Decoupled Compressed Cache: Exploiting Spatial Locality for Energy-Optimized Compressed Caching

Somayeh Sardashti
Computer Sciences Department
University of Wisconsin-Madison
somayeh@cs.wisc.edu

David A. Wood
Computer Sciences Department
University of Wisconsin-Madison
david@cs.wisc.edu

ABSTRACT

In multicore processor systems, last-level caches (LLCs) play a crucial role in reducing system energy by i) filtering out expensive accesses to main memory and ii) reducing the time spent executing in high-power states. Cache compression can increase effective cache capacity and reduce misses, improve performance, and potentially reduce system energy. However, previous compressed cache designs have demonstrated only limited benefits due to internal fragmentation and limited tags.

In this paper, we propose the Decoupled Compressed Cache (DCC), which exploits spatial locality to improve both the performance and energy-efficiency of cache compression. DCC uses decoupled super-blocks and non-contiguous sub-block allocation to decrease tag overhead without increasing internal fragmentation. Non-contiguous sub-blocks also eliminate the need for energy-expensive re-compaction when a block's size changes. Compared to earlier compressed caches, DCC increases normalized effective capacity to a maximum of 4 and an average of 2.2 for a wide range of workloads. A further optimized Co-DCC (Co-Compacted DCC) design improves the average normalized effective capacity to 2.6 by co-compacting the compressed blocks in a super-block. Our simulations show that DCC nearly doubles the benefits of previous compressed caches with similar area overhead. We also demonstrate a practical DCC design based on a recent commercial LLC design.

Categories and Subject Descriptors

B.3.2 [**Design Styles**]: Cache Memories.

General Terms

Performance, Design.

Keywords

Compression, Cache Design, Multicore, Energy Efficiency.

1. INTRODUCTION

Future computer systems face continuing power and energy challenges as the power per transistor scales more slowly than transistor density [12]. Caches, long used to reduce effective memory latency and increase effective bandwidth, play an increasingly important role in reducing memory system energy. Keckler [21] shows that last-level caches (LLCs) are especially important, since obtaining operands of a double-precision multiply-add from off-chip memory requires approximately 200x the energy of the operation. Thus improving effective cache utilization is important not only for system performance, but also for system energy.

Increasing LLC size can improve performance for most workloads, but comes at significant area cost. For example, the well-known "square root" power law [18] predicts that doubling LLC size will reduce misses by ~30%, on average. But it obviously doubles LLC area, which already accounts for 15–30% of the die area of most processors [10].

Cache compression seeks to increase effective cache size—by compressing and compacting cache blocks—while incurring smaller area overheads [2][5][9][22]. For example, previously proposed techniques have the potential to double effective LLC capacity, while increasing LLC area by only ~8%. Unfortunately, as summarized in Table 1 (simulation and workload details in Section 6), previous compressed cache designs (i.e., FixedC and VSC-2X, described below) fail to achieve this potential, increasing normalized effective capacity by only 1.5–1.7, on average, compared to the same size conventional LLC (Baseline).

FixedC and VSC-2X perform less well than doubling LLC size for two main reasons. First, internal fragmentation results from the way blocks are mapped to sets and, importantly, how compressed blocks are compacted within a set. For example, fixed compression designs (denoted FixedC) only compact two compressed blocks if each can fit in half the space of an

Table 1. Summary of results of the compressed caches

	LLC Area Overhead (%)	Norm Max Capacity	Norm Capacity	Norm LLC Miss Rate	Norm Runtime	Norm System Energy
Baseline	0	1	1	1	1	1
FixedC	~8%	2	1.5	0.91	0.96	0.96
VSC-2X	~8%	2	1.7	0.89	0.95	0.97
2X Baseline	100%	2	1.9	0.87	0.93	0.94
DCC	~8%	4	2.2	0.82	0.90	0.92
Co-DCC	~18%	4	2.6	0.75	0.86	0.88

MICRO-46, December 07 - 11 2013, Davis, CA, USA

uncompressed block [9][22]. Variable size compression (VSC-2X) compacts compressed blocks into a variable number of sub-blocks, e.g., 0–4 16-byte sub-blocks [2]. VSC-2X reduces internal fragmentation (within a set), since all blocks in a set share the same pool of sub-blocks. Table 1 shows that VSC-2X's better compaction improves average normalized capacity from 1.5 to 1.7. Second, to limit area overhead, both FixedC and VSC-2X only double the number of tags (plus some additional metadata) and thus can at most double effective cache capacity. Thus, on average, VSC-2X's effective capacity is significantly less than a double-sized conventional LLC (labeled 2X Baseline). Note that doubling LLC size only increases effective capacity by 1.9 times, since some workloads do not fully utilize the larger cache.

In this paper, we propose Decoupled Compressed Cache (DCC). DCC uses decoupled super-blocks (also known as sectors [31])[1] to increase the maximum effective capacity to four times the uncompressed capacity, while using comparable area overhead to previous cache compression techniques. DCC uses super-blocks—four aligned contiguous cache blocks that share a single address tag—to reduce tag overhead. Each 64-byte block in a super-block is compressed and then compacted into 0–4 16-byte sub-blocks. DCC decouples the address tags—allowing any sub-block in a set to map to any tag in that set—to reduce fragmentation within a super-block [31]. Decoupling also allows sub-blocks of a block to be non-contiguous, eliminating the re-compaction overheads of previous variable size compressed caches [2]. An optimized Co-DCC design further reduces internal fragmentation (and increases effective capacity) by compacting the compressed blocks from a super-block into the same set of sub-blocks.

Using the GEMS full-system simulator [26] to model an 8-core multicore system with a baseline 8MB LLC, we show that DCC can improve average performance and system energy by 10% and 8%, respectively. Importantly, this is better than a conventional LLC of twice the capacity, and uses only 8% more area than Baseline. In comparison with FixedC and VSC-2X, DCC nearly doubles the performance and energy benefits for comparable area overheads. Co-DCC further reduces runtimes and system energy, but at the cost of some additional complexity.

This paper makes the following contributions:

- DCC uses decoupled super-blocks to increase the effective number of tags with low overhead and little internal fragmentation.
- DCC stores compressed data in non-contiguous sub-blocks to eliminate re-compaction overheads when a block's compressed size changes.
- Co-DCC further reduces internal fragmentation by compacting the blocks of a super-block into the same set of sub-blocks.
- DCC and Co-DCC provide more effective capacity, on average, than a conventional cache of twice the size, while increasing cache area by only 8% and 18%, respectively.
- We present a concrete design for (Co-)DCC and show how it can be integrated into a recent commercial LLC design (AMD Bulldozer LLC) with little additional complexity.

In the rest of the paper, we present the background on compressed caches in Section 2, show the potential in exploiting spatial locality for improving compression effectiveness in Section 3, present DCC design in Section 4, describe hardware complexities in Section 5, describe the evaluation methodology and experimental results in Section 6, discuss the related work in Section 7, and conclude the paper in Section 8.

2. BACKGROUND ON COMPRESSED CACHES

Data compression has the potential to increase the effective capacity of every level of the memory hierarchy [1][2][15]. Compression is widely used to increase the capacity of disk storage systems. IBM's MXT effectively doubles main memory capacity [1]. Cache compression has been studied for every level of the cache hierarchy to increase effective capacity, reduce miss rates, improve performance [2][9][16][24] and reduce energy [22][23]. Cache compression is harder than other levels of the memory hierarchy, since performance is sensitive to cache latency, especially for L1 and L2 caches. But as multicore systems move to having three or more levels of cache, the sensitivity to LLC latency decreases allowing systems to consider more effective, longer latency (de-)compression algorithms.

Figure 1(a) illustrates the trade-off between decompression latency and compression ratio (i.e., original size over compressed size) for three compression algorithms. A simple zero-block detection algorithm (denoted ZERO) has single-cycle decompression latency, but only achieves an average compression ratio of 1.5 and only really benefits a few workloads [14]. Adding a more complex significance-based compression algorithm (denoted FPC+Z), works for a broader range of workloads and improves the average compression ratio to 2.8, but increases decompression latency to five cycles [2]. Finally, adding dictionary-based compression (denoted CPACK+Z), increases the average compression ratio to 3.9 and the decompression latency to 9 cycles (see section 5.1) [9]. However, because multi-megabyte LLCs already have relatively long access times (e.g., 30 cycles) and very high miss penalties (e.g., greater than 150 cycles and ~60 nJ), the benefit of higher compression ratio has the potential to outweigh the longer decompression pipeline. In the remainder of this paper, we use CPACK+Z, which combines the C-PACK algorithm [9] with zero-block detection [14]. In Section 5, we describe these compression algorithms in detail.

Such a high compression ratio suggests the potential for a similarly large normalized effective cache capacity; that is, the number of compressed blocks stored divided by maximum number of uncompressed blocks that could be stored. Unfortunately, compressed caches fail to achieve this potential for three main reasons. First, all hardware caches map blocks into sets, introducing an internal fragmentation problem since a compressed block must (at least in current designs) be stored entirely within one set. In Figure 1(b), the BytePack column represents an idealized compressed cache with infinite tags that compacts compressed blocks on arbitrary byte boundaries. BytePack degrades average normalized effective capacity to 3.1, on average. Second, practical compressed caches introduce a second internal fragmentation problem by compacting compressed blocks into one or more sub-blocks, rather than storing compressed data on arbitrary byte boundaries. The column labeled

[1] This paper uses the unambiguous terms super-block, block, and sub-block, rather than the original, but sometimes confusing terms sectors, blocks, and segments.

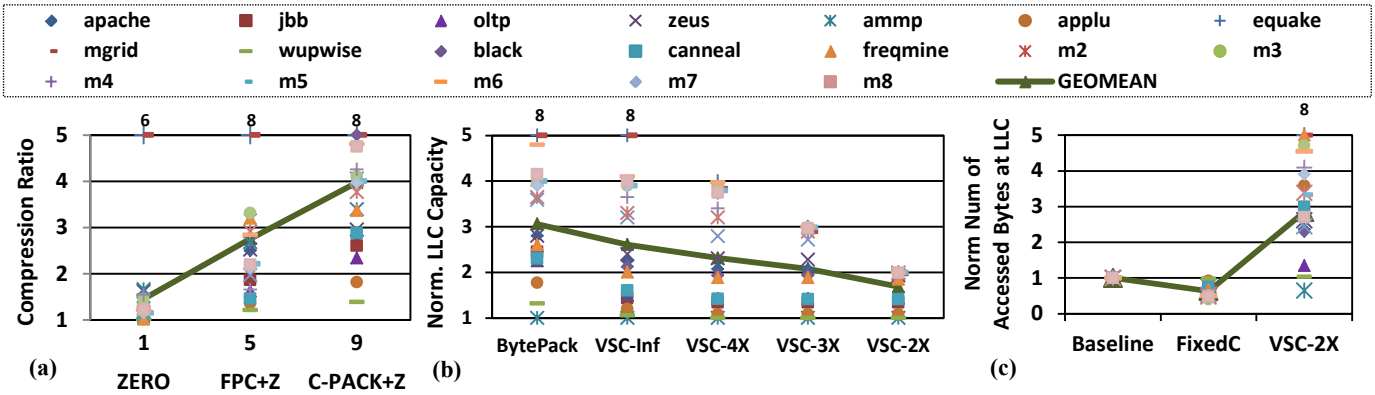

Figure 1. (a) Compression ratio of different algorithms (b) Normalized LLC effective capacity of different compressed caches (c) VSC overhead on the number of LLC accessed bytes

VSC-Inf in Figure 1(b) illustrates that compacting compressed blocks into 0–4 16-byte sub-blocks (but with infinite tags per set) degrades normalized effective capacity from 3.1 to 2.6, on average. Finally, compressed caches have a fixed number of tags per set. The remaining columns in Figure 1(b) illustrates that reducing the number of tags from infinite to a more practical twice Baseline, degrades the normalized effective capacity from 2.6 to 1.7, on average.

Figure 1(b) results suggest two approaches to unlocking the potential of cache compression. First, reduce the internal fragmentation within a set. However, this must also be done with care in today's energy constrained environment. VSC-2X relaxes the mapping constraint between tags and data and compacts compressed blocks into a variable number of contiguous sub-blocks [2]. VSC-2X can compact more blocks in the cache than a simple FixedC compressed cache, which only compacts compressed blocks into half blocks (i.e., 32-byte sub-blocks). However, VSC needs to repack the sub-blocks in a set whenever a block's size changes to make contiguous free space. This can significantly increase the cache bank occupancy and dynamic energy. Figure 1(c) shows the average number of accessed bytes at LLC normalized to Baseline. FixedC decreases the average number of accessed bytes by 36% compared to Baseline due to accessing shorter compressed blocks. On the other hand, VSC-2X increases the number of accessed bytes at LLC by nearly a factor of three since it needs to repack sets (copying almost half a set, on average). This significantly increases LLC dynamic energy, as discussed in Section 6.

The second approach to improving cache compression is to increase the number of tags per set. Figure 1(b) shows that increasing the tags from twice the baseline to four times the baseline increases the normalized effective capacity from 1.7 to 2.3, on average. However, done naively, this significantly increases the area overhead. Figure 2 shows the area overhead (compared to Baseline) versus normalized effective capacity. A variable size compression cache (VSC-2X) with twice as many tags as Baseline [2] increases LLC area by 8%. However, quadrupling the number of tags (VSC-4X) increases LLC area by ~21%, which would increase total die size by 3–6%.

In summary, we seek a new LLC compressed cache design that (a) increases the number of tags per set, without significantly increasing the metadata overhead and (b) improves the energy efficiency of compacting compressed blocks to reduce internal fragmentation.

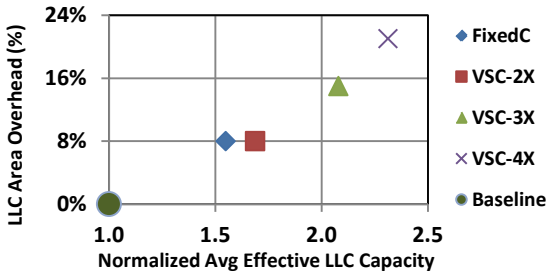

Figure 2. LLC area overhead vs. Norm LLC capacity

3. EXPLOITING SPATIAL LOCALITY

Although current multicore caches typically support a single block size, most workloads exhibit spatial locality at multiple granularities. Figure 3 shows the distribution of neighboring blocks in a conventional LLC with a tag per 64-byte block (workloads and simulation parameters described in Section 6). Neighboring blocks are defined as those in a 4-block aligned super-block (i.e., aligned 256-byte region). The graph shows the fraction of blocks that are part of a Quad (all four blocks in a super-block co-reside in the cache), Trios (three blocks out of four co-reside), Pairs (two blocks out of four co-reside), and Singletons (only one block out of four resides in the cache). Pairs and Trios are not necessarily contiguous blocks, but represent two or three blocks, respectively, that could share a super-block tag. Although access patterns differ, the majority of cache blocks reside as part of a Quad, Trio, or Pair. For applications with streaming access patterns (e.g. mgrid) Quads account for essentially all the blocks. Other workloads exhibit up to 29% singletons (canneal), but Quads or Trios account for over 50% of blocks for all but two of our workloads (canneal and gcc).

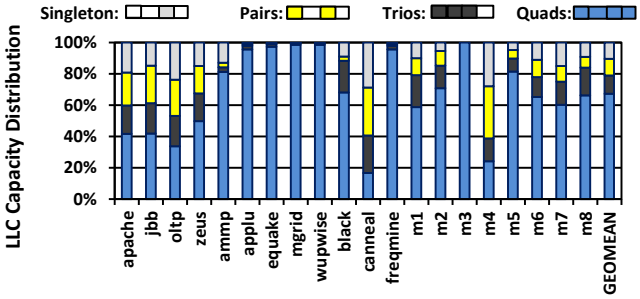

Figure 3. Distribution of LLC cache blocks

Super-blocks (also known as sectors [31]) have long exploited coarser-grain spatial locality to reduce tag overhead [25][30][31][39]. Super-blocks associate one address tag with multiple cache blocks, replicating only the per-block metadata such as coherence state. Figure 4(a) shows one set of a 4-way-associative super-block cache (SC), with 4-block super-blocks. Using 4-block super-blocks reduces tag area by 70% compared to a conventional cache. However, Figure 4(a) illustrates that Singletons, Pairs, and Trios—such as, super-blocks D, C, and A, respectively—result in internal fragmentation, which can lead to significantly higher miss rates [31].

Seznec showed that decoupling super-block tags from data blocks helps reduce internal fragmentation [31]. Decoupled super-block caches (DSC) increase the number of super-block tags per set and use per-block back pointers to identify the corresponding tag. Figure 4(b) illustrates how decoupling can reduce fragmentation by allowing two Singletons (i.e., blocks F1 and G3) to share the same super-block. DSC uses more tag space than SC, but less than a conventional cache since back pointers are small.

In this work, we use decoupled super-block tags to improve cache compression in two ways. First, super-blocks reduce tag overhead, permitting more tags per set for comparable overhead. Second, decoupling tags and data reduces internal fragmentation and, importantly, eliminates re-compaction when the size of a compressed block changes.

4. DECOUPLED COMPRESSED CACHE

In this section, we describe Decoupled Compressed Cache (DCC) and Co-Compacted DCC (Co-DCC) designs in detail. While these designs may be applicable to other levels of the cache hierarchy, we target LLC in this work.

4.1 DCC Architecture

Figure 5 shows DCC architecture. To improve compression effectiveness at LLC, DCC exploits super-blocks and manages the cache at three granularities: coarse-grain super-blocks, single cache blocks, and fine-grain sub-blocks. DCC tracks super-blocks, which are groups of aligned, contiguous cache blocks (Figure 5 (d)), while it compresses and stores single cache blocks as variable number of sub-blocks. Figure 5 (a) shows the key components of DCC architecture for a small 2-way-set associative cache with 4-block super-blocks, 64-byte blocks, and 16-byte sub-blocks. DCC consists of Tag Array, Sub-Blocked Back Pointer Array, and Sub-Blocked Data Array. DCC is indexed using the super-block address bits (Set Index in Figure 5 (e)). Note that like all super-block caches, this index uses higher order bits. In this way, all blocks of the same super-block are mapped to the same data set.

DCC tracks super-blocks to fit more compressed blocks in the cache while limiting tag area overhead. DCC explicitly tracks super-blocks through the tag array. The tag array is a largely conventional super-block tag array. Figure 5 (b) shows one tag entry that consists of one tag per super-block (Super-block tag) and coherence state (CState) and compression status (Comp) for each block of the super-block. Since blocks of a super-block share a tag address, the tag array can map more blocks compared to the same size conventional cache without incurring high area overhead. DCC holds as many super-block tags as the maximum number of uncompressed blocks that could be stored. For example, in Figure 5, for a 2-way-associative cache, it holds two super-block tags in each set of the tag array. In this way, each set in the tag array can map eight blocks (i.e., 2 super-blocks * 4 blocks/super-block), while maximum of two uncompressed blocks can fit in each set. In the worst case scenario, when there is no spatial locality (i.e., all singletons) or cached data is uncompressible, DCC can still utilize all the cache data space, for example, by tracking two singletons per set in Figure 5 (a).

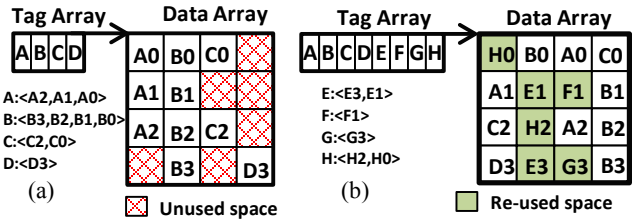

Figure 4. (a) Sectored Cache (b) Decoupled Sectored Cache

DCC compresses and compacts cache blocks into variable number of data sub-blocks. It dynamically allocates these sub-blocks in the sub-blocked data array. The data array is a mostly conventional cache data array, organized in sub-blocks. In Figure 5 (a), it provides eight 16-byte sub-blocks per set, for a total of 128 bytes. This is only one quarter of the data space mapped by each set in the tag array (i.e., 2 super-blocks * 4 blocks/super-block * 64 bytes/block = 512). Thus using this configuration the tag array has the potential to map four times as many blocks as can fit in the same size uncompressed data array.

DCC decouples sub-blocks from the address tag to eliminate the expensive re-compaction when a block's size changes. DCC allocates sub-blocks of a block in the data array not necessarily in contiguous space (unlike VSC [2]) but in order. For example, in Figure 5 (a), block A0 is compressed into two sub-blocks (A0.1 and A0.0) that are stored in the sub-block #5 and the sub-block #1 in the data array. DCC uses the sub-blocked back pointer array as one level of indirection to identify sub-blocks of a block. Each back pointer entry (BPE) in the sub-blocked back pointer array corresponds to one data sub-block in the data array and identifies the owner block. A BPE stores its corresponding block's tag ID and block ID (Figure 5 (c)). Tag ID (e.g., 1 bit for a 2-way-associative cache) refers to the super-block tag entry matching this block in the same set of the tag array (e.g., 1 in Figure 5 (a)). (Co-)DCC derives Block ID (e.g., 2 bits for a 4-block super-block) from the address of the accessing block (Blk# in Figure 5 (e)). The back pointer array slightly increases LLC area (discussed in Section 6.2); however, it enables low overhead variable size compression. In the next sub-section, we explain how blocks are allocated and looked up in DCC in detail.

4.2 DCC Lookup Process

Figure 6 shows DCC lookup procedure for different scenarios. On a cache lookup, both the tag array and the back pointer array are accessed in parallel. In the common case of a cache hit, both the block and its corresponding super-block are found available (i.e., tag matched and block is valid). In the event of a cache hit, the result of the tag array and the back pointer array lookup determines which sub-blocks of the data array belong to the accessing block. On a read, those sub-blocks are read out next, and the corresponding tag entry and BPEs are updated. In Figure 5, for example, on a read access to block A0, Tag A, Index A, and block ID (e.g., #0) are derived from the address (Figure 5 (e)). The corresponding set of the tag array and the back pointer array (indexed by Index A) are read out. The tag match and sub-block selection logic then identify whether the block is available and where its sub-blocks locate in the data array. For instance, the tag

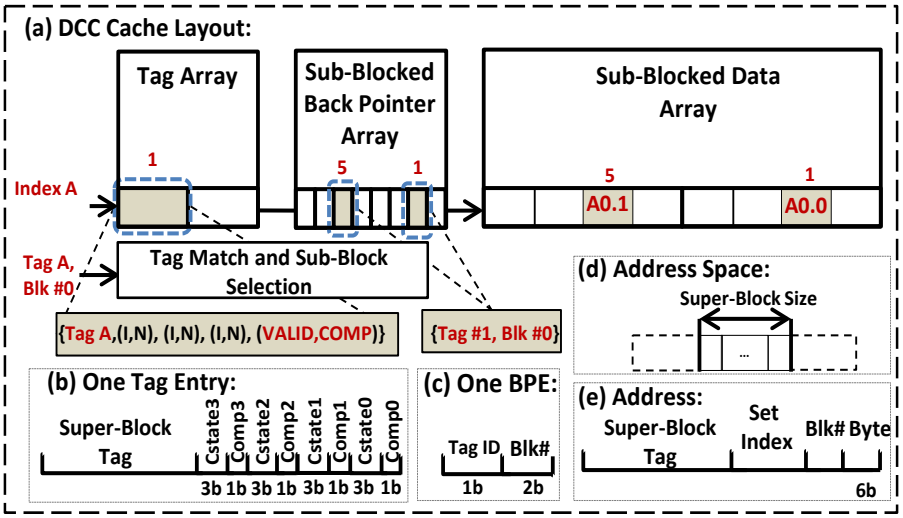

Figure 5. DCC cache design

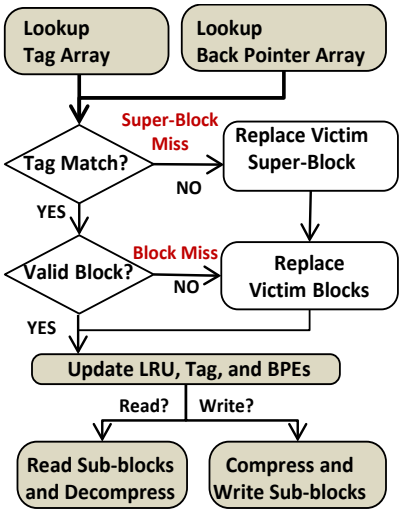

Figure 6. DCC cache lookup

entry #1 in the tag array matches super-block A, and its block #0 is available (CState #0 is valid). The sub-block selection logic finds the matched BPEs (BPEs #5 and #1 for A0) using the matched tag ID (e.g., 1 for A) and the block ID (e.g., 0 for A0). Since there is a one-to-one correspondence between BPEs and data sub-blocks, the corresponding sub-blocks are then read out of the data array (e.g., the sub-blocks #5 and #1 for A0) and decompressed.

On the other hand, in case of a cache miss, DCC allocates the compressed block in the data array. A cache miss occurs when the block is not available in the cache even if its super-block is available. If its super-block is available (Block Miss in Figure 6), the accessing block will be allocated in the data array, and its corresponding tag entry and BPEs will be updated. This might need replacing one or more cache blocks to make enough space for this block. If its super-block is not available (Super-Block Miss in Figure 6), we might need to replace another super-block (e.g., the least recently used one). In this case, the blocks belonging to the victim super-block are evicted from LLC as well. We handle the eviction process in the background by storing the victim super-blocks in a small buffer until all of their blocks are evicted from the cache. In this way, their tag entries can be released to allocate the new super-blocks.

Unlike conventional caches, on a write (or update) to a compressed cache, the new compressed size could be different. To fit a larger block, (Co-)DCC might need more sub-blocks, which may force a block (or a super-block) eviction. On the other hand, if the new compressed size is smaller, (Co-)DCC would deallocate the unused sub-blocks, and update the corresponding tag entry and BPEs.

When (Co-)DCC allocates a block or updates an existing block, it allocates from a list of free sub-blocks in the corresponding set. In the presented design, free sub-blocks are those pointing to invalid blocks. Since both the tag array and the back pointer array are accessed in parallel on a cache lookup, the cache controller gets the list of free data sub-blocks by finding those whose corresponding BPEs are pointing to invalid blocks. Thus, the cache controller always makes sure free sub-blocks are pointing to invalid blocks. An alternative design is to use an extra bit per BPE representing its validity, which might slightly increase area.

4.3 Co-DCC: Dynamically Co-Compacting Super-Blocks

To further improve cache compression, we present Co-DCC that optimizes DCC to reduce internal fragmentation. DCC, similar to previous variable size compressed caches [2][16], compresses and compacts small cache blocks into one or more sub-blocks, leading to internal fragmentation in sub-blocks (e.g., a 16-byte sub-block is allocated for an 8-byte compressed block). Compressing cache blocks and compacting them to byte granularity (BytePack) eliminates internal fragmentation but at the cost of higher hardware overheads (discussed in Section 6.2). Using larger block sizes can also help reducing internal fragmentation by packing more data in the same space. However, increasing cache block size can lead to cache pollution and higher energy overheads [38]. Since for many applications, neighboring blocks co-reside at LLC, by managing cache at super-block granularity, super-blocks (e.g., a quad) can be treated as one large block. Co-DCC exploits this opportunity and dynamically compacts compressed blocks of one super block into the same set of sub-blocks. By co-compacting super-blocks, Co-DCC can get some of the benefits of BytePack with much lower overheads, as shown in Section 6. In the next sub-section, we show how Co-DCC works, and how it can be integrated to DCC design with small changes.

4.3.1 Co-DCC Design

Co-DCC operates mostly similar to DCC, except for co-compacting super-blocks. When allocating a block to an existing super-block, Co-DCC compacts and stores the compressed block with the existing blocks of the same super-block. Figure 7 shows an example of how Co-DCC works for the same configuration used in Figure 5. In this example, Co-DCC stores and co-compacts A0, A1 and A2, which belong to the super-block A, in chronological order in a 2-way set associative data set. When allocating block A1, since it fits in less than a sub-block, it shares a sub-block with A0 (in the sub-block #5). When A2 is allocated, A2 can also share some space (A2.0) with A0 and A1 in the sub-block #5. Its remaining sub-blocks (A2.1 and A2.2) need to be allocated in free sub-blocks of the set. In this example, there is not enough in-order space available for A2 if we want to share sub-block #5 among these blocks. Therefore, unlike DCC that stores the sub-blocks of a block in order, Co-DCC stores them not necessarily in order but in a round-robin fashion (e.g., block

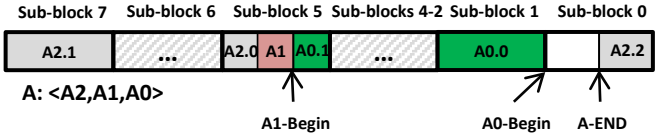

Figure 7. Co-DCC co-compaction example

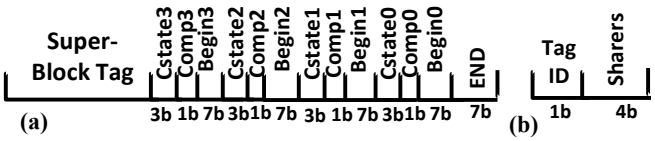

Figure 8. (a) One Co-DCC tag entry (b) One Co-DCC BPE

A2 in Figure 7). In this way, it will not need to move blocks if there is not enough in-order space available when co-compacting them. However, this design can slightly increase access latency (described in Section 5).

Co-DCC can be integrated with DCC design with some small changes in the tag array and the back pointer array. Figure 8 shows one Co-DCC tag entry and one BPE for the same configuration used in Figure 5. Since the first byte of a compressed block can be stored anywhere in a data sub-block (e.g., A2.0 in Figure 7), Co-DCC tracks each block's starting byte separately in its corresponding tag entry (e.g., 7-bit Begin0 in Figure 8(a)). Co-DCC also tracks the last occupied byte of each super-block in its corresponding tag entry (7-bit End in Figure 8(a)). When allocating a new block to an existing super-block, Co-DCC stores it next to this last byte if there is free space in that sub-block, and updates this pointer.

Unlike DCC, where each data sub-block belongs to only one block, Co-DCC can share one sub-block among multiple blocks of the same super-block. For example, A0.1, A1, and A2.0 share the sub-block #5 in Figure 7. Therefore, each Co-DCC BPE tracks its sharers by storing a small bit-vector (e.g., 4-bit Sharers in Figure 8(b)). Each bit of the sharers bit-vector shows if its corresponding block shares that sub-block. This information will slightly increase LLC area (Section 6.2), but allows Co-DCC to fit more blocks in the cache by reducing internal fragmentation.

5. HARDWARE COMPLEXITIES

In this section, we first describe the compression algorithms we have used throughout this paper. We then describe how (Co-)DCC can be integrated with a modern cache design.

5.1 Compression Algorithms

Multiple compression algorithms have been proposed for cache compression, which have reasonably low overheads [2][9][14][22][29][35][37]. (Co-)DCC is mainly independent of compression algorithm in use. Therefore, in this paper, we study three representative algorithms:

ZERO (Z): ZERO detects blocks containing all zeros (i.e., null data) and stores only a tag for those blocks [14]. This technique has very simple (de-)compression hardware [14]. Since zeros are common in cached data [14], all the algorithms in this paper detect zero blocks.

FPC+Z: FPC is a significance-based compression algorithm [2] that exploits the fact that many values are small and do not require the full space allocated for them (e.g., small integers). FPC also compacts zeros and repeated bytes. FPC+Z detects zero blocks as well. FPC decompresses a 64-byte line in five cycles, has 0.183 mm^2 area, and 0.273 W power consumption in 45nm [11].

Table 2. C-PACK+Z Overheads

Parameters	Compressor	Decompressor
Pipeline Depth	6	2
Latency (cycles)	16	9
Area (mm^2)	0.016	0.016
Power Consumption (mW)	25.84	19.01

C-PACK+Z: C-PACK is a pattern-based partial dictionary match compression algorithm [9]. It compresses data by both statically (for fixed data patterns) and dynamically (using a 16-entry dictionary) detecting frequently appearing data. The original paper [9] presented a compressor that is designed as a 3-stage pipeline, and an un-pipelined decompressor. They showed that C-PACK runs at 1.2GHz in 65nm [9]. We extend C-PACK with zero block detection (C-PACK+Z). To make C-PACK+Z run at 3.2GHz in 32nm, we doubled the pipeline depth (decompression from un-pipelined to a 2-stage pipeline). C-PACK(+Z) takes 16 bytes as input and produces 8 bytes as output per cycle. Table 2 summarizes C-PACK+Z overheads in 32nm, which are scaled from 65nm [9] using ITRS [20] and include the overhead of deeper pipelines. The critical loop in both compression and decompression involves reading and updating the dictionary. Pipelining this operation requires classical register bypass logic to forward dependent updates. We conservatively assumed 1.5X larger area and power due to extra pipe registers and bypass circuits. We use C-PACK+Z in this paper, since it has higher compression ratio, low overheads and a more practical design [9].

5.2 Cache Design Complexities

(Co-)DCC can be integrated into LLC of a recent commercial design with relatively little additional complexity and more importantly no need for an alignment network. The AMD Bulldozer implements an 8MB LLC that is broken into four 2MB sub-caches, each sub-cache consists of four banks that can independently service cache accesses [34]. Figure 9 illustrates the data array of one bank in LLC and shows how it is divided into 4 sequential regions (SR). Each sequential region runs one phase (i.e., half a cycle) behind the previous region and contains a quarter of a cache block (i.e., 16 bytes). Figure 9 shows how block A0's four 16-byte sub-blocks (e.g., A0.0–A0.3) are distributed to the same row in each sequential region. Each subsequent sequential region receives the address a half cycle later and takes a half cycle longer to return the data. Thus, a 64-byte block is returned in a burst of four cycles on the same data bus. For example, A0.1 is returned one cycle after A0.0 in Figure 10(a).

DCC requires only a small change to the data array to allow non-contiguous sub-blocks. In Figure 9, block B1 is compressed into 2 sub-blocks (B1.0 and B1.1), stored in sequential regions #1 and #2, but not in the same row. To select the correct sub-block, DCC must send additional address lines (i.e., 4 bits for a 16-way-associative cache) to each sequential region (illustrated by the dotted lines in Figure 9). DCC must also enforce the constraint that a compressed block's sub-blocks are allocated to different sequential regions to prevent sequential region conflicts.

Figure 10(b) illustrates DCC timing when reading block B1. As described in Section 4, the back pointer array is accessed in

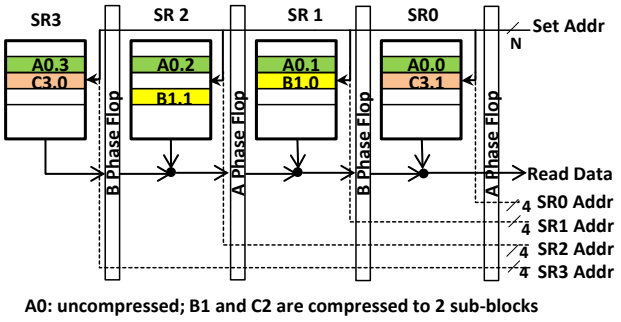

Figure 9. (Co-)DCC Data Array Organization

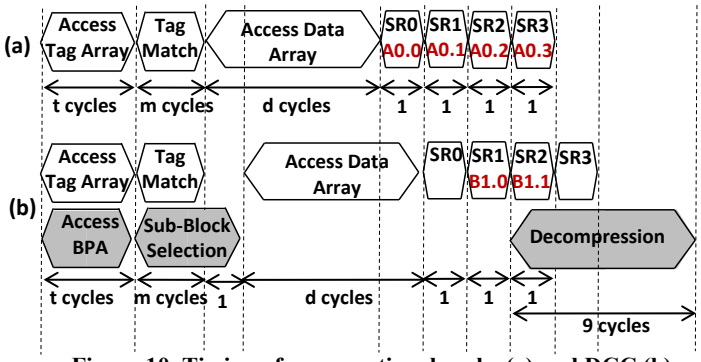

Figure 10. Timing of a conventional cache (a) and DCC (b)

parallel with the tag array. The sub-block selection logic finds the BPEs corresponding to this block using its block ID (derived from its address) and the matched tag ID, which is found by the tag match logic. The sub-block selection logic can only be partially overlapped with the tag match logic since it needs the matched tag ID. To calculate the latency overhead of the sub-block selection, we implemented the tag match and the sub-selection logic in Verilog, synthesized in 45nm and scaled to 32nm [20]. The sub-block selection logic adds less than half a cycle to the critical path, which we conservatively assume increases the access latency by one cycle. Figure 10(b) shows how the matching sub-blocks are returned and fed directly into the decompression logic, which accepts 16-byte per cycle and has a small FIFO buffer to rate match. Decompression starts as soon as the first sub-block arrives (e.g., B1.0), which depends upon which sequential region it resides in. Since sub-block B1.0 resides in sequential region 1, there is one extra cycle (worst case is 3 cycles). Note that because the decompression latency is deterministic (9 cycles), DCC can determine at the end of sub-block selection when the data will be ready and whether the decompression hardware can be bypassed. Thus, even though completion times vary, DCC has ample time to arbitrate for the response network.

Figure 9 also shows how block C3 is allocated by Co-DCC. Co-DCC also stores sub-blocks of a block in different regions, but allocates them in round-robin fashion and not necessarily in order. Therefore, Co-DCC cannot necessarily start decompression as soon as it reads the first sub-block (e.g., C3.1 will be read out first before C3.0). To handle these cases, Co-DCC must buffer the sub-blocks and pass them to the decompression logic in order. The decompression logic must also pre-align the first sub-block, since the compressed block doesn't necessarily start in the first byte. The reordering and pre-alignment add up to 3 additional cycles compared to DCC.

6. EVALUATION

6.1 Experimental Methodology

We evaluate (Co-)DCC using a full-system simulator based on GEMS [26]. We model a multicore system with three levels of cache hierarchy (Table 3) [10]. We use an 8MB LLC that is broken into 8 banks, each divided into 4 sequential regions. Note that although we use a different cache configuration than AMD Bulldozer LLC, we model the timing and allocation constraints of sequential regions at LLC in detail, as discussed in Section 5. We use CACTI [19] to model power at 32nm. We also use a detailed DRAM power model developed based on the Micron Corporation power model [27] with energy per operation listed in Table 3. In this section, we report total system energy that include energy consumption of processors (cores + caches), on-chip network, and off-chip memory.

Table 4 shows the configurations we use. For (Co-)DCC, we use 4-block super-blocks, 64-byte blocks, and 16-byte sub-blocks. With these parameters, DCC has similar area overhead as FixedC and VSC-2X (Section 6.2). Alternative super-block and sub-block sizes can be used. We use 4-block super-blocks, since not all workloads would benefit from larger super-blocks due to their limited spatial locality. Using smaller sub-blocks also potentially improves compression effectiveness by reducing internal fragmentation, but at the cost of higher hardware complexities and overheads (discussed in Section 6.2).

Our evaluations use representative multi-threaded and multi-programmed workloads from Commercial workloads [3], SPEC-OMP [6], PARSEC [8], and mixes of SPEC CPU2006 benchmarks, summarized in Table 5. We evaluate eight multi-programmed workloads with different mixes of compute-bound and memory intensive benchmarks. Each workload consists of 8 threads evenly divided among the named Spec2006 benchmarks. For example, cactus-mcf-milc-bwaves runs two copies of each of the four benchmarks.

Figure 11 shows the sensitivity of our workloads to LLC capacity and LLC access latency. Compressed caches in general benefit cache capacity sensitive workloads by providing higher effective cache capacity. On the other hand, they might hurt cache latency sensitive workloads due to the decompression latency. We categorize our workloads as cache latency sensitive if they observe more than 1% runtime slowdown compared to Baseline when we use the same size cache with 9 extra LLC access latency, which represents the decompression latency. Many of our workloads (e.g., freqmine and oltp) are sensitive to cache latency and observe up to 6% (for oltp) slow down with the slower cache. We also categorize our workloads that observe more than 2% speedup with double LLC capacity (with the same access latency as Baseline) as cache capacity sensitive. Our workloads have a wide range of sensitivity to cache capacity (maximum 22% speedup for apache). Among our workloads, ammp, applu, blackscholes, and libquantum are cache insensitive. We run each workload for approximately 500M instructions with warmed up caches. We use a work-related metric, run each workload for a fixed number of transactions/iterations and report the average over multiple runs to address workload variability [4].

6.2 (Co-)DCC Area and Power Overheads

Compressed caches can increase cache area due to their extra metadata. Table 6 shows the quantitative area overheads of DCC, Co-DCC, FixedC and VSC-2X over the same size conventional

Table 3. Simulation parameters

Cores	OOO, 3.2 GHz, 4-wide issue, 128-entry Instruction Window.
L1I$/L1D$	Private, 32-KB, 8-way, 2 cycles, HP transistors.
L2 $	Private, 256-KB, 8-way, 10 cycles, HP transistors.
L3 $	Shared, 8-MB, 16-way, 8 banks, 30 cycles, LSTP transistors.
Main Memory	4GB, 16 Banks, 800 MHz bus frequency DDR3, 60.35 nJ per Read, 66.5 nJ per Write, and 4.25W static power.

Table 4. Configurations

Baseline	Conventional **16-way-associative 8MB LLC.**
2X Baseline	Conventional **32-way-associative 16MB LLC.**
FixedC	**2x tags** per set (i.e., **32** tags per set). Each cache block is compressed to half if compressible.
VSC-2X	**2x tags** per set (i.e., **32** tags per set). A block is compressed into 0-4 **16-byte sub-blocks**.
DCC	**Same number of tags** per set (i.e., **16** tags per set). Each tag tracks up to 4 blocks (**4-block Super-Blocks**). Blocks are compressed individually to 0-4 **16-byte sub-blocks**.
Co-DCC	Similar to DCC, except it dynamically co-compacts blocks of the same super-blocks.

Table 5. Workloads

Suite	Workloads
Commercial	apache, jbb, oltp, zeus
SPEC-OMP	ammp, applu, equake, mgrid, wupwise
PARSEC	blackscholes, canneal, freqmine
Spec2006 (denoted as m1-m8)	bzip2, libquantum-bzip2, libquantum, gcc, astar-bwaves, cactus-mcf-milc-bwaves, gcc-omnetpp-mcf-bwaves-lbm-milc-cactus-bzip, omnetpp-lbm

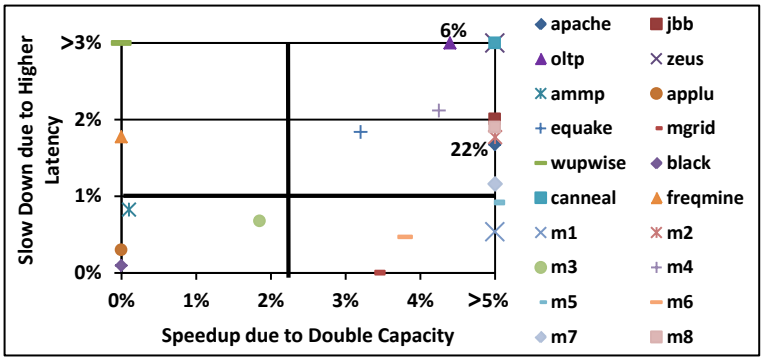

Figure 11. Cache sensitivity of our workloads

cache (16-way-associative 8MB LLC) with the parameters in Table 3 and Table 4. DCC uses the same number of tags as Baseline, but almost doubles the per-block metadata largely due to the back pointers. However, since the data array is much larger than the tag array, Cacti calculates the overall LLC area overhead as about ~6% [9]. DCC's area overhead is similar to FixedC and VSC-2X, which track twice as many tags per set (e.g., 32 tags per 16 blocks). Co-DCC increases metadata stored per block, as discussed in Section 4.3, resulting to 16% area overhead compared to Baseline. Co-DCC still has less area overhead than naively quadrupling the number of tags (VSC-4X). It also incurs much lower overhead compared to a DCC configuration with no packing constraint (DCC-BytePack). BytePack can increase compression effectiveness by reducing internal fragmentation (Figure 1(b)). However, using 1-byte sub-blocks requires 16 times more BPEs per set than (Co-)DCC with 16-byte sub-blocks. BytePack would also require a complex alignment network to compact the bytes into 16-byte sub-blocks before passing them to the decompression hardware. Table 6 also includes the area overhead of (de-)compression units. Since C-PACK+Z's decompressors produce 8 bytes per cycle, we match the cache bandwidth by considering two decompressors per cache bank. Since compression is not on the critical path, we consider one compressor per bank. For LLC configuration in Table 3, we need 8 compressors and 16 decompressors resulting to an extra 1.8% area overhead.

Compressed caches can also increase LLC per-access dynamic power and LLC static power due to their extra metadata. DCC, similar to FixedC and VSC-2X, increases LLC per-access dynamic power by 2% and LLC static power by 6%. Co-DCC also incurs 6% overhead on LLC per-access dynamic power and 16% LLC static power overhead [19]. We model these overheads as well as the power overheads of (de-)compression in detail.

6.3 Improved Cache Efficiency

Result 1: By exploiting spatial locality, DCC achieves on average 2.2 times (up to 4 times) higher LLC effective capacity compared to Baseline, resulting to 18% lower LLC miss rate on average and up to 38% lower LLC miss rate.

Result 2: Co-DCC further improves the effective cache capacity by co-compacting the blocks in a super-block. It achieves on average 2.6 times and up to 4 times higher effective capacity and on average 24% and up to 42% lower LLC miss rate.

Result 3: (Co)-DCC provides significantly higher effective cache capacity and lower miss rate than FixedC and VSC-2X. (Co-)DCC also performs on average better than 2X Baseline with much lower area overhead.

Compressed caches improve the cache effective capacity by

Table 6. LLC area overheads of different compressed caches over the conventional cache

Components	DCC	Co-DCC	FixedC/VSC-2X	VSC-3X	VSC-4X	DCC-BytePack
Tag Array	2.1%	11.3%	6.3%	12.7%	18.8%	2.1%
Back Pointer Array	4.4%	5.4%	0%	0%	0%	70.6%
Compressors	0.6%	0.6%	0.6%	0.6%	0.6%	0.6%
Decompressors	1.2%	1.2%	1.2%	1.2%	1.2%	1.2%
Total Area Overhead	**8.3%**	**18.5%**	**8.1%**	**14.5%**	**20.6%**	**74.5%**

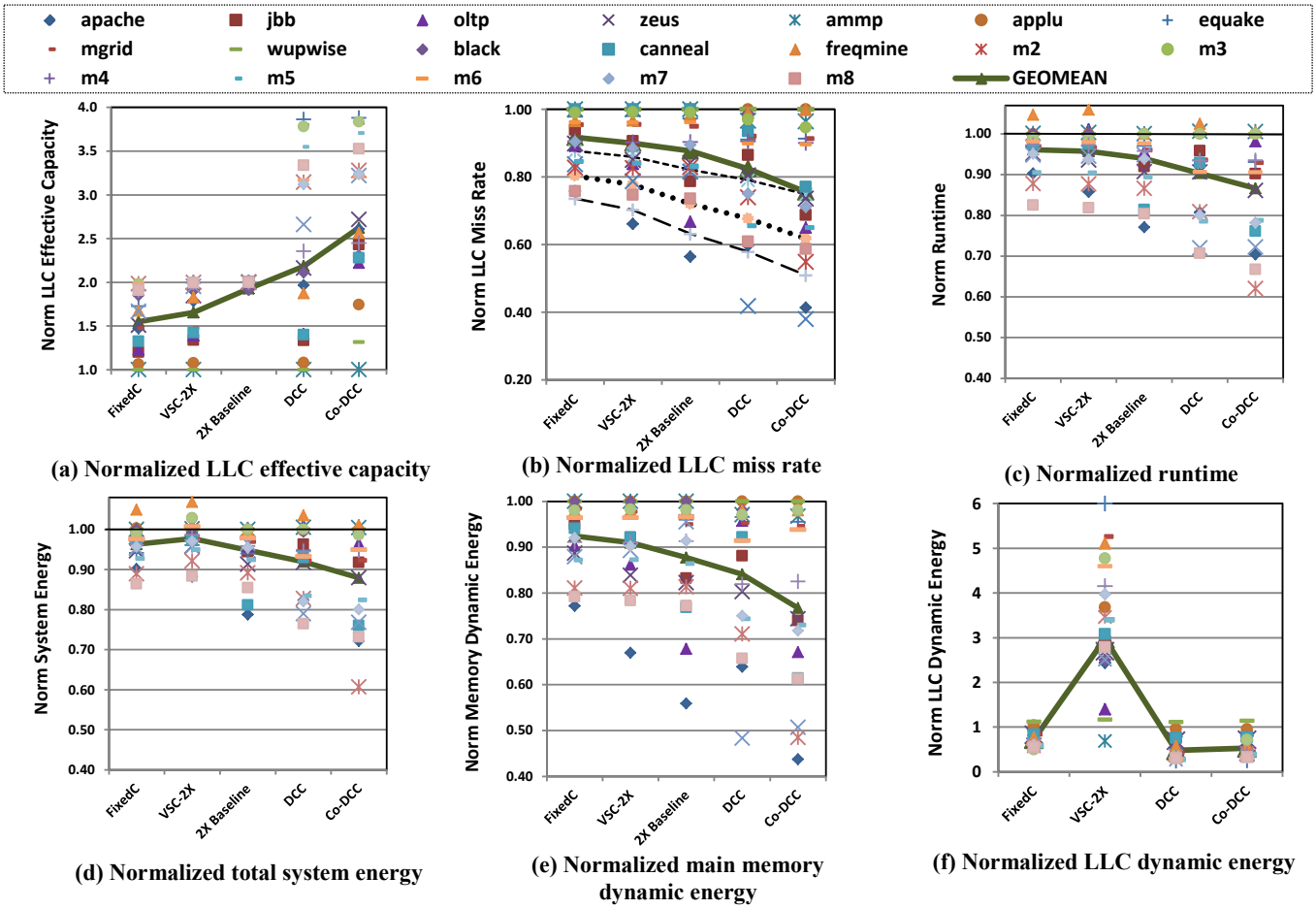

Figure 12. LLC effective capacity, LLC miss rate, system performance and energy normalized to Baseline

fitting more blocks in the same space. They can achieve the benefits of larger cache sizes with lower area and power overheads. Figure 12(a) and Figure 12(b) plot LLC effective capacity and LLC miss rate of different techniques normalized to Baseline. We calculate the effective cache capacity by counting valid LLC cache blocks periodically. We measure LLC miss rate as the total number of misses per kilo executed instructions (MPKI). Figure 12(b) also plots the average LLC miss rate reduction predicted with power law for miss rate [18] in dashed lines. This model [18] predicts the cache miss rate to be inversely proportional to the increased capacity with an scaling factor typically set to 0.5 (i.e., "square root" power law), 0.3, or 0.7 (the higher the scaling factor, the lower the predicted miss rate). The average improvement we found for our workloads is less than what these models predict, since our workloads represent a wide range of cache sensitivities and we are not picking only highly cache sensitive ones.

DCC can significantly improve LLC effective capacity and LLC miss rate for many applications by fitting more compressed blocks. On average, DCC provides 2.2x (i.e., 17.6MB) higher effective capacity and 18% lower LLC miss rate compared to Baseline. DCC benefits differ per workload, depending on its sensitivity to cache capacity, compression ratio, and spatial locality. It achieves highest benefits for cache sensitive workloads with good compressibility and spatial locality (e.g., apache and omnetpp-lbm/m8). Workloads with low spatial locality (e.g., canneal) or low compression ratio (e.g., wupwise) observe lower improvements. Cache insensitive workloads (e.g., blackscholes) also do not benefit from compression.

Co-DCC further improves compression effectiveness by reducing internal fragmentation within data sets. Co-DCC achieves, on average, 2.6x higher effective capacity (i.e., 20.8MB) and 24% lower miss rate than Baseline. By fitting more compressed blocks in the cache, compared to DCC, Co-DCC can further reduce LLC miss rate for almost half of our workloads, including commercial workloads (e.g., 18% lower miss rate for jbb), canneal, and some of our Spec2006 mixes (e.g., 19% lower miss rate for libquantum-bzip2/m2). By co-compacting super-blocks, Co-DCC gets some of the benefits of the idealized BytePack (Figure 1(b)) with much lower hardware overheads, as discussed in Section 6.2.

Compared to FixedC and VSC-2X, (Co-)DCC provides higher LLC effective capacity and lower miss rate. Both FixedC and VSC-2X can at most double effective cache capacity compared to Baseline (i.e., 16MB). FixedC achieves on average 1.5x higher effective capacity and 8% lower miss rate than Baseline. VSC-2X provides slightly higher benefits (1.7x effective capacity, and 10% lower miss rate). Increasing VSC tag space can improve its benefits. For example, VSC-4X has similar miss rate reduction as DCC, but with 2.6x higher area overhead.

Compared to 2X Baseline, (Co-)DCC effectively more than doubles cache capacity with lower overheads. DCC achieves higher LLC effective capacity than 2X-Baseline for majority of our workloads. It provides lower LLC miss rate reduction than

2X-Baseline (within 27%) for apache, jbb, oltp and gcc, which have lower compression ratio and spatial locality compared to other workloads. For these workloads, Co-DCC provides similar or better LLC miss rate reduction than 2X-Baseline by reducing internal fragmentation.

6.4 Overall Performance and Energy

Result 4: DCC and Co-DCC improve LLC efficiency and boost system performance by 10% (up to 29%) and 14% (up to 38%) on average, respectively.

Result 5: DCC and Co-DCC save on average 8% (up to 24%) and 12% (up to 39%) of system energy, respectively, due to shorter runtime and fewer accesses to the main memory.

Result 6: DCC and Co-DCC achieve respectively 2.5x and 3.5x higher performance improvements, and 2.2x and 3.3x higher system energy improvements compared to FixedC and VSC-2X.

Result 7: (Co-)DCC also improves LLC dynamic energy by about 50% on average due to accessing fewer bytes. On the other hand, VSC-2X hurts LLC dynamic energy for majority of our workloads due to its need for energy-expensive re-compactions.

By improving LLC utilization and reducing accesses to the main memory (i.e., the lower LLC miss rate), (Co-)DCC significantly improves system performance over Baseline. Figure 12(c) plots runtime of different techniques normalized to Baseline. DCC and Co-DCC improve performance by 10% (up to 29% for omnetpp-lbm/m8) and 14% (up to 38% for libquantum-bzip2/m3) on average, respectively. For cache sensitive applications with medium-to-high compressibility and medium-to-high spatial locality (e.g., apache and zeus), (Co-)DCC achieves significant performance improvements by fitting more blocks in the cache. They provide lower improvements for applications with low spatial locality and low compression ratio (e.g., canneal and gcc). On the other hand, compressed caches, including (Co-)DCC, can hurt performance of workloads sensitive to LLC access latency (e.g., freqmine) due to the decompression latency. (Co-)DCC hurts performance by less than 3% (for freqmine). Cache insensitive workloads also do not benefit from compressed caches. An adaptive technique can be employed to further reduce these overheads [2], which is orthogonal to our proposals.

(Co-)DCC significantly outperforms FixedC, VSC-2X and 2X-Baseline by effectively more than doubling the cache capacity. FixedC and VSC-2X limit compression effectiveness in improving system performance, achieving on average 4% and 5% performance improvements, respectively. (Co-)DCC outperforms 2X-Basline for majority of our workloads. 2X-Baseline performs better than DCC for six of our workloads (within 11% for canneal). These workloads have lower spatial locality (e.g. canneal), lower compression ratio (e.g., jbb), or higher sensitivity to cache latency (e.g., freqmine) than the rest of our workloads. Co-DCC improves performance for more workloads, providing slightly lower performance than 2X-Baseline only for three workloads (within 3% for freqmine).

(Co-)DCC improves system energy both due to shorter runtime and fewer accesses to the main memory. Figure 12(d) shows the total system energy of different techniques. DCC and Co-DCC reduce the total system energy by 8% (up to 24% for omnetpp-lbm/m8) and 12% (up to 39% for libquantum-bzip/m2) on average, respectively. Figure 12(e) plots the main memory dynamic energy for these techniques. (Co-)DCC significantly reduces the main memory dynamic energy by reducing the number of cache misses. This contributes to (Co-)DCC higher system energy improvements as well. Compared to FixedC and VSC-2X, (Co-)DCC achieves higher energy savings. Although VSC-2X provides slightly higher performance and lower main memory dynamic energy consumption than FixedC, its system energy saving is less due to its high overheads on LLC dynamic energy. Figure 12(f) shows the dynamic energy of different compressed caches normalized to Baseline. FixedC, DCC and Co-DCC improve LLC dynamic energy by 27%, 52% and 46% on average over Baseline, respectively. On the other hand, VSC-2X significantly increases LLC dynamic energy (about 3x) by increasing the number of cache accesses (Figure 1(c)).

We also measured the sensitivity of (Co-)DCC to different design parameters including decompression latency and LLC access latency. Our simulations (not shown here) show that reducing decompression latency (for the same C-PACK+Z algorithm) from 9 cycles to 3 cycles only slightly increases (Co-)DCC performance. It achieves on average 1% and up to 3% higher performance than the results shown in Figure 12(c). We also studied the sensitivity of (Co-)DCC to LLC cache access latency. Our simulation results (not shown here) show that even reducing LLC access latency to 20 cycles (33% faster LLC) does not significantly impact (Co-)DCC results.

7. RELATED WORK

Exploiting Spatial Locality in Caches. This work builds upon previous dual-grain caches namely the Region Tracker [39], the sectored cache [25], sector pool cache [30], and the decoupled-sectored cache [31], mostly discussed in Section 3. RegionTracker also manages cache at dual-granularities of memory regions (e.g., 1KB) and cache blocks [39]. Unlike our proposal, Region Tracker [39] aims facilitating collection of coarse-grain information.

Cache Compression. In Section 2, we described some of related work on compressed caches. Hallnor et al. extend their earlier indirect index cache [17] to support compression (IIC-C) [16]. IIC-C uses a software-managed hash table to provide full associativity and forward pointers to associate tags with variable number of sub-blocks anywhere in the data array [16], eliminating the need for repacking. However, for an 8MB LLC with 64-byte blocks, 16-byte sub-blocks, and doubled number of tags, their scheme incurs about 24% area overhead (26% considering (de-)compressors), while it at most doubles effective capacity. Further increasing the number of tags will make its area overhead even worse. J. S. Lee, et al. [24] compress block pairs and store them in a single line if both lines compress by 50% or more. In this way, they free a cache block in an adjacent set; however, they need to check two sets for a potential hit on every access, which increases power overheads. In addition, their technique limits compressibility by failing to take advantage of lines that compress by less than 50%. Naffziger and Kover's patent [28] describes a compressed cache design that uses forward pointers to associate tags with data sub-blocks. The overhead is less than IIC-C [16], since pointers only refer to sub-blocks within a set (not the entire cache) and are thus much smaller. In one embodiment they also serialize the sub-block accesses. However, as far as we can tell, this scheme has never been evaluated in the public literature.

Some techniques aim reducing cache dynamic power consumption by compression. For example, Dynamic zero compression (DZC) [33] only stores non-zero bytes in the cache. It reduces L1 cache dynamic power, but does not increase cache

effective size. Similarly, FVC [35] reduces cache dynamic power by accessing half of the block if compressed. Significance-compression [22] helps both cache power and system energy by accessing half of the cache block if compressed, and packing more blocks in the cache. In a recent work, Residue cache [23] aims reducing L2 cache area and power in single processor embedded systems. They compress cache blocks and store them in half size in L2 cache. For uncompressed cache blocks, they store another half in a small cache, called residue cache.

S. Baek, et al. [7] proposes a size-aware compressed cache management to improve the performance of compressed caches. Their proposal is orthogonal to ours, and can be integrated with (Co-)DCC. G. Pekhimenko, et al. [29] also proposes a new compression algorithm with lower complexities than previously proposed ones, but requires an adder per word. (Co-)DCC is mainly independent of compression algorithms, and can use their proposed algorithm as well.

Compression is also used at main memory [1][15]. The decoupled zero-compressed memory [13] manages the main memory as a decoupled sectored set-associative cache. It detects null blocks to improve performance, which is more limited than compression-based schemes.

8. CONCLUSIONS

In this paper, we propose Decoupled Compressed Cache that exploits spatial locality to improve both the performance and energy-efficiency of cache compression. DCC manages the cache at three granularities, tracking super-blocks while dynamically compressing and allocating single blocks as variable number of sub-blocks. It addresses the issues with conventional compressed caches, and achieves significantly higher LLC effective cache capacity while incurring low area overheads. It also decouples sub-blocks from the address tag to eliminate energy-expensive re-compaction when a block's size changes. A further optimized design (Co-DCC) reduces internal fragmentation in the cache by co-compacting super-blocks. We show that on average, DCC and Co-DCC reduce system energy by 8% and 12%, respectively, and improve performance by 10% and 14%, respectively, compared to the same size conventional cache. (Co-)DCC nearly doubles compression benefits compared to previous proposals with comparable overheads.

9. ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation (CNS-0916725, CCF-1017650, CNS-1117280, and CCF-1218323) and a University of Wisconsin Vilas award. The views expressed herein are not necessarily those of the NSF. Professor Wood has a significant financial interest in AMD. The authors would like to acknowledge Hamid Reza Ghasemi, members of the Multifacet research group, and our anonymous reviewers for their comments on the paper.

10. REFERENCES

[1] Abali, B. et al. 2001. Performance of Hardware Compressed Main Memory. In Proceedings of the 7th IEEE Symposium on High-Performance Computer Architecture.

[2] Alameldeen, A. and Wood, D. 2004. Adaptive Cache Compression for High-Performance Processors. In Proceedings of the 31st Annual International Symposium on Computer Architecture.

[3] Alameldeen, A. et al. 2003. Simulating a $2M Commercial Server on a $2K PC. IEEE Computer.

[4] Alameldeen, A. and Wood, D. 2003. Variability in Architectural Simulations of Multi-threaded Workloads. In Proceedings of the Ninth IEEE Symposium on High-Performance Computer Architecture.

[5] Arelakis, A., Stenström, P. 2012. A Case for a Value-Aware Cache. IEEE Computer Architecture Letters.

[6] Aslot, V. et al. 2001. SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. In Workshop on OpenMP Applications and Tools.

[7] Baek, S. et al. 2013. ECM:Effective Capacity Maximizer for High-Performance Compressed Caching. In Proceedings of IEEE Symposium on High-Performance Computer Architecture.

[8] Bienia, C. et al. 2009. PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors. In Workshop on Modeling, Benchmarking and Simulation.

[9] Chen, X. et al. 2010. C-pack: a high-performance microprocessor cache compression algorithm, IEEE Transactions on VLSI Systems.

[10] Intel Core i7 Processors http://www.intel.com/products/processor/corei7/

[11] Das, R. et al. 2008. Performance and Power Optimization through Data Compression in Network-on-Chip Architectures, International Symposium on High Performance Computer Architecture.

[12] Dennard R. et al. 1974. Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions. IEEE Journal of Solid-State Circuits.

[13] Dusser, J. et al. 2011. Decoupled Zero-Compressed Memory. In Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers.

[14] Dusser, J. et al. 2009. Zero content augmented cache. In Proceedings of the 23rd international conference on Supercomputing.

[15] Ekman, M. and Stenstrom, P. 2005. A robust main-memory compression scheme. SIGARCH Computer Architecture News.

[16] Hallnor, E. et al. 2005. A Unified Compressed Memory Hierarchy. In Proceedings of the 11th International Symposium on High-Performance Computer Architecture.

[17] Hallnor, E. et al. 2000. A Fully Associative Software-Managed Cache Design. In Proceedings of the 27th Annual International Symposium on Computer Architecture.

[18] Hartstein, A. et al. 2008. On the Nature of Cache Miss Behavior: Is It √2? J. Instruction-Level Parallelism 10.

[19] CACTI: http://www.hpl.hp.com/research/cacti/

[20] ITRS. International technology roadmap for semiconductors, 2010 update, 2011. URL http://www.itrs.net

[21] Keckler, S. 2011. Life After Dennard and How I Learned to Love the Picojoule. In Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture.

[22] Kim, N. et al. 2002. Low-Energy Data Cache Using Sign Compression and Cache Line Bisection. Second Annual workshop on Memory Performance Issues.

[23] Kim, S. et al. 2011. Residue Cache: A Low-Energy Low-Area L2 Cache Architecture via Compression and Partial Hits. In Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture.

[24] Lee, J. et al. 2000. An on-chip cache compression technique to reduce decompression overhead and design complexity. Journal of Systems Architecture.

[25] Liptay, J. 1968. Structural Aspects of the System/360 Model85 Part II: The Cache. IBM Systems Journal.

[26] Martin, M. et al. 2005. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. Computer Architecture News.

[27] 2007. Calculating memory system power for DDR3. Technical Report TN-41-01. Micron Technology.

[28] Naffziger, S. et al. 2002. Apparatus for cache compression engine for data compression of on-chip caches to increase effective cache size. US patent 6,640,283.

[29] Pekhimenkoy, G. et al. 2012. Base-Delta-Immediate Compression: Practical Data Compression for On-Chip Caches. In Proceedings of the 21st international conference on Parallel architectures and compilation techniques.

[30] Rothman, J. et al. 1999. The Pool of Subsectors Cache Design. International Conference on Supercomputing.

[31] Seznec, A. 1994. Decoupled sectored caches: Conciliating low tag implementation cost and low miss ratio. International Symposium on Computer Architecture.

[32] Tremaine, R. et al. 2001. IBM Memory Expansion Technology (MXT). IBM Journal of Research and Development.

[33] Villa, L. et al. 2000. Dynamic zero compression for cache energy reduction. In Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture.

[34] Weiss, D. et al. 2011. An 8MB Level-3 Cache in 32nm SOI with Column-Select Aliasing. Solid-State Circuits Conference Digest of Technical Papers.

[35] Yang, J. et al. 2002. Energy Efficient Frequent Value Data Cache Design. In Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture.

[36] Yang, J. et al. 2002. Frequent Value Locality and its Applications. ACM Transactions on Embedded Computing Systems.

[37] Yang, J. et al. 2000. Frequent Value Compression in Data Caches. In Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture.

[38] Yoon, D. et al. 2011. Adaptive granularity memory systems: A tradeoff between storage efficiency and throughput. In Proceeding of the 38th Annual International Symposium on Computer Architecture.

[39] Zebchuk, J. et al. 2007. A Framework for Coarse-Grain Optimizations in the On-Chip Memory Hierarchy. In Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitectur