Enabling Transparent Memory-Compression for Commodity Memory Systems

Vinson Young* , Sanjay Kariyappa* , Moinuddin K. Qureshi

Georgia Institute of Technology
{vyoung,sanjaykariyappa,moin}@gatech.edu

***Abstract*—*Transparent Memory-Compression (TMC)* allows the system to obtain the bandwidth benefits of memory compression in an OS-transparent manner. Unfortunately, prior designs for TMC (MemZip) rely on using non-commodity memory modules, which can limit their adoption. We show that TMC can be implemented with commodity memories by storing multiple compressed lines in a single memory location and retrieving all these lines in a single memory access, thereby increasing the effective memory bandwidth. TMC requires metadata to specify the compressibility and location of the line. Unfortunately, even with dedicated metadata caches, maintaining and accessing this metadata incurs significant bandwidth overheads and causes slowdown. Our goal is to enable TMC for commodity memories by eliminating the bandwidth overheads of metadata accesses.**

This paper proposes *PTMC (Practical and Transparent Memory-Compression)*, a simple design for obtaining bandwidth benefits of memory compression while relying only on commodity (non-ECC) memory modules and avoiding any OS support. Our design uses a novel *inline-metadata* mechanism, whereby the compressibility of the line can be determined by scanning the line for a special *marker* word, eliminating the overheads of metadata access. We also develop a low-cost *Line Location Predictor (LLP)* that can determine the location of the line with 98% accuracy and a dynamic solution that disables compression if the benefits of compression are smaller than the overheads. Our evaluations show that PTMC provides a speedup of up to 73%, is robust (no slowdown for any workload), and can be implemented with a total storage overhead of less than 300 bytes.

***Index Terms*—Memory; Compression; Bandwidth; DRAM;**

I. Introduction

As modern compute systems pack more and more cores on the processor chip, the memory systems must also scale proportionally in terms of bandwidth in order to supply data to all the cores. Unfortunately, memory bandwidth is dictated by the pin count of the processor chip, and this limited memory bandwidth is one of the bottlenecks for system performance. Data compression is a promising solution for increasing the effective bandwidth of the memory system. Prior works on memory compression [1] [2] [3] aim to obtain both the capacity and bandwidth benefits from compression, trying to accommodate as many pages as possible in the main memory, depending on the compressibility of the data. As the effective memory capacity of such designs can change at runtime, these designs need support from the Operating System (OS) or the hypervisor, to handle the dynamically changing memory capacity. Unfortunately, this means that such memory compression solutions are not viable unless both the hardware vendors (e.g. Intel, AMD etc.) and the OS vendors (Microsoft, Linux etc.) can coordinate with each other on the interfaces, or such solutions will be limited to systems where the same vendor provides both the hardware and the OS. Ideally, we want a memory-compression design that can be implemented entirely in hardware, without requiring any OS/hypervisor support. *Transparent Memory-Compression (TMC)* can provide bandwidth benefits of memory compression in an OS-transparent manner by trying to exploit only the increased bandwidth and not the extra capacity.[1]

An example of TMC is the MemZip [5] design that tries to increase the memory bandwidth using hardware-based compression and avoids any OS support. Unfortunately, MemZip requires significant changes to the memory organization and the memory access protocols. Instead of striping the line across all the chips on a memory DIMM, MemZip places the entire line in one chip, and changes the number of bursts required to stream out the line, depending on the compressibility of the line. MemZip requires significant changes to the data organization of commodity memories and the memory controller to support variable burst lengths. Ideally, we want to enable TMC for commodity memory modules while retaining conventional data organization and bus protocols.

We observe that TMC can be implemented with commodity memories by storing multiple compressed lines in a single memory location and retrieving all these lines in a single memory access, thereby increasing the effective bandwidth. For example, if two neighboring lines A and B are compressible then they can both be stored in the location of A, and a single access to A can provide both A and B, as shown in Figure 1(a). If the lines are not compressible (such as lines X and Y) then they can be resident in their respective locations.

Compression can change both the size and location of the line. Without additional information, the memory controller would not know how to interpret the data obtained from the memory (compressed or uncompressed). For example, an access to A can either give only A (uncompressed) or both A and B (compressed). Therefore, all compressed-memory designs require additional metadata information that indicates the compression status of each line. Thus, for lines A and B the compression status would be 1 (compressed) and for lines X and Y the compression status would be 0 (uncompressed).

*These authors contributed equally to this work.

[1]In fact, Qualcomm's Centriq [4] system was recently announced with a feature that tries to provide higher bandwidth through memory compression while forgoing the extra capacity available from memory compression. Centriq's design relies on a large linesize of 128 bytes, striping this wide line across two channels, having ECC DIMMs in each channel to track compression status, and obtaining the 128-byte line from one channel if the line is compressible. Ideally, we want to obtain bandwidth benefits independent of linesize, without relying on ECC DIMMs, and without getting limited to 2x compression ratio. Nonetheless, the Centriq announcement shows the commercial appeal of Transparent Memory-Compression.

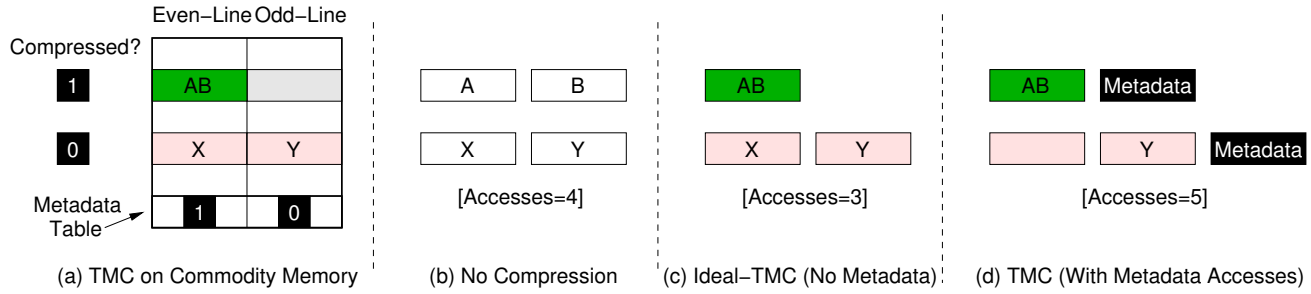

Fig. 1. (a) Transparent Memory-Compression (TMC) design for commodity memories. Sequence of accesses to read 4 lines: A, B, X, Y for (b) uncompressed memory, (c) ideal compressed memory, and (d) TMC with metadata lookups. Note that the metadata overhead is not unique to TMC and also occurs in all prior designs of compressed memory.

Conventional designs for memory compression typically store the per-line metadata in a Metadata Table, which is stored in a dedicated region of memory. Unfortunately, accessing the metadata can incur significant bandwidth overhead, even in the presence of dedicated metadata caches [3] and can cause significant slowdown (as much as 49%).

We explain the problem of bandwidth overhead of metadata with an example. Figure 1(b-d) shows three memory systems, each servicing four memory requests A, B, X and Y. A and B are compressible and can reside in one line, whereas X and Y are incompressible. For the baseline system (b), servicing these four requests would require four memory accesses. For an idealized compressed memory system (c) (that does not require metadata lookup), lines A and B can be obtained in a single access, whereas X and Y would require one access each, for a total of 3 accesses for all the four lines. However, when we account for metadata lookup (d), it could take up to 5 accesses to read and interpret all the lines, causing degradation relative to an uncompressed scheme.

Our goal is to implement TMC on commodity memory modules and without incurring the bandwidth overheads of metadata accesses. To this end, we propose *PTMC (Practical and Transparent Memory Compression)*, an efficient hardware-based main-memory compression design for improving bandwidth. Our design eliminates metadata lookup by decoupling and separately solving the issue of (i) how to interpret the data, and (ii) where to look for the data. Overall, our design makes the following three contributions.

Contribution-1: Inline-Metadata to Reduce Overheads To efficiently identify if the line retrieved from memory is compressed or not, we propose a novel *inline-metadata* scheme, whereby compressed lines are required to contain a special value, called a *marker* value at the end. We leverage the insight that compressed data rarely uses the full 64-byte space, so we can store compressed data within 60 bytes and use the remaining four bytes to store the marker. On a read, if the line contains the marker it is considered a compressed line and uncompressed otherwise. The likelihood that an uncompressed line coincidentally matches with a marker is quite small (less than one in a billion with a 4-byte marker), and PTMC handles such rare cases simply by identifying lines that cause marker collisions on a write and storing such lines in an inverted form (more details in Section IV-C).

Contribution-2: Low-Cost Predictor for Line Location The inline-metadata scheme eliminates the need to do a separate metadata lookup. However, we still need an efficient way to predict the location of the line, as the location may depend on compressibility. For example, in Figure 1(b), when A and B are compressible, both A and B are resident in the location of A. Therefore, the location of A remains unchanged regardless of compression. However, for B, the location depends on compressibility. We propose a history-based *Line Location Predictor (LLP)*, that can identify the correct location of the line with a high accuracy (98%). The LLP is based on the observation that lines within a page tend to have similar compressibility. The prediction of LLP is verified using the inline-marker obtained from the retrieved line, therefore, in the common case (of correct LLP prediction) memory access is performed with only a single memory access.

Contribution-3: Dynamic Compression for Robustness We observe that even after eliminating the bandwidth overheads of the metadata lookup, some workloads still have slowdown with compression due to the inherent bandwidth overheads associated with compressing memory. For example, the maintenance operation of compressing and writing back clean-lines incurs bandwidth overhead, as those lines would not be written to memory in an uncompressed design. For workloads with poor reuse, this bandwidth overhead of writing compressed data does not get amortized by the subsequent accesses. To avoid performance degradation in such scenarios, we develop *Dynamic-PTMC*, a sampling-based scheme that can dynamically disable compression based on the cost-benefit analysis. Dynamic-PTMC ensures no slowdown for workloads that do not benefit from compression.[2]

To the best of our knowledge, PTMC is **the only** design that provides bandwidth benefits of memory compression without requiring any OS support and while using only commodity (non-ECC) DIMMs and interfaces. It does so while retaining support for 64-byte linesize and is also applicable to single channel systems. Our evaluations show that PTMC provides a speedup of up to 73%, while ensuring no slowdown for *any* workloads. PTMC can be implemented with only minor changes to the memory controller and incurs a total storage overhead of less than 300 bytes.

[2]Such dynamic-compression designs become viable only with inline-metadata and not with table-based metadata (Section V-A).

II. Background and Motivation

We first provide the background on hardware-based memory compression, then discuss an address mapping scheme that can extend TMC for commodity memories, followed by discussion on the bandwidth overhead of metadata accesses, and the potential available from an idealized design that does not incur metadata overheads. We finally provide an insight that can avoid the metadata lookups.

A. Hardware-Based Memory-Compression

Compression exploits the redundancy in data values to store the contents in reduced space. There are several compression algorithms [6] [7] [8] [9] [10] [11] [12] in literature to compress memory contents. A hardware-based memory compression design uses these algorithms to increase not only the capacity of the main memory but also the bandwidth (as the data can be obtained with fewer accesses). Figure 2 describes the architecture of a typical hardware-based memory compression design. Without loss of generality, we assume that the data obtained from compressed memory is decompressed at the memory controller before being sent to the cache hierarchy. As each line in memory can either be compressed or uncompressed, depending on the data values, there is metadata required for each line to identify the compression status (and location) of the line. This per-line metadata is stored in a memory-mapped table and cached on-chip in the metadata cache.

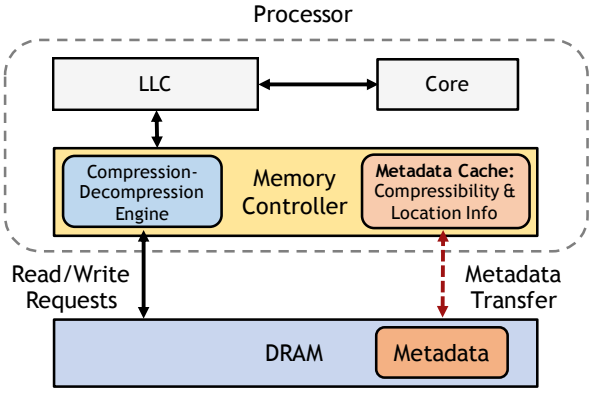

Fig. 2. Overview of Typical Hardware-Based Memory Compression. Metadata for compression status of each line is stored in memory and cached on-chip.

B. Address Mapping for Enabling TMC

Transparent Memory-Compression (TMC) can provide bandwidth benefits of memory compression in an OS-transparent manner. While prior design [5] for TMC was developed in the context of non-commodity memory modules, we are interested in developing a solution that works with commodity DIMMs. We observe that TMC can be implemented with commodity memories by co-locating multiple compressible lines at a single location and streaming out all these lines in a single memory accesses. Figure 3 shows a simple memory mapping scheme that can enable TMC for commodity memories without changing the standard burst length.

If the lines are compressible, we can place up to 4 adjacent compressed lines in one location[3] and stream out all these lines in one access. This allows us to retrieve 4 lines at the bandwidth cost of a single memory access and thus constitutes bandwidth-free prefetches (our design installs all the freely obtained lines in the L3 cache as we found that doing so improves performance). If the lines become incompressible, they get stored in an uncompressed format in the original location, without having to relocate the entire page.

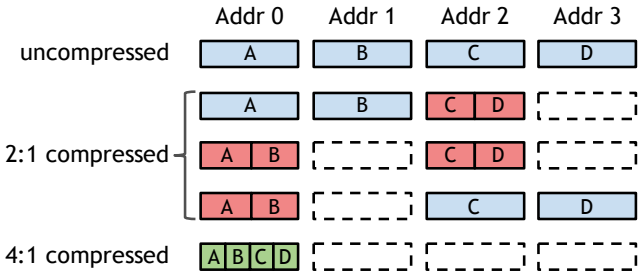

Fig. 3. Address mapping to enable TMC for commodity memories (for supporting up to 4x compression).

C. The Challenge of Metadata Accesses

An access to TMC on commodity memory would obtain a 64-byte line; however, the memory controller would not know if the line contains compressed data or not. For example, an access to A would provide both A and B, if both lines are compressible, and only A if the lines are uncompressed. Simply obtaining the line from location A is insufficient to provide the information about compressibility of the line. To help interpret the lines read, conventional designs keep track of the *Compression Status Information (CSI)* for each line in a separate region in memory, which we refer to as the *Metadata-Table*. Compressed memory needs the CSI to not only *interpret* the data line read from memory, but also to determine the *location* of the data line. Even if we provisioned only one bit per line in memory, the size of this Metadata-Table would be quite large (e.g., 32 MB). We can keep the metadata for TMC in a memory-mapped table as well, similar to Figure 2. TMC needs to access this metadata to determine the location (and contents) of the line before servicing the read. The metadata cache can alleviate some of these accesses; however, on a miss in the metadata cache, the bandwidth overhead of accessing the metadata from the Metadata-Table is still incurred.

D. Bandwidth Overhead of Metadata

Figure 4 shows the breakdown of the bandwidth consumed by designs with table-based metadata design, normalized to the uncompressed memory. In general, compression is effective at reducing the number of requests for data. However, depending

[3]When a cacheline gets evicted from LLC, the memory controller checks if (a) the neighboring cachelines are present in the LLC, and if (b) the group of 2 or 4 cachelines can be compressed to the size of a single uncompressed cacheline (64 Bytes). If so, the memory controller compacts them together and issues a write containing the 2 or 4 compressed lines to one physical location.

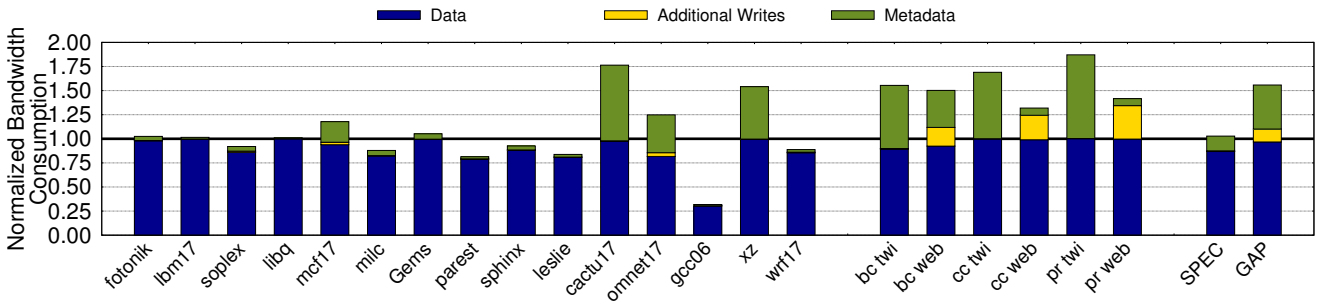

Fig. 4. Bandwidth consumption for data, additional writes to compress data, and metadata for TMC design with table-based metadata, normalized to uncompressed memory. Metadata accesses incur significant bandwidth overhead.

on the workload, the metadata cache can have poor hit-rate, and require frequent access to obtain the metadata. These extra metadata accesses can constitute a significant bandwidth overhead. For example, *xz* needs over 50% extra bandwidth just to fetch the metadata. Thus, schemes that require a separate metadata lookup can end up degrading performance relative to uncompressed memory. It is important that the implementation of compressed memory does not degrade the performance of such workloads and solving the challenge of bandwidth bloat due to metadata access is key to developing an effective design.

E. Potential for Improvement

In order to evaluate the impact of metadata, we compare the performance of a TMC design with Metadata-Table access, to an idealized design that does not require metadata access. The table-based design is equipped with a dedicated 32KB of metadata cache and is designed to capture spatial locality (similar to prior work). In contrast, the idealized compression scheme does not maintain any metadata and simply streams out lines in the same location that are compressed together. Figure 5 compares the performance of the two designs.

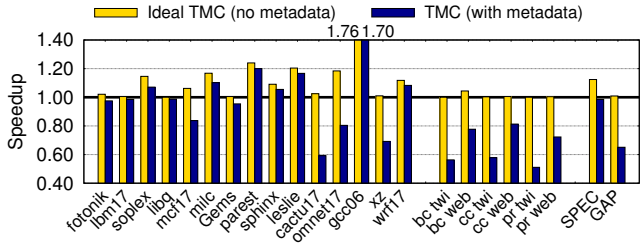

Fig. 5. Speedup from ideal TMC (no metadata lookup) and TMC with metadata (w/ metadata cache), relative to uncompressed memory.

While an ideal design can provide a speedup of 12.3% on SPEC, prior designs fall short of achieving this speedup due to the overheads associated with additional metadata-accesses. In fact, prior schemes degrade performance significantly (up to 49%) for many Graph (GAP) workloads. Therefore, unless bandwidth overhead of metadata can be reduced, we can deem TMC design as not viable for commodity memories. The goal of our paper is to enable TMC for commodity memories by mitigating the bandwidth overhead of metadata accesses. We provide an insight for a practical solution.

F. Insight: Store Metadata in Unused Space

To reduce the metadata access of compressed memory, we leverage the insight that not all the space of the 64 byte line is used by compressed memory. For example, when we are trying to compress two lines (A and B) they must fit within 64 bytes; however, the compressed size could still be smaller than 64 bytes (and not large enough to store additional lines C and D). We can leverage the unused space in the compressed memory line to store metadata information within the line. For example, we could require that the compressed lines store a 4-byte marker (a predefined value) at the end of the line, and the space available to store the compressed lines would now get reduced to 60 bytes. Figure 6 shows the probability of a pair of adjacent lines compressing to $\leq$64B and $\leq$60B. As the probability of compressing pairs of lines to $\leq$64B and $\leq$60B are 38% and 36%, respectively, we find that reserving space for this marker does not substantially impact the likelihood of compressing two lines together and thus would not have a significant impact on compression ratio.

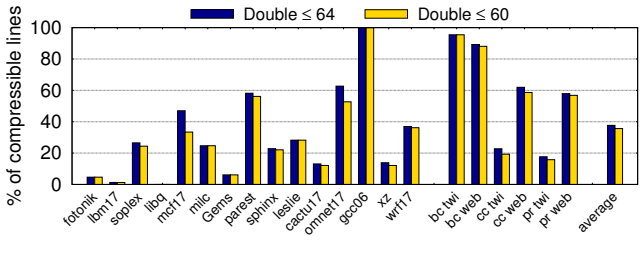

Fig. 6. Probability of a pair of adjacent lines compressing to $\leq$64B and $\leq$60B. We can use 4B to store metadata, without significantly affecting compressibility.

We can use this insight to store the metadata within the line, and avoid the bandwidth overheads of accessing the metadata separately. If the line obtained from memory contains the marker value, the line is deemed compressed, whereas, if the line does not have the marker value, then it is deemed uncompressed. However, there could be a case where the uncompressed line coincidentally stores the marker value. A practical solution must efficiently handle such collisions, even though such collisions are expected to be extremely rare. We discuss our methodology before discussing our solution.

III. Methodology

A. Framework and Configuration

We use USIMM [13], an x86 simulator with detailed memory system model. Table I shows the configuration used in our study. We assume a three-level cache hierarchy (L1, L2, L3 being on-chip SRAM caches). All caches use line-size of 64 bytes. The DRAM model is based on DDR4. We model a virtual memory system to perform virtual to physical address translations, and this ensures that the memory accesses of different cores do not map to the same physical page. Note that, other than the virtual memory translation, the OS is not extended to provide any support to enable compressed memory.

For compression, we use a hybrid of FPC and BDI algorithms and compress with the one that gives better compression. We assume a decompression latency of 5 cycles in our evaluations. Information about the compression algorithm used and the compression-specific metadata (e.g. base for BDI) are stored within the compressed line, and are counted towards determining the size of the compressed line.

TABLE I
System Configuration

Processors	8 cores; 3.2GHz, 4-wide OoO
Last-Level Cache	8MB, 16-way
Compression Algorithm	FPC + BDI
Main Memory	
Capacity	16GB
Bus Frequency	800MHz (DDR 1.6GHz)
Configuration	2 channel, 2x rank, 64-bit bus
tCAS-tRCD-tRP-tRAS	11-11-11-39 ns

B. Workloads

We use a representative slice of 1-billion instructions selected by PinPoints [14], from benchmarks suites that include SPEC 2006 [15], SPEC 2017 [16], and GAP [17]. We evaluate all SPEC 2006 and SPEC 2017 workloads, and mark '06 or '17 to denote the version when the workload is common to both. We additionally run GAP suite, which is graph analytics with real data sets (`twitter`, `web sk-2005`). We show detailed evaluation of the workloads with at least five misses per thousand instructions (MPKI). The evaluations execute benchmarks in rate mode, where all eight cores execute the same benchmark. Table II shows L3 miss rates and memory footprints of the workloads we have evaluated in detail. In addition, we also include 6 mixed workloads by randomly mixing the SPEC and GAP workloads.

We perform timing simulation until each benchmark in a workload executes at least 1 billion instructions. We use weighted speedup to measure aggregate performance of the workload normalized to the baseline uncompressed memory and report geometric mean for the average speedup across the 15 High-MPKI SPEC workloads (7 SPEC2006, 8 SPEC2017). Additionally, we evaluate GAP and MIX workloads to show that our compressed memory design does not degrade performance (in contrast to prior work) for workloads with limited spatial locality. For other workloads that are not memory bound, we present full results of all 64 benchmarks evaluated (29 SPEC2006, 23 SPEC2017, 6 GAP, 6 MIX) in Section VI-B.

TABLE II
Workload Characteristics

Suite	Workload	L3 MPKI	Footprint
SPEC	fotonik	26.2	6.8 GB
	lbm17	25.5	3.4 GB
	soplex	23.3	2.1 GB
	libq	23.1	418 MB
	mcf17	22.8	4.4 GB
	milc	21.9	3.1 GB
	Gems	17.2	5.8 GB
	parest	16.4	465 MB
	sphinx	11.9	223 MB
	leslie	11.9	861 MB
	cactu17	10.6	2.1 GB
	omnet17	8.6	1.9 GB
	gcc06	5.8	205 MB
	xz	5.7	943 MB
	wrf17	5.2	798 MB
GAP	bc twi	66.6	9.2 GB
	bc web	7.4	10.0 GB
	cc twi	101.8	6.0 GB
	cc web	8.1	5.3 GB
	pr twi	144.8	8.3 GB
	pr web	13.1	8.2 GB

IV. Design of Practical TMC

To enable Transparent Memory-Compression for commodity memory system while avoiding the bandwidth overhead of metadata lookups, we propose *Practical Transparent Memory Compression (PTMC)*. PTMC tries to obtain bandwidth benefits using memory compression without requiring OS support, without changes to bus protocol, and while maintaining the existing organization for the memory modules. Additionally, our design avoids the use of a metadata-table using novel in-lined markers, effectively side-stepping the overheads of metadata access inherent to prior designs. We start by providing a brief overview of PTMC and then explain the role of the different components of the design.

A. Overview of PTMC

PTMC is based on two key innovations: (1) inline-metadata and (2) line-location prediction (LLP). Figure 7 shows the overview of PTMC. Instead of keeping metadata explicitly in a table, inline-metadata leverages the unused space in compressed memory lines to store a marker (4-byte value in our design). If a line retrieved from memory has the marker value, it is deemed to be compressed, and uncompressed otherwise. Due to the address mapping used for TMC, the location of the line may depend on the compressibility status of the line (for example, a line B may be present along with line A at the location of A, if both lines are compressible). Therefore, for obtaining line B, TMC must access the original location of B if B is incompressible and A otherwise. To avoid the requirement of looking up both places, we develop a low-cost line location predictor that can correctly predict the compressibility status (and location) of the line with high accuracy.[4]

[4]The proposed LLP can also be used for compressed memory designs that use table-based metadata. However, for such designs LLP does not reduce the bandwidth overhead of metadata accesses, as metadata is required for verifying LLP predictions. Our inline-metadata makes it possible to verify LLP predictions with a single access to the data line, thus avoiding bandwidth of metadata lookups.

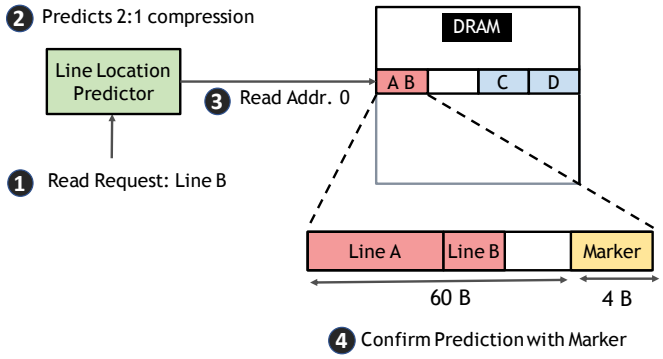

Fig. 7. Overview of PTMC: Line location is predicted with LLP and confirmed using the in-line Marker.

Figure 7 illustrates the sequence of events involved in a read operation for PTMC with an example. A read request to Line B is serviced by first consulting with the Line Location Predictor to determine its location. In this case, the LLP predicts that Line B is 2:1 compressed, and hence the read request is sent to the address location of line A. The address mapping scheme ensures that if lines A and B can be compressed together, they must be resident at the location of line A. Once the data is retrieved from memory, the prediction is confirmed by checking the value of the marker (the last 4 Bytes) to ensure that the retrieved line indeed contains the compressed line for B. If the line does not contain the marker, then there is a misprediction of the LLP, and we access line B from the original location. As LLP accuracy is high, the second access is quite uncommon. Note that there is no need for location prediction while accessing line A, as it is always resident in the same location, regardless of compressibility.

The two components of PTMC: 1. Line Location Predictor and 2. Inline-Metadata work together to obtain the bandwidth benefits of compression while avoiding the overheads associated with accessing metadata. We now provide a detailed explanation of each of these individual components.

B. Line Location Predictor

The address mapping scheme (discussed in Section II-B) ensures that we can determine the location of the line if we know the compressibility of the line with its neighbors. Thus, the Line location Predictor (LLP) can locate the compressed line by predicting its compression status. To design a low-cost LLP, we exploit the observation that lines within a page tend to have similar compressibility [3] [18].[5] Figure 8 shows LLP organization. LLP contains the *Last Compressibility Table (LCT)*, that tracks the last compression status seen for a given index. The LCT is indexed with the hash of the page address. So, for a given access, the index corresponding to the page address is used to predict the compressibility, then line location. The LCT is used only when a prediction is needed (e.g., Line A is always resident in one location and does not need prediction). We use a 512-entry LCT with storage cost of 128B.

For prior designs with metadata-table, if there is a hit in the metadata cache, then location of the line is known and

[5]We describe a dynamic fallback solution in Section V-A that can disable compression if compressibility fails to be predictable [19].

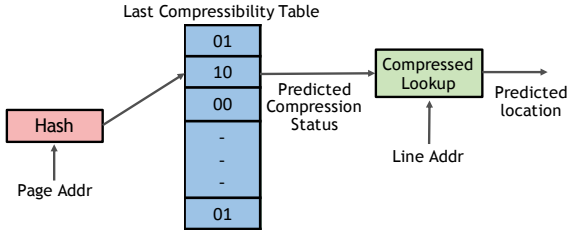

Fig. 8. Line Location Predictor uses line address and compressibility-prediction (based on last-time compressibility) to predict location.

the line can be retrieved in one memory access. However, a miss in the metadata cache results in two memory accesses: one for the metadata and the second for the data. Figure 9 compares the hit-rate of the metadata cache (32KB) with the prediction accuracy of the LLP (128 bytes). Even though the LLP is quite small, it provides an accuracy of 98%, much higher than the hit-rate of the metadata cache. On an LLP misprediction (determined by our Inline-Metadata), we re-issue the request to the other possible locations of the line and the corresponding entry in the LCT is updated with the correct compression status of the line.

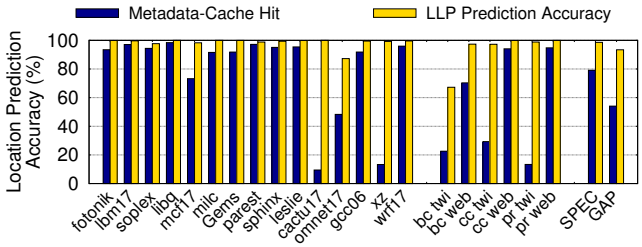

Fig. 9. Probability of finding line in one access for prior designs with metadata cache vs PTMC with LLP.

C. Inline-Metadata

The LLP predicts the compression status and the location of the line in memory. However, we need a way to detect LLP mispredictions so that we don't read incorrect data from memory. It would be nice if the data that was read also conveys information about the compression status of the line. This would let us confirm the compression status (and hence the prediction) without the need for extra metadata accesses. Our proposal is to inline the metadata within the 64B of the line that is read from memory using special 4 Byte value called the *marker*.

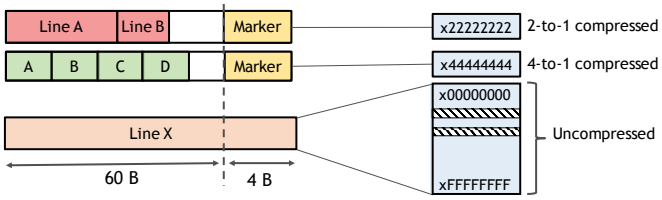

Fig. 10. Inline metadata using markers: Compressed lines always contain a marker in the last four bytes.

Figure 10 shows the inline-metadata design using markers, for lines that are compressible with 2-to-1 compression, 4-to-1 compression, or no compression. If the line contains two compressed lines (e.g. A and B both reside in A), then the line is required to contain the marker corresponding to 2-to-1 compression (e.g., x22222222) in the last four bytes. Similarly,

if the line contains four compressed lines (A, B, C, and D, all reside in A), then the line is required to contain the marker corresponding to 4-to-1 compression (e.g., x44444444). Marker reduces the available space for storing compressed lines to 60 bytes. If the compressed lines cannot fit within 60 bytes, then it is stored in uncompressed form.[6]

An incompressible line is stored in its original form, without any space reserved for the marker. The probability that the uncompressed line coincidentally matches with the 32-bit marker is quite small (less than 1 in a billion). Our solution handles such extremely rare cases of collision with marker values by storing such lines in an inverted form. This ensures that the only lines in memory that contain the marker value (in the last four bytes) are the compressed lines.

Determining Compression Status with Markers: When a line is retrieved, the memory controller scans the last four bytes for a match with the markers. If there is a match with either the 2-to-1 marker or the 4-to-1 marker, we know that the line contains compressed data for either two lines, or four lines, respectively. If there is no match, the line is deemed to store uncompressed data. Thus, with a single access, PTMC obtains both the data and the compression status.

Handling Collisions with Marker via Inversion: We define a *marker collision* as the scenario where the data in an uncompressed line (last four bytes) matches with one of the markers. Our design generates per-line markers (more details in *Attack-Resilient Marker Codes*), so the likelihood of marker collision is quite rare (less than one in a billion). However, we still need a way to handle it without incurring significant storage or complexity. PTMC handles marker collisions simply by inverting the uncompressed line and writing the inverted line to memory, as shown in Figure 11. Doing so ensures that the only lines in memory that contain the marker (in last four bytes) are compressed lines.

A dedicated on-chip structure, called the *Line Inversion Table (LIT)*, keeps track of all the lines in memory that are stored in an inverted form. The likelihood that multiple lines resident in memory concurrently encounter marker collisions is negligibly small. For example, if the system continuously writes to memory, then it will take more than 10 million years to obtain a scenario where more than 16 lines are concurrently stored in inverted form. Therefore, for our 16GB memory, we provision a 16-entry LIT in PTMC.

When a line is fetched from memory, it is not only checked against the marker, but also against the complement of the marker. If the line matches with the inverted value of the marker, then we know that the line is uncompressed. However, we do not know if the retrieved data is the original data for the line or if the line was stored in memory in an inverted form due to a collision with the marker. In such cases, we consult the LIT. If the line address is present in the LIT, then we know

[6]We choose a 4B marker because our baseline 16GB memory contains 2^{28} lines, so a 32-bit marker would cause less than 1 colliding line residing in memory on average, while limiting the space used for storing the marker. For systems with hundreds of gigabytes of memory, we recommend a 5B marker.

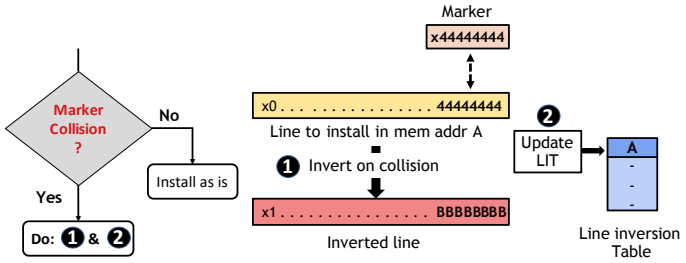

Fig. 11. Line inversion handles collisions of uncompressed lines with marker.

the line was stored in an inverted fashion and we will write the reverted value in the LLC. Otherwise, the data obtained from the memory is written as-is to the LLC.

On a write to the memory, if the line address is present in the LIT, and the last four bytes of the line no longer match with any of the markers, then we write the line in its original form and remove this line address from the LIT. Each entry in the LIT contains a valid bit and the line address (30 bits), so our 16-entry LIT incurs a storage overheads of only 64 bytes. We recommend that the size of the LIT be increased in proportion to the memory size.

Efficiently Handling LIT Overflows: In the extremely rare cases LIT can overflow, we have two solutions to handle this scenario: (Option-1) Make the LIT memory-mapped (one inversion-bit for every line in memory, stored in memory) and this can support every line in memory having a collision. On marker-collision, the memory system has to make two accesses: one access to the memory, and another to the LIT to resolve collision. Under adversarial settings, the worst-case effect would simply be twice the bandwidth consumption. We implement updates to the LIT by resetting the LIT entry when lines with marker-collisions are brought into the LLC and marking these cachelines as dirty. On eviction, these lines will be forced to go through the marker-collision check and will appropriately set the corresponding LIT entry. (Option-2) On an LIT overflow, PTMC can regenerate new marker values using the random number generator, encode the entire memory with new marker values, and resume execution. As LIT overflows are rare (once per 10 million years), the latency of handling LIT overflows does not affect performance.

Attack-Resilient Marker Codes: The markers in Figure 10 were chosen for simplicity of explanation. We generate per-line markers based on a hash of the line address and the global marker values. However, markers generated from simple address based hash functions can be a target for a Denial-of-Service Attack. An adversary with knowledge of the hash function can write data values intended to cause frequent LIT overflows resulting in severe performance degradation. We address this vulnerability by using a cryptographically secure hash function (e.g. such as DES [20], given that marker generation can happen off-the-critical path) to generate marker values on a per-line basis. This would make the marker values impractical to guess without knowledge of the secret-keys of the hash function, which are generated randomly for each machine. Furthermore, the secret-keys are regenerated in the event of an LIT overflow which changes the per-line markers.

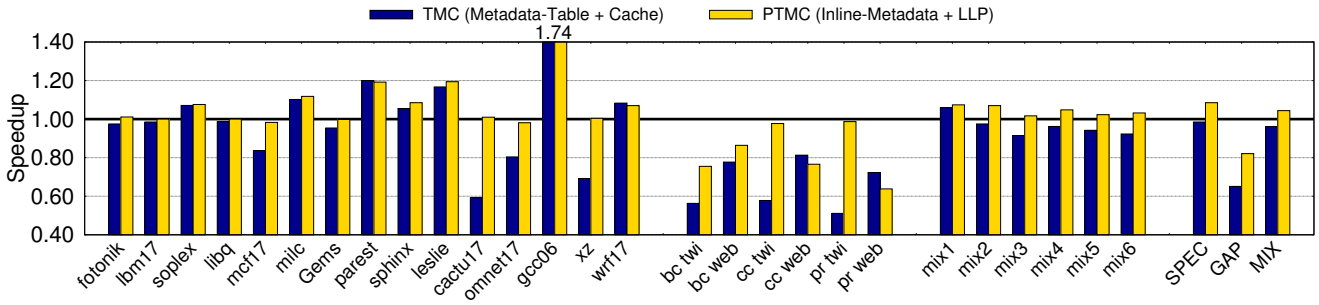

Fig. 12. Speedup of prior TMC designs with metadata-table vs PTMC with inlined-metadata and line-location-predictor, normalized to uncompressed memory. PTMC eliminates metadata lookup and improves performance.

Efficiently Invalidating Stale Copies of Data: Compression can relocate the lines, and, when lines get moved, they can leave behind a potentially stale copy of the line. For example, in Figure 13, if adjacent lines A and B became compressible (into values A' and B'), we could move B' and store lines A' and B' together in one physical location. However, an old value of B would still exist in the previous location. An LLP misprediction can result in an erroneous access to this stale data value, which can still be interpreted as valid. Keeping all locations of the line in sync requires significant bandwidth overheads. Therefore, we simply mark such lines as invalid using a special 64-byte marker value, called *Invalid Line Marker (Marker-IL)*. Marker-IL is also initialized at boot time using a randomly generated value. Per-line Marker-IL can be generated as in Section IV-C. Collisions with Marker-IL are extremely rare (1 in 2^{512} probability, less than one in quadrillion years), and are also handled using line inversion, and are tracked by the LIT.

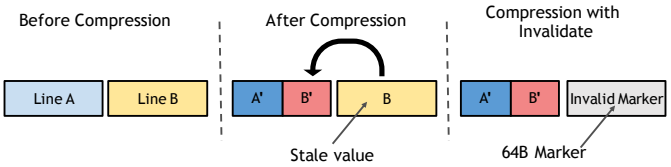

Fig. 13. Compression relocates lines and can create copies of data. We mark such lines as invalid to ensure correct operation.

Handling Updates to Compressed Lines: An update to a compressed line can render the entire group (of two or four lines) from compressible to incompressible. Such updates must be performed carefully so that the data of the other line(s) in the group gets relocated to their original location(s). To accomplish this, we need to know the compressibility of the line when the line was obtained from memory. To track this information, we provision 2-bits in the tag store of the LLC that denotes the compression level when the data was read from memory. On an eviction, we can determine if the lines were previously uncompressed, 2-to-1 compressed, or 4-to-1 compressed by checking these two bits, and we can send writes and invalidates (when applicable) appropriately.

Ganged Eviction: Write-back of a cacheline that belongs to a compressed group can require a read-modify-write operation if the other cachelines in the group are not present in the cache. Our design avoids this by using a ganged-eviction scheme which forces the eviction of all members of a compressed group if one of its members gets evicted. This ensures that all the members of a compressed group are either simultaneously present or absent from the LLC, effectively avoiding the need for read-modify-write. Our evaluations show that ganged eviction has negligible impact on the LLC hit rate.[7]

D. Speedup of PTMC

PTMC, with its inline-metadata and LLP, can accomplish the task of locating and interpreting lines, without the need for a separate metadata lookup. Figure 12 shows the performance of PTMC (with inline-metadata + LLP) compared to a practical implementation of prior TMC designs (with metadata-table and metadata-cache). PTMC eliminates the metadata lookup, which significantly helps both compressible and incompressible workloads. For SPEC and MIX workloads, PTMC provides substantial speedup. However, for Graph (GAP) workloads, PTMC still causes a slowdown. We investigate bandwidth of PTMC to determine the cause.

E. Bandwidth Breakdown of PTMC

Figure 14 shows the bandwidth consumption of PTMC (with inline-metadata + LLP), normalized to uncompressed memory. The components of bandwidth consumption of PTMC are data, second access due to LLP mispredictions, and clean writebacks + invalidates (for writing compressed data). High location prediction accuracy means we are able to effectively remove the cost of metadata lookup, except for *bc_twi*. For Graph workloads, the inherent cost of compression (i.e., compressing and writing back clean lines, and invalidating) is the dominant source of bandwidth overhead and the cause for performance degradation. We develop an effective scheme to disable compression when compression degrades performance.

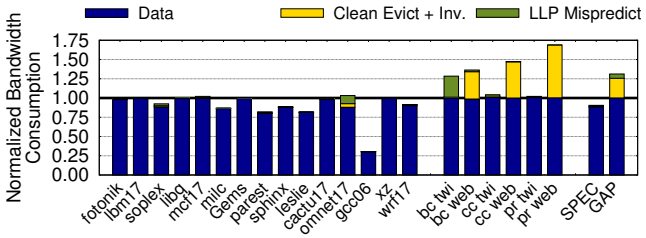

Fig. 14. Bandwidth consumption for PTMC approach, normalized to uncompressed memory.

[7]We have evaluated a more complex scheme that retains lines (requires re-fetching adjacent lines to recompress lines together), and found the difference to be minimal. This is because the remaining lines are often similarly old and likely to be evicted soon after.

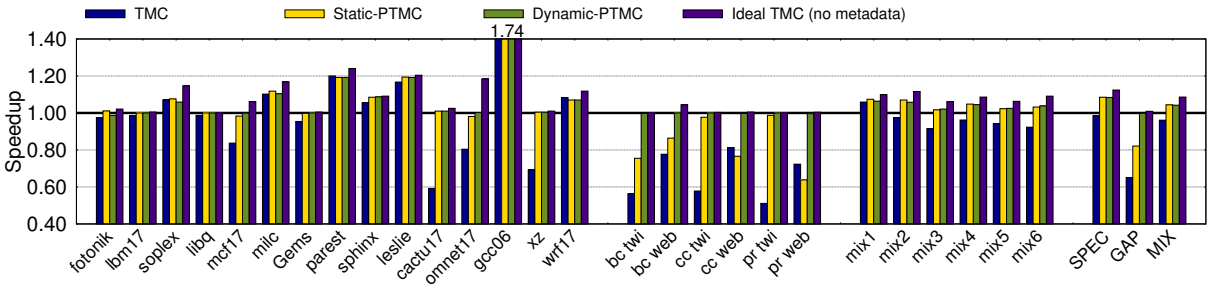

Fig. 15. Speedup of Static-PTMC, Dynamic-PTMC, and Ideal TMC, normalized to uncompressed memory. Dynamic-PTMC avoids slowdown for workloads that do not benefit from compression, and performs similar to Ideal.

V. PTMC: Dynamic Design

Thus far, we have focused only on avoiding the metadata access overheads of compressed memory. However, even after eliminating all of the bandwidth overheads of metadata, there is still slodown for several workloads. Compression requires additional writebacks to memory which can consume additional bandwidth. For example, when a cacheline is found to be compressible on eviction from LLC, it needs to be written back in its compressed form to memory. What could have been a clean evict in an uncompressed memory is now an additional writeback, which becomes a bandwidth overhead.[8] Additionally, PTMC requires invalidates to be sent, which further adds to the bandwidth cost of implementing compression. In general, if the workload has enough reuse and spatial locality, the bandwidth cost of compression yields bandwidth savings in the long run. But for a workload with poor reuse and spatial locality (such as several Graph workloads), the cost of compression does not get recovered, causing performance degradation.

A. Design of Dynamic-PTMC

To get around the inherent cost of compression, we need an effective mechanism to disable compression when the costs of compression outweigh the benefits. Fortunately, due to our robust inline-metadata approach, we can remove the cost of compression by simply deciding to stop actively compressing lines. This is in contrast to prior approaches as they would need to globally decompress memory to disable their major cost of compression, metadata-table access. We simply need an effective mechanism to determine when to disable compression to enable robust performance. We make such a decision by comparing at runtime the "bandwidth cost of doing compression" with the "bandwidth benefits from compression," and enabling/disabling compression based on this cost-benefit analysis. We call this design *Dynamic-PTMC*.

Bandwidth Cost of Compression: The bandwidth overhead of compression comes from sending *extra writebacks* (compressed writebacks from clean locations), sending *invalidates*, and sending requests to *mispredicted* locations. These are additional requests incurred due to compression that could have been avoided if we had used an uncompressed design.

Bandwidth Benefits of Compression: Compression provides bandwidth benefits by enabling bandwidth-free prefetching. On reading a compressed line, adjacent lines get fetched without any extra bandwidth. This saves bandwidth if the prefetched lines are useful. Tracking *useful prefetches* can allow us to determine the benefits from compression.

Dynamic-PTMC monitors the bandwidth costs and benefits of compression at run-time, to determine if compression should be enabled or disabled. To efficiently implement Dynamic-PTMC, we use set-sampling, whereby a small fraction of sets in the LLC (1% in our study) always implement compression and we track the cost-benefit statistics only for the sampled sets. The decision for the remaining (99%) of the sets is determined by the cost-benefit analysis on the sampled sets, as shown in Figure 16. To track the cost and benefit of compression, we use a simple saturating counter. The counter is decremented on seeing the bandwidth *cost* and is incremented on seeing the bandwidth *benefit* of compression. The Most Significant Bit (MSB) of the counter determines if the compression should be enabled or disabled for the remaining sets. We use a 12-bit counter in our design. We extend Dynamic-PTMC to support per-core decision by maintaining per core counters and a 3-bit storage per line in the sampled sets to identify requesting core.

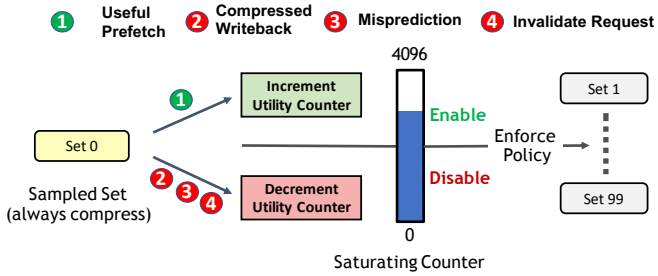

Fig. 16. Dynamic-PTMC analyzes cost-benefit of compression on sampled sets to decide compression policy.

B. Effectiveness of Dynamic-PTMC

Figure 15 shows the speedup of PTMC (that always tries to compress), and Dynamic-PTMC. PTMC without Dynamic optimization provides speedup for SPEC; however, it causes slowdown for GAP. Meanwhile, Dynamic-PTMC eliminates all of the degradations, ensuring robust performance – the design is able to obtain performance when compression is beneficial and avoid degradation when compression is harmful, be it from increased writes or low location-prediction accuracy. Dynamic-PTMC provides speedup of up to 74% (with worst-case slowdown within 1%), nearing two-thirds of the performance of an idealized TMC design that does not incur any bandwidth overheads. Thus, Dynamic-PTMC is a robust and practical way to implement hardware-based main memory compression.

[8]PTMC installs new pages in an uncompressed form to avoid inaccurate prefetches. We have performed evaluations where compressible lines are initialized compressed, but we find many cases where bringing in neighboring but never-used-together lines causes cache pollution and degrades performance. As our primary goal is to design a robust no-hurt memory compression scheme, we opted for a conservative approach of initializing uncompressed.

VI. Results and Analysis

A. Storage Overhead of PTMC Structures

PTMC can be implemented with minor changes at the memory controller. Table III shows the storage overheads required for Dynamic-PTMC. The total storage of the additional structures at the memory controller is less than 300 bytes. In addition to these structures, PTMC needs 2-bits in the tag-store of each line in LLC to track prior-compressibility (<0.1% overhead). And, per-core Dynamic-PTMC needs 4-bits per line in sampled sets (1%) for reuse and core id.

TABLE III
Storage Overhead of PTMC Structures

Structure	Storage Cost
Marker for 2-to-1	4 Bytes
Marker for 4-to-1	4 Bytes
Marker for Invalid Line	64 Bytes
Line Inversion Table (LIT)	64 Bytes
Line Location Predictor (LLP)	128 Bytes
Dynamic-PTMC counter	12 Bytes
Total	276 bytes

B. Extended Evaluation

We perform our study on 27 workloads that are memory intensive. Figure 17 shows the speedup with Dynamic-PTMC across an extended set of 64 workloads (29 SPEC2006, 23 SPEC2017, 6 GAP, and 6 mixes), including ones that are not memory intensive. Dynamic-PTMC is robust in terms of performance, as it avoids degradation for any of the workloads while retaining improvement when compression helps.

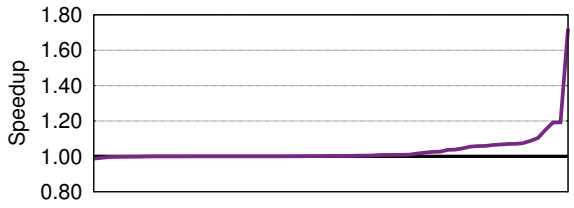

Fig. 17. Speedup of Dynamic-PTMC for 64 workloads, sorted by speedup.

C. Impact on Energy and Power

Figure 18 shows the power, energy consumption and energy-delay-product (EDP) of a system using Dynamic-PTMC, normalized to a baseline uncompressed main memory. Energy consumption is reduced as a consequence of fewer number of requests to main memory. Overall, Dynamic-PTMC reduces energy by 5% and improves EDP by 10%.

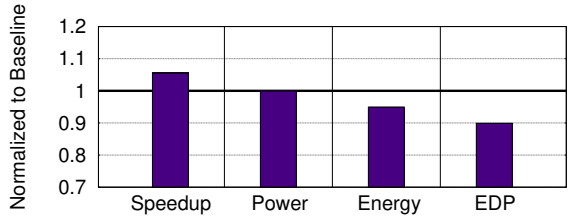

Fig. 18. Dynamic-PTMC impact on energy and power

D. Sensitivity to Memory Channels

PTMC offers bandwidth-free adjacent-line prefetch, which are latency benefits that exist regardless of the number of memory channels. Table IV shows that PTMC provides consistent speedup, even with larger number of channels.

TABLE IV
Sensitivity to Number of Memory Channels

Num. Channels	Avg. SPEC Speedup
1	8.1%
2	**8.5%**
4	7.8%

E. Impact of PTMC on Hit-Rate of L3

Table V shows the hit rate of the L3 cache for the baseline (uncompressed memory), and a system using PTMC. Under PTMC, L3 Hit-rate is improved significantly. Thus, the adjacent lines obtained due to compression with PTMC are useful, and installing them in the L3 cache provides speedup.

TABLE V
Effect of PTMC on L3 Hit-rate

	Baseline	Dynamic-PTMC
SPEC	17.3%	23.9%
GAP	15.7%	15.7%
MIX	13.3%	15.8%

F. Comparison to Larger Fetch for L3

PTMC can install adjacent lines from the memory to the L3 cache. While this may resemble prefetching, we note there is a fundamental difference. PTMC installs additional lines in L3 only when those lines are obtained without bandwidth overhead. Meanwhile, prefetches result in extra memory accesses which incur additional bandwidth. We compare the performance of next-line prefetching and Dynamic-PTMC in Table VI. PTMC avoids the bandwidth overheads of next-line prefetching and provides robust speedup across all workloads.

TABLE VI
Comparison of PTMC to Next-Line Prefetch

	Next-Line Prefetch	Dynamic-PTMC
SPEC	-5.7%	+8.5%
GAP	-21.1%	+0.0%
MIX	-7.3%	+4.2%

G. Applicability to Multi-Socket or DMA

Every access to a particular DRAM channel is managed by its corresponding memory controller, even if the access is on behalf of another socket or a DMA request. As we implement PTMC in the memory controller, every write to or response from DRAM can be intercepted and inverted appropriately. Thus, PTMC works even in multi-socket or DMA scenarios.

VII. Related Work

To the best of our knowledge, this is the first paper to propose a robust hardware-based main-memory compression for bandwidth improvement, without requiring any OS-support and without causing changes to the memory organization and protocols. We discuss prior research related to our study.

A. Low-Latency Compression Algorithms

As decompression latency is in the critical path of memory accesses, hardware compression techniques typically use simple per-line compression schemes [6] [7] [8] [9] [10] [11] [12]. We evaluate PTMC using a hybrid compression using FPC [6] and BDI [10]. However, PTMC is orthogonal to compression algorithm and can be implemented with any compression algorithm, including dictionary-based [21] [22] [23] [24] [25].

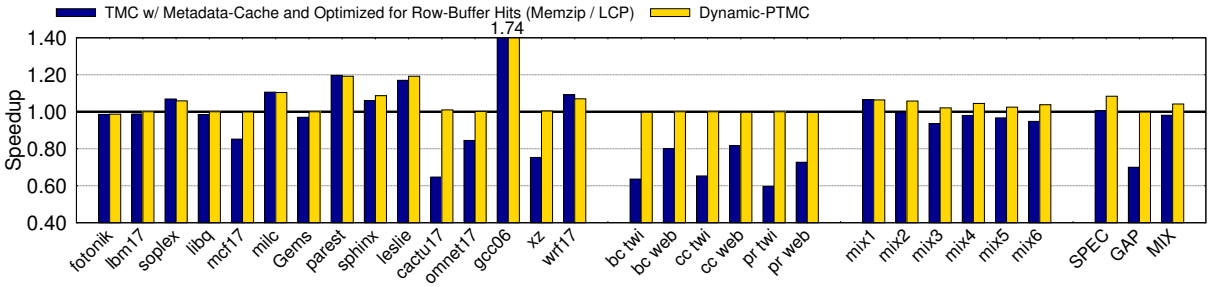

Fig. 19. Performance of prior memory compression schemes that use a metadata-table (Memzip, LCP), optimized for row-buffer hits and using a 32KB metadata cache. Prior approaches require significant bandwidth to retrieve metadata.

B. Main Memory Compression for Bandwidth

Hardware-based memory compression has been applied to improve the bandwidth of main memory [5] [26] [27] [28]. These works attempt to save memory bandwidth by sending compressed data across links in fewer number of bursts. However, commodity memory has a minimum access granularity, so such proposals can achieve bandwidth savings only if their systems use non-traditional DRAM organizations (such as MiniRank [29] [30] [31]) or modify the bus protocols or both. Meanwhile, PTMC enables compressed memory systems using existing memory devices and protocols.

Prior studies [3] [5] advocate reducing the latency for metadata lookups by placing the metadata in the same row as the data, and using a metadata-cache to filter metadata lookups. However, improving row buffer locality does not reduce the raw bandwidth required to obtain metadata – such designs still suffer from significant bandwidth problems. For comparison, we implement a TMC with metadata optimized to reside in the same row as data, and also using a 32KB metadata cache. Figure 19 compares the performance of Dynamic-PTMC to prior row-buffer-optimized designs. The bandwidth overhead to obtain metadata is still significant, causing slowdown. Whereas, Dynamic-PTMC improves performance.

The idea of in-lining metadata using a marker value was developed independently and concurrently[9] by the recent work called Attaché [28]. While PTMC and Attaché [28] share the similarity of using in-line metadata and a predictor for estimating compressibility, there are three key differences. First, Attaché requires non-commodity memories (mini-rank) to save bandwidth, whereas PTMC is built using commodity memories. Second, Attaché saves bandwidth directly by reducing the number of bursts, whereas PTMC uses a constant number of bursts and relies on spatial locality to provide bandwidth reduction. Third, PTMC proposes a novel dynamic mechanism that can disable compression when compression hurts (due to extra writes or poor location-prediction accuracy) whereas Attaché did not consist of any such mechanism.

C. Main Memory Compression for Capacity or Reliability

Hardware-based memory compression has been applied to increase the capacity of main memory [1] [2] [3]. To locate the line, these approaches extend the page table entries to include information on the compressibility of the page. These approaches are attractive as they allow locating and interpreting lines using the TLB. However, such approaches inherently require software-support (from the OS or hypervisor) that limit their applicability. We want a design that can be built entirely in hardware, without any OS support.

COP [33] proposes in-lining ECC into compressed lines and uses the ECC as markers to identify compressed lines. Unfortunately, COP is designed to provide reliability at low cost and provides no performance benefit if the system does not need ECC or already has an ECC-DIMM. Whereas, PTMC is designed to provide bandwidth benefits by fetching multiple lines, and helps regardless of whether the system has ECC-DIMM or not. Furthermore, COP relies on a fairly complex mechanism to handle marker collisions (locking lines in the LLC, memory-mapped linked-list etc.), whereas, PTMC handles marker collisions efficiently via data inversion.

D. SRAM-Cache Compression

Prior work has looked at using compression to increase capacity of on-chip SRAM caches. Cache compression is typically done by accommodating more ways in a cache set and statically allocating more tags [11] [34]. Recent proposals, such as SCC, investigate reducing SRAM tag overhead by sharing tags across sets [35] [36] [37] [38]. Compressed caches typically obtain compression metadata by storing it beside the tag and retrieving them along with tag accesses. These approaches do not scale for memory, as there is no tag space or tag lookup to enable easy metadata access.

The address mapping in PTMC is inspired by the placement in SCC [36], in that the location of the line gets determined by compressibility. However, unlike SCC, our placement ensures that a significant fraction of lines do not change their locations, regardless of their compression status. Furthermore, SCC requires skewed-associative lookup of all possible positions, which is possible to do in a cache; however, such probes of all possible placement locations would incur intolerable bandwidth overheads in main memory.

E. Adaptive Cache-Compression

Prior works have looked at adaptive or dynamic cache compression [11] [34] [39] [40] to avoid performance degradation due to latency overheads of decompression or due to extra misses caused by sub-optimal replacement in compressed caches. These designs primarily target cache hit rate. Whereas, our main memory proposal targets bandwidth overheads inherent in memory compression (metadata or compressed writes). Additionally, fine-grain adaptive memory compression has been previously unexplored, as prior approaches have had no capability to turn off compression.

[9] An earlier version of our paper was submitted to MICRO 2018 [32].

F. Predicting Cache Indices

Several studies have looked at predicting indices in associative caches [41] [42] [43] [44] [45] [46] [47] [48] [18]. A cache can verify such predictions simply by checking the tag, and issuing a second request in case of a misprediction. Our work is quite different from these, in that we try to predict the location for *memory*. Since memory does not provide tags to identify the data like caches, our solution is to inline the compressibility information of the line using Inline-Metadata. The Line Location Predictor utilizes this inline-metadata to verify the location prediction and issue a request to an alternate location on a misprediction.

VIII. Conclusions

This paper proposes *PTMC (Practical and Transparent Memory-Compression)*, a simple design for obtaining bandwidth benefits of memory compression while relying only on commodity (non-ECC) memory modules and avoiding any OS support. Our design uses a novel *inline-metadata* mechanism, whereby the compressibility of the line can be determined by scanning the line for a special *marker* word, eliminating the need for a separate metadata access. We also develop a low-cost *Line Location Predictor (LLP)* that can determine the location of the line with 98% accuracy. Finally, we develop a dynamic solution that disables compression if the cost of compression is higher than the benefits. Our evaluations show that PTMC provides a speedup of up to 73%, is robust (no slowdown for any workload), and can be implemented with a storage overhead of less than 300 bytes.

Acknowledgements

We thank the anonymous reviewers and our colleagues from the Memory Systems Lab for their feedback. This work was partially supported by a gift from Intel and from a grant from the Semiconductor Research Corporation (SRC).

References

[1] B. Abali *et al.*, "Performance of hardware compressed main memory," in *HPCA*, 2001.
[2] M. Ekman and P. Stenstrom, "A robust main-memory compression scheme," in *ISCA*, 2005.
[3] G. Pekhimenko *et al.*, "Linearly compressed pages: A low-complexity, low-latency main memory compression framework," in *MICRO*, 2013.
[4] Qualcomm, "Qualcomm centriq 2400 processor," https://www.qualcomm.com/media/documents/files/qualcomm-centriq-2400-processor.pdf, 2017.
[5] A. Shafiee *et al.*, "Memzip: Exploring unconventional benefits from memory compression," in *HPCA*, 2014.
[6] A.R. Alameldeen and D.A. Wood, "Frequent pattern compression: A significance-based compression scheme for l2 caches," *Dept. Comp. Scie., Univ. Wisconsin-Madison, Tech. Rep*, vol. 1500, 2004.
[7] J. Dusser, T. Piquet, and A. Seznec, "Zero-content augmented caches," in *ICS*, 2009.
[8] Y. Zhang, J. Yang, and R. Gupta, "Frequent value locality and value-centric data cache design," in *SIGOPS*, 2000.
[9] J. Yang, Y. Zhang, and R. Gupta, "Frequent value compression in data caches," in *MICRO*, 2000.
[10] G. Pekhimenko *et al.*, "Base-delta-immediate compression: practical data compression for on-chip caches," in *PACT*, 2012.
[11] A.R. Alameldeen, D. Wood *et al.*, "Adaptive cache compression for high-performance processors," in *ISCA*, 2004.
[12] J. Kim *et al.*, "Bit-plane compression: Transforming delta for better compression in many-core architectures," in *ISCA*, 2016.
[13] N. Chatterjee *et al.*, "Usimm: the utah simulated memory module," *University of Utah, Tech. Rep*, 2012.
[14] H. Patil *et al.*, "Pinpointing representative portions of large intel itanium programs with dynamic instrumentation," in *MICRO*, 2004.
[15] J.L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, Sep. 2006.
[16] S.P.E. Corporation, "Spec cpu 2017," 2017, accessed: 2017-11-10.
[17] S. Beamer, K. Asanovic, and D.A. Patterson, "The GAP benchmark suite," *CoRR*, vol. abs/1508.03619, 2015.
[18] V. Young, P.J. Nair, and M.K. Qureshi, "Dice: Compressing dram caches for bandwidth and capacity," in *ISCA*, 2017.
[19] S. Sardashti and D.A. Wood, "Could compression be of general use? evaluating memory compression across domains," *TACO*, 2017.
[20] D. Coppersmith, "The data encryption standard (des) and its strength against attacks," *IBM J. Res. Dev.*, vol. 38, May 1994.
[21] X. Chen *et al.*, "C-pack: A high-performance microprocessor cache compression algorithm," *VLSI*, 2010.
[22] T.M. Nguyen and D. Wentzlaff, "Morc: A manycore-oriented compressed cache," in *MICRO*, 2015.
[23] A. Arelakis and P. Stenstrom, "Sc2: A statistical compression cache scheme," in *ISCA*, 2014.
[24] A. Arelakis, F. Dahlgren, and P. Stenstrom, "Hycomp: A hybrid cache compression method for selection of data-type-specific compression methods," in *MICRO*, 2015.
[25] Y. Tian *et al.*, "Last-level cache deduplication," in *ICS*, 2014.
[26] V. Sathish, M.J. Schulte, and N.S. Kim, "Lossless and lossy memory i/o link compression for improving performance of gpgpu workloads," in *PACT*, 2012.
[27] H. Kim *et al.*, "Reducing network-on-chip energy consumption through spatial locality speculation," in *NOCS*, 2011.
[28] M.H.S. Hong *et al.*, "Attaché: Towards ideal memory compression by mitigating metadata bandwidth overheads," in *MICRO*, 2018.
[29] H. Zheng *et al.*, "Mini-rank: Adaptive dram architecture for improving memory power efficiency," in *MICRO*, 2008.
[30] D.H. Yoon *et al.*, "Adaptive granularity memory systems: A tradeoff between storage efficiency and throughput," in *ISCA*, 2011.
[31] J.H. Ahn *et al.*, "Future scaling of processor-memory interfaces," in *SC*, 2009.
[32] V. Young, S. Kariyappa, and M. Qureshi, "CRAM: Efficient Hardware-Based Memory Compression for Bandwidth Enhancement," https://arxiv.org/abs/1807.07685, July 2018.
[33] D.J. Palframan, N.S. Kim, and M.H. Lipasti, "Cop: To compress and protect main memory," in *ISCA*, 2015.
[34] J. Guar, A.R. Alameldeen, and S. Subramoney, "Base-victim compression: An opportunistic cache compression architecture," in *ISCA*, 2016.
[35] S. Sardashti and D.A. Wood, "Decoupled compressed cache: Exploiting spatial locality for energy-optimized compressed caching," in *MICRO*, 2013.
[36] S. Sardashti *et al.*, "Skewed compressed caches," in *MICRO*, 2014.
[37] B. Panda and A. Seznec, "Dictionary sharing: An efficient cache compression scheme for compressed caches," in *MICRO*, 2016.
[38] S. Sardashti, A. Seznec, and D.A. Wood, "Yet another compressed cache: A low-cost yet effective compressed cache," *TACO*, 2016.
[39] Y. Xie and G.H. Loh, "Thread-aware dynamic shared cache compression in multi-core processors," in *ICCD*, 2011.
[40] S. Kim *et al.*, "Transparent dual memory compression architecture," in *PACT*, 2017.
[41] A. Agarwal, J. Hennessy, and M. Horowitz, "Cache performance of operating system and multiprogramming workloads," *TOCS*, 1988.
[42] A. Agarwal and S.D. Pudar, "Column-associative caches: A technique for reducing the miss rate of direct-mapped caches," *ISCA*, 1993.
[43] J.J. Valls *et al.*, "Ps-cache: An energy-efficient cache design for chip multiprocessors," *J. Supercomput.*, 2015.
[44] B. Calder, D. Grunwald, and J. Emer, "Predictive sequential associative cache," in *HPCA*, 1996.
[45] D.H. Albonesi, "Selective cache ways: On-demand cache resource allocation," in *MICRO*, 1999.
[46] M.D. Powell *et al.*, "Reducing set-associative cache energy via way-prediction and selective direct-mapping," in *MICRO*, 2001.
[47] H.C. Chen and J.S. Chiang, "Low-power way-predicting cache using valid-bit pre-decision for parallel architectures," in *AINA*, 2005.
[48] A. Deb *et al.*, "Enabling technologies for memory compression: Metadata, mapping, and prediction," in *ICCD*, 2016.