DICE: Compressing DRAM Caches for Bandwidth and Capacity

Vinson Young Prashant J. Nair Moinuddin K. Qureshi
School of Electrical and Computer Engineering, Georgia Institute of Technology
{vyoung,pnair6,moin}@gatech.edu

ABSTRACT

This paper investigates compression for DRAM caches. As the capacity of DRAM cache is typically large, prior techniques on cache compression, which solely focus on improving cache capacity, provide only a marginal benefit. We show that more performance benefit can be obtained if the compression of the DRAM cache is tailored to provide higher bandwidth. If a DRAM cache can provide two compressed lines in a single access, and both lines are useful, the effective bandwidth of the DRAM cache would double. Unfortunately, it is not straight-forward to compress DRAM caches for bandwidth. The typically used Traditional Set Indexing (TSI) maps consecutive lines to consecutive sets, so the multiple compressed lines obtained from the set are from spatially distant locations and unlikely to be used within a short period of each other. We can change the indexing of the cache to place consecutive lines in the same set to improve bandwidth; however, when the data is incompressible, such spatial indexing reduces effective capacity and causes significant slowdown.

Ideally, we would like to have spatial indexing when the data is compressible and TSI otherwise. To this end, we propose *Dynamic-Indexing Cache comprEssion (DICE)*, a dynamic design that can adapt between spatial indexing and TSI, depending on the compressibility of the data. We also propose low-cost *Cache Index Predictors (CIP)* that can accurately predict the cache indexing scheme on access in order to avoid probing both indices for retrieving a given cache line. Our studies with a 1GB DRAM cache, on a wide range of workloads (including SPEC and Graph), show that DICE improves performance by 19.0% and reduces energy-delay-product by 36% on average. DICE is within 3% of a design that has double the capacity and double the bandwidth. DICE incurs a storage overhead of less than 1KB and does not rely on any OS support.

CCS CONCEPTS

• **Hardware** → **Memory and dense storage**;

KEYWORDS

Stacked DRAM, compression, bandwidth, memory.

ACM Reference format:
Vinson Young Prashant J. Nair Moinuddin K. Qureshi. 2017. DICE: Compressing DRAM Caches for Bandwidth and Capacity. In *Proceedings of ISCA '17, Toronto, ON, Canada, June 24-28, 2017,* 12 pages.
https://doi.org/10.1145/3079856.3080243

ISCA '17, June 24-28, 2017, Toronto, ON, Canada

ACM ISBN 978-1-4503-4892-8/17/06…$15.00
https://doi.org/10.1145/3079856.3080243

1 INTRODUCTION

As modern compute systems pack more and more cores on the processor chip, their memory systems must scale proportionally in terms of both capacity and bandwidth in order to store and supply data to all the cores. Advancements in packaging and interconnect technology have enabled stacking several DRAM modules thereby offering 4-8x higher bandwidth than conventional DIMM-based DDR memories. Examples of DRAM based stacked memory technology include High Bandwidth Memory, Hybrid Memory Cube, and MCDRAM in Intel's Knights Landing [25, 40, 41]. However, while stacked memories offer 4-8x higher bandwidth, these technologies do not yet have enough capacity to fully replace conventional DDR-based DIMMs. So, future memory systems are likely to consist of heterogeneous organizations that use both high-bandwidth stacked memories and high-capacity DDR memories. An attractive option is to architect stacked DRAM as a DRAM cache and place it between on-die caches and DDR-based DIMMs [12–15, 19, 21, 22, 24, 32, 39].

Architecting stacked DRAM as a hardware managed cache has several challenges, including designing and accessing a tag storage of several megabytes. For example, a 1GB DRAM cache contains 16 million lines, which would need 64MB of storage for tags. Therefore, practical designs of gigascale DRAM caches place tags inlined with data in the DRAM array [32, 40], and organize the cache as a direct-mapped cache to reduce lookup latency. We notice that as any given bit within the DRAM cache can act as a tag bit or data bit, we can implement data compression within DRAM caches inexpensively. Extra tags needed for accommodating compressed lines can be dynamically allocated in the DRAM. Therefore, compression is well-suited for DRAM caches as it can be implemented at low cost.

Several prior proposals [4, 34, 35] have looked at compression in the context of SRAM caches. These proposals focus solely on increasing the effective capacity of the cache as a means to improve performance. However, if the cache is large enough to hold the uncompressed working set of the applications, then these proposals would not provide any performance benefit. As the size of the DRAM cache is typically quite large compared to an SRAM cache, prior schemes [8, 16] that focus solely on increasing the capacity of DRAM cache have limited performance benefit. Besides compression for capacity, we can also use compression for bandwidth. For example, if a compressed DRAM cache can provide two useful lines per each access, then the effective bandwidth of the DRAM cache would double. Furthermore, compressing the DRAM cache for bandwidth can still provide performance even if the DRAM cache is large enough to hold the working set of the application. In this paper, we advocate compressing DRAM cache primarily for bandwidth and secondarily for capacity. Unfortunately, compressing DRAM caches for bandwidth is not straight-forward.

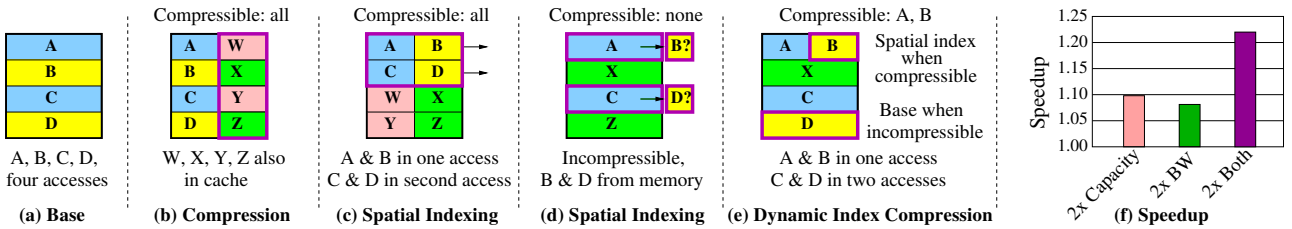

Figure 1: Considerations in compressing DRAM caches (a) Baseline system with four lines (A-D) (b) Compression for capacity (c) Spatial indexing for bandwidth (d) Slowdown from spatial indexing when data is incompressible (e) Dynamic index compression based on compressibility (A, B use spatial index) (f) Potential speedup from doubling DRAM cache capacity, bandwidth, and both.

We explain the considerations in compressing DRAM caches with an example. Figure 1(a) shows the baseline uncompressed cache storing lines A-D. The baseline uses Traditional Set Indexing (TSI) that maps consecutive lines to consecutive sets. There are four more lines (W-Z) in the working set that are used less frequently than A-D. A straight-forward method to compress DRAM caches is to compress the lines that map to the same set together, if they both can be compressed to within the same set. If the data is compressible, we can expect all the eight lines (A-D and W-Z) to be resident in the cache, as shown in Figure 1(b). A single access to the cache can obtain two lines (A and W for example). However, even though we can get two lines with one access, such a design compresses purely for capacity, as two lines mapping to the same set (A and W for example) would spatially be GBs apart in main memory – and hence are unlikely to be used within a short period of each other. Our studies show that compressing DRAM cache only for capacity gives limited performance improvement (7%). Given that a DRAM cache provides disproportional bandwidth as compared to its capacity, if we can compress for both capacity and bandwidth, then as shown in Figure 1(f), we can potentially get much higher performance (22%).

To obtain higher effective bandwidth, it would be desirable to obtain two spatially neighboring lines in one access from the DRAM cache. Figure 1(c) shows a spatial indexing scheme that maps two consecutive lines into the same set to get more bandwidth and capacity. When lines are compressible, on an access to A, both line A and line B are received with one access, improving effective bandwidth. However, if lines are incompressible, in Figure 1(d), only one of A and B can be resident at a time, and the other would be fetched from memory, degrading performance. Ideally, we want to use spatial indexing only when data is compressible, and fallback to TSI otherwise, as in Figure 1(e). To this end, we propose *Dynamic-Indexing Cache Compression (DICE)*, a design that dynamically switches between spatial indexing and TSI based on compressibility. DICE uses spatial indexing when data is compressible and TSI otherwise.

As DICE supports two indexing schemes, switching between spatial indexing and TSI should ideally be quick. We propose a novel indexing scheme called *Bandwidth Aware Indexing (BAI)* that maps consecutive lines into the same set, while ensuring that half of the lines remain in same location as TSI. For the half of lines where BAI is different from TSI, the line could be in either of the two locations (depending on BAI or TSI). For such lines, DICE uses the compressed size of the line to determine if the line should be installed with BAI or TSI. If the compressed line is smaller than a given threshold (36B), it is installed using BAI, otherwise using TSI.

On a cache access, the line could potentially be in two separate sets, depending on the indexing scheme. It would be bandwidth inefficient to check for the line in two locations on every access. To overcome this, we propose *Cache Indexing Predictors (CIP)*, that can predict the right indexing scheme for the given access. We find that compressibility is heavily correlated for lines in a given page, so if we have a table that keeps track of the last index policy used for a given page, we can get high accuracy (94%) at low storage overheads (<1KB). DICE employs history-based CIP for index prediction on cache accesses, and performs the lookup of the second location only on misprediction.

We evaluate DICE on a wide variety of workloads (SPEC, GAP), and find that DICE is robust and improves performance across workloads of varying compressibility, working set sizes, and access patterns. Our studies on a 1GB DRAM cache show that DICE provides on average 19% speedup and 36% reduction in energy-delay-product, approaching 21.9% speedup of a double-capacity double-bandwidth cache, while requiring less than 1 kilobyte of storage overhead and no change to OS.

Overall, this paper makes the following contributions:

(1) We advocate that the compression of DRAM cache should focus on obtaining not only the capacity benefits but also the bandwidth benefits. We show how compression can be implemented on DRAM cache without incurring extra storage for tags, needing multiple accesses for getting tags, or requiring OS support.

(2) To obtain bandwidth benefits, we propose *Bandwidth-Aware Indexing (BAI)*, a scheme that maps consecutive lines in the same cache set, and improves the effective bandwidth. One of the nice property of BAI is that it ensures that half of the lines in the cache can still reside in the same location as with Traditional Set Indexing (TSI), facilitating dynamic adaptation of indexing schemes.

(3) We show that while BAI provides benefits for compressible workloads, it can degrade performance for incompressible workloads. To this end, we propose a dynamic scheme called *Dynamic-Indexing Cache Compression (DICE)* that employs BAI and TSI based on data compressibility.

(4) We also propose low-cost and accurate (<1KB) cache index predictors (CIP) to predict the cache indexing scheme used for a given access. CIP assists each access by avoiding looking up two adjacent potential locations where the given cache line may reside.

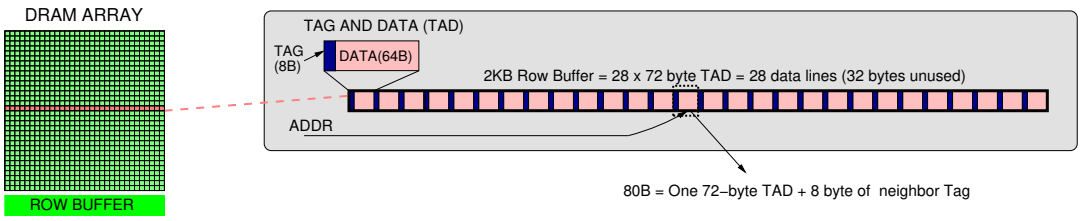

Figure 2: Organization of DRAM Cache configured as Alloy Cache. Each access indexes the direct-mapped location and obtains one Tag and Data (TAD) entry for low hit latency.

2 BACKGROUND AND MOTIVATION

The emerging stacked DRAM technology allows the creation of DRAM caches, as an intermediate level between on-chip caches and main memory [24][32][22][21][40]. We investigate compression for increasing the logical capacity of these caches as well as to improve overall bandwidth. In this section, we discuss background on on-chip cache compression, main memory compression, and the organization of typical DRAM caches, which makes it amenable to implement compression in a relatively inexpensive and straight-forward manner.

2.1 Compressing On-Chip SRAM Caches

Compression exploits redundancy in data values to increase effective capacity of a given substrate. Prior work has looked at compression for improving the capacity of SRAM caches [4, 28, 34, 35]. As decompression latency is in the critical path of cache access, these proposals use simple compression schemes such as Frequent Pattern Compression (FPC) [5], Base-Delta-Immediate (BDI) [31], CPACK [11], and ZCA [17], that can perform decompression within a minimal number of cycles. In a typical compressed cache design [4], the cache line is compressed into well-defined segments (e.g. one-fourth or one-half line), and the cache is provisioned with extra tag-store entries to reference the extra lines that can be stored (18% storage overhead for 4x ways).[1] The extra tags allow the cache to index up to 4x number of lines than before. However, we note that SRAM designs compress only for capacity (sometimes at the cost of bandwidth). If we could improve both capacity and bandwidth, we could get much higher performance.

2.2 Compressing Main Memory

Compression has also been applied to main memory for increasing memory capacity [1]. Recent work in hardware-assisted main-memory compression has looked at implementing memory compression with low-latency and high bandwidth. For example, Linearly Compressed Pages (LCP) [30] proposes to spatially compress all lines in a page to one-fourth their size and store exception storage elsewhere if lines cannot be compressed to one-fourth the size.

One of the primary advantages of LCP is that a single memory access can obtain multiple memory lines, when data is compressible. Therefore, LCP can utilize compression not only for capacity but also for bandwidth benefits. Unfortunately, the page mapping and organization of LCP must be done using the OS, as the OS is required to know the compressed page size, in order to access the adjusted offset in the page correctly and in order to use physical main memory capacity fully. In addition, when data is incompressible, the main memory must send a second request to obtain data from the exception storage, which incurs both latency and bandwidth overheads. Thus, the disadvantages of LCP are that it requires significant OS support, and it has costly handling of incompressible lines. Ideally, we would like to get benefits of compression without relying on OS support, or suffering high-latency access when data is not compressible.

[1] Recent studies on SRAM cache compression propose sharing tags between a larger number of sets (say 4x sets, called superblocks) to reduce tag and metadata overhead [28, 34, 35]. However, when applied on DRAM caches, these designs increase the number of sets that must be checked on each access, which can waste bandwidth. We analyze these proposals in detail in Section 7.3.

2.3 Organization of Practical DRAM Caches

Recent research has looked at enabling fine-grained (64B line size) DRAM caches in a low cost manner [24, 32]. Given that the tag-store required for these caches often is in the range of several tens of MB, these studies propose to co-locate tag-store entries with the data line. Without loss of generality, in this paper, we consider Alloy Cache as a representative example of DRAM cache organization, given that a similar "direct-mapped, 64B linesize, tags part of line" design is used for the DRAM cache in Knights Landing [40].

Alloy Cache architects the cache as a direct-mapped structure and alloys the tag and data together to form a single entity called *Tag and Data (TAD)*, as shown in Figure 2. On a cache access, the TAD entry corresponding to the set of the direct-mapped cache is transferred. On a tag hit, data is obtained from the TAD without the need for an additional access to the DRAM cache. The size of the TAD is assumed to be 72 bytes (8B tag plus 64B data). Infact, given the bus of stacked DRAM is 16B, the cache transfers 80B, so the tag-entry of the neighboring line is obtained for a given access without additional bandwidth overheads. Thus, Alloy Cache design obviates need for SRAM-based tag storage. In addition, designs that store tag within the data array lend themselves to additional optimizations as the controller has the freedom to interpret bits as either tag bits or data bits.

2.4 Compressing DRAM Cache is Almost Free

While DRAM caches have much larger capacity than on-chip caches, there is still performance benefit if we can improve capacity. As shown in Figure 1(f), doubling the capacity of the DRAM cache could potentially provide an improvement of about 10%, on average. We note that DRAM caches are provided mainly to improve the system bandwidth. If we could use compression to increase the effective bandwidth of the DRAM cache as well, then we could get even higher performance benefit. For example, doubling the capacity and bandwidth of the DRAM cache can provide 22% performance on average. Fortunately, the organization and hardware-management of DRAM caches avoid the limitations of on-chip SRAM cache compression and main-memory compression, as shown in Table 1.

Table 1: Comparison of different forms of compression

Module to Compress	Improve Capacity Only?	Tag Overhead?	OS support Needed?
On-Chip Cache	Yes	Yes	No
Main Memory	No	No	Yes
DRAM Cache	No	No	No

Unlike SRAM caches, DRAM caches perform similar to DIMM-based DRAM counterparts and are sensitive to bandwidth. Designs that improve only capacity quickly meet diminishing returns on a 1GB DRAM cache. As such, we take inspiration from main-memory compression to compress for bandwidth.

Unlike compression for SRAM caches, DRAM cache compression can be done without requiring any extra storage for the additional tags. The extra tag-store entries can be created dynamically within the DRAM array, as the memory controller has the freedom to interpret any bit as either a tag bit or a data bit. This allows the compressed DRAM cache to hold several compressed lines in a given set, without being constrained by the size of the tag-store.

Unlike main memory compression, which requires support from the OS to maintain page mapping and to evict pages in case data is incompressible, DRAM caches can be managed entirely in a software-transparent manner.

Thus, we can implement compression on DRAM caches for almost free as the tag-store needed to support extra capacity can be obtained from the DRAM array. We would simply need compression and decompression logic to enable compression for DRAM caches. While implementing cache compression for DRAM for capacity is straight-forward, we would ideally want a design that provides both capacity and bandwidth benefits. This paper proposes such a design. We discuss our methodology before we present our solution.

3 METHODOLOGY

3.1 Configuration

We use USIMM [10], an x86 simulator with detailed memory system model. We modified USIMM to include a DRAM cache. Table 2 shows the configuration used in our study. We assume a four-level cache hierarchy (L1, L2, L3 being on-chip SRAM caches and L4 being off-chip DRAM cache), with 64B line size. We model a virtual memory system to perform virtual to physical address translations.

Table 2: Baseline Configuration

Processors	
Number of cores	8 cores
Core type	4-wide 3.2GHz out-of-order
L1/L2 (private)	32KB/256KB (8-way each)
L3 cache (shared)	8MB (1MB per core)
DRAM Cache	
Capacity	1GB
Configuration	4 channel, 128-bit bus
Bus Frequency	800MHz (DDR 1.6GHz)
Banks	16 banks per channel
tCAS-tRCD-tRP-tRAS	44-44-44-112 CPU cycles
Read/Write Queue	96 entries per channel
Main Memory (DDR DRAM)	
Capacity	32GB
Configuration	1 channel, 64-bit bus
Bus Frequency	800MHz (DDR 1.6GHz)
Banks	16 banks per channel
tCAS-tRCD-tRP-tRAS	44-44-44-112 CPU cycles
Read/Write Queue	96 entries

We use Alloy Cache for the L4 cache, and results are normalized to Alloy Cache unless stated otherwise. Cache misses fill all levels of the hierarchy. We equip Alloy Cache with a MAP-I predictor to overcome tag lookup latency for cache misses. We assume a heterogeneous memory system with DRAM cache using HBM technology [41] and main memory using conventional DDR-based DIMM technology, corresponding to $1/8^{th}$ scale of Knights Landing [40]. In accordance with stacked memory specifications, we assume same access latency for both DRAM technologies. However, the bandwidth of stacked-DRAM is 8x higher than main-memory, with 4x channels and 2x bus width.

3.2 Workloads

We use a representative slice of 4 billion instructions selected by PinPoints [29], for benchmarks from SPEC 2006 and GAP [9]. For SPEC, we perform studies on all the 16 benchmarks that have at least 2 Miss Per Thousand Instructions (MPKI) out of L3 cache. In addition, we run GAP suite (Graph Algorithm Platform) to show server workloads with real data sets (twitter, web sk-2005). We run all 30 suggested configurations, and present a sample where speedup is representative of GAP suite. We perform evaluations by executing benchmarks in rate mode, where all eight cores execute the same benchmark. In addition to rate-mode workloads, we also evaluate four 8-thread mixed workloads, which are created by randomly choosing 8 out of the 16 SPEC benchmarks. Table 3 shows the L3 miss rates and footprints of the 8-core rate-mode workloads used in our study.

Table 3: Workload Characteristics

Suite	Workload (8-copies)	L3 MPKI	Footprint
SPEC	mcf	53.6	13.2 GB
	lbm	27.5	3.2 GB
	soplex	26.8	1.9 GB
	milc	25.7	2.9 GB
	gcc	22.7	264 MB
	libq	22.2	256 MB
	Gems	17.2	6.4 GB
	omnetpp	16.4	1.3 GB
	leslie3d	14.6	624 MB
	sphinx	12.9	128 MB
	zeusmp	5.2	2.9 GB
	wrf	5.1	1.4 GB
	cactus	4.9	3.3 GB
	astar	4.5	1.1 GB
	bzip2	3.6	2.5 GB
	xalanc	2.2	1.9 GB
GAP	bc twitter	69.7	19.7 GB
	bc web	17.7	25.0 GB
	cc twitter	93.9	14.3 GB
	cc web	9.4	16.0 GB
	pr twitter	112.9	23.1 GB
	pr web	16.7	25.2 GB

We perform timing simulation until all benchmarks in the workload execute at least 4 billion instructions each. We use weighted speedup to measure aggregate performance of the workload normalized to baseline. We use geometric mean to report average speedup across workloads, and use RATE to denote average over 16 spec rate-mode workloads, MIX for the 4 mixed workloads, GAP for its 6 workloads, and ALL26 to denote average over all 26 workloads.

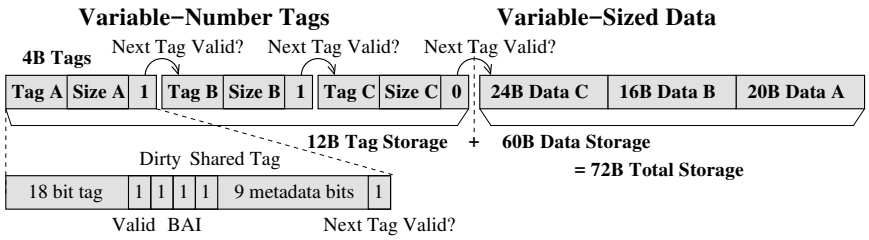

Figure 5: Proposed format for storing multiple compressed lines in 72B Alloy set. Space for tags can be allocated as needed (up to 28).

4 COMPRESSED DRAM CACHE

4.1 Overview: Organization and Working

As practical implementations of DRAM caches use the same DRAM array for storing tags as well as data, we can simply architect a compressed DRAM cache by providing L4 cache controller with compression and decompression logic as shown in Figure 3. In our compressed DRAM cache design, only the L4 is compressed, and the data in other parts of the memory system (such as the L3 cache or main memory) remain in normal uncompressed form; thus, cache compression can be implemented local to L4 cache controller without requiring changes to the other parts of the system.

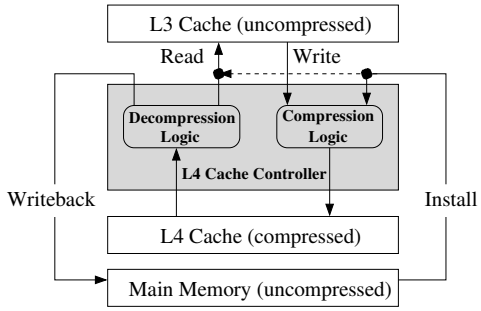

Figure 3: Design of compressed DRAM-cache. Compression can be implemented with L4-controller-local changes.

On an L3 read from the compressed L4 cache, the L4 controller obtains a 72B TAD from the Alloy Cache. Decompressing the TAD can provide multiple lines with a single access. The system can decide to install these lines in L3, or only the requested line.

The L4 cache controller compresses data before L3 writebacks to L4, and before L4 installs from main memory. For writes, the L4 cache controller compresses the data to see how much space is required to store the line, and reads from L4 cache to check what lines are resident. If the compressed line can be stored in the unused space of the 72B TAD, it is appended and written. If the compressed line cannot fit, then resident entries are evicted (and written back to memory if dirty) until enough space is made available for the line.

4.2 Potential for Compression

In our evaluations, we use two low-latency compression algorithms Frequent Pattern Compression[5] and Base-Delta-Immediate[31]. FPC and BDI's decompression latency is expected to be 1-5 cycles. We use both FPC and BDI, and compress with the policy that gives better compression ratio. Bits denoting the compression algorithm used are stored inside the space allocated for tags. If two adjacent lines are compressed together, we share tags [34, 35] and bases [31].

Figure 4 shows the compressibility of lines with FPC+BDI. We analyze the lines being installed in DRAM cache, and measure the fraction of lines that get compressed to half the size (32B), 36B (tag not needed due to tag sharing), or the likelihood of compressing two adjacent lines into 68B. For some workloads, such as *mcf*, *omnet*, and *astar* the potential for compressibility is high. Whereas, workloads such as *lbm*, *libq*, and *Gems* have little potential for compression. We find that on average 52% of two adjacent lines can be compressed within a single 72B physical line of Alloy Cache.

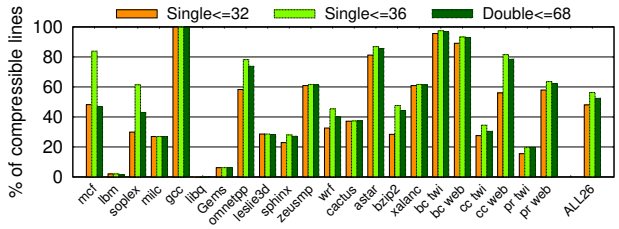

Figure 4: Fraction of compressible lines per workload. 52% of two adjacent lines compress to ≤68B (72B TAD).

4.3 Flexible Data format for Compression

To enable compression, we need not change the organization of the DRAM cache. The DRAM cache provides 72B per set. It is up to the memory controller to interpret those 72B as either tag or data. For supporting compression, we use a format that allows the number of tag-store entries to increase dynamically to accommodate storing extra lines with the same set. We implement this by having one bit per tag denoting that the next 4B should be interpreted as tag or data.

Figure 5 shows the format of the tag and data entry used in our design. We note that even though the Alloy Cache provisioned 8 bytes for tag-store (so that TAD is aligned at the bus boundary), the tag entry for such a large cache need not be large. For example, for a 48-bit physical address space, a 1GB direct-mapped cache requires only 18 bits of tag. With a valid bit and a dirty bit, we only need 20 bits for tag-store entry for the baseline cache. For compressed design, each tag entry has a *Next Tag Valid* bit to inform whether the next 4B is tag or data. This allows us to store arbitrary number of tags. A *BAI* bit is added to distinguish the direct-mapped line vs. an adjacent line that is spatially compressed together with it (more on this in Section 5). A *Shared tag* bit is used to save tag space when spatially contiguous lines are compressed in the same index [34, 35]. We use up to 9 bits for compression algorithm metadata (FPC/BDI). Our design can accommodate up to 28 compressed lines per set.

4.4 Speedup from Compression for Capacity

Our compressed cache design tries to accommodate more lines in each set of Alloy Cache, if data is compressible. For our baseline Alloy Cache, we assume that each set of Alloy Cache is determined by the conventional cache indexing scheme that places consecutive lines in consecutive sets. We call this set selection as *Traditional*

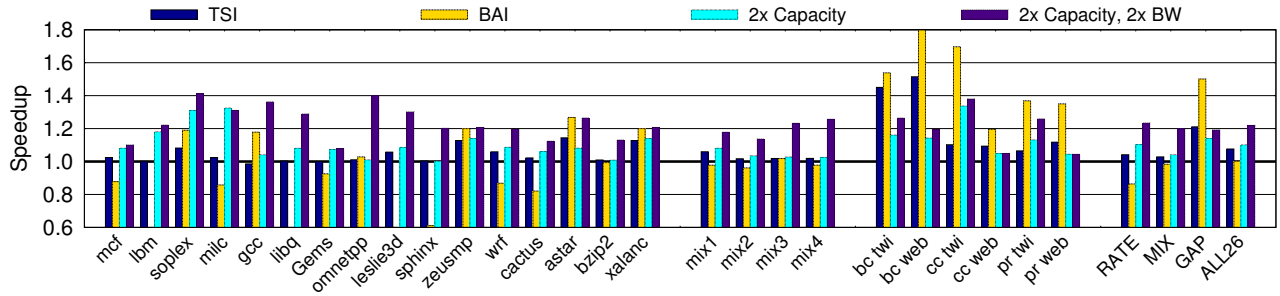

Figure 7: Speedup from Traditional Set Indexing and Bandwidth-Aware Indexing, compared to doubling the cache capacity and bandwidth. BAI improves bandwidth for compressible workloads but causes slowdown for others due to thrashing.

Set Indexing (TSI), in Figure 6(a). Lines that map to the same set under TSI are separated by several GB in physical memory. If these lines can be compressed, then they can reside in the same cache set, and an access to the set will obtain these lines with a single access. Unfortunately, lines that are spatially far away are unlikely to be accessed within a short period of each other, and should not be installed in the L3 cache. Thus, this form of compression is purely for capacity benefits, and not for bandwidth. Figure 7 shows performance improvement of a cache compressed with TSI. Unfortunately, compression for capacity alone has limited performance benefit, as it provides a speedup of 7%. We observe that compressing for both capacity and bandwidth has higher potential for speedup. Therefore, we seek a design that can improve both capacity and bandwidth.

Set 0	A0, A8
Set 1	A1, A9
Set 2	A2, A10
Set 3	A3, A11
Set 4	A4, A12
Set 5	A5, A13
Set 6	A6, A14
Set 7	A7, A15

(a) TSI

Set 0	**A0,** A1
Set 1	A2, A3
Set 2	A4, A5
Set 3	A6, A7
Set 4	A8, A9
Set 5	A10, A11
Set 6	A12, A13
Set 7	A14, **A15**

(b) NSI

Set 0	**A0,** A1,
Set 1	A8, **A9**
Set 2	**A2,** A3
Set 3	A10, **A11**
Set 4	**A4,** A5
Set 5	A12, **A13**
Set 6	**A6,** A7
Set 7	A14, **A15**

(c) BAI

Figure 6: Mapping 16 consecutive lines A0-A15 in a cache with 8 sets under (a) TSI (b) NSI (c) BAI. Purple boxes indicate lines that remain in the same set as TSI.

4.5 Bandwidth-Aware Indexing (BAI)

Spatially nearby lines are more likely to be accessed within a short period of each other. Therefore, if we could change the cache indexing such that spatially neighboring lines can be resident in the same set, then with compression we can obtain multiple useful lines per access, improving both capacity and bandwidth. A simple method to have two consecutive lines in the same set is to ignore the least significant bit of line address while indexing the cache, as in Figure 6(b). We call such a method of cache indexing as *Naive Spatial Indexing (NSI)*. When lines are compressible, NSI is successful in having two consecutive lines map to same set. This improves both capacity and bandwidth. Unfortunately, when lines are incompressible, the spatially close lines fight for space in the same set, degrading performance (by as much as 63%).

We explain shortcoming of NSI with an example. Figure 6 shows a cache with 8 sets, labeled Set 0 to Set 7. We have a workload with sixteen consecutive lines A0-A15, where lines A0-A7 are frequently accessed. Figure 6(a) shows the mapping with TSI and Figure 6(b) shows mapping with NSI. When lines are compressible, both TSI and NSI can fit all 8 frequently used lines (A0-A7), and NSI can stream out these lines in half the number of accesses. Unfortunately, if lines are incompressible, NSI can accommodate only four lines out of A0-A7 at any time, causing thrashing. This thrashing can degrade performance of NSI to worse than that of uncompressed cache, which is undesirable. As such, we aim to have a dynamic policy to switch between TSI and NSI depending on compressibility, to get capacity and bandwidth when lines are compressible, but avoid slowdown when lines are incompressible.

However, switching between NSI and TSI is costly, as nearly all the lines are in different positions, as shown in Figure 6(b). To address this, we propose *Bandwidth-Aware Indexing (BAI)* that ensures consecutive lines map to the same set, while half of lines retain the same position as in TSI, as shown in Figure 6(c). BAI retains capacity and bandwidth benefits of NSI when lines are compressible, as consecutive lines map to same set. And, it allows quick switching to TSI when lines are incompressible.

Another key feature of BAI is that BAI is guaranteed to be either the same set, or neighboring set as under TSI. Thus, both locations of the line (under BAI or TSI) are guaranteed to be in the same row buffer. Furthermore, by design, Alloy Cache streams out tags of the neighboring set, so we can determine if line is resident in either of the two locations with a single access.

4.6 Effectiveness of Bandwidth-Aware Index

BAI can get benefits of both capacity and bandwidth – as multiple lines obtained from a single cache access are likely to be useful, reducing accesses to the DRAM cache. Unfortunately, when lines are incompressible, BAI (and NSI) performs poorly compared to TSI. For example, if our access stream contained only 8 lines (A0-A7), TSI would be able to accommodate all lines, where BAI (and NSI) would be able to accommodate only four lines at any time.

Figure 7 compares the speedup from BAI with TSI, and also to doubling the cache capacity and bandwidth. BAI improves performance significantly for compressible workloads such as *soplex*, *gcc*, *zeusmp*, and *astar*. This happens because compression allows the cache to have more effective capacity and bandwidth. Unfortunately, for workloads such as *mcf*, *lbm*, *libq*, and *sphinx*, there is significant performance degradation with BAI, as spatially contiguous lines end up fighting for the same set. Ideally, we would like to use BAI as much as possible when lines are compressible, but use TSI when lines are incompressible. To this end, we propose a dynamic indexing scheme for compressed caches that can adapt cache indexing based on data compressibility.

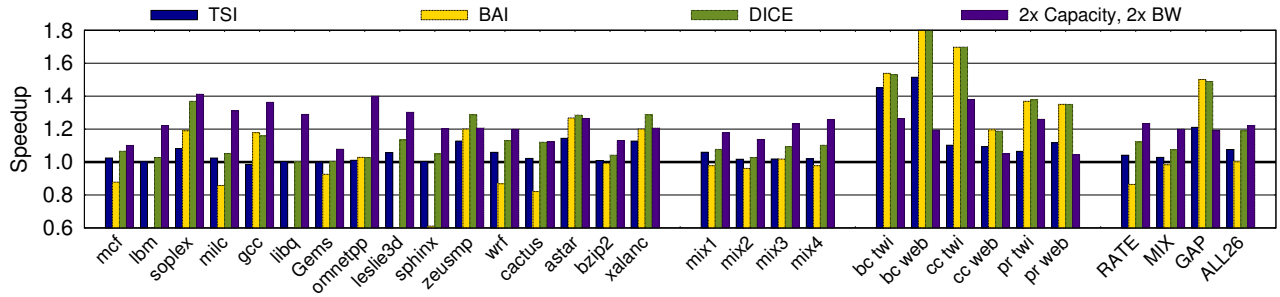

Figure 10: Speedup of compressing DRAM cache with TSI, BAI, and DICE. DICE's dynamic selection helps it to outperform both TSI and BAI. DICE provides an average speedup of 19.0%, nearing the 21.9% of a double-capacity double-bandwidth cache.

5 DYNAMIC-INDEXING COMPRESSION

We develop a compressed cache indexing policy that maximizes both bandwidth and capacity while ensuring no performance degradation compared to baseline uncompressed cache. To do so, we propose a dynamic indexing scheme called *Dynamic-Indexing Cache Compression (DICE)* that switches between two indexing policies, Traditional Set Indexing (TSI), and Bandwidth-Aware Indexing (BAI) depending on data compressibility. We present overview and working of DICE, then discuss effectiveness of our solution.

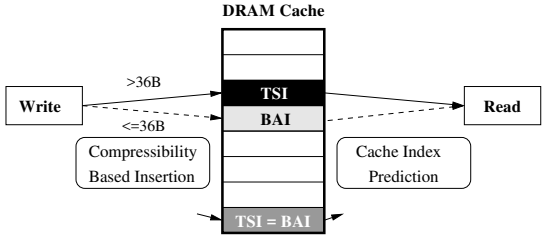

Figure 8: Design of DICE. DICE is implemented by deciding index policy on write, and predicting index policy on read.

5.1 DICE: Overview

DICE allows the cache to adapt its indexing scheme between TSI and BAI. Therefore, a given cache line can be present in either of the two locations, determined either by TSI or BAI. Fortunately, our BAI scheme is designed such that both of these sets would be either neighboring to each other, or be the same set, as shown in Figure 8. On a write access (due to install or writeback), we must decide which indexing policy to use. We develop a compressibility-based scheme to make this decision. Similarly, on a read access, we predict which indexing scheme is likely to have been used, and access that location. We exploit the property that an access to the Alloy Cache also brings the 8 byte tag information from the neighboring set. Therefore, we can find out if the index prediction is incorrect (the requested line is in neighboring set due to alternate indexing scheme) or is just a miss, and send a second access only if the line is guaranteed to be in the alternate location. The effectiveness of DICE depends on developing simple and effective mechanism for deciding index policy on writes and predicting index policy on reads. Note that, given BAI is designed such that the set index of 50% of the lines remain invariant between BAI and TSI, we need to decide insertion index (on write) and predict index policy (on read) for only the remaining 50% of the lines.

5.2 Deciding Cache Index Policy on Insertion

To decide the index policy at insertion, we leverage the observation that lines within a page are usually compressible to similar sizes [30]. As BAI gets benefits of both capacity and bandwidth when lines are compressible, we want the insertion policy to favor BAI when two lines are likely to compress together. If a line compresses to $\leq$ 36B, its neighboring line is also likely to compress to $\leq$ 36B. In these cases, we insert into BAI. Conversely, if a line is incompressible (say it compresses to 60 bytes), then its neighboring line is unlikely to be able to be compressed with it, so we insert using TSI.

We propose a simple mechanism that selects insertion policy based on size of compressed line. If a line compresses to $\leq$*Threshold*, we insert the line using BAI. Otherwise, we insert it using TSI. We study different threshold values for deciding index policy and determine that a threshold of 36B provides the best performance. In our studies, we use a default threshold of 36B. Sensitivity to threshold is performed in Section 6.2.

5.3 Cache Index Prediction (CIP)

As DICE makes the decision to use either TSI or BAI at insertion time, a cache can become mixed with some lines using TSI and others using BAI. To retrieve a line in the cache, we may need to look up both the possible locations. Unfortunately, doing so would consume more bandwidth and incur high latency. We develop a predictor to determine which location to access first. Note that, as a single access to Alloy Cache also gets tag information of the neighboring set, we can determine the location of the line in a single access to either set. If the line is not in either location (*miss*), a second access is not required. Only if the requested line is found in the adjacent set is a second access issued.

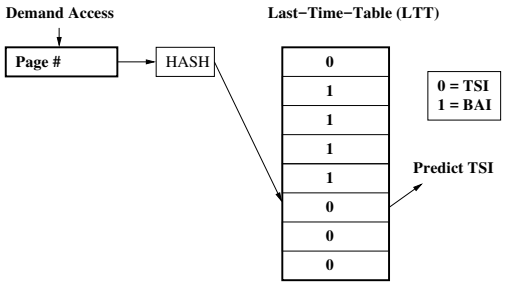

Figure 9: History-based Cache Index Predictor. CIP tracks history at page granularity.

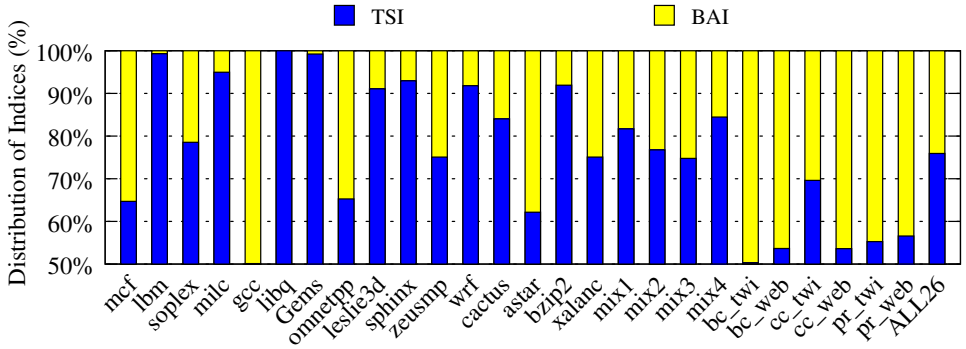

Figure 11: Distribution of BAI and TSI for a cache compressed with DICE. Note that for 50% of accesses, we do not need to make install decisions or do index prediction as TSI and BAI refer to the same set, hence the y-axis starts at 50%.

As misprediction incurs extra latency and bandwidth, we would like rate of misprediction to be low. We develop *Cache Index Predictors (CIP)* for reads and writes that can accurately predict cache index with low storage overhead. For reads, we design a page-based CIP predictor that uses last-time information for predictions, as shown in Figure 9. CIP leverages observation that lines within a page have similar compressibility [30]. If a page is compressible, lines within that page will likely be in BAI. CIP contains a *Last Time Table (LTT)* that tracks last outcome for a page. Given LTT has limited entries, we hash page address to index LTT.

We vary the number of entries in LTT and find the accuracy increases from 93.2% (512 entries) to 94.1% (8192 entries). For reads, we use a default LTT of 2048 entries (256B), which has an average accuracy of 93.8%. For writes, we predict index based on compressibility of data (same as insertion policy), which has an accuracy of 95%.

5.4 Impact on System Performance

Figure 10 shows the speedup of cache compression with TSI, BAI and DICE, and compare it with a cache that has double the capacity and bandwidth. Recall that compressing with TSI provides only capacity benefits and not bandwidth. Therefore, compression with TSI provides a marginal benefit of 7% on average. We observe that TSI always provides a hit rate that is either better than or equal to baseline uncompressed cache, so no workloads experience slowdown.

Compression using BAI tries to get both higher capacity and higher bandwidth. Therefore, there is significant performance improvement for workloads such as *gcc* and *cc twi*, where optimizing for capacity alone provided negligible benefits. Unfortunately, for workloads such as *lbm* and *libq*, the data is incompressible and the increased contention due to the indexing of BAI causes significant increase in cache misses, resulting in performance degradation. Overall, BAI performs similar to baseline (0.1% speedup), on average.

With DICE, the cache performs as well as BAI when BAI performs well, and similar to TSI for incompressible workloads, causing no degradation compared to baseline. In addition, there are several standouts (such as *soplex*, *leslie3d*, *zeusmp*, *wrf*, and *cactus*) when DICE performs better than either BAI or TSI independently, as it is able to use BAI for compressible regions of memory, and TSI for incompressible regions of memory. The dynamic selection of DICE helps it to outperform the two static indexing schemes. Overall, our DICE design incurs an SRAM overhead of less than 1 kilobyte yet provides 19.0% speedup, which is close to the 21.9% performance improvement of a double-capacity double-bandwidth DRAM cache.

6 RESULTS AND ANALYSIS

6.1 Distribution of TSI and BAI with DICE

With DICE, the cache can use TSI, or BAI, or a combination of TSI and BAI across the cache sets. Figure 11 shows the distribution of BAI and TSI. We separate the cases where location of the line remains invariant between BAI and TSI (50% of lines). From the remaining lines, we see a skew of 52% towards TSI and 48% towards BAI. This is due to incompressible workloads such as *libq* that cause almost the entire cache to use TSI to avoid performance degradation.

6.2 Sensitivity to Insertion Threshold

DICE uses compressibility of data to determine index policy on insertion. We use a default threshold of 36B to determine if BAI or TSI should be used. Table 4 shows the speedup of DICE as the threshold is changed from 32B to 36B to 40B. Note that a threshold of 0 will degenerate DICE to always use TSI, and a threshold of 64 will degenerate DICE to always use BAI. We find that the performance is maximized for a threshold of 36B. This is because BDI often compresses a single line to 36B, but double-line compresses it to 68B, which can fit in BAI if the tags are shared.

Table 4: Sensitivity to DICE threshold

	$\leq$ 32B	$\leq$ 36B	$\leq$ 40B
SPEC RATE	+10.6%	+12.2%	+11.1%
SPEC MIX	+6.4%	+7.5%	+7.4%
GAP	+47.6%	+48.9%	+49.1%
GMEAN26	+17.5%	+19.0%	+18.3%

6.3 Impact on DRAM Cache Capacity

Compression increases effective capacity of cache by storing more lines within the same physical space. Table 5 shows average capacity of DRAM cache when compressed with TSI, BAI, or DICE. We estimate effective capacity by checking number of valid lines in each set every 50M instructions.

Table 5: Effective Capacity of TSI/BAI/DICE

	TSI	BAI	DICE
SPEC RATE	1.07x	1.16x	1.13x
SPEC MIX	1.12x	1.28x	1.24x
GAP	2.00x	5.57x	5.06x
GMEAN26	1.24x	1.69x	1.62x

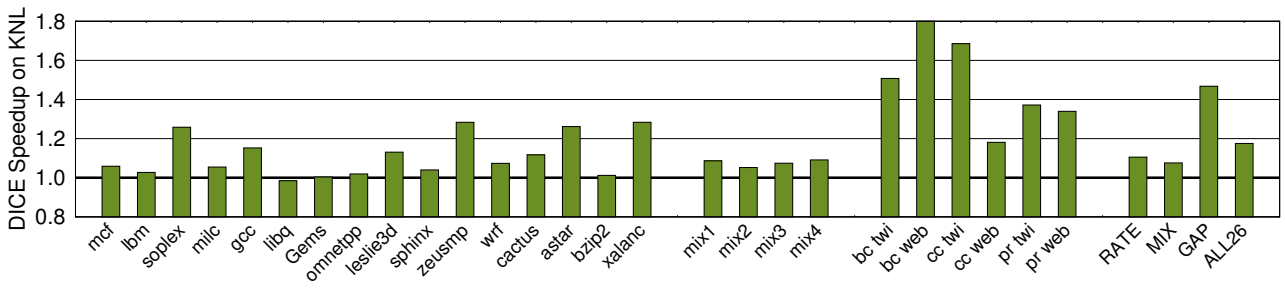

Figure 12: DICE on a DRAM cache design based on Intel Knights Landing. Note that even on KNL, DICE improves average performance by 17.5% (within 2% of the speedup of DICE on an Alloy Cache).

DICE and BAI have higher compression ratios due to two reasons: First, TSI compresses lines from different pages within the same set. These lines are less likely to have similar compressibility. DICE and BAI, on the other hand, often compress together lines from the same page, which are likely to have similar compressibility and hence are more likely to fit within the same set. Second, DICE and BAI improve compression ratio due to tag and base-sharing (BDI), which amortizes tag and metadata overhead. Unlike SRAM-based cache compression, which is limited by tag-store entries, our design can store up to 28 logical compressible lines in one physical line and provide higher capacity (e.g. *GAP*).

While many workloads, such as those in *GAP*, see capacity benefits, other workloads, such as *libq*, have poor compressibility. DICE increases effective cache capacity by 62%, on average.

6.4 Impact of DICE on Hit-Rate of L3

If data is compressible, then BAI can provide two spatially-contiguous lines with a single access to the L4 cache. As these lines are spatially close, we install both lines in the L3 cache as they are likely to be used within a short period of each other, improving L3 hit rate.

Table 6 shows the hit rate of the L3 cache for a baseline system (uncompressed L4), and a system using DICE. For the baseline system, the average L3 hit rate is 37.0%, and it is improved to 43.6% with DICE. Thus, the adjacent lines obtained due to compression with DICE are useful, and installing them in the L3 cache provides performance benefits.

Table 6: Effect of DICE on L3 hit rate

	BASE	DICE
SPEC RATE	34.7%	43.0%
SPEC MIX	61.6%	67.2%
GAP	26.9%	29.4%
AVG26	37.0%	43.6%

6.5 Comparison to Larger Fetch for L3

DICE can send adjacent lines from the L4 cache proactively to the L3 cache. While this may have some resemblance to nextline prefetch or 2x-width line fetch, we note that there is a fundamental difference. DICE sends the adjacent line from L4 to L3, only when that line is obtained without any bandwidth overheads. However, prefetches result in an independent cache request which incurs extra bandwidth. We compare our proposal, with alternative designs for L3 cache that try to either get a wider granularity line in the L3 cache (128 bytes, with two separate 64 byte requests) or next line prefetching in the L3 cache (demand request is followed by a prefetch for the next line).

Table 7 shows the performance of wide-granularity fetch at L3 cache, next-line prefetch in L3 cache, and compare it with DICE (in L4) and a combination of DICE (in L4) plus next-line prefetch in L3 cache. We find that designs that simply try to get an extra line in the L3 cache (due to wider fetch or next-line prefetch) give marginal benefits of 1.9% and 1.6%, on average. DICE, which inherently provides an extra line to the L3 cache when such a line is obtained without bandwidth overheads, provides speedup of 19.0%. Nonetheless, the L3 optimizations are orthogonal to DICE and can be combined for greater benefit. For example, using DICE with next line prefetch increases speedup to 20.9%.

Table 7: Comparison of DICE to Prefetch

	128B-PF	Nextline-PF	DICE	DICE+NL
SPEC RATE	+3.2%	+2.6%	+12.2%	+16.7%
SPEC MIX	+1.2%	+1.9%	+7.5%	+7.7%
GAP	-1.1%	-1.1%	+48.9%	+43.4%
GMEAN26	+1.9%	+1.6%	+19.0%	+20.9%

6.6 DICE on Intel's Knights Landing (KNL)

In our studies, we assumed a baseline Alloy Cache configuration, which obtains a 72-byte TAD per each access by transferring 80 bytes over 5 bursts. In this section, we study DICE on the DRAM cache used in Intel's Knights Landing (KNL). The DRAM Cache in KNL uses 64B cacheline with tags stored in ECC [40]. In this design, the 3D memory is equipped with additional lanes for ECC, and each access obtains a 72-byte TAD over four bursts. The TAD is used for tag, data, and ECC; however, such a design does not provide tag information for the neighboring line. Nonetheless, CIP still predicts read and write locations correctly 94% and 95% of the time, respectively. However, to ensure correctness, misses now need to check both indices when BAI≠TSI (50% of the time).

Fortunately, neighboring lines are likely to be accessed together, so the miss probes for both accesses are often merged by the controller, therefore the effective performance impact of checking the alternate location is mitigated. Figure 12 shows that DICE on KNL configuration achieves 17.5% speedup, which corresponds to most of the 19.0% benefit of DICE on Alloy Cache.[2]

[2]KNL provisions 8GB of MCDRAM capacity and 400GBps bus bandwidth shared between 64 cores. We model a system that is 1/8th the size of KNL and has 1GB capacity and 100GBps shared between 8 cores.

6.7 Non Memory-Intensive Workloads

In our studies, we only considered benchmarks that had an L3 cache MPKI ≥ 2, as they tend to be sensitive to optimizing memory system. Alternatively, if a workload fits in on-chip caches, then such workload would not benefit from improving off-chip memory.

Figure 13 shows performance impact of DICE on the non memory-intensive SPEC benchmarks excluded from our detailed study. As many of these benchmarks fit in L3 cache, they do not see benefit. However, more importantly, DICE does not degrade performance for any of them. On average, DICE improves performance by 2% on these workloads.

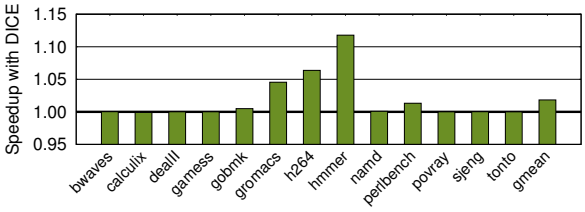

Figure 13: Speedup of DICE on non-memory-intensive applications (L3 MPKI < 2). DICE does not degrade these workloads.

6.8 Sensitivity to Capacity, BW, and Latency

Table 8 shows sensitivity of DICE to varying the capacity, bandwidth, and latency of the DRAM cache, normalized to their respective uncompressed designs. For a 2GB DRAM cache, DICE retains its bandwidth benefits for a speedup of 13.2%. For a 2x-channel DRAM cache, denoted by 2x BW in Table 8, DICE performs well at 24.5% speedup. We note that specifications for stacked DRAM state that stacked DRAM latency remains same as DIMM-based counterparts. Nonetheless, we perform sensitivity study on a half-latency DRAM cache. For a half-latency DRAM cache, DICE is able to alleviate the increased memory pressure (by increasing L4 hit rate) caused by the lower-latency DRAM cache for a speedup of 24.4%. Overall, DICE is robust and benefits a wide range of DRAM configurations.

Table 8: Sensitivity of DICE on different caches

	Base(1GB)	2x Capacity	2x BW	50% Latency
SPEC RATE	+12.2%	+8.7%	+13.3%	+13.5%
SPEC MIX	+7.5%	+4.7%	+8.2%	+9.1%
GAP	+48.9%	+32.6%	+75.9%	+73.5%
GMEAN26	+19.0%	+13.2%	+24.5%	+24.4%

6.9 Impact of DICE on Energy

Figure 14 shows L4+Memory power, energy consumption, and energy-delay-product (EDP) of a system with TSI, BAI, and DICE, normalized to the baseline. TSI increases L4 hit rate, which reduces memory energy consumption. BAI improves performance and energy for compressible workloads, but hurts incompressible ones, making its performance similar to baseline but energy worse. DICE improves both L3 and L4 hit rate leading to a reduction in both stacked DRAM and memory energy consumption. Overall, DICE reduces energy consumption by 24% and EDP by 36%.

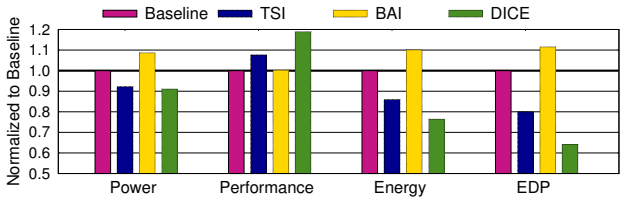

Figure 14: Impact of DICE on energy. DICE reduces DRAM cache and memory accesses, reducing off-chip energy by 24%.

7 RELATED WORK

To our knowledge, this is the first paper to advocate compressing DRAM cache primarily for bandwidth and secondarily for capacity. We show that compression can be implemented without increasing tag storage overhead or affecting tag access. We also show that it is important to design DRAM cache compression to exploit both capacity and bandwidth. We discuss prior research proposals related to our study.

7.1 Low-Latency Compression Algorithms

As decompression latency is in the critical path of memory accesses, memory systems rely on simple data compression schemes [4, 5, 17, 23, 31, 42]. We evaluate DICE using a hybrid compression scheme based on FPC and BDI. However, DICE is orthogonal to the type of data compression scheme used and can be used in conjunction with any data compression scheme, including ones that employ dictionary-based compression [6, 7, 11, 26].

7.2 Main Memory Compression

Hardware-based memory compression has been applied to increase the effective capacity of main memory [1, 18, 30]. Memzip [36, 38] proposes to send compressed data across links in smaller bursts, and send additional ECC or metadata bits when there is still room in a burst length. These proposals try to increase the bandwidth of the memory system. However, they either require OS support or give up on the capacity benefits.

A recent work on DRAM cache compression for a PCM + DRAM hybrid system [16] uses IBM MXT main memory compression as a baseline. This work has a key shortcoming inherited from main memory compression. It requires an additional serialized access to find compressed size and offset, before finally accessing the data. This comes with double the bandwidth usage and double the latency per access, which we show to be ineffective in Section 7.3. Another work [8] on PCM + DRAM hybrid system is based on SRAM cache compression and assumes an associative DRAM cache. This requires an additional serialized lookup of tag and is thus also latency and bandwidth-inefficient. Our proposal, on the other hand, provides both capacity and bandwidth benefits without relying on OS support or needing serialized tag lookup.

7.3 Compressing SRAM Caches

Prior work has looked at using compression to increase capacity of on-chip SRAM caches. Cache compression is typically done by

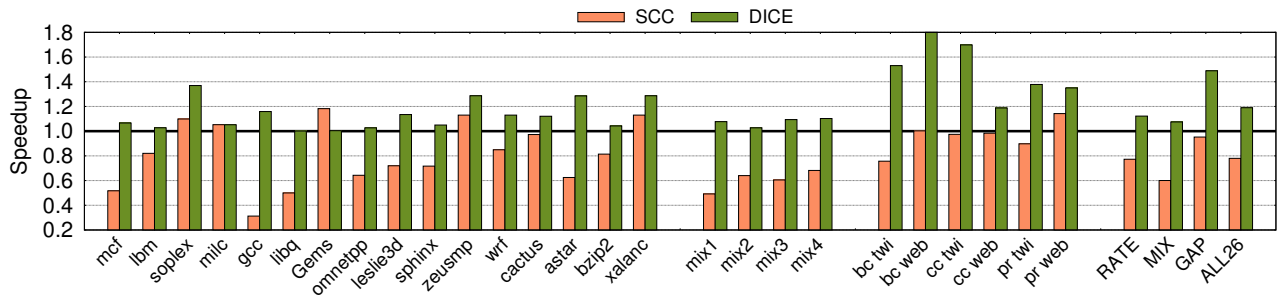

Figure 15: Skewed Compressed Cache (SCC) on DRAM cache. SCC causes 22% slowdown due to extra tag accesses.

accommodating additional ways in a given cache set and statically allocating more tag-store entries [4, 20]. These proposals optimize purely for hit rate, while we find that DRAM caches are more sensitive to bandwidth. As such, 11.4% of our 19.0% speedup is from bandwidth benefits, not capacity (DICE over TSI). Recent proposals, such as Skewed Compressed Cache (SCC), investigate reducing SRAM tag overhead by sharing tags across spatially-contiguous sets in what are called *superblocks* [27, 28, 34, 35]. For a 4x-superblock 8-way physical cache, these proposals use 32 physical tags to address up to 128 compressed lines by sharing the tags of neighboring sets. Unfortunately, an access in SCC requires skewed associative lookup for different locations in the cache. While this may be practical to do in an SRAM cache, it incurs prohibitively high bandwidth overhead to lookup multiple locations in DRAM cache to service each request.

We evaluate SCC in the context of DRAM caches, to highlight need for bandwidth efficiency in compressing DRAM caches. Figure 15 shows the speedup from compressing the DRAM cache with SCC and DICE. Each request in SCC incurs four accesses to DRAM cache (3 for tags and one for data), whereas a request in DICE requires only one access to the DRAM cache in the common case (second only in case of CIP misprediction). On average, SCC causes 22% slowdown, whereas DICE provides 19% speedup.

7.4 Multiple Index to Reduce Conflict Misses

DICE uses multiple indexing schemes (TSI and BAI) in order to get bandwidth benefits of spatial indexing and avoid slowdown when data is incompressible. Prior work in designing direct-mapped L1 caches have also looked at using multiple indexing schemes in order to reduce conflict misses. On a miss, these designs [2, 3] check an alternative location (a faraway set in the cache) to find the conflicting line. Unfortunately, such a design that always requires a second access in case of a cache miss would incur high latency (on hits in second location) and high bandwidth (from extra accesses due to a second lookup on a miss). Schemes that rely on looking up multiple locations in parallel [33, 37] to reduce conflict misses would reduce latency overheads but would incur significant bandwidth overheads for DRAM cache. Unlike these proposals, DICE avoids the latency and bandwidth of second lookup via index prediction and exploiting properties of a DRAM cache. Furthermore, the multiple indexing schemes in DICE are not aimed at reducing conflict miss but for increasing cache bandwidth for compressed lines.

8 CONCLUSIONS

This paper looked at compression as a means of increasing the bandwidth of DRAM caches, while also obtaining capacity benefits. We exploit the fact that practical DRAM caches are likely to store tags within the DRAM array, so they can support compression seamlessly, as the tag-store entries required for the additional capacity created due to compression can be accommodated within the DRAM substrate without the need for any SRAM overheads. Furthermore, as DRAM caches are managed entirely in hardware, we can do DRAM cache compression in a software-invisible manner and avoid the OS changes that are necessary for compressing main memory.

Our study showed that for maximizing performance it is important that DRAM caches perform compression for enhancing both the capacity and the bandwidth. We note that traditional methods to perform cache compression are aimed at solely increasing the cache capacity and provide only marginal benefits. To this end, our paper proposes to change the cache indexing dynamically to a bandwidth-enhancing scheme called *Bandwidth-Aware Indexing (BAI)*. We show that while BAI can improve both capacity and bandwidth when data is compressible, it can degrade performance when data is not compressible. We propose *Dynamic-Indexing Cache Compression (DICE)* that dynamically changes cache indexing depending on compressibility of line. To avoid looking up two locations for a line, we develop low-cost *Cache Index Predictors (CIP)* that can accurately predict index for the line using history information. Our evaluations show that DICE improves performance of a 1GB DRAM cache by 19.0% and reduces EDP by 36%, while incurring storage overhead of less than 1 kilobyte and without requiring OS support.

ACKNOWLEDGMENTS

We thank Chia-Chen Chou, Alaa Alameldeen, Rajat Agarwal, and Swamit Tannu for comments and feedback. This work was supported in part by a gift from Intel, NSF grant 1319587, and the Center for Future Architecture Research (C-FAR), one of the six SRC STARnet Centers, sponsored by MARCO and DARPA.

REFERENCES

[1] Bulent Abali, Hubertus Franke, Xiaowei Shen, Dan E. Poff, and T. Basil Smith. 2001. Performance of hardware compressed main memory. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*. 73–81. https://doi.org/10.1109/HPCA.2001.903253

[2] Anant Agarwal, John Hennessy, and Mark Horowitz. 1988. Cache Performance of Operating System and Multiprogramming Workloads. *ACM Trans. Comput. Syst.* 6, 4 (Nov. 1988), 393–431. https://doi.org/10.1145/48012.48037

[3] Anant Agarwal and Steven D. Pudar. 1993. Column-associative Caches: A Technique For Reducing The Miss Rate Of Direct-mapped Caches. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*. 179–190. https://doi.org/10.1109/ISCA.1993.698559

[4] Alaa R. Alameldeen and David A. Wood. 2004. Adaptive Cache Compression for High-Performance Processors. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA '04)*. IEEE Computer Society, Washington, DC, USA, 212–. http://dl.acm.org/citation.cfm?id=998680.1006719

[5] Alaa R Alameldeen and David A Wood. 2004. Frequent pattern compression: A significance-based compression scheme for L2 caches. *Dept. Comp. Scie., Univ. Wisconsin-Madison, Tech. Rep* 1500 (2004).

[6] Angelos Arelakis, Fredrik Dahlgren, and Per Stenstrom. 2015. HyComp: A Hybrid Cache Compression Method for Selection of Data-type-specific Compression Methods. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 38–49. https://doi.org/10.1145/2830772.2830823

[7] Angelos Arelakis and Per Stenstrom. 2014. SC2: A statistical compression cache scheme. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*. 145–156. https://doi.org/10.1109/ISCA.2014.6853231

[8] Seungcheol Baek, Hyung Gyu Lee, Chrysostomos Nicopoulos, and Jongman Kim. 2014. Designing Hybrid DRAM/PCM Main Memory Systems Utilizing Dual-Phase Compression. *ACM Trans. Des. Autom. Electron. Syst.* 20, 1, Article 11 (Nov. 2014), 31 pages. https://doi.org/10.1145/2658989

[9] Scott Beamer, Krste Asanovic, and David A. Patterson. 2015. The GAP Benchmark Suite. *CoRR* abs/1508.03619 (2015). http://arxiv.org/abs/1508.03619

[10] Niladrish Chatterjee, Rajeev Balasubramonian, Manjunath Shevgoor, S Pugsley, A Udipi, Ali Shafiee, Kshitij Sudan, Manu Awasthi, and Zeshan Chishti. 2012. Usimm: the utah simulated memory module. *University of Utah, Tech. Rep* (2012).

[11] Xi Chen, Lei Yang, Robert P. Dick, Li Shang, and Haris Lekatsas. 2010. C-pack: A High-performance Microprocessor Cache Compression Algorithm. *IEEE Trans. Very Large Scale Integr. Syst.* 18, 8 (Aug. 2010), 1196–1208. https://doi.org/10.1109/TVLSI.2009.2020989

[12] Chiachen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. 2014. CAMEO: A Two-Level Memory Organization with Capacity of Main Memory and Flexibility of Hardware-Managed Cache. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 1–12. https://doi.org/10.1109/MICRO.2014.63

[13] Chiachen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. 2015. *BATMAN: Maximizing Bandwidth Utilization of Hybrid Memory Systems*. Technical Report TR-CARET-2015-01. School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, Georgia. 12 pages. http://www.jaleels.org/ajaleel/publications/techreport-BATMAN.pdf

[14] Chiachen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. 2015. BEAR: Techniques for Mitigating Bandwidth Bloat in Gigascale DRAM Caches. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 198–210. https://doi.org/10.1145/2749469.2750387

[15] Chiachen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. 2016. CANDY: Enabling coherent DRAM caches for multi-node systems. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. https://doi.org/10.1109/MICRO.2016.7783738

[16] Yu Du, Miao Zhou, Bruce Childers, Rami Melhem, and Daniel Mossé. 2013. Delta-compressed Caching for Overcoming the Write Bandwidth Limitation of Hybrid Main Memory. *ACM Trans. Archit. Code Optim.* 9, 4, Article 55 (Jan. 2013), 20 pages. https://doi.org/10.1145/2400682.2400714

[17] Julien Dusser, Thomas Piquet, and André Seznec. 2009. Zero-content Augmented Caches. In *Proceedings of the 23rd International Conference on Supercomputing (ICS '09)*. ACM, New York, NY, USA, 46–55. https://doi.org/10.1145/1542275.1542288

[18] Magnus Ekman and Per Stenstrom. 2005. A Robust Main-Memory Compression Scheme. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA '05)*. IEEE Computer Society, Washington, DC, USA, 74–85. https://doi.org/10.1109/ISCA.2005.6

[19] Sean Franey and Mikko Lipasti. 2015. Tag tables. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 514–525. https://doi.org/10.1109/HPCA.2015.7056059

[20] Jayesh Gaur, Alaa R. Alameldeen, and Sreenivas Subramoney. 2016. Base-Victim Compression: An Opportunistic Cache Compression Architecture. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 317–328. https://doi.org/10.1109/ISCA.2016.36

[21] Djordje Jevdjic, Gabriel H. Loh, Cansu Kaynak, and Babak Falsafi. 2014. Unison Cache: A Scalable and Effective Die-Stacked DRAM Cache. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 25–37. https://doi.org/10.1109/MICRO.2014.51

[22] Djordje Jevdjic, Stavros Volos, and Babak Falsafi. 2013. Die-stacked DRAM Caches for Servers: Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 404–415. https://doi.org/10.1145/2485922.2485957

[23] Jungrae Kim, Micahel Sullivan, Esha Choukse, and Mattan Erez. 2016. Bit-Plane Compression: Transforming Data for Better Compression in Many-Core Architectures. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 329–340. https://doi.org/10.1109/ISCA.2016.37

[24] Gabriel H. Loh and Mark D. Hill. 2011. Efficiently Enabling Conventional Block Sizes for Very Large Die-stacked DRAM Caches. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. ACM, New York, NY, USA, 454–464. https://doi.org/10.1145/2155620.2155673

[25] Micron. 2013. HMC Gen2. *Micron* (2013). http://www.micron.com/products/hybrid-memory-cube

[26] Tri M. Nguyen and David Wentzlaff. 2015. MORC: A Manycore-oriented Compressed Cache. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 76–88. https://doi.org/10.1145/2830772.2830828

[27] Shingo Ohya. 2016. Skewed Compressed DRAM Cache Ni Yori. (2016).

[28] Biswabandan Panda and André Seznec Seznec. 2016. Dictionary sharing: An efficient cache compression scheme for compressed caches. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. https://doi.org/10.1109/MICRO.2016.7783704

[29] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. 2004. Pinpointing Representative Portions of Large Intel Itanium Programs with Dynamic Instrumentation. In *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*. 81–92. https://doi.org/10.1109/MICRO.2004.28

[30] Gennady Pekhimenko, Vivek Seshadri, Yoongu Kim, Hongyi Xin, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2013. Linearly Compressed Pages: A Low-complexity, Low-latency Main Memory Compression Framework. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 172–184. https://doi.org/10.1145/2540708.2540724

[31] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Michael A. Kozuch, Phillip B. Gibbons, and Todd C. Mowry. 2012. Base-delta-immediate compression: Practical data compression for on-chip caches. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 377–388.

[32] Moinuddin K. Qureshi and Gabriel H. Loh. 2012. Fundamental Latency Trade-off in Architecting DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE Computer Society, Washington, DC, USA, 235–246. https://doi.org/10.1109/MICRO.2012.30

[33] Daniel Sanchez and Christos Kozyrakis. 2010. The ZCache: Decoupling Ways and Associativity. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. 187–198. https://doi.org/10.1109/MICRO.2010.20

[34] Somayeh Sardashti, André Seznec, and David A. Wood. 2014. Skewed Compressed Caches. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 331–342. https://doi.org/10.1109/MICRO.2014.41

[35] Somayeh Sardashti and David A. Wood. 2013. Decoupled compressed cache: Exploiting spatial locality for energy-optimized compressed caching. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 62–73.

[36] Vijay Sathish, Michael J. Schulte, and Nam Sung Kim. 2012. Lossless and Lossy Memory I/O Link Compression for Improving Performance of GPGPU Workloads. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT '12)*. ACM, New York, NY, USA, 325–334. https://doi.org/10.1145/2370816.2370864

[37] André Seznec. 1993. A case For Two-way Skewed-associative Caches. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*. 169–178. https://doi.org/10.1109/ISCA.1993.698558

[38] Ali Shafiee, Meysam Taassori, Rajeev Balasubramonian, and Al Davis. 2014. MemZip: Exploring unconventional benefits from memory compression. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 638–649. https://doi.org/10.1109/HPCA.2014.6835972

[39] Jaewoong Sim, Gabriel Loh, Hyesoon Kim, Mike OConnor, and Mithuna Thottethodi. 2012. A Mostly-Clean DRAM Cache for Effective Hit Speculation and Self-Balancing Dispatch. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. 247–257. https://doi.org/10.1109/MICRO.2012.31

[40] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. 2016. Knights Landing: Second-Generation Intel Xeon Phi Product. *IEEE Micro* 36, 2 (Mar 2016), 34–46. https://doi.org/10.1109/MM.2016.25

[41] JEDEC Standard. 2013. High bandwidth memory (hbm) dram. *JESD235* (2013).

[42] Youtao Zhang, Jun Yang, and Rajiv Gupta. 2000. Frequent Value Locality and Value-centric Data Cache Design. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*. ACM, New York, NY, USA, 150–159. https://doi.org/10.1145/378993.379235