compression scheme. Additional cycle overheads are estimated with the number of translation cache misses per kilo-instructions(TCMPKI), the number of DMC-LZ hits per kilo-instructions(LZHPKI), and the number of zero-block hits per kilo-instruction(ZHPKI).

$$\begin{aligned} Overhead &= \alpha \times (TCMPKI - ZHPKI) \\ &\quad + (\beta \times LZHPKI) \end{aligned}$$

where α is the average memory access latency from logic die and β is the average DMC-LZ decompression latency. Since $Overhead$ is based on kilo-instructions, CPI without compression can be derived.

$$NoCompressionCPI = \frac{CurCPI \times 1000 + Overhead}{1000}$$

where $NoCompressionCPI$ refers to CPI without compression and $CurCPI$ refers to current CPI computed in CPU. Based on the current CPI and estimated CPI without compression, the performance degradation is estimated. However, in our benchmark applications, no application suffered from any notable performance degradation, and the estimation method did not trigger the disabling of compression.

CPU Processor	1-4 cores, out-of-order x86 ISA, 4GHz
CPU L1-D cache	I cache 32KB, D cache 32KB, 64B cache-line, 8-way
CPU L2 cache	2 MB per core, 64B cache-line, 32-way
Translation cache	4096 entries per core, 32KB block for LCP, 1KB for LZ
Epoch length	50 million cycle
Memory Config	4GB HMC, 32 vaults, 8 banks/vault, 4KB row buffer
DRAM Timing	tCK=1.25, tRP=11, tCCD=4, tBURST=4, tRCD=11, tCL=11, tCMD=1, tWR=12

Table II: Simulated system configuration

VI. Evaluation

A. Methodology

Our evaluation uses a timing accurate simulator that combines McSimA+ [32] and GEMS [33]. Using a PIN-based driver, the core architecture is modeled with McSimA+, while the cache hierarchy is modeled with GEMS. We configured the DRAM model with the 4GB HMC parameters. Table II presents the detailed simulation parameters for cores, cache hierarchy, and HMC. Since the accurate compression model requires the actual memory contents during the execution, we use an execution driven simulation. In this study, we evaluate both a single core model and a multi-core model. Moderate private 2MB LLC size per core and one HMC module is set to serve our models, not to overprovision the resources.

We use the SPEC CPU2006 [34], five workloads from TPC-H [35], and five workloads from NPB [36] suite as our benchmark applications. The TPC-H benchmark database is created at scale factor 4.0 and the NPB benchmark is set to class C. After fast-forwarding each application to the representative phase, the results is collected by running 2 billion instructions for a single core model and 4 billion instructions for a multi-core model.

Unlike previous sections, the simulator compresses only the memory pages touched by the application since the application launch. The other untouched memory pages are excluded from the compression ratio measurement, to evaluate the proposed scheme conservatively. Note that with the proposed dual compression scheme, compressing the untouched pages with `DMC-LZ` improves the compression ratio significantly, but those pages are not included in our measurement not to exaggerate the effect. During the fast-forwarding execution, the dual compression schemes are applied approximately without an accurate timing model. With the approximate modeling, all touched memory regions are compressed either by `DMC-LCP` or `DMC-LZ` at the beginning of the timing simulation, depending on the latest access time of each region.

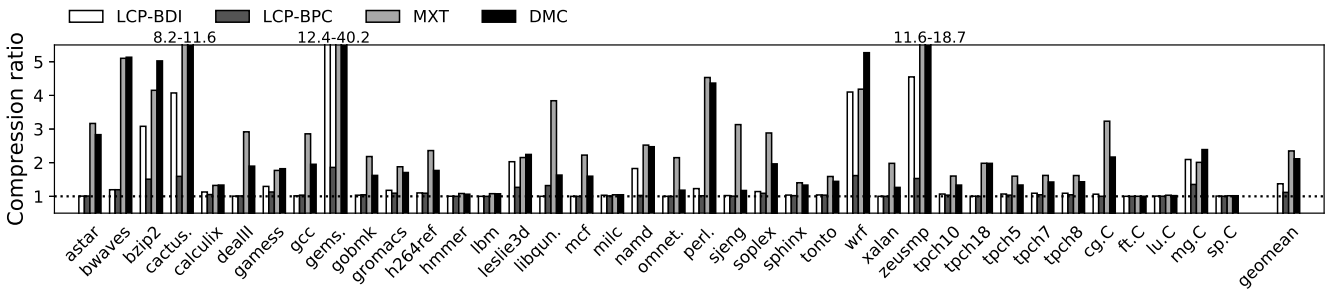

Figure 10: Single-core memory compression ratios

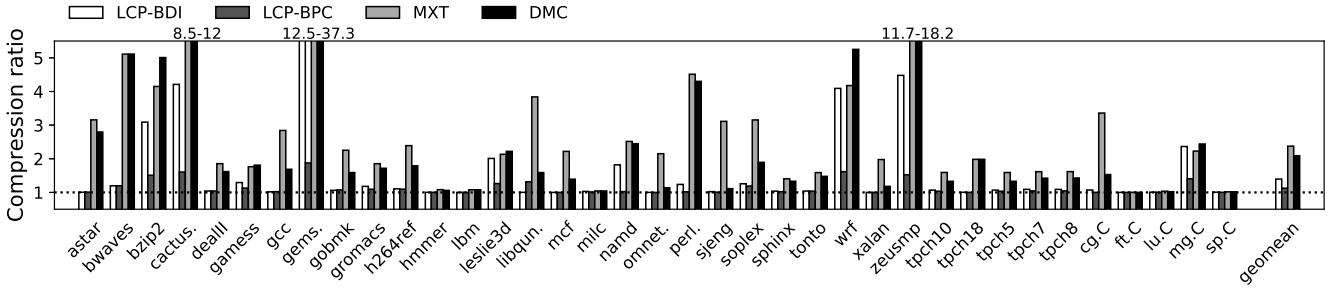

Figure 11: Multi-core memory compression ratios

DMC transcompress from DMC−LCP to DMC−LZ for every 50 million cycle epoch. In the following IPC and compression ratio results for both single-core and multi-core models, we chose a maximum limit of 2,400 transcompressions per epoch to avoid the bandwidth waste in memory. 2,400 transcompressions per epoch corresponds to 75MB transcompressions per 50 million cycles.

For comparison, we evaluate three prior techniques, MXT, LCP−BDI, and LCP−BPC. For the MXT and LCP−BDI configurations, we use decompression latencies of 64 cycles and 1 cycle respectively, the same latencies as DMC−LZ and DMC−LCP. LCP−BPC requires 7 cycle latency for decompression. Address translation latency is not applied to LCP−BDI and LCP−BPC, as they are originated from the OS-managed mapping. For DMC and MXT, Address translation cache miss adds dynamic latency of memory access, whereas one cycle for translation cache hit.

B. Compression Ratio

Figure 10 and Figure 11 present compression ratios with the proposed DMC for single-core and multi-core models respectively, compared to prior compression techniques. In the graph, the first two bars (LCP−BDI, and LCP−BPC) show the compression ratio of LCP with two different cacheline compression algorithms discussed in Section 2. For the LCP results, the compression unit is the cacheline size of 64B, but all the cachelines within a 1KB block must take up the same compressed space. The MXT bar presents the compression ratio, when the entire memory is compressed with the LZ compression used by the MXT architecture. DMC presents the results with the proposed dual compression.

The best compression ratios of the two latency-optimized compression schemes (LCP−BDI, and LCP−BPC) is less than 1.4. The highest average compression ratio among the techniques is LCP−BDI with 1.37 and 1.39 for single-core and multi-core respectively. However, if the entire memory is compressed with the capacity-optimized MXT compression, the average compression ratio can improve to 2.35 for single-core and 2.37 for multi-core, confirming the large compression ratio gap between the latency-optimized and capacity-optimized compression techniques. Note that LCP−BPC is one of promising compression techniques for GPU workloads with 128B cacheline. However, the mechanism is less suitable for CPU workloads with 64B cacheline as shown in the figure.

The proposed DMC provides a significantly higher compression ratio, 2.11 for single-core and 2.05 for multi-core, than the latency-optimized one. With the compression ratio of 2.11, 4GB memory can operate as more than 8.4GB memory on average. Although its average compression ratio is still slightly lower than the compression ratio of MXT, it eliminates the performance degradation of MXT, as will be discussed in the next section.

C. Performance

Along with the compression ratio, the next important aspect is the performance degradation for supporting memory compression. Figure 12 and Figure 13 present the normalized IPC (instruction per cycles) compared to the

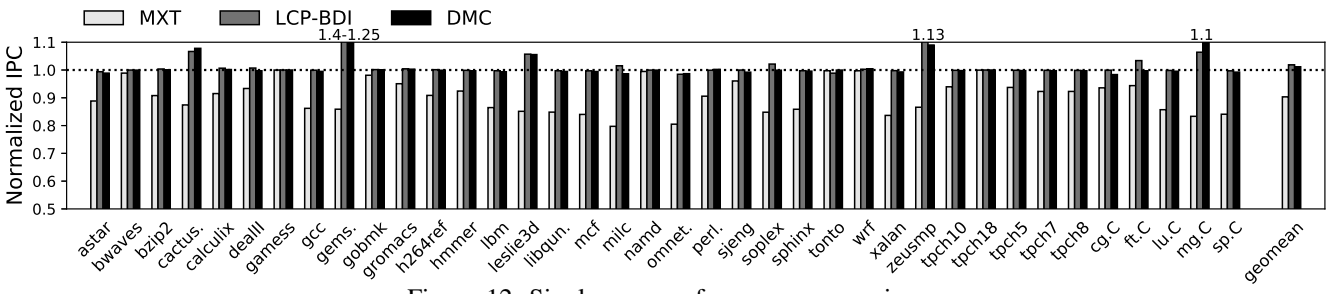

Figure 12: Single-core performance comparison

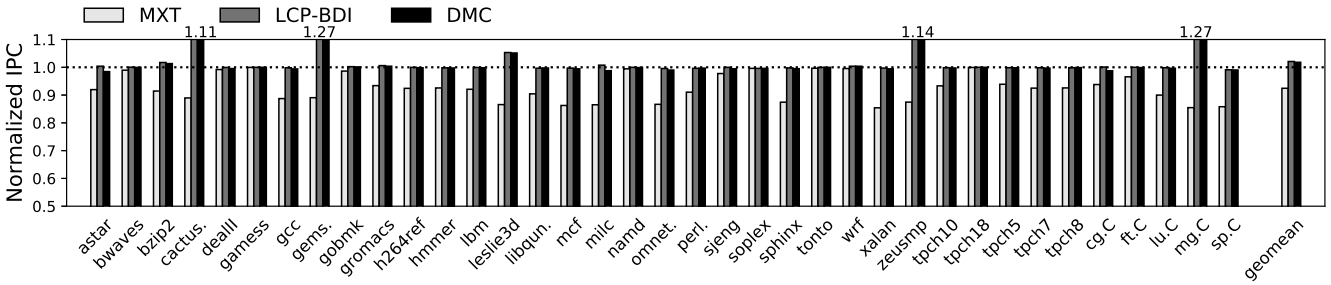

Figure 13: Multi-core performance comparison

baseline system without memory compression in single-core and multi-core models, respectively.

The performance with the capacity-optimized scheme (MXT in the figure) exhibits a significant performance degradation, compared to the baseline. It can reduce the performance by 10.0% for single-core and 7.6% for multi-core on average. For memory intensive applications, such as milc, omnetpp, xalan, mcf, mg.C and sp.C the performance degradations are from 16.0% to 21.3% for single-core, and from 13.4% to 14.6% for multi-core.

Although the performances with the latency-optimized LCP-BPC scheme is not shown, its performance is close to that of the baseline system; LCP-BPC takes 7 cycles to decompress instead of 1 cycle in LCP-BDI. LCP-BDI and DMC does not access memory with zero value which improves the overall IPC. As shown in the result, both LCP-BDI and DMC slightly improved performance compared to the baseline due to the zero value optimization. LCP-BDI was improved by 1.8%, and 2.0% and DMC was improved by 1.2%, and 1.8% for single-core and multi-core respectively. While performance is slightly better than the baseline, DMC expands the memory capacity by more than two times as shown in the previous section.

With DMC, the worst IPC degradation scenario among the benchmarks is cg.C in single-core evaluation. However, the performance degradation is only 1.7%.

D. Effect of Transcompression Limit

Limiting the number of transcompressions can affect both compression ratio and performance. If the only a small number of transcompressions is allowed, the reduced DMC-LZ compressed blocks could result in the reduction of compression ratios, while it can potentially improve IPCs at the cost of compression efficiency. Thus, we evaluated the effect of transcompression limit from 4,800 transcompressions (150MB) to 1,200 (37.5MB) per epoch.

	1200	2400	4800	∞
Single-core	2.09	2.11	2.12	2.13
Multi-core	2.03	2.05	2.06	2.09

Table III: The effect of transcompression limit on compression ratios: from 1,200 to unlimited transcompressions per epoch

	1200	2400	4800	∞
Single-core	+1.2%	+1.2%	+1.2%	+1.2%
Multi-core	+1.8%	+1.8%	+1.8%	+1.8%

Table IV: The effect of transcompression limit on IPCs

Table III shows how limiting the number of transcompressions influences the compression ratio. The effect of transcompression limit on compression ratio is minimal. Even if transcompression is limited to 1200 per epoch, compression ratios are still above 2.0 for both single-core and multi-core simulations. The compression ratio drops only by 6% in multi-core simulation with 1200 transcompressions compared to that without the limit. Table IV presents the normalized IPCs with transcompression limits. As shown in the table, limiting the number of transcompressions has almost no effects on the performance.

E. Translation Cache Sensitivity

	1K	4K	8K
geomean	96.1%	98.4%	98.8%

(a) Average hit rates

Applications	1K	4K	8K
gamess	93.9%	94.0%	94.4%
mcf	92.3%	95.6%	96.7%
omnetpp	87.4%	98.2%	99.4%
sjeng	43.4%	83.1%	92.5%
cg.c	87.4%	90.9%	92.3%

(b) Workloads under 95% hit rates with 1K entries

Table V: Translation cache hit rates

In addition to the default 4K entries for the translation cache, this section presents the translation efficiency with 1K and 8K entries. Table V presents the translation cache hit rates with the 32KB block size for `DMC-LCP`, when the number of translation cache entries is increased from 1024 to 4096 and 8192 entries. The table shows the average translation cache hit rate of all workloads, and the hit rates of the workloads with less than 95% hit rates with 1K entries. As shown in the table, with the proposed optimization, the translation cache misses are almost eliminated even with 1K entries, except for the five applications. With 4096 entries, even for the five applications, the translation cache misses are significantly reduced.

VII. Conclusion

In this study, we proposed a transparent dual compression technique to achieve both high compression ratio and low decompression latency. Utilizing the skewed access patterns existing in common applications, recently accessed blocks are compressed with the latency-optimized DMC-LCP algorithm, and infrequently accessed blocks are compressed the capacity-optimized DMC-LZ algorithm. To reduce the overhead of periodic status checking and extra address translation, we proposed several optimizations for the dual compression architecture, including the hierarchical access check and transcompression with region unit. With the proposed architecture, the compression ratios improve significantly to 2.11 for single-core and to 2.05 for multi-core on average, more than 50% higher compared to that of the latency-optimized scheme. At the same time, DMC eliminates the performance degradation of the capacity-optimized compression.

Acknowledgment

This work is supported by the National Research Foundation of Korea (NRF-2016R1A2B4013352) and by the Institute for Information & communications Technology Promotion (IITP-2017-0-00466). Both grants are funded by the Ministry of Science, ICT and future Planning (MSIP), Korea. Jaehyuk Huh is also supported by Microsoft Research Asia University Relations. We thank Jungrae Kim for his support for the BPC compression.

References

[1] M. Ekman and P. Stenstrom, "A robust main-memory compression scheme," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA '05)*. IEEE, 2005.

[2] R. B. Tremaine, P. A. Franaszek, J. T. Robinson, C. O. Schulz, T. B. Smith, M. E. Wazlowski, and P. M. Bland, "IBM memory expansion technology (MXT)," *IBM Journal of Research and Development*, 2001.

[3] R. B. Tremaine, T. B. Smith, M. Wazlowski, D. Har, K.-K. Mak, and S. Arramreddy, "Pinnacle: Ibm mxt in a memory controller chip," *IEEE Micro*, 2001.

[4] 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 Transactions on Architecture and Code Optimization (TACO)*, 2015.

[5] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-delta-immediate compression: practical data compression for on-chip caches," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT '12)*. ACM, 2012.

[6] J. Kim, M. Sullivan, E. Choukse, and M. Erez, "Bit-Plane Compression: Transforming Data for Better Compression in Many-core Architectures," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '16)*. IEEE, 2016.

[7] A. R. Alameldeen and D. A. Wood, "Frequent pattern compression: A significance-based compression scheme for l2 caches," *Technical Report 1500, Computer Sciences Department, University of Wisconsin-Madison*, 2004.

[8] J. Yang, Y. Zhang, and R. Gupta, "Frequent value compression in data caches," in *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-33)*. ACM, 2000.

[9] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, 1977.

[10] G. Pekhimenko, 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 *Proceedings of the 46th annual ACM/IEEE international symposium on Microarchitecture (MICRO-46)*. ACM, 2013.

[11] J. A. Storer and T. G. Szymanski, "Data compression via textual substitution," *Journal of the ACM (JACM)*, 1982.

[12] J. Yang and R. Gupta, "Energy efficient frequent value data cache design," in *Proceedings of the 35rd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-35)*. IEEE, 2002.

[13] S. Ali, T. Meysam, B. Rajeev, and A. Davis, "MemZip: Exploring unconventional benefits from memory compression," in *International Symposium on High Performance Computer Architecture (HPCA '14)*. IEEE, 2014.

[14] D. J. Palframan, N. S. Kim, and M. H. Lipasti, "Cop: To compress and protect main memory," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, 2015.

[15] J. Kim, M. Sullivan, S.-L. Gong, and M. Erez, "Frugal ecc: Efficient and versatile memory error protection through fine-grained compression," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM, 2015.

[16] A. Arelakis and P. Stenstrom, "SC2: a statistical compression cache scheme," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE, 2014.

[17] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, 1952.

[18] X. Chen, L. Yang, R. P. Dick, L. Shang, and H. Lekatsas, "C-pack: A high-performance microprocessor cache compression algorithm," *Transactions on Very Large Scale Integration Systems (VLSI)*, 2010.

[19] S. Balakrishnan and G. S. Sohi, "Exploiting value locality in physical register files," in *Proceedings of the 36rd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-36)*. IEEE, 2003.

[20] J. Dusser, T. Piquet, and A. Seznec, "Zero-content augmented caches," in *Proceedings of the 23rd International Conference on Supercomputing (ICS '09)*. ACM, 2009.

[21] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis, "The case for compressed caching in virtual memory systems," in *Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC '99)*. ACM, 1999.

[22] D. Hepkin, "Active memory expansion: Overview and usage guide," *POW03037-USEN-00, http://www.ibm.com/systems/power/hardware/whitepapers/ am_exp.html*, 2010.

[23] J. Seth, "The zswap compressed swap cache," https://lwn.net/-Articles/537422/.

[24] M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh, "Heterogeneous memory architectures: A hw/sw approach for mixing die-stacked and off-package memories," in *International Symposium on High Performance Computer Architecture (HPCA '15)*. IEEE, 2015.

[25] J. T. Pawlowski, "Hybrid memory cube (HMC)," in *Hot Chips (HC23)*, 2011.

[26] J. Jeddeloh and B. Keeth, "Hybrid memory cube new DRAM architecture increases density and performance," in *Symposium on VLSI Technology (VLSIT)*. IEEE, 2012.

[27] "Hybrid Memory Cube Specification 2.1," http://www.hybridmemorycube.org/files/SiteDownloads/HMC-30G-VSR_HMCC_Specification_Rev2.1_20151105.pdf.

[28] J. Yang and R. Gupta, "Frequent value locality and its applications," *Transactions on Embedded Computing Systems (TECS)*, 2002.

[29] A. R. Alameldeen and D. A. Wood, "Adaptive cache compression for high-performance processors," in *Proceedings of the 31nd Annual International Symposium on Computer Architecture (ISCA '04)*. IEEE, 2004.

[30] M. M. Islam and P. Stenstrom, "Zero-value caches: Cancelling loads that return zero," in *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT '09)*. IEEE, 2009.

[31] L. Villa, M. Zhang, and K. Asanović, "Dynamic zero compression for cache energy reduction," in *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-33)*. ACM, 2000.

[32] J. H. Ahn, S. Li, O. Seongil, and N. P. Jouppi, "McSimA+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling," in *International Symposium on Performance Analysis of Systems and Software (ISPASS '13)*. IEEE, 2013.

[33] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *ACM SIGARCH Computer Architecture News*, 2005.

[34] "SPEC CPU 2006 benchmarks," https://www.spec.org/cpu-2006/.

[35] "Transaction Processing Performance Council," http://www.-tpc.org/tpch/.

[36] "The NAS parallel benchmarks," https://www.nas.nasa.gov/-publications/npb.html.