2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture

Base-Victim Compression: An Opportunistic Cache Compression Architecture

Jayesh Gaur, Alaa R. Alameldeen, Sreenivas Subramoney
Intel Corporation
Email: jayesh.gaur@intel.com, alaa.r.alameldeen@intel.com, Sreenivas.subramoney@intel.com

***Abstract*—The memory wall has motivated many enhancements to cache management policies aimed at reducing misses. Cache compression has been proposed to increase effective cache capacity, which potentially reduces capacity and conflict misses. However, complexity in cache compression implementations could increase cache power and access latency. On the other hand, advanced cache replacement mechanisms use heuristics to reduce misses, leading to significant performance gains. Both cache compression and replacement policies should collaborate to improve performance.**

In this paper, we demonstrate that cache compression and replacement policies can interact negatively. In many workloads, performance gains from replacement policies are lost due to the need to alter the replacement policy to accommodate compression. This leads to sub-optimal replacement policies that could lose performance compared to an uncompressed cache. We introduce a novel, opportunistic cache compression mechanism, Base-Victim, based on an efficient cache design. Our compression architecture improves performance on top of advanced cache replacement policies, and guarantees a hit rate at least as high as that of an uncompressed cache. For cache-sensitive applications, Base-Victim achieves an average 7.3% performance gain for single-threaded workloads, and 8.7% gain for four-thread multi-program workload mixes.

Keywords-cache compression; cache replacement policies

I. Introduction

Advances in process technology have significantly improved CPU performance over the last few decades. Unfortunately, commodity memory technology has not improved at the same pace. This creates a significant overhead whenever a CPU load request has to be fulfilled from memory. This "memory wall" motivated innovation and enhancements in cache architecture to reduce the performance penalty of memory accesses. Cache management mechanisms aim at reducing memory misses through timely prefetching, more accurate replacement policies ([7], [13], [14], [19], [20], [21], [31], [38], [39]), and larger effective cache capacity with compression ([1], [3], [9], [11], [16], [28], [32], [33]).

Ideally, different cache management mechanisms should collaborate to reduce misses and improve performance. Prior work has shown, for example, that last-level cache (LLC) compression and prefetching interact positively [2], leading to greater performance improvements than achieved by either mechanism alone. Unfortunately, this ideal relationship does not hold for all cache management policies. Some optimizations to improve one mechanism could negatively impact the performance of another mechanism.

In this paper, we focus on efficient last-level cache compression to improve performance with low overheads. We show that there are negative interactions between cache replacement and compression mechanisms that lead to worse performance when both are implemented together. In many cases, optimizing cache capacity increases from compression leads to sub-optimal replacement decisions, which hurt performance gains from state-of-the-art replacement policies. Prior work has addressed compressed-cache replacement policies ([4], [29]), but not in the context of negative interactions with compression.

Another significant implementation bottleneck for cache compression is due to the complexity of implementing decoupled variable-sector caches which result in significant performance and power overheads. Energy-efficient and area-efficient caches implement various mechanisms to reduce cache power, such as sub-banking ([15], [37]). Such mechanisms partition the cache into sub-banks, where only the bank that contains the requested data is activated. A similar mechanism is implemented in commercial products [6]. However, compressed cache architectures require activating segments from multiple ways to read a single cache line, which increases the power and area of the tightly-packed data array. This places a significant roadblock for any practical cache compression implementation.

We propose a cache compression architecture that avoids both major drawbacks for prior compression work. To avoid negative interactions with replacement policies, we guarantee that all lines that would have existed in an uncompressed cache at any point in the program's execution would remain in the compressed cache. To avoid power and area overhead of compressed cache architectures, we ensure the data array is unmodified by enforcing the association of two tags with one physical way. In our compressed cache (Base-Victim) architecture, each physical way can include data from at most two logical cache lines. The first (base) line is the line that would exist without compression. The second (victim) line is opportunistically kept in place if it can be compressed to fit with an existing base line. Our architecture sacrifices the flexibility of general variable-sector cache architectures to achieve a more efficient design while still maintaining the performance gains of advanced replacement policies.

In this paper, we make the following main contributions:

1) We show how compression can interact negatively with advanced cache replacement policies, leading to significant performance losses.
2) We highlight some overlooked overheads of general compression architectures.
3) We propose an opportunistic (Base-Victim) compression architecture that avoids the negative interactions with replacement policies and the overheads of prior compression work. By design, our architecture guarantees equal or higher cache hit rates for any program vs. an uncompressed cache.

1063-6897/16 $31.00 © 2016 IEEE
DOI 10.1109/ISCA.2016.36

4) We show that our proposal achieves significant performance improvements for both single-thread and multi-program workloads. Opportunistic compression adds 8.5% LLC area overhead by doubling tags, but gains an average performance equivalent to a 50% increase in cache capacity. This performance is achieved without reducing the hit rate on any workload relative to an uncompressed cache.

In the remainder of this paper, we present some background on cache compression architectures in Section II. We discuss negative interactions between cache compression and replacement policies in Section III. We propose our Base-Victim Compression Architecture in Section IV. We explain our evaluation methodology in Section V, and present results in Section VI. We highlight some related work in Section VII, and conclude in Section VIII.

II. Background

Cache design has recently focused on two main aspects: performance improvement and energy efficiency. Both are essential for modern general-purpose CPU caches that have to operate across many market segments.

Energy efficiency is an essential design requirement for caches to reduce overall system power. Caches constitute a significant fraction of die area, which motivates mechanisms to reduce static (idle) power. Caches are also frequently accessed since a large percentage of instructions are loads and stores, which increases dynamic power. To avoid incurring significant dynamic power in caches, many mechanisms have been proposed to reduce the fraction of cache cells powered up on each access. Cache sub-banking [37] has been proposed to fetch the requested sub-line instead of the whole logical line. Chang et al. [6] show how this is implemented in a commercial processor. For a 16-way set associative cache, each cache access requires accessing only one of the 16 sub-arrays in each group, and one of the eight blocks in the accessed sub-array is powered up. This means that the fraction of blocks that need to be powered up for each cache access is very small [6].

Performance improvements in caches come from better architectures and better cache management policies. For example, prefetching reduces cache misses by predicting future accesses. Replacement and insertion policies attempt to keep more useful blocks in the cache. Cache compression (our focus in this paper) can also reduce cache misses.

Cache compression has been proposed to increase the effective cache capacity, thereby reducing misses, without significantly increasing the cache area. Due to the sensitivity of many workloads to cache hit latency, prior work has focused on compression algorithms with fast decompression latency, and architectures that don't significantly increase cache overhead.

Cache compression algorithms have focused on achieving the highest compression ratio at a low decompression overhead. Examples of these compression algorithms include Frequent Pattern Compression [1], Cache Packer (C-PACK) [9], and Base-Delta-Immediate (BDI) compression [28]. We describe these and other algorithms in Section VII. In our work, we use BDI as the baseline compression algorithm due to its fast decompression latency.

Another important aspect of prior cache compression work is implementing a cache architecture that supports storing compressed lines. Such an architecture needs to allow for more logical lines to be stored than physical lines without sacrificing cache complexity or area. Many of these proposals decoupled the tag and data arrays based on the decoupled sectored cache [34], allowing data lines of variable sizes to be stored in the data array. One proposal is the Decoupled Variable-Sector Cache (VSC) [1] which supports twice as many tags as physical data lines, but allows data lines to be compacted within a set. In this architecture, a compressed cache line can start at any 8-byte segment boundary within a set, and can potentially span more than one physical line.

Unfortunately, VSC suffers from three drawbacks. First, it requires re-compaction of the set whenever a compressed line size increases, which incurs a significant read-modify-write overhead. Second, it requires significant changes to the SRAM data array since a hit to a compressed line could require activating more than one physical line. This negates energy savings of sub-banked caches [37] which are used to reduce energy in commercial processor caches [6]. Third, cache replacement becomes quite complicated since VSC goes through logical lines in LRU order, and evicts as many lines as needed to fit the incoming line. To illustrate this third drawback, an incoming 64B line that is uncompressed (i.e., has a size of eight 8B segments) might need to evict two or more lines from the LRU stack. For example, if the LRU lines have sizes of 2, 3, 2, and 5 segments, consecutively, all four lines at the bottom of the LRU stack need to be evicted and (potentially) written back to memory.

A more recent proposal, the Decoupled Compressed Cache (DCC) [32] addresses the first drawback of VSC by eliminating the need to re-compact compressed lines whenever a line changes its compressed size. By adding a level of indirection, DCC saves a significant amount of overhead that would be incurred due to re-compaction. DCC achieves significant energy savings compared to VSC [32]. A more efficient version, the Skewed Compressed Cache (SCC) was proposed by Sardashti et al. [33]. SCC eliminates many overheads of DCC by avoiding DCC's backward pointers, and reduces the latency of tag-data indirection.

However, both DCC and SCC still suffer from VSC's second and third drawbacks. Both require making changes to the data array to enable multiple segment access, and both require complex replacement mechanisms that could evict multiple logical lines on a cache fill (though SCC simplifies the replacement policy path). A hit to a compressed line might require activating more than one 16B segment that span multiple physical lines. To avoid significant energy increases, DCC proposes accessing the cache at 16B sub-bank granularity, which would require more logic circuits for control signals and sense-amps for each sub-bank. DCC

would suffer from additional latency due to either (a) the need to re-compact segments from different physical lines before the line is sent to the CPU (if segments are all activated and read at the same time); or (b) more stages/cycles in the pipeline if segments are activated one at a time.

To avoid such drawbacks that would substantially increase cache area, we chose to implement a simpler two-tags-per-way architecture that we explain in the next two sections. Our simple architecture enforces the association of each physical cache line with two logical tags to avoid making changes to the data array design.

III. INTERACTIONS BETWEEN CACHE COMPRESSION AND REPLACEMENT

To avoid the negative power and complexity implications of compressed caches, we choose to implement a simple architecture that associates two tags with each physical way in the data array. Figure 1 presents a high-level architecture that implements this association on a simple two-way set-associative baseline.

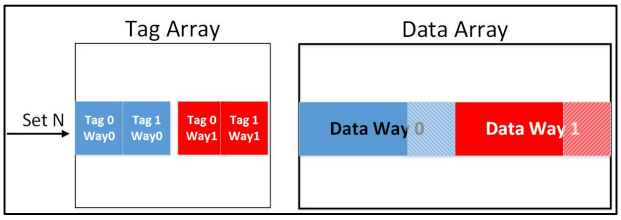

Figure 1. Two-Tag Architecture for a 2-way cache. Each physical way has two tags associated with it, and can store up to two logical lines (striped part represents second compressed line). The blue tags 0 and 1 are always associated with way 0, while the red tags 0 and 1 are associated with way 1.

This architecture does not require any changes to the physical data array. On a cache access, all tags that correspond to the correct set are compared, and only one data way is identified on a hit. That physical data line would include either one or two (compressed) logical lines. However, the whole data line is accessed from the data array using the same steps as an uncompressed cache. When the data is read out of the data array, it can then be decompressed, and the appropriate line is sent to the CPU and lower-level caches. The only additional overheads required for this architecture are (a) the extra tag area, access latency and power; and (b) compression and decompression latencies and power. However, the main advantage is that it doesn't require any additional area or floorplan changes to the data array.

Unfortunately, this simple architecture introduces some negative interactions with cache replacement policies, which could lead to losing the performance gains we achieve from advanced cache replacement mechanisms. In the following example, we show two different scenarios where compression negatively impacts LRU replacement.

Example. Figure 2 shows a 4-way cache (8-way with compression) where the most recently used (MRU) line is partnered with the least recently used (LRU) line. In this example, physical way 0 has two logical lines allocated to it: the MRU line (LRU stack position 0) and the LRU line (LRU stack position 6 since one line is uncompressed). The MRU line size is 6 segments and the LRU line size is 2 segments, where each physical 64B line can store eight 8B segments.

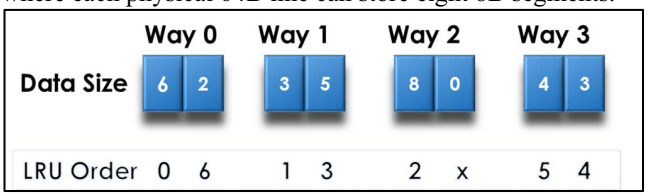

Figure 2. Example for MRU and LRU lines sharing the same physical way.

If an incoming fill line of size 6 segments needs to be allocated, LRU replacement indicates it should replace the LRU line (corresponding to tag 1 of way 0). However, since the partner of the victim line has six segments, the incoming line cannot fit with the other line in way 0, which happens to be the MRU line. There are two options for replacement in this case:

(1) Evict all logical lines in the physical way associated with the LRU logical line to allow room for the incoming fill line. We call this policy **partner line victimization**. In this example, the MRU line is evicted, which could cause significant losses in many workloads where the MRU line is the most frequently hit.

(2) Go through all logical lines in LRU stack order to identify a victim of appropriate size to be evicted. For this example, the first line that would fit the incoming line is at LRU stack position 2 (corresponding to tag 0 of way 2). Choosing this line for eviction means that LRU replacement is no longer maintained, since this line would remain in an uncompressed cache. Breaking LRU replacement (or other advanced replacement policies) can lead to significant glass jaws in some applications.

In Section VI, we show that many workloads suffer from large losses due to these negative interactions that offset any gains from a compressed cache. It should be noted that VSC avoids these negative interactions by replacing as many lines from the bottom of the LRU stack as possible (in this example, both LRU positions 6 and 5) to allow enough room for the incoming line. However, this requires re-compaction (i.e., defragmentation) of the cache lines in the set before the incoming line can be inserted. This incurs power overheads, latency increases and higher logic complexity for the replacement policy.

While this example focuses on LRU to provide a clear explanation, the performance losses are even more significant for advanced replacement policies as compression may require changing the replacement stack order. In the next section, we present an opportunistic cache compression architecture, Base-Victim, which retains the performance gains for advanced cache replacement policies, and opportunistically provides performance gains when cache lines can be efficiently compressed.

IV. Base-Victim Cache Compression Architecture

A. High-Level Architecture

To avoid the negative interactions that occur between compression and cache replacement policies, we propose an opportunistic cache compression policy where the non-compressed cache state is maintained, and victim lines are only opportunistically kept if they fit. We ensure that the baseline replacement policy performance is maintained by logically partitioning the cache into a Baseline (B) Cache and a Victim (V) Cache. Figure 3 shows how we treat tag 1 in each way of a set as belonging to the Victim Cache. The Victim Cache only holds lines that would have been evicted from the baseline uncompressed cache, but are only kept around because they could be compressed.

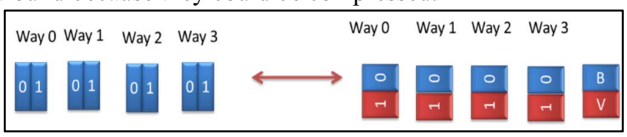

Figure 3. Logical Partitioning of LLC set into a Baseline (B) set and a Victim (V) set. This is an example for a 4-way (uncompressed) cache that becomes an 8-way cache with compression.

In the Baseline Cache, we strictly enforce the baseline insertion and replacement policy on all tag 0's of each set. *By design, this architecture cannot have a higher miss rate than an uncompressed cache with the same replacement policy.* Whenever a modified line is replaced from the Baseline Cache, its data is written back to memory to make it a clean line. This operation could include sending back-invalidates to the L1 and L2 caches. We then attempt to opportunistically insert the clean replaced line into the Victim Cache.

By only allowing clean lines in the Victim Cache, we can silently evict these lines on subsequent modifications to their partner Baseline Cache line with no additional memory traffic. This ensures that we only do (at most) one writeback for every cache fill operation. We show how this simplifies our implementation in the next sub-section. However, this comes at the expense of not saving writeback traffic to memory. Our architecture only saves memory read miss traffic, but we incur the same number of memory writebacks compared to an uncompressed cache.

B. Baseline and Victim Cache Implementation

In this section, we address different scenarios for hits and misses to compressed cache lines. To simplify our explanation, we use LRU replacement in the Baseline Cache, and random replacement in the Victim Cache. We ensure all Victim Cache lines are clean. The following sub-sections highlight how our Base-Victim opportunistic compression architecture handles various scenarios of hits and misses.

1) Compressed LLC Miss

On a miss to the compressed LLC, a replacement victim is identified from the Baseline Cache based on the baseline replacement policy. If the replaced line is modified, its data is written back to memory to make it a clean line. The incoming line occupies the same location as the chosen replacement victim. If the incoming line can be compressed to fit with the existing line in the same way from the Victim Cache, the Victim Cache line is retained; otherwise, the Victim Cache line is silently evicted (since it is a clean line). The replaced line from the Baseline Cache is then opportunistically stored to any way that would fit it in the Victim Cache, or evicted if it cannot fit in any victim way.

For our Victim Cache implementation, we use a replacement policy inspired by ECM [4]. We first search for the way that can fit the victim line. Then among all the candidates, we select the way with the largest size of the base partner line. However, for examples in this section, we assume random replacement in the Victim Cache. We study variations of Victim Cache policies in Section VI.B.4.

Compressed LLC Miss Example. Figure 4 shows an example scenario where the processor requests a line Z that misses the compressed LLC. We assume LRU replacement in the Baseline Cache, and random replacement in the Victim Cache to simplify the explanation of these scenarios. However, this could be handled using any other replacement policy.

In this example, neither the Baseline (B) nor the Victim (V) sets contain line Z, so we do the following steps:

(1) We identify the LRU victim B from way 3 of the Baseline Cache.
(2) If B is modified, its data is written back to memory. For inclusive caches, back invalidations are issued to the L1 and L2 caches to ensure the most recent data is written back.
(3) Since Z requires 6 segments of space, the partner line Y sharing physical way 3 has to be victimized as well. Y is silently evicted since all Victim Cache lines are clean.
(4) Z is inserted into way 3 of the Baseline Cache.
(5) We randomly pick a victim in the Victim Cache section (V) that can accommodate the victim from the Baseline Cache – E, in this case, from way 1. Note that X (from way 2) could not have been picked since it frees up only 2 segments whereas B needs 3 free segments. The only other possible victim is F (Way 0).
(6) E is evicted (with no writeback since it is clean), and B is stored in way 1 of the Victim Cache.

Note that next time the processor makes a request to line B, it will be installed back in the Baseline Cache, i.e., gets treated as a fill into the Baseline Cache which could trigger either the eviction of the baseline LRU way or its insertion in the Victim Cache, as shown in the next sub-section. Hence, the Victim cache behaves as a cache of victim blocks that would have been evicted without compression. We ensure that the victim cache is always clean with respect to memory. If the cache is inclusive, we send appropriate back-invalidations to the L1 and L2 caches, and write back any modified data to memory. Therefore, we only need to perform (at most) one writeback for each cache fill, which

simplifies the replacement logic. This is less complex than current proposals (e.g., VSC) that may require multiple evictions to fill in one line.

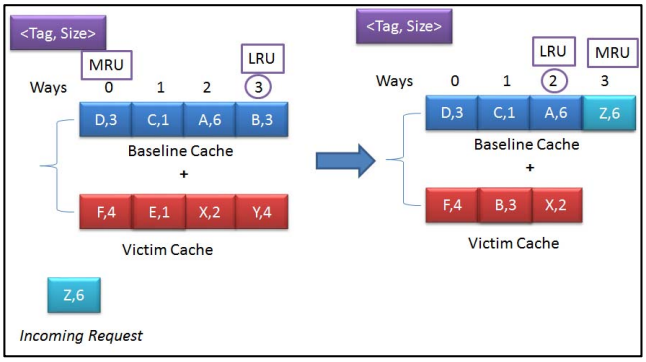

Figure 4. Compressed LLC Miss Example. The LRU order is 0 (MRU), 1, 2, and 3 (LRU). The compressed size of a block (in 8B segments) is shown after the comma. The left side shows the state before inserting Z, and the right side shows the state after inserting Z. Note that B could be inserted in ways 0 or 1 in the victim cache (depending on the replacement policy of the victim cache). Here with random replacement, B replaces E in way 1.

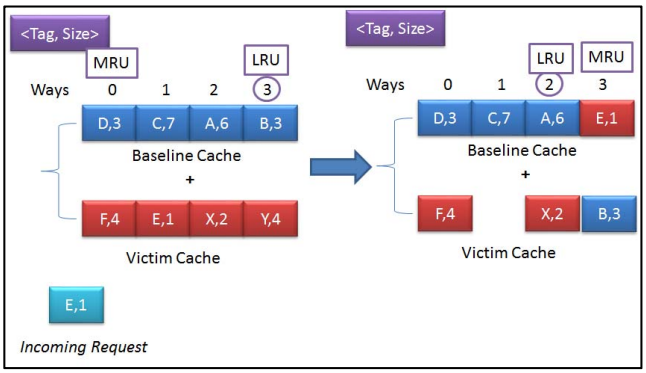

Figure 5. Compressed LLC Hit Into a Victim Cache Line. The left side represents the state before accessing E, and the right side represents the state after accessing E.

2) *Read Hit to the Victim Cache*

On a read that hits in the Victim Cache, we need to promote that line to the Baseline Cache. We therefore choose a replacement victim in the Baseline Cache using the baseline replacement policy. The promoted victim line occupies the replaced line's place. The Victim Cache partner line for the replaced way is only kept if it fits with the promoted line. The replaced line from the Baseline Cache is then opportunistically stored in any way that would fit it in the Victim Cache, or evicted if it does not fit.

Compressed LLC Victim Read Hit Example. Figure 5 shows an example scenario where the processor requests a line E which hits in the Victim Cache (V). E is a line that would not be in the cache without compression. However, we still need to maintain the baseline replacement policy to avoid losing performance. In this case:

(1) The LRU victim B is chosen from way 3 of the Baseline Cache.

(2) If B is modified, its data is written back to memory. For inclusive caches, back invalidations are issued to the L1 and L2 caches to ensure the most recent data is written back.

(3) E is de-allocated from the Victim Cache and installed in the Baseline Cache.

(4) B is then inserted in the Victim Cache in way 3 (which becomes the MRU way) by silently evicting its partner line Y since both B and E fit in the same physical line. To simplify implementation, we do not attempt to relocate the victim line Y in the Victim Cache.

By always moving lines that hit in the Victim Cache into the Baseline Cache, we ensure that the Baseline Cache (B) of the Compressed LLC always mirrors the state of an Uncompressed LLC.

3) *Write Hit to the Victim Cache*

This case will not occur for inclusive caches. In our architecture, the Victim Cache lines are always clean. Before we insert a Baseline Cache replacement victim into the Victim Cache, we send appropriate back-invalidations to the L1 and L2 caches, and write any modified data to memory. As a consequence, we do not reduce writebacks to memory, and subsequent writes to those victim lines will need to acquire the appropriate (exclusive) permissions.

However, our architecture can be implemented for non-inclusive caches where we do not enforce the restriction that Victim Cache lines are clean. In that case, the Victim Cache write hit is handled in exactly the same way as a Victim Cache read hit. The only exception is that the newly written line in the Victim Cache needs to be compressed, potentially to a different size. It is then promoted to the Baseline Cache, and the replaced line from the Baseline Cache is opportunistically stored in the Victim Cache if it fits in any of the victim ways, or evicted otherwise.

4) *Read Hit to the Baseline Cache*

This case is handled in exactly the same way as a read hit in an uncompressed cache, except that the line is decompressed before being sent back to the CPU and lower level caches. The replacement bits in the metadata of the Baseline Cache are modified as they would be in an uncompressed cache. Line sizes do not change on a read, so no data movement occurs in the Baseline and Victim caches.

5) *Write Hit to the Baseline Cache*

This case is handled in exactly the same way as a write hit in an uncompressed cache, except that the victim partner line sharing the same physical way is evicted if the Baseline Cache line grows so the new size would not fit the existing victim. To simplify our implementation, we evict the Victim Cache line right away even if it was the MRU line in the Victim Cache set. Since our victim cache is always clean with respect to memory, this silent eviction only requires changing the state of the Victim Cache line to "Invalid". However, it is

possible to implement a more complex policy where the victim line can be moved to a different way.

C. Area Overheads

For a 2MB 16-way uncompressed cache, each way requires 64B of data, 31 bits for the address tag (assuming 48 bit addresses that include 6 offset bits and 11 index bits), and an additional byte for metadata including replacement, coherence state and tracking bits. The opportunistic compressed cache adds an additional address tag for the Victim Cache. The Victim Cache is always clean and uses random replacement, so it only needs one "valid" bit of metadata (no other coherence or replacement bits are needed). However, to simplify our implementation, we need to add size information to both the Baseline Cache and Victim Cache to simplify victim selection and partner line victimization. This adds 4 bits of metadata to each tag to align compressed lines at 4-byte boundaries (i.e., to support 16 different sizes for compressed lines). It should be noted that our evaluation is based on 4B segments (not 8B as in examples in this section). Overall, we add an additional 31-bit address tag and 9 extra bits of metadata (4x2 for size, and 1 valid bit) for each original way. The area overhead for this is 40b/(39b+512b) = 7.3% of the original (tag + data) array size. We use the compression and decompression logic area estimates from [32] where that logic accounts for 1.2% additional area. With these estimates, the overall area overhead (as a fraction of cache area) is 8.5% for a 2MB cache.

V. Evaluation Methodology

We evaluated our proposed architecture using a cycle-accurate execution-driven x86 simulator. We model a 4 GHz 4-way dynamically scheduled out-of-order issue core, similar to the state-of-the-art Intel® Core™ processor [17]. Each core has its own private L1 and L2 caches. We model a 32KB L1 instruction cache, a 32 KB 8-way L1 data cache, and a unified 256 KB 8-way L2 cache. For single-thread studies, we model a 2 MB 16-way last-level cache (LLC). For multi-program simulations, we model a 4MB 16-way LLC. All caches in the hierarchy use a 64B line size. The LLC is inclusive of the core caches and uses 1-bit Not Recently Used (NRU) [14] as the replacement policy. The load-to-use latencies for L1, L2 and L3 are 3 cycles, 10 cycles, and 24 cycles respectively. For all simulations, we model the main memory as two channels of DDR3-1600. The DRAM has timing parameters of 15-15-15-34 (tCL-tRCD-tRP-tRAS). We model per-core aggressive multi-stream instruction and data prefetchers for the L1, L2 and LLC.

Our single-threaded traces are drawn from four workload categories as outlined in Table I. We use 100 traces representing different execution phases of benchmarks in these categories. Each trace is run for 200 million instructions. We report performance in terms of instructions per cycle (IPC). Out of these 100 traces, we found 60 traces to be sensitive to cache performance. All analysis and results will be presented for these 60 traces. In Section VI.B.5, we will show the performance impact on the remaining 40 traces. We use the geometric mean to present average normalized IPC and miss rate ratios across traces.

We also present results for 20 4-way multi-programmed workloads prepared by mixing four representative single-threaded traces from the workload categories. Within a mix, each thread executes 100 million instructions. If a thread finishes its performance simulation phase early, it continues executing so that we can model the shared LLC contention properly. The mix terminates when every thread has finished its performance simulation phase. Hence, a minimum of 400 million instructions are retired. We report performance as the weighted speedup of all threads.

TABLE I: WORKLOADS

Category	Total Traces	Benchmarks
SPECCPU 2006 FP (FSPEC) [35]	30	CactusADM, Milc, LBM, Wrf, Sphinx3, GemsFDTD, Soplex, Calculix, Bwaves
SPECCPU 2006 Integer (ISPEC) [35]	29	Xalancbmk, Sjeng, Gobmk, Omnetpp, Astar, Gcc, Libquantum, Mcf
Productivity	14	Sysmark[5] , Winrar, Win-compression
Client	27	Octane Browser Benchmarks [27], Speech Recognition, Cinebench [10], 3DMark [12]

We use Base Delta Immediate (BDI) as our LLC compression algorithm [28]. We align the compressed data to a 32-bit (i.e., 4 byte) boundary. We added two cycles for the decompression latency and an additional cycle for tag lookup (since tags have been doubled). Uncompressed and Zero lines do not suffer any decompression latency since we add the size information to the tag metadata: Zero blocks and uncompressed blocks can be detected from the data size field when we read the tag and metadata, and therefore we do not need to decompress the data read from the LLC data array for these block types.

For the majority of our results, we use the Not-Recently-Used replacement policy in the Baseline Cache. We also use a replacement policy inspired by ECM [4] for the Victim Cache. However, we discuss sensitivity to Baseline Cache and Victim Cache replacement policies in Section VI.B.

Earlier proposals like VSC-2X [1] and DCC [32] rely on distributing compressed cache lines across ways in a given set. When simulated on functional cache models, these policies come close to an 80% increase in cache capacity. This is significantly higher than our opportunistic Base-Victim architecture. Unfortunately, as discussed in Section II, these architectures require significant changes to the SRAM data array layout, potentially needing deeper cache pipelines that will add more latency for all cache lookups. This makes it difficult to compare these policies to our opportunistic Base-Victim architecture. We therefore do not compare the IPC from such policies to our proposal. The two-tag architecture does not require any changes to SRAM data

array (it only changes the controller), so comparing it to an uncompressed cache is more straightforward.

VI. Results

A. Single-Core Performance

We split the single-threaded benchmarks into two categories depending on their compression ratios. Of the 60 cache-sensitive traces, 10 traces have an average compressed block size higher than 75% of the uncompressed size. These 10 traces are not expected to gain much performance from compression. For the remaining 50 traces, the average block size after compression is 50% of the uncompressed size, so we classify them as compression-friendly. On average across all 60 traces, the compressed block size is 55% of the uncompressed size.

We first show the performance of the simple two-tag architecture where we always victimize partner lines that don't fit with the fill line. Figure 6 shows a line graph for this scheme. The graph shows the normalized performance (IPC Ratio) and normalized read miss rate (Memory Read Ratio) compared to an uncompressed cache baseline. For workloads that have significantly higher miss rate, normalized IPC is much lower than the baseline. The opposite is true for workloads that have significantly lower miss rates. As discussed earlier in Section III, there are significant negative outliers because of partner line victimization. Despite the capacity increases due to compression, this scheme loses 12% performance, on average, over an uncompressed baseline. 37 out of the 60 traces have a lower IPC (IPC ratio less than 1) compared to the uncompressed cache.

To reduce the partner line victimization problem, we use a replacement policy similar to ECM [4] tailored to our baseline two tag architecture. We search for a tag (based on NRU) which does not need to evict its partner. Among all such victim candidates, we chose the one with the largest compressed size. Figure 7 shows the performance of the modified two-tag architecture. On average, this scheme gives 4.7% performance gain for compression-friendly benchmarks. However it causes a 3.8% performance loss for workloads that don't compress well. There are significant negative outliers (up to 14%), which also correlates with more DRAM traffic. Nearly half the traces (27 out of 60) lose performance compared to an uncompressed cache. Moreover, this scheme causes more back-invalidations than the baseline. This is because, in an inclusive cache hierarchy, partner lines that have been victimized could still be present in the L1/L2 caches and hence need to be back-invalidated.

We also tried out other variants of the replacement and insertion policies that were found to be inferior to the modified policy shown in Figure 7, and are hence not shown in this paper. As discussed in Section III, pairing and replacement policies are difficult to maintain since LRU and compressibility may not correlate. As a result, all variants that we tried out had significant negative outliers.

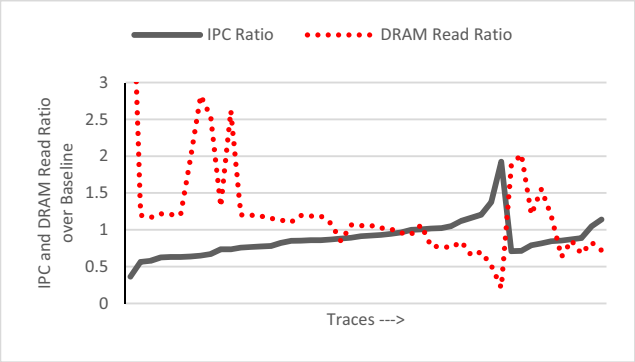

Figure 6. Normalized IPC and DRAM Read Ratios compared to a 2MB uncompressed baseline for the two-tag architecture. The left side shows gains for compression friendly traces, and right side shows traces that have low compression. Higher DRAM read ratios usually correspond to lower performance (IPC).

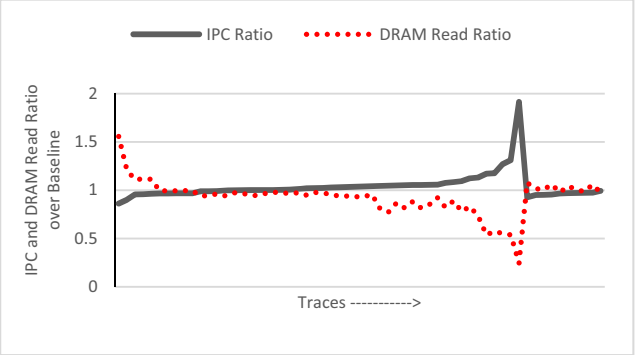

Figure 7. Normalized IPC and DRAM Read Ratio compared to a 2MB LLC for the modified two-tag architecture.

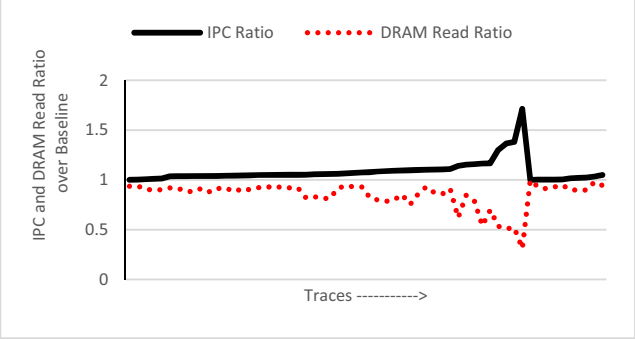

Figure 8. Normalized Performance and DRAM Read Ratio compared to a 2MB Baseline for the Opportunistic Cache Compression. LLC misses are always less than or equal to baseline.

Figure 8 shows the performance (IPC) improvement with our opportunistic Base-Victim compression architecture across all workloads. The graph shows that there is only one negative outlier out of 60 (that loses 0.01% IPC). We also show that reads from memory are always lower than or the same as baseline. The small losses are caused only by the additional latency due to decompression and tag lookup.

For compression-friendly benchmarks, opportunistic compression gives an average (geometric mean)

improvement of 8.5% and reduces read misses by an average of 16%. Figure 9 shows the average performance improvement per workload category. We also show the gains from a 3MB uncompressed cache. We construct a 3MB cache by adding 8 ways to a 2MB, 16-way baseline. We add an extra cycle of latency because of the increase in tag and data array sizes. On average, a 3MB uncompressed cache also gives a performance improvement of 8.5% and reduces read misses by 16.1% for compression-friendly traces. Opportunistic compression hence, gives a performance similar to a 50% larger (3MB) LLC. In other words, by adding 8.5% to LLC area, we gain performance equivalent to a 50% increase in cache capacity.

For the 10 traces that are not compression-friendly, opportunistic compression gives an average IPC improvement of 1.45%. This is expected, since compressibility of these workloads limit their potential performance gain. Across all cache-sensitive workloads, opportunistic compression gives an average 7.3% performance gain over a 2MB uncompressed cache, whereas a 3MB uncompressed cache gains an average 8.1%.

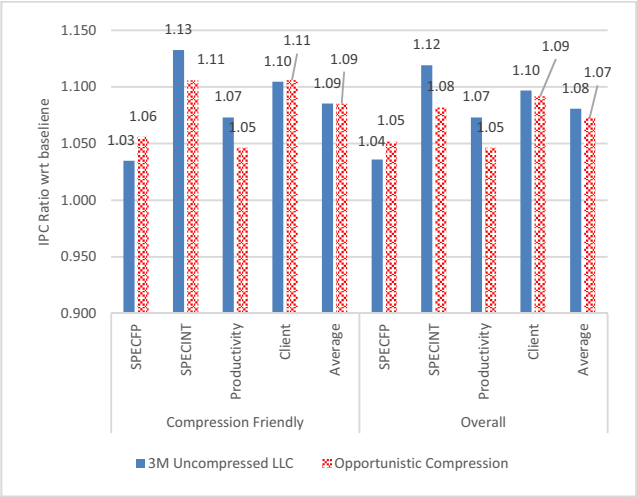

Figure 9. Performance Per category for Opportunistic compression as compared to a 2MB Uncompressed LLC. The left side shows performance for compressible workloads, and the right side shows overall performance for all cache-sensitive workloads.

B. Sensitivity Analysis

1) Effect of LLC Associativity

In our opportunistic compression architecture, we use 32 tags, instead of 16, in each set, and add an additional 1 cycle of latency for tag access. To analyze the sensitivity of our results to baseline associativity, we simulated a 16-tags-per-set version of the opportunistic compressed cache. In that version, the baseline (uncompressed) associativity would be 8-way, and compression adds 8 additional ways per set. Compared to a baseline 2MB 16-way set-associative cache, the 16-way opportunistic compression version gives an average performance gain of 6.2% (vs. 7.3% for the 32-way version). On the other hand, increasing the associativity of the baseline uncompressed cache from 16 to 32 yields insignificant (almost zero) performance gains.

2) Effect of Base Replacement policy

We used a 1-bit NRU replacement policy for this study. Significant prior work has been done to improve replacement policies for the LLC. In this section we study the impact of advanced replacement policies when applied to the Baseline Cache.

We consider two recent replacement policies. The first is SRRIP [20] that uses 2 bits per cache line for managing ages. The second is CHAR [7] that uses set-dueling for learning workload cache behavior and then sends downgrade hints on L2 cache evictions. The CHAR replacement policy can manipulate the ages, not just on fill or hit, but also based on hints. We implement the CHAR replacement policy with 1 bit ages and not on top of SRRIP, as was done in [7]. Since the Base-Victim architecture maintains the Baseline Cache behavior, advanced replacement policies can be seamlessly integrated.

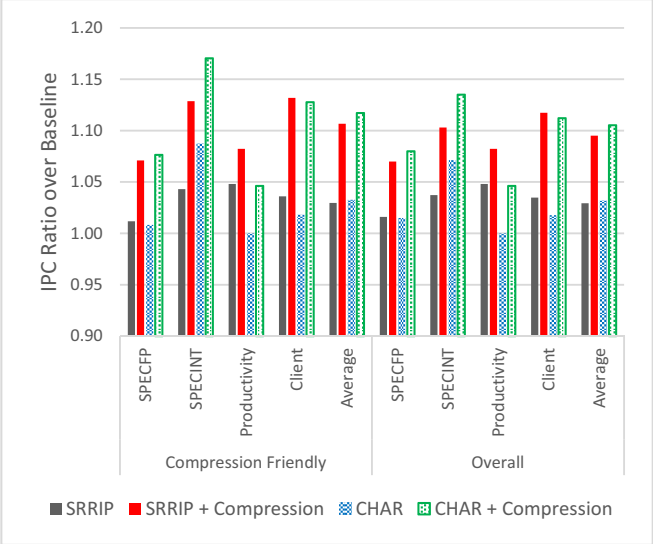

Figure 10. Impact of Baseline Cache replacement policy on opportunistic compression.

Figure 10 shows the performance gains for our opportunistic compression architectures with these replacement policies as baseline. On top of a 1-bit NRU replacement policy, SRRIP gives a 2.9% gain in performance and CHAR gives a 3.2% performance gain. Opportunistic cache compression gives 6.4% gain on top of SRRIP-managed baseline, and 7.2% gain on top of a CHAR-managed baseline. We found that, as in the case of NRU replacement, the baseline hit rate is not decreased and there are no negative outliers. This illustrates that our opportunistic (Base-Victim) compression architecture is synergistic with advanced replacement policies. We maintain performance improvements of replacement, and gain more performance because of compression.

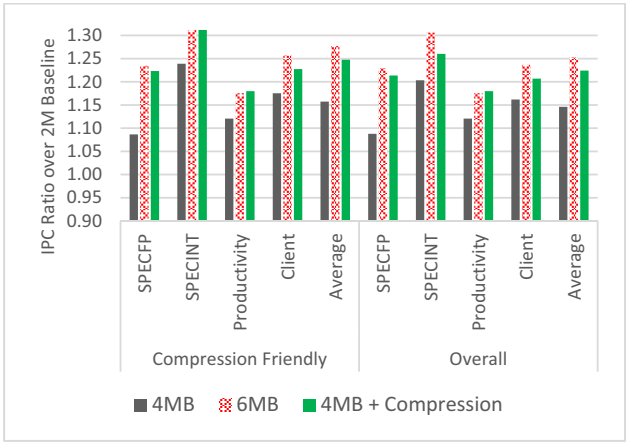

Figure 11. Sensitivity to LLC Size. We show IPC gains over a 2MB uncompressed cache for a 4MB uncompressed cache, a 6MB uncompressed cache, and 4MB compressed cache. The left side shows workloads with good compression, and the right shows all cache-sensitive workloads.

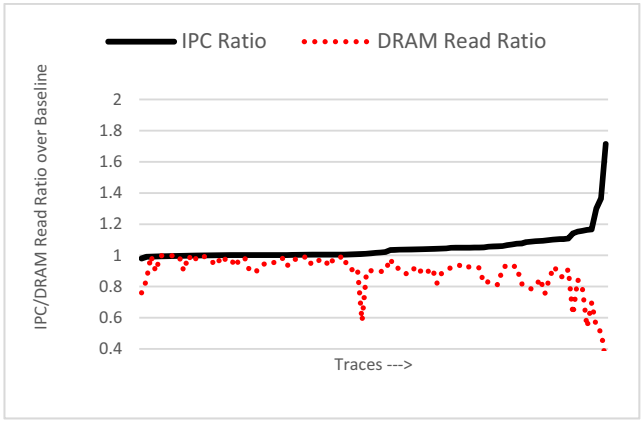

Figure 12. Normalized IPC and DRAM Read Ratio as compared to a 2MB uncompressed Baseline for all 100 traces.

3) *Effect of increasing LLC Size*

Figure 11 shows the performance gains for Opportunistic compression for a 4MB cache. A 4MB uncompressed cache gains 15.8% over a 2MB uncompressed cache. Opportunistic compression provides an additional 6.8% performance gain on top of it. A 50% larger (6MB) cache achieves 9% average performance gain.

4) *Effect of Victim Cache Replacement Policy*

We observe that for compression-friendly benchmarks, the average compressed size is close to 50% of uncompressed size (i.e., 2X compression ratio), but effective cache capacity only increases by 50% (i.e., 1.5X). This suggests that the Victim Cache needs to be managed better.

We tried various other variations of Victim Cache replacement, including LRU and a mix of LRU and size-based replacement. Unfortunately, none of these variants showed any significant improvement over the current policy. We leave the exploration of better Victim Cache replacement policies for future work.

5) *Results on full trace list*

In the previous sub-sections, we only considered the traces that were sensitive to LLC performance. Figure 12 shows a line graph for all 100 traces including cache-insensitive workloads. There are no significant negative outliers for opportunistic cache compression. Overall, our opportunistic compressed cache architecture gains an average 4.3% performance over an uncompressed cache baseline; whereas a 50% larger (3MB) uncompressed cache gains an average performance of 4.9%.

C. *Multi-Core Performance*

The results for 4-way multi-program (MP) workloads are summarized in Figure 13. We use normalized weighted speedup $\sum_{i=0}^{n-1}\left(\frac{IPC_{new}}{IPC_{base}}\right)$as the metric for reporting multi-program workload performance. Compared to a 4MB baseline, opportunistic compression gains 8.7% performance, on average, whereas a 6MB (50% larger) cache gains 9%. Compared to an 8MB baseline, it gains 11.2% performance, on average, whereas a 12MB (50% larger) cache gains 15.7%. As we observed in single-core results, there are no negative outliers for our opportunistic compression architecture. The hit rate for all the multi-program mixes is at least as high as the hit rate of an uncompressed cache.

D. *Power Analysis*

Cache compression saves power by reducing the number of requests that go to memory. However, it also adds power because of the compression and decompression logic, as well as leakage and active power needed for the extra tags and metadata. Furthermore, our opportunistic cache compression mechanism needs to migrate compressed cache lines from the Baseline Cache to the Victim Cache and vice-versa. Since the base way and victim way may be physically distinct ways, data should be read out from the Baseline Cache and written into the Victim Cache. Similarly, on a Victim Cache hit, data should be read out and moved to the Baseline Cache. Both operations require additional power. On a cache fill or write-back, compressed data will be written to the cache without overwriting an existing partner line.

Most SRAM implementations have byte or word enable controls. If that is the case, these word enables need to be enabled for only the specified number of words that are being written to save power. However, if word enables are not available, then a read modify write operation needs to be done every time a fill or a writeback happens to ensure the integrity of the compressed partner line.

For 2MB single-thread simulations, opportunistic compression saves 16% of all reads to memory, but does not save any writes to memory (as the victim cache is clean). This results in a 12% average reduction in memory bandwidth, since write bandwidth is not affected. On the

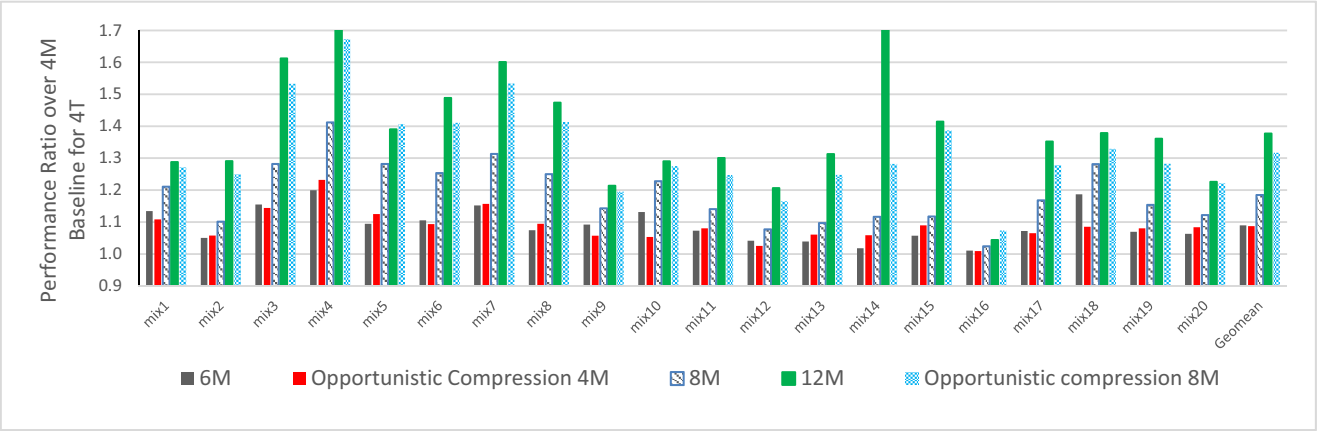

Figure 13. Performance gain for multi-program workload mixes (4 threads each) with 4MB uncompressed cache as the baseline.

other hand, it adds about 31% additional accesses to LLC, primarily because of the additional hits and necessary data movement between the Baseline Cache and Victim Cache.

To estimate the power impact of opportunistic compression, we use the Micron power calculator [25] for estimating the DRAM array energy in the main memory. We use CACTI [26] to estimate the dynamic and leakage energy expended by the tag/state SRAM of the LLC, assuming a 22-nm process. The compression/decompression power numbers for BDI are scaled to a 22nm process based on [23].

Overall, opportunistic cache compression saves 6.5% energy in the memory + cache subsystem. However if word enables are not present, the energy savings drop to 2.2%, primarily because of read modify write operations on fills and writebacks. Figure 14 shows the energy ratio as compared to baseline, for all 100 traces. Also plotted for correlation is the DRAM read ratio for each trace. Power savings are highest when DRAM bandwidth reduction (i.e., reduction in read miss traffic) is substantial. There are a few traces where we increase power as compared to the baseline (by up to 2.3%). For such traces, the reduction in DRAM bandwidth is not sufficient to offset power loss because of compression and migration. Without word enables, there are even more outliers and we see up to 6% higher power on some traces.

VII. Related Work

A. Cache Compression Architectures and Algorithms

Prior work has explored a number of hardware compression algorithms to compress cache data. These algorithms are orthogonal to our opportunistic compressed cache architecture, since we can use any of the previously proposed compression algorithms. The only difference would be in the compressibility, area and latency overheads of the chosen compression algorithm.

Many compression architectures are motivated by the concept of value locality introduced by Lipasti, et al. [24], which indicates the likelihood that a previously-seen value will be re-observed for the same storage location. Yang and Gupta [40] propose augmenting the data array of the L1 cache with a frequent value cache (FVC) that stores the most frequent data values, therefore saving the energy needed for accessing the L1 data array. Islam and Stenstrom [18] apply a similar concept to speed up loads of zero values through their Zero-Value Cache (ZVC). At the block level, Dusser, et al. [11] propose augmenting the data cache with a zero-content cache that stores the addresses of null (zero) blocks, therefore increasing effective cache capacity. Tian et al. [36] propose cache line de-duplication, where duplicate cache blocks are detected, and a level of indirection is used to allow multiple addresses to access the same data block.

Several compressed cache architecture proposals decouple tags and data to allow variable-sized lines to co-exist in the cache. Hallnor and Reinhardt [16] extend their indirect index cache proposal to support compression by allocating variable amounts of storage to different cache blocks. Pekhimenko et al. [29] explore extending the V-Way cache [30] to support multiple line sizes. We discussed the Decoupled Variable-Segment Cache (VSC) [1], the Decoupled Compressed Cache (DCC) [32] and the Skewed Compressed Cache (SCC) [33] proposals in Section II.

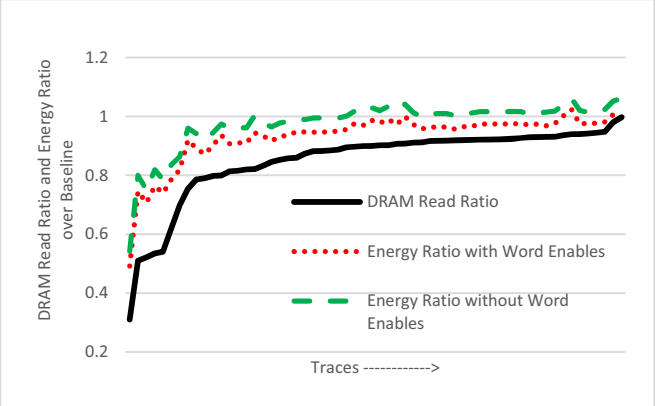

Figure 14. Energy Reduction for Opportunistic Compression compared to uncompressed baseline. Having word enables in the cache, helps reduce read modify writes and saves substantial power.

Alameldeen and Wood [1] propose Frequent Pattern Compression (FPC), a significance-based algorithm that detects frequent patterns in each 32-bit word of a cache line, and compresses the line if it can save enough bits. Chen, et al. [9] propose the Cache Packer (C-Pack) algorithm that augments frequent pattern detection with a dynamic dictionary to detect redundancy within a cache line. Pekhimenko, et al. [28] propose the Base-Delta-Immediate (BDI) compression algorithm that detects whether all words in a cache line are within a short delta from a base word, and compresses cache lines with enough redundancy. Arelakis and Stenstrom [3] propose a statistical compression algorithm (SC^2) based on Huffman encoding that takes advantage of the low variability of data values over time and across applications. We chose BDI as the compression algorithm in our opportunistic compressed cache implementation since it has a fast decompression latency and a high compression ratio.

Our Base-Victim compressed cache architecture opportunistically keeps replacement victims from the base cache without sacrificing performance gains from advanced replacement policies. Kim, et al. [22] compress two lines if corresponding words can be sign-compressed to half their size. Chen, et al. [8] combine two lines if their combined compressed sizes can fit in one physical line, similar to this paper. However, cache insertion policy for both proposals does not preserve performance from advanced cache replacement policies, since lines that would exist in the baseline uncompressed cache could be evicted in many cases.

B. Cache Management and Replacement Policies

Prior work has investigated many cache replacement and management policies that target reducing cache miss rate. These policies are orthogonal to our opportunistic compressed cache architecture, since we always maintain the same cache insertion/replacement policy in the Baseline Cache. The key feature of our opportunistic Base-Victim compressed cache architecture is that we guarantee that all hits in an uncompressed cache would still be hits in our compressed cache, regardless of the underlying baseline replacement and/or insertion policy.

Prior proposals have focused on detecting blocks that are unlikely to be used in the future and picking those blocks as replacement victims. This could be achieved through insertion policies ([14], [19], [20], [31], [38], [39]) that insert blocks that are likely to be reused in a location that is unlikely to be evicted quickly. Many replacement policies detect dead blocks that are unlikely to be reused before eviction, and either completely bypass those blocks, or insert them in a vulnerable position ([7], [13], [20], [21], [39]). More advanced replacement/insertion policies use more information to predict future reuse, such as SHiP [38] which uses the data block's program counter (PC) value. However, most replacement policies do not depend on having the PC available at the last-level cache.

C. Interactions between Compression and Cache Management Policies

In much of the earlier work on cache compression, the interactions between compression and cache management policies were not considered. Prior proposals [1] assumed LRU as the underlying replacement policy, and allowed more than one block to be evicted on a cache fill, as we explained in more detail in Section II.

Two recent proposals consider tailoring cache management and replacement policies for a compressed cache. Baek, et al. [4] propose the Effective Capacity Maximizer (ECM) that attempts to maximize the compressed cache capacity by optimizing the replacement policy. In this paper, we use a replacement policy in our Victim Cache that is inspired by ECM since we also try to maximize the capacity of the Victim Cache. Pekhimenko, et al. [29] propose novel Compression-Aware Management Policies (CAMP) for compressed caches that use compressed size as an indicator to predict future reuse. They propose eviction and insertion policies to keep lines with the highest reuse probability in the cache. CAMP is evaluated on top of a V-Way cache [30] where tags and data are decoupled through a level of indirection, which is more complex than our two-tag baseline architecture. Our opportunistic compressed cache architecture can be adopted to implement CAMP in the Baseline Cache, which could be addressed in future work.

VIII. CONCLUSIONS

In this paper, we demonstrate that cache compression and replacement policies can interact negatively. Many workloads suffer performance losses due to the need to alter the replacement policy to accommodate compression, leading to sub-optimal replacement policies. We introduce a novel, opportunistic cache compression mechanism, Base-Victim, based on an efficient cache design. Our compression implementation retains the performance gains due to advanced cache replacement policies, while still increasing effective cache capacity. We guarantee that the cache hit rate will be at least as high as an uncompressed cache for all workloads. For cache-sensitive applications, Base-Victim achieves an average 7.3% performance gain for single-threaded workloads and an average 8.7% gain for four-thread multi-program workload mixes.

ACKNOWLEDGMENT

We thank Komal Jothi and Ragavendra Natarajan for their help during early phases of this work. We thank Shih-Lien Lu and the anonymous reviewers for their valuable and constructive feedback.

REFERENCES

[1] Alaa R. Alameldeen and David A. Wood, "Adaptive Cache Compression for High-Performance Processors," International Symposium on Computer Architecture (ISCA-31), pp. 212-223, Munich, Germany, June 2004.

[2] Alaa R. Alameldeen and David A. Wood, "Interactions Between Compression and Prefetching in Chip Multiprocessors," International Symposium on High-Performance Computer Architecture (HPCA-13), pp. 228-239, Phoenix, AZ, February 2007.

[3] A. Arelakis and P. Stenstrom, "SC2: A Statistical Compression Cache Scheme," International Symposium on Computer Architecture (ISCA-41), pp. 145-156, Minneapolis, MN, June 2014.

[4] Seungcheol Baek, Hyung Gyu Lee, Chrysostomos Nicopoulos, Junghee Lee, and Jongman Kim, "ECM: Effective Capacity Maximizer for High-Performance Compressed Caching," International Symposium on High-Performance Computer Architecture (HPCA-19), pp. 131-142, Shenzhen, China, February 2013.

[5] Business Applications Performance Corporation (BAPCo), "Sysmark 2014," Whitepaper, 2014. https://bapco.com/wp-content/uploads/2015/09/SYSmark2014Whitepaper_1.0.pdf

[6] Jonathan Chang, Ming Huang, Jonathan Shoemaker, John Benoit, Szu-Liang Chen, Wei Chen, Siufu Chiu, Raghuraman Ganesan, Gloria Leong, Venkata Lukka, Stefan Rusu, and Durgesh Srivastava, "The 65-nm 16-MB Shared On-Die L3 Cache for the Dual-Core Intel Xeon Processor 7100 Series," IEEE Journal of Solid-State Circuits, Vol. 42, No. 4, pp. 846-852, April 2007.

[7] Mainak Chaudhuri, Jayesh Gaur, Nithiyanandan Bashyam, Sreenivas Subramoney, and Joseph Nuzman, "Introducing Hierarchy-awareness in Replacement and Bypass Algorithms for Last-level Caches," International Conference on Parallel Architectures and Compilation Techniques (PACT), pp 293-304, Minneapolis, MN, September 2012.

[8] X. Chen, L. Yang, H. Lekatsas, R. P. Dick,and L. Shang, "Design and Implementation of a High-Performance Microprocessor Cache Compression Algorithm," Data Compression Conference (DCC), pp 43-52, March 2008.

[9] X. Chen, L Yang, R. P. Dick, L. Shang, and H. Lekatsas, "C-Pack: A High-Performance Microprocessor Cache Compression Algoithm," IEEE Transactions on VLSI Systems, Vol. 18, No. 8, pp. 1196-1208, 2010.

[10] CineBench, http://www.maxon.net/products/cinebench/overview.html

[11] Julien Dusser, Thomas Piquet and Andre Seznec, "Zero-Content Augmented Caches," 23rd International Conference on Supercomputing (ICS'09), pp. 46-55, 2009.

[12] Futuremark, "3DMARK 11 Benchmark," http://www.futuremark.com/benchmarks/3dmark11

[13] Hongliang Gao and Chris Wilkerson, "A Dueling Segmented LRU Replacement Algorithm with Adaptive Bypassing," First JILP Workshop on Computer Architecture Competitions, June 2010.

[14] Jayesh Gaur, Mainak Chaudhuri, and Sreenivas Subramoney, "Bypass and Insertion Algorithms for Exclusive Last-level Caches," International Symposium on Computer Architecture (ISCA-38), pp. 81-92, June 2011.

[15] K. Ghose and M.B. Kamble, "Reducing Power in Superscalar Processor Caches Using Subbanking, Multiple Line Buffers and Bit-Line Segmentation," International Symposium on Low Power Electronics and Design (ISLPED), pp. 70-75, 1999.

[16] E.G. Hallnor and S.K. Reinhardt, "A Unified Compressed Memory Hierarchy," International Symposium on High-Performance Computer Architecture, pp. 201-212, 2005.

[17] Intel Corporation, "Intel Core i7 Processor," www.intel.com/products/processor/corei7/index.htm

[18] M. M. Islam and Per Stenstrom, "Zero-Value Caches; Cancelling Loads that Return Zero" International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 237-245, Raleigh, NC, September 2009.

[19] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon C. Steely Jr., and Joel Emer, "Adaptive Insertion Policies for Managing Shared Caches," International Conference on Parallel Architecture and Compilation Techniques (PACT'08), pp. 208-219, 2008.

[20] Aamer Jaleel, Kevin Theobald, Simon C. Steely Jr, and Joel Emer, "High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP)," International Symposium on Computer Architecture (ISCA-37), pp. 60-71, Saint Malo, France, June 2010.

[21] Samira M. Khan, Yingying Tian, and Daniel A. Jiménez, "Dead Block Replacement and Bypass with a Sampling Predictor," International Symposium on Microarchitecture (MICRO-43), pp. 175-186, December 2010.

[22] Nam-Sung Kim, Todd Austin, and Trevor Mudge, "Low-Energy Data Cache Design Using Sign Compression and Cache Line Bisection," 2nd Workshop on Memory Performance Issues (WMPI'02), May 2002.

[23] Sangpil Lee, Keunsoo Kim, Gunjae Koo, Hyeran Jeon, Won Woo Ro, and Murali Annavaram, "Warped-Compression: Enabling Power Efficient GPUs through Register Compression," International Symposium on Computer Architecture (ISCA-42), pp. 502-514, Portland, OR, June 2015.

[24] H. Lipasti, Christopher B. Wilkerson and John Paul Shen, "Value Locality and Load Value Prediction," International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII), pp. 138-147, Cambridge, MA, 1996.

[25] Micron Technology Inc., "Calculating Memory System Power for DDR3. Micron Technical Note TN-41-01. http://www.micron.com/products/support/power-calc.

[26] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi, "CACTI 6.0: A Tool to Model Large Caches," HP Labs Technical Report HPL-2009-85, HP Laboratories, 2009.

[27] Octane, https://developers.google.com/octane/?hl=en

[28] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Michael A. Kozuch, Phillip B. Gibbons, and Todd C. Mowry, "Base-Delta-Immediate Compression: Practical Data Compression for On-Chip Caches," International Conference on Parallel Architectures and Compilation Techniques (PACT), Minneapolis, MN, September 2012.

[29] Gennady Pekhimenko, Tyler Huberty, Rui Cai, Onur Mutlu, Phillip P. Gibbons, Michael A. Kozuch, and Todd C. Mowry, "Exploiting Compressed Block Size as an Indicator of Future Reuse," International Symposium on High-Performance Computer Architecture (HPCA-21), pp. 51-63, February 2015.

[30] Moinuddin K. Qureshi, David Thompson, and Yale N. Patt, "The V-Way Cache: Demand Based Associativity via Global Replacement," International Symposium on Computer Architecture (ISCA-32), pp. 544-555, Madison, WI, June 2005.

[31] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely Jr., and Joel Emer, "Adaptive Insertion Policies for High Performance Caching," International Symposium on Computer Architecture (ISCA-34), pp. 381-391, San Diego, CA, June 2007.

[32] Somayeh Sardashti and David A. Wood, "Decoupled Compressed Cache: Exploiting Spatial Locality for Energy-Optimized Compressed Caching," International Symposium on Microarchitecture, Davis, CA, December 2013.

[33] Somayeh Sardashti, Andre Seznec, and David A. Wood, "Skewed Compressed Caches," 47th International Symposium on Microarchitecture (MICRO-47), pp. 331–342, Washington, D.C., December 2014.

[34] Andre Seznec, "Decoupled Sectored Caches," IEEE Transactions on Computers, Vol. 46, No. 2, pp. 210–215, February 1997.

[35] SPEC CPU 2006 Benchmarks. http://www.spec.org/cpu2006

[36] Yingying Tian, Samira M. Khan, Daniel A. Jimenez, and Gabriel H. Loh, "Last-level cache deduplication," 28th International Conference on Supercomputing (ICS '14), pp. 53–62, 2014.

[37] C. L. Su and A.M. Despain, "Cache Designs for Energy Efficiency," 28th Annual Hawaii International Conference of System Sciences, 1995.

[38] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C. Steely Jr., and Joel Emer, "SHiP: Signature-based Hit Predictor for High Performance Caching," International Symposium on Microarchitecture (MICRO-44), pp. 430-441, December 2011.

[39] Yuejian Xie and Gabriel H. Loh, "PIPP: Promotion/Insertion Pseudo-partitioning of Multi-core Shared Caches," International Symposium on Computer Architecture (ISCA-36), pp. 174-183, Austin, TX, June 2009.

[40] Jun Yang and Rajiv Gupta, "Energy Efficient Frequent Value Cache Design," International Symposium on Microarchitecture (MICRO-35), pp. 197-207, Istanbul, Turkey, December 2002.