Decoupled Compressed Cache: Exploiting Spatial Locality for Energy Optimization

The authors propose decoupled compressed cache (DCC) to improve performance and energy efficiency of cache compression. DCC uses decoupled superblocks and noncontiguous sub-block allocation to decrease tag overhead and internal fragmentation and to eliminate the need for energy-expensive recompaction caused by changes in compressed block size. The authors also demonstrate a practical design based on a recent commercial last-level cache design.

Somayeh Sardashti
David A. Wood
University of Wisconsin-Madison

Caches, especially last-level caches (LLCs), have long been used to reduce effective memory latency and increase effective bandwidth. They also play an increasingly important role in reducing memory system energy. Increasing LLC size can improve performance for most workloads, but the improvement comes at significant area cost. Cache compression, however, seeks to increase effective cache size—by compressing and compacting cache blocks—while incurring small area overheads.[1,2] Unfortunately, previous designs limit compression benefits because of internal fragmentation, limited tags, and energy-expensive recompaction that occurs when a block's size changes.

We propose decoupled compressed cache (DCC), a technique that uses decoupled superblocks (also known as sectors[3]) to increase the maximum effective capacity to four times the uncompressed capacity, while using area overhead comparable to previous cache-compression techniques. DCC uses superblocks—four aligned, contiguous cache blocks that share a single address tag—to reduce tag overhead. Each 64-byte block in a superblock is compressed separately and then compacted into zero to four 16-byte sub-blocks (or segments). DCC decouples the address tags by increasing the number of tags and allowing any sub-block in a set to map to any tag in that set to reduce fragmentation within a superblock.[3]

Decoupling allows sub-blocks of a block to be noncontiguous, thereby eliminating the recompaction overheads of previous variable-size compressed caches.[2] An optimized co-compacted DCC (Co-DCC) design further reduces internal fragmentation (and increases effective capacity) by allocating the compressed blocks from a superblock into the same set of data sub-blocks.

0272-1732/14/$31.00 © 2014 IEEE Published by the IEEE Computer Society

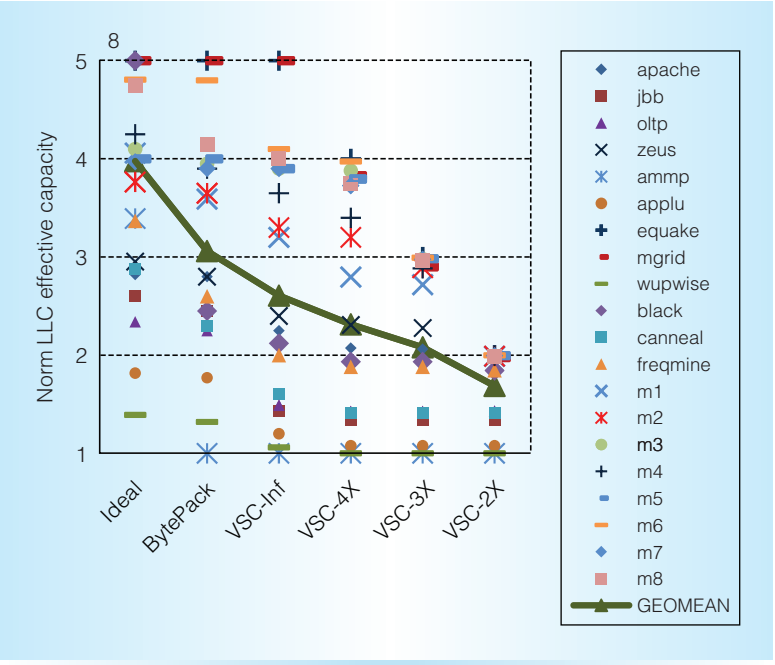

Figure 1. Normalized effective capacity of different compressed cache designs. Although an ideal compressed cache (Ideal) has the potential to significantly increase effective cache size, previous designs (such as VSC-2X) reduce compression effectiveness by limiting the number of tags and internal fragmentation.

This article makes the following contributions:

- DCC uses decoupled superblocks to increase the effective number of tags with low overhead.
- DCC stores compressed data in non-contiguous sub-blocks to eliminate re-compaction overheads when a block's compressed size changes.
- DCC provides more effective capacity, on average, than a conventional cache of twice the size, while slightly increasing cache area. Viewed another way, DCC allows a designer to get approximately the same cache performance with about half the area.
- Co-DCC further reduces internal fragmentation by compacting the blocks of a superblock and allocating them into the same set of data sub-blocks.

In this article, we also present a concrete design for Co-DCC and show how it can be integrated into a recent commercial LLC design with little additional complexity.

Potential and limits of compressed caching

Although some data (and most instructions) are difficult to compress, most workloads are highly compressible. In this article, we use C-PACK+Z, a dictionary-based algorithm[4] with nine-cycle decompression latency. C-PACK+Z achieves an average compression ratio (that is, the original size over compressed size) of 3.9. Thus, compression has the potential to nearly quadruple cache size (shown as "Ideal" in Figure 1).

Previous compressed cache designs fail to achieve this potential for three main reasons. First, caches must compact compressed blocks into sets, which introduces an internal fragmentation problem. In Figure 1, BytePack represents an idealized compressed cache with infinite tags, which compacts compressed blocks on arbitrary byte boundaries. BytePack degrades normalized effective capacity to 3.1 on average. Second, practical compressed caches introduce another internal fragmentation problem by compacting compressed blocks into one or more sub-blocks, rather than storing compressed data on arbitrary byte boundaries.[2] Variable-size compression (VSC) techniques relax the mapping constraint between tags and data and compact compressed blocks into a variable number of contiguous sub-blocks.[2] The column labeled VSC-Inf in Figure 1 illustrates that compacting compressed blocks into zero to four 16-byte sub-blocks (but with infinite tags per set) degrades normalized effective capacity from 3.1 to 2.6 on average. Third, practical compressed caches have a fixed number of tags per set. The remaining columns in Figure 1 illustrate that reducing the number of tags, from infinite to a more practical two times the baseline, degrades the average normalized effective capacity from 2.6 to 1.7. Furthermore, VSC is not energy efficient. It must repack the sub-blocks in a set whenever a block's size changes to make contiguous free space. This action can increase LLC dynamic energy by a factor of nearly three, on average.

Decoupled compressed cache overview

DCC uses decoupled superblock tags (see the "Exploiting Spatial Locality" sidebar for

Exploiting Spatial Locality

Superblocks (also known as sectors) have long exploited coarse-grained spatial locality to reduce tag overhead. Superblocks associate one address tag with multiple cache blocks, replicating only the per-block metadata, such as the coherence state. Figure A1 shows one set of a four-way associative sectored cache (SC), with four-block superblocks. Using four-block superblocks reduces the tag area by 70 percent compared with a conventional cache. However, Figure A1 illustrates that singletons, pairs, and trios—such as superblocks D, C, and A, respectively—result in internal fragmentation, which can lead to significantly higher miss rates.

Seznec showed that decoupling superblock tags from data blocks helps reduce internal fragmentation.[1] Decoupled sectored (or superblock) caches (DSC) increase the number of superblock tags per set and use per-block back pointers to identify the corresponding tag. Figure A2 illustrates how decoupling can reduce fragmentation by letting two singletons (that is, blocks F1 and G3) share the same superblock. DSC uses more tag space than SC but less than a conventional cache because back pointers are small.

Reference

1. A. Seznec, "Decoupled Sectored Caches: Conciliating Low Tag Implementation Cost and Low Miss Ratio," *Proc. 21st Ann. Int'l Symp. Computer Architecture*, 1994, pp. 384-393.

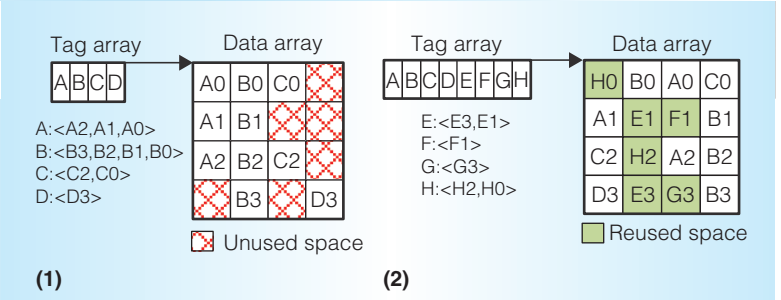

Figure A. Sectored cache (1) and decoupled sectored cache (2). DSC reduces internal fragmentation and can fit more blocks in the cache (Block E to Block H).

more information) to improve cache compression in two ways. First, superblocks reduce tag overhead, permitting more tags per set for comparable overhead. Second, decoupling tags and data reduces internal fragmentation and, importantly, eliminates recompaction when the size of a compressed block changes.

Figure 2 shows how DCC exploits superblocks and manages the cache at three granularities: coarse-grained superblocks, singular cache blocks, and fine-grained sub-blocks. DCC tracks superblocks, which are groups of aligned, contiguous cache blocks (Figure 2d), while it compresses and stores each cache block as a variable number of sub-blocks. Figure 2a shows the key components of DCC for a small, two-way set associative cache with four-block superblocks, 64-byte blocks, and 16-byte sub-blocks. DCC consists of a tag array, a sub-blocked back pointer array, and a sub-blocked data array. DCC is indexed using the superblock address bits ("Set Index" in Figure 2e).

DCC tracks superblocks to fit more compressed blocks into the cache while limiting tag area overhead. DCC explicitly tracks superblocks using a largely conventional superblock tag array. Each tag entry (Figure 2b) consists of one tag per superblock and per-block coherence (C-state) and compression (Comp) states. Because blocks of a superblock share an address tag, the tag array can map more blocks compared with the same size conventional cache without incurring high area overhead. DCC holds as many superblock tags as the maximum number of uncompressed blocks that can be stored. For example, Figure 2a shows a two-way associative cache with four-block superblocks. Each set in the tag array can map eight blocks (that is, 2 superblocks × 4 blocks/superblock), while a maximum of two uncompressed blocks can fit in each set. In the worst-case scenario—when there is no spatial locality (that is, all singletons) or when cached data is uncompressible—DCC can still utilize all the cache data space, for example, by tracking two singletons per set.

Although DCC tracks blocks of a superblock using one tag entry, it allocates or evicts the blocks to and from the data array separately. The data array is a mostly conventional cache data array, organized into sub-blocks.

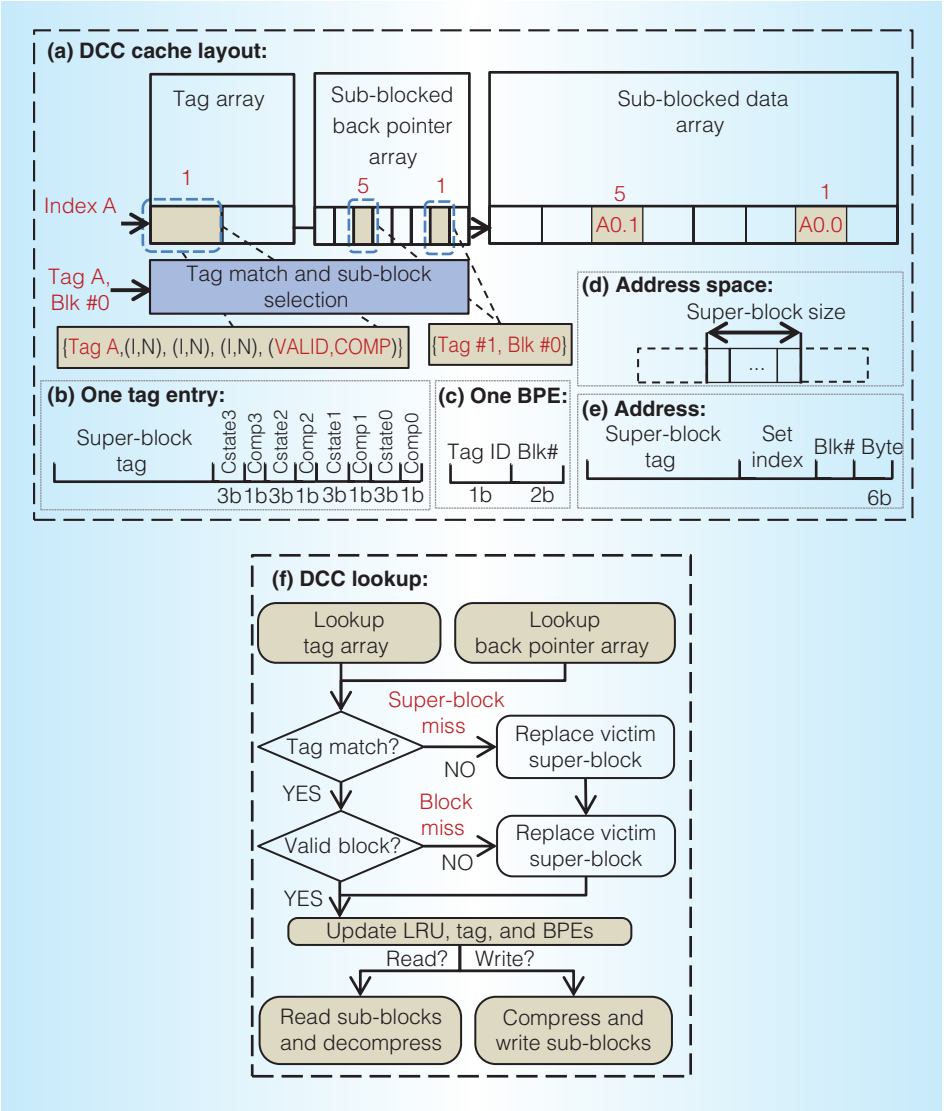

Figure 2. A decoupled compressed cache. DCC cache layout (a); one tag entry (b); one back pointer entry (BPE) (c); address space (d); address (e); and DCC lookup process (f). DCC exploits superblocks and manages the cache at multiple granularities: coarse-grained superblocks, singular cache blocks, and fine-grained sub-blocks.

DCC compacts compressed blocks into a variable number of noncontiguous sub-blocks in the sub-blocked data array. Figure 2a shows block A0 compressed into two sub-blocks (A0.1 and A0.0), which are stored in sub-blocks 5 and 1 in the data array. DCC decouples sub-blocks from the superblock tag using a back pointer array as a level of indirection. Each back pointer entry corresponds to one sub-block in the data array and identifies the owner block (Figure 2c). The back pointer array slightly increases LLC area (see the "Compressed Cache Overheads" sidebar); however, it enables low-overhead, variable-size compression. DCC's decoupled design allows a block's sub-blocks to be noncontiguous, thus eliminating the need for recompaction when a block's size changes.

Co-DCC optimizes DCC to further reduce internal fragmentation. Co-DCC treats blocks from the same superblock as one large block and dynamically allocates them into the same set of data sub-blocks, thereby reducing internal fragmentation within sub-block

Compressed Cache Overheads

Compressed caches can increase cache area because of their extra metadata. Table A shows the quantitative area overheads of decoupled compressed cache (DCC), co-compacted DCC (Co-DCC), and previous work (FixedC and VSC-2X) over the same size conventional cache (16-way associative, 8-Mbyte last-level cache [LLC]). DCC tracks four-block superblocks and almost doubles the per-block metadata largely due to the back pointers. However, because the data array is much larger than the tag array, Cacti calculates the overall LLC area overhead as about 6 percent. DCC's area overhead is similar to FixedC and VSC-2X, which track twice as many tags per set (for example, 32 tags per 16 blocks). Co-DCC further increases metadata stored per block, resulting in 16 percent area overhead compared to the baseline.

Table A also includes the area overhead of (de-)compression units. Because C-PACK+Z's decompressors produce 8 bytes per cycle, we match the cache bandwidth by considering two decompressors per cache bank. Compression is not on the critical path, so we consider one compressor per bank. Thus, for an LLC with eight banks, we need eight compressors and 16 decompressors, resulting in an extra 1.8 percent area overhead.

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

Table A. Last-level cache (LLC) area overheads of different compressed caches.

Components	DCC (%)	Co-DCC (%)	FixedC/ VSC-2X (%)
Tag array	2.1	11.3	6.3
Back pointer array	4.4	5.4	0
Compressors	0.6	0.6	0.6
Decompressors	1.2	1.2	1.2
Total area overhead	8.3	18.5	8.1

boundaries. Co-DCC increases overhead and complexity in exchange for better cache compression.

Figure 2f illustrates the DCC lookup procedure. On a cache lookup, 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 superblock are found available (that is, 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, the corresponding sub-blocks are then read out of the data array and decompressed. On a write, the new, compressed size might be larger, resulting in a block (or a superblock) eviction if sufficient space is not available. On the other hand, in case of a cache miss, DCC allocates the compressed block in the data array. If its superblock is not available, DCC allocates it first in the tag array.

A practical design for DCC

DCC can be integrated into the LLC of a recent commercial design with relatively little additional complexity and, more importantly, no need for an alignment network. The AMD Bulldozer processor implements an 8-Mbyte LLC that is broken into four 2-Mbyte subcaches. Each subcache consists of four banks that can independently service cache accesses.[5] Figure 3 illustrates the data array of one bank in LLC and shows how it is divided into four sequential regions (SRs). Each sequential region runs one phase (that is, half a cycle) behind the previous region and contains a quarter of a cache block (that is, 16 bytes). Figure 3 shows how block A0's four 16-byte sub-blocks (A0.0 to A0.3) are distributed to the same row in each sequential region. Each subsequent sequential region receives the address half a cycle later and takes half a 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 4a.

DCC requires only a small change to the data array to allow noncontiguous sub-blocks. In Figure 3, block B1 is compressed into two sub-blocks (B1.0 and B1.1), which are stored in sequential regions 1 and 2, but

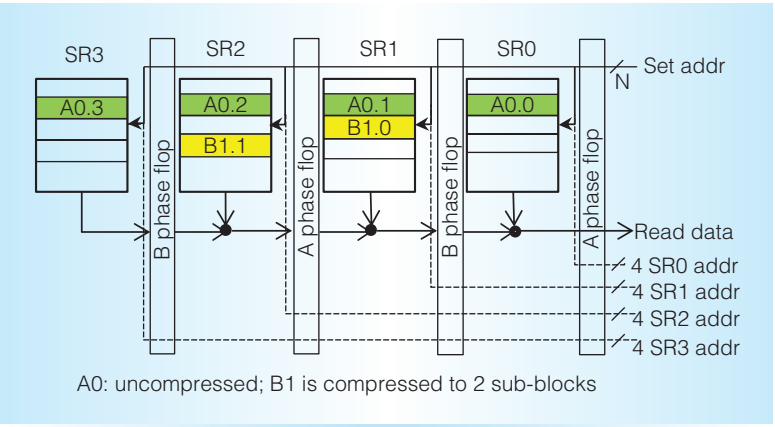

Figure 3. DCC data array organization. The data array is divided into four sequential regions, each containing a sub-block of a cache block.

not in the same row. To select the correct sub-block, DCC must send additional address lines (4 bits for a 16-way associative cache) to each sequential region (illustrated by the dotted lines in Figure 3). 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 4b illustrates DCC timing when reading block B1. The back pointer array is accessed in parallel with the tag array. The sub-block selection logic then finds the back pointer entries 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 because it needs the matched tag ID. To calculate the latency overhead of the sub-block selection, we implemented the tag match and the subselection logic in Verilog, synthesized in 45 nm, and scaled to 32 nm.[6] 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 4b shows how the matching sub-blocks are returned and fed directly into the decompression logic, which accepts 16 bytes per cycle and has a small first-in, first-out (FIFO) buffer to rate match. Decompression starts as soon as the first sub-block arrives (for example, B1.0), depending on which sequential region it resides in. Because sub-block B1.0 resides in sequential region 1, there is one extra cycle (worst case is three cycles). Note that because the decompression latency is deterministic (nine 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.

Evaluation

We evaluate DCC using a full-system simulator based on GEMS.[7] We model a multicore system with eight out-of-order cores; per-core private 32-Kbyte, 8-way L1 instruction and data caches;; per-core private 256-Kbyte, 8-way L2 caches; and one shared 8-Mbyte, 16-way L3 cache.[8] We use CACTI 6.5[9] to model power at 32 nm. We also use a detailed DRAM power model based on Micron Technology's power model.[10] In this section, we report total system energy, which includes energy consumption of processors (cores and caches), on-chip network, and off-chip memory. For DCC and Co-DCC, we use four-block superblocks, 64-byte blocks, and 16-byte sub-blocks. With these parameters, DCC has similar area overhead as FixedC, which doubles the number of tags and compresses a block to half, if possible, and VSC-2X, which doubles tags but compresses a block into zero to four 16-byte sub-blocks (see the "Compressed Cache Overheads" sidebar).

Our evaluations use representative multithreaded and multiprogrammed workloads from commercial workloads (*apache, jbb, oltp, zeus*),[11] SPEC-OMP (*ammp, applu, equake, mgrid, wupwise*),[12] Parsec (*blackscholes, canneal, freqmine*),[13] and mixes of SPEC CPU2006 benchmarks denoted as m1 to m8 (*bzip2, libquantum-bzip2, libquantum, gcc, astar-bwaves, cactus-mcf-milc-bwaves, gcc-omnetpp-mcf-bwaves-lbm-milc-cactus-bzip, omnetpp-lbm*).

Improved cache efficiency

Compressed caches improve the cache's effective capacity by fitting more blocks into the same space. They can achieve the benefits of larger cache sizes with lower area and power overheads.

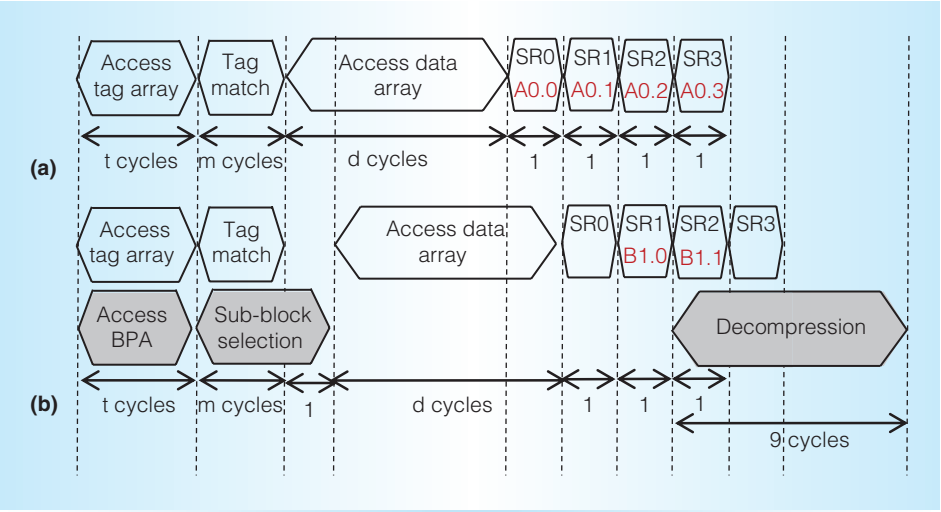

Figure 4. Timing of a conventional cache (a) and DCC (b). A 64-byte block is returned in a burst of four cycles on the same data bus. With DCC, only the matching sub-blocks are read and fed directly into the decompression logic.

- *Result 1:* By exploiting spatial locality, DCC achieves on average 2.2× (and up to 4×) higher LLC effective capacity compared to the baseline, resulting in 18 percent lower LLC miss rate on average and up to 38 percent lower LLC miss rate.
- *Result 2:* Co-DCC further improves the effective cache capacity by reducing internal fragmentation within data sets. It achieves on average 2.6× (and up to 4×) higher effective capacity and 24 percent (up to 42 percent) lower LLC miss rate.
- *Result 3:* DCC and Co-DCC provide significantly higher effective cache capacity and lower miss rate than FixedC and VSC-2X. DCC and Co-DCC also perform better on average than a cache with twice the capacity (2× baseline) while incurring much lower area overhead.

Figure 5a shows LLC effective capacity of different techniques normalized to baseline. We calculate the effective cache capacity by counting valid LLC cache blocks periodically. DCC can significantly improve LLC effective capacity and LLC miss rate (misses per kilo executed instructions [MPKI]) for many applications by fitting more compressed blocks. DCC benefits differ per workload, depending on workload sensitivity to cache capacity, compression ratio, and spatial locality. It achieves the greatest benefit for cache-sensitive workloads with good compressibility and spatial locality (such as *apache* and *omnetpp-lbm/m8*). Workloads with low spatial locality (such as *canneal*) or low compression ratio (such as *wupwise*) observe lower improvements. Cache-insensitive workloads (such as *blackscholes*) also do not benefit from compression.

Overall performance and energy

By improving LLC utilization and reducing accesses to the main memory (that is, the lower LLC miss rate), DCC and Co-DCC significantly improve system performance and energy.

- *Result 4:* DCC and Co-DCC improve LLC efficiency and boost system performance by 10 percent (up to 29 percent) and 14 percent (up to 38 percent) on average, respectively.
- *Result 5:* DCC and Co-DCC save on average 8 percent (up to 24 percent) and 12 percent (up to 39 percent) of system energy, respectively, because of shorter runtime and fewer accesses to the main memory.

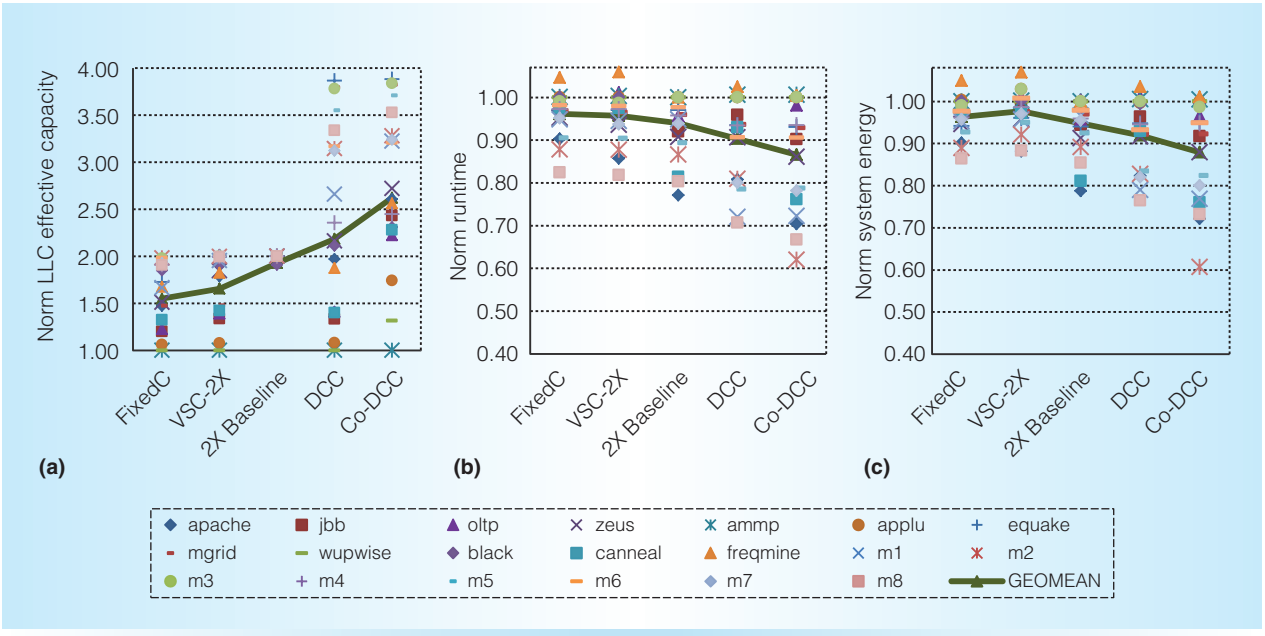

Figure 5. Normalized LLC effective capacity (a); normalized runtime (b); normalized total system energy (c). DCC and Co-DCC improve LLC utilization, resulting in higher performance and energy improvements than previous work and 2x baseline.

- *Result 6:* DCC and Co-DCC achieve 2.5× and 3.5× higher performance improvements, respectively, and 2.2× and 3.3× higher system energy improvements compared with FixedC and VSC-2X.
- *Result 7:* DCC and Co-DCC also improve LLC dynamic energy by about 50 percent on average by accessing fewer bytes. However, VSC-2X hurts LLC dynamic energy for the majority of our workloads because of its need for energy-expensive recompactions.

Figure 5b shows that DCC outperforms baseline, FixedC, VSC-2X, and 2× baseline by effectively more than doubling the cache capacity. DCC and Co-DCC also improve system energy owing to shorter runtime and fewer accesses to the main memory. Figure 5c shows the total system energy of different techniques. DCC and Co-DCC significantly reduce the main memory dynamic energy by reducing the number of cache misses, which contributes to greater system energy improvements as well. Unlike VSC-2X, which hurts LLC dynamic energy because of recompaction, DCC and Co-DCC eliminate the need for recompaction and can even save LLC dynamic energy by accessing fewer bytes when reading or writing compressed data.

DCC demonstrates the potential for compression to increase the effective capacity of last-level caches, improving both performance and energy efficiency. Alternatively, DCC can reduce the chip area required for a given cache capacity, thereby reducing implementation cost. Future work should explore algorithms that better compress instructions and floating point data, as well as extending compression to all levels of the cache and memory hierarchy.

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. We thank Hamid Reza Ghasemi, Dan Gibson, members of the Multifacet research group, and the anonymous reviewers for their comments on the article.

References

1. S. Sardashti and D. Wood. "Decoupled Compressed Cache: Exploiting Spatial Locality for Energy-Optimized Compressed Caching," *Proc. 46th Ann. IEEE/ACM Int'l Symp. Microarchitecture*, 2013, pp. 62-73.
2. A. Alameldeen and D. Wood. "Adaptive Cache Compression for High-Performance Processors," *Proc. 31st Ann. Int'l Symp. Computer Architecture*, 2004, pp. 212-223.
3. A. Seznec, "Decoupled Sectored Caches: Conciliating Low Tag Implementation Cost and Low Miss Ratio," *Proc. 21st Ann. Int'l Symp. Computer Architecture*, 1994, pp. 384-393.
4. X. Chen et al., "C-Pack: A High-Performance Microprocessor Cache Compression Algorithm," *IEEE Trans. VLSI Systems*, vol. 18, no. 18, 2010, pp. 1196-1208.
5. D. Weiss et al., "An 8MB Level-3 Cache in 32nm SOI with Column-Select Aliasing," *Proc. Solid-State Circuits Conf.*, 2011, pp. 258-260.
6. *International Technology Roadmap for Semiconductors, 2010 Update*, ITRS, 2011; www.itrs.net.
7. M. Martin et al., "Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset," *Computer Architecture News*, 2005, vol. 33, no. 4, pp. 92-99.
8. "4th Generation Intel Core i7 Processors," Intel Corporation; www.intel.com/products/processor/corei7.
9. "CACTI: An Integrated Cache and Memory Access Time, Cycle Time, Area, Leakage, and Dynamic Power Model," HP Labs Research; www.hpl.hp.com/research/cacti.
10. "*Calculating Memory System Power for DDR3*," tech. note TN-41-01, Micron Technology, 2007.
11. A. Alameldeen et al., "Simulating a $2M Commercial Server on a $2K PC," *IEEE Computer*, vol. 36, no. 2, 2003, pp. 50-57.
12. V. Aslot et al., "SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance," *Proc. Int'l Workshop OpenMP Applications and Tools: OpenMP Shared Memory Parallel Programming*, 2001, pp. 1-10.
13. C. Bienia and K. Li, "PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors," *Proc. 5th Ann. Workshop Modeling, Benchmarking and Simulation*, 2009, pp. 47-55.

Somayeh Sardashti is a PhD candidate in the Department of Computer Sciences at the University of Wisconsin-Madison. Her research interest is computer architecture, specifically energy-optimized memory hierarchies. Sardashti has an MS in computer science from the University of Wisconsin-Madison and an MS in computer engineering from the University of Tehran. She is a member of IEEE and the ACM.

David A. Wood is a professor in the Department of Computer Sciences and the Department of Electrical and Computer Engineering at the University of Wisconsin-Madison. His research interests include techniques for improving the performance and energy efficiency of multiprocessor and heterogeneous computing systems. Wood has a PhD in computer science from the University of California, Berkeley. He is a fellow of IEEE and the ACM, and a member of the IEEE Computer Society.

Direct questions and comments about this article to Somayeh Sardashti, Department of Computer Science, University of Wisconsin-Madison, 1210 West Dayton Street, Madison, WI 53706-1685; somayeh@cs.wisc.edu.