Skewed Compressed Cache

Somayeh Sardashti, André Seznec, David A. Wood

Skewed Compressed Caches

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

André Seznec
IRISA/INRIA
Andre.Seznec@inria.fr

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

Abstract— **Cache compression seeks the benefits of a larger cache with the area and power of a smaller cache. Ideally, a compressed cache increases effective capacity by tightly compacting compressed blocks, has low tag and metadata overheads, and allows fast lookups. Previous compressed cache designs, however, fail to achieve all these goals.**

In this paper, we propose the Skewed Compressed Cache (SCC), a new hardware compressed cache that lowers overheads and increases performance. SCC tracks super-blocks to reduce tag overhead, compacts blocks into a variable number of sub-blocks to reduce internal fragmentation, but retains a direct tag-data mapping to find blocks quickly and eliminate extra metadata (i.e., no backward pointers). SCC does this using novel sparse super-block tags and a skewed associative mapping that takes compressed size into account. In our experiments, SCC provides on average 8% (up to 22%) higher performance, and on average 6% (up to 20%) lower total energy, achieving the benefits of the recent Decoupled Compressed Cache [26] with a factor of 4 lower area overhead and lower design complexity.

Keywords-component; cache design; compression; performance; energy;

I. INTRODUCTION

In modern processors, last level caches (LLCs) mitigate the limited bandwidth, high latency, and high energy of off-chip main memory. Increasing LLC size can improve system performance and energy, but at the cost of high area and power overheads. Cache compression seeks to achieve the benefits of larger caches, using the area and power of a smaller cache. Compressed caches increase effective capacity by compressing and compacting cache blocks.

A compressed cache design must balance three frequently-conflicting goals: i) tightly compacting variable-size compressed blocks, ii) keeping tag and other metadata overheads low, and iii) allowing fast (e.g., parallel tag/data) lookups. Previous compressed cache designs achieved at most two of these three goals. The earliest compressed caches do not support variable compressed block sizes [25][29][16], allowing fast lookups with relatively low area overheads, but achieve lower compression effectiveness due to internal fragmentation. More recent designs [26][1][13] improve compression effectiveness using variable-size compressed blocks, but at the cost of extra metadata and indirection latency to locate a compressed block.

In this paper, we propose Skewed Compressed Cache (SCC), which achieves all three goals. SCC exploits the fact that most workloads exhibit both (1) spatial locality (i.e., neighboring blocks tend to reside in the cache at the same time), and (2) compression locality (i.e., neighboring blocks tend to compress similarly) [14]. SCC exploits spatial locality by tracking super-blocks, e.g., an aligned, adjacent group of blocks (e.g., eight 64-byte blocks). Using super-blocks allows SCC to track up to eight times as many compressed blocks with little additional metadata. SCC exploits compression locality by compacting neighboring blocks with similar compression ratio into the same physical data entry, tracking them with one tag.

SCC does this using a novel *sparse super-block tag*, which tracks anywhere from one block to all blocks in a super-block, depending upon their compressibility. For example, a single sparse super-block tag can track: all eight blocks in a super-block, if each block is compressible to 8 bytes; four adjacent blocks, if each is compressible to 16 bytes; two adjacent blocks, if each is compressible to 32 bytes; and only one block, if it is not compressible. By allowing variable compressed block sizes—8, 16, 32, and (uncompressed) 64 bytes—SCC is able to tightly compact blocks and achieve high compression effectiveness.

Using sparse super-block tags allows SCC to retain a direct, one-to-one tag-data mapping, but also means that more than one tag may be needed to map blocks from the same super-block. SCC minimizes conflicts between blocks using two forms of skewing. First, it maps blocks to different cache ways based on their compressibility, using different index hash functions for each cache way [11]. To spread all the different compressed sizes across all the cache ways, the hash function used to index a given way is a function of the block address. Second, SCC skews compressed blocks across sets within a cache way to decrease conflicts [2][3] and increase effective cache capacity.

Compared to recent previous work, SCC eliminates the extra metadata needed to locate a block (e.g., the backward pointers in Decoupled Compressed Caches (DCC) [26][27]), reducing tag and metadata overhead. SCC's direct tag-data mapping allows a simpler data access path with no extra latency for a tag-data indirection. SCC also simplifies cache replacement. On a conflict, SCC always replaces one sparse super-block tag and all of the one to eight adjacent blocks packed into the corresponding data entry. This is much simpler than DCC, which may need to replace blocks that correspond to multiple super-blocks as DCC tracks all blocks of a super-block with only one tag.

Compared to conventional uncompressed caches, SCC improves cache miss rate by increasing effective capacity and reducing conflicts. In our experiments, SCC improves system performance and energy by on average 8% and 6%

respectively, and up to 22% and 20% respectively. Compared to DCC, SCC achieves comparable or better performance, with a factor of four lower area overhead, a simpler data access path, and a simpler replacement policy.

This paper makes the following contributions:

(1) SCC is the first compressed cache design to achieve all three goals: variable-size compressed blocks, low tag and metadata overhead, and fast lookups.

(2) SCC exploits spatial and compression locality to pack neighboring blocks with similar compressibility in the same data entry, tracking them with one sparse super-block tag.

(3) SCC enables a simple direct tag-data mapping using a skewed-associative lookup, simultaneously searching different cache ways for different sized compressed blocks.

(4) SCC achieves performance and energy comparable to that of a conventional cache with twice the capacity and associativity, while increasing the area by only 1.5%.

This paper is organized as follows. We discuss background on compressed caching in Section 2, skewed associative caching in Section 3. Then, Section 4 presents our proposal: the Skewed Compressed Cache. Section 5 explains our simulation infrastructure and workloads. In Section 6, we discuss the overheads of compressed caches. We present our evaluations in Section 7. Finally, Section 8 discusses related work, and Section 9 concludes the paper.

II. COMPRESSED CACHING

Cache compression is a promising technique for expanding effective cache capacity with little area overhead. Compressed caches can achieve the benefits of larger caches using the area and power of a smaller cache by compressing and compacting more cache blocks in the same cache area. Designing a compressed cache typically has two main parts: a compression algorithm to compress blocks, and a compaction mechanism to fit compressed blocks in the cache. In this work, we focus on compacting compressed blocks in the last-level cache. Prior work has proposed different compression algorithms that tradeoff compression ratio (i.e., original size over compressed size) and decompression latency. We use the C-PACK+Z compression algorithm because it has been shown to have a good compression ratio with moderate decompression latency and hardware overheads [26][31]. In general, SCC is largely independent of the compression algorithm in use.

To achieve the potentials of a given compression algorithm, the compaction mechanism plays a critical role. Table 1 shows a taxonomy of the current state of the art. Previous proposals differ on two main design factors: (1) how to provide the additional tags, and (2) how to find the corresponding block given a matching tag. Traditionally, compressed caches double the number of tags to track up to 2x cache blocks in the cache [1][13]. Further increasing the number of tags is costly as LLC is already one of the largest on-chip components. More recently, DCC [26] shows that tracking super-blocks is an effective technique to increase the number of tags with low area overhead. DCC uses the same number of tags as a regular cache (e.g., 16 tags per set in a 16-way associative cache) but at coarser granularity.

TABLE 1: COMPRESSED CACHE TAXONOMY

		Number of Tags	
		2x Block Tags	***1x Super-Block Tags***
Tag-Data Mapping	***One-to-one or Direct***	FixedC [25][16]	**SCC [new]**
	Decoupled	VSC [1] IIC-C [13]	DCC [26]

Each tag tracks a 4-block super-block, and can map up to four cache blocks. Tracking super-blocks only slightly increases tag area compared to the same size regular cache.

The subsequent issue is how to find a block given a matched tag (i.e., tag-data mapping). Traditional caches maintain a direct one-to-one relationship between tags and data, so a matching tag implicitly identifies the corresponding data. The earliest compressed caches maintain such a direct relationship by allowing only one compressed size (i.e., half the block size) [25][16]. More recent designs, VSC [1], IIC-C [13], and DCC [26], reduce internal fragmentation by supporting variable-size blocks. They compact compressed blocks into a variable number of sub-blocks (e.g., 0–4 16-byte sub-blocks). Supporting multiple compressed sizes requires a decoupled tag-data mapping that adds a level of indirection to find a block. This requires additional metadata—compressed size in VSC, forwarding pointers in IIC-C, and backward pointers in DCC. This additional metadata allows the cache to compact variable-size compressed blocks in the cache, but at the cost of increased area overhead and design complexity.

III. SKEWED ASSOCIATIVE CACHING

SCC builds on ideas first introduced for skewed-associative caches. In a conventional N-way set-associative cache, each way is indexed using the same index hash function. Thus conflict misses arise when more than N cache blocks compete for space in a given set. Increasing associativity reduces conflict misses, but typically causes an increase in cache access latency and energy cost. Skewed associative caches [2][3] index each way with a different hash function, spreading out accesses and reducing conflict misses.

Figure 1(a) shows a simple 2-way associative cache, which indexes all cache ways with the same function. In this example, blocks A, B, and C all map to the same set. Thus, only two of these blocks can stay in the cache at any time. Figure 1 (b) illustrates a skewed associative cache, which indexes each cache way with a different hash function. In this example, even though blocks A, B, and C map to the same set using function f1, they map to different sets using function f2 in the second cache way. In this way, all three of these blocks can reside in the cache at the same time. By distributing blocks across the sets, skewed associative caches typically exhibit miss ratios comparable to a conventional set-associative cache with twice the ways [2][3].

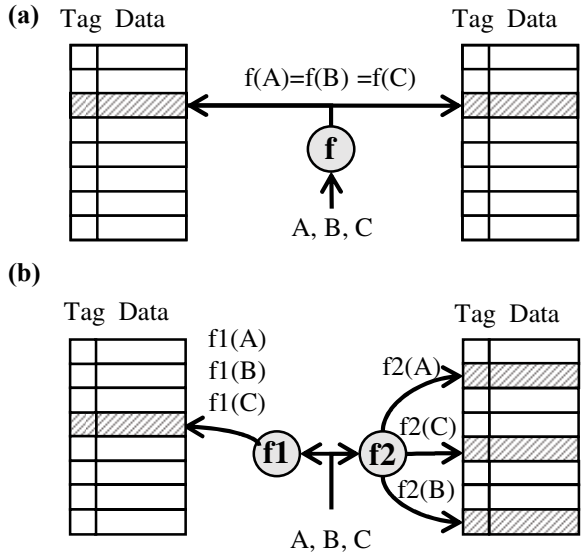


Figure 1. (a) two-way set associative cache (b) skewed associative cache

Skewed associativity has also been used to support multiple page sizes in the same TLB [3], at the cost of reduced associativity for each page size. Using different, page-size specific hash functions for each way, such a TLB can look for different size page table entries in parallel. In this work, we use a similar skewing technique but use compressed size, rather than page size, to select the appropriate way and hash-function combinations.

IV. Skewed Compressed Cache

Previously proposed compressed caches either do not support variable-size compressed blocks [25][29][16] or need extra metadata to find a compressed block, increasing overhead and complexity [26][1][13]. SCC stores neighboring compressed blocks in a power-of-two number of sub-blocks (e.g., 1, 2, 4, or 8 8-byte sub-blocks), using sparse super-block tags and a skewed associative mapping that preserves a one-to-one direct mapping between tags and data.

SCC builds on the observation that most workloads exhibit (1) spatial locality, i.e., neighboring blocks tend to simultaneously reside in the cache, and (2) compression locality, i.e., neighboring blocks often have similar compressibility [14]. SCC exploits both types of locality to compact neighboring blocks with similar compressibility in one physical data entry (i.e., 64 bytes) if possible. Otherwise, it stores neighbors separately.

SCC differs from a conventional cache by storing a sparse super-block tag per data entry. Like a conventional super-block (aka sector) cache, SCC's tags provide additional metadata that can track the state of a group of neighboring blocks (e.g., up to eight aligned, adjacent blocks). However, SCC's tags are sparse because—based on the compressibility of the blocks—they may map only 1 (uncompressed), 2, or 4 compressed blocks. This allows SCC to maintain a conventional one-to-one relationship between a tag and its corresponding data entry (e.g., 64 bytes).

SCC only maps neighboring blocks with similar compressibility to the same data entry. For example, if two aligned, adjacent blocks are each compressible to half their original size, SCC will allocate them in one data entry. This allows a block's offset within a data entry to be directly determined using the appropriate address bits. This eliminates the need for additional metadata (e.g., backward pointers [26]) to locate a block.

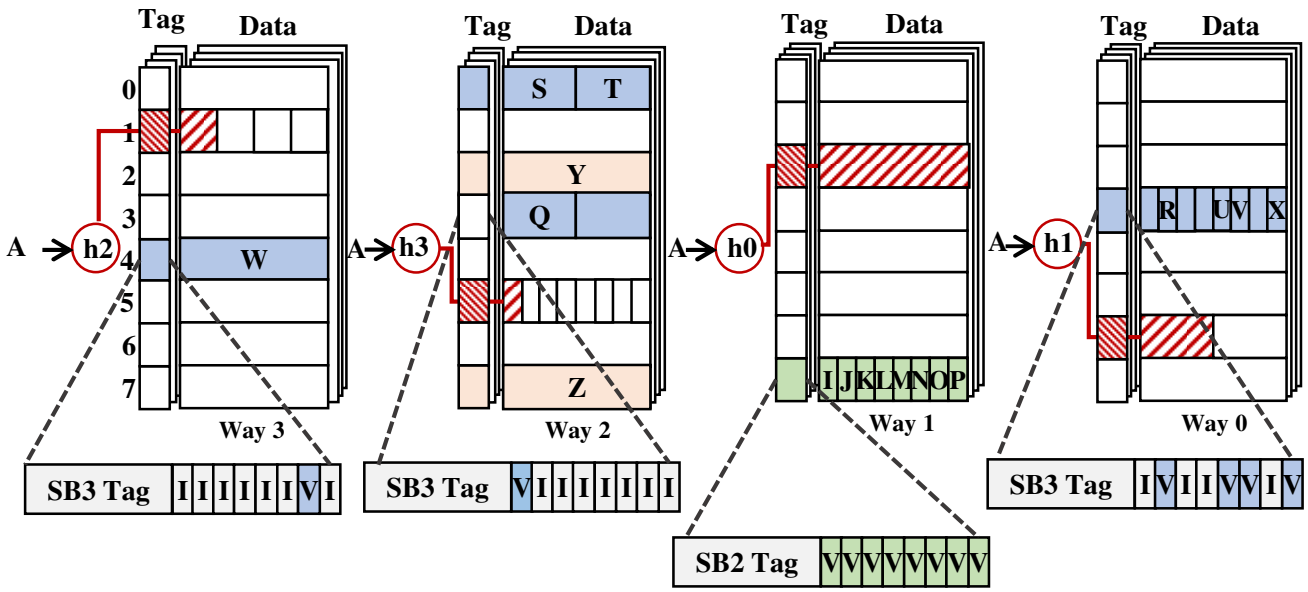
SCC's cache lookup function is made more complicated because the amount of data mapped by a sparse super-block tag depends upon the blocks' compressibility. SCC handles this by using a block's compressibility and a few address bits to determine in which cache way(s) to place the block. For example, for a given super-block, uncompressed blocks might map to cache way #0, blocks compressed to half size might map to cache way #2, etc. Using address bits in the placement decision allows different super-blocks to map blocks with different compressibility to different cache ways. This is important, as it permits the entire cache to be utilized even if all blocks compress to the same size.

To prevent conflicts between blocks in the same super-block, SCC use different hash functions to access ways holding different size compressed blocks. On a cache lookup, the same address bits determine which hash function should be used for each cache way. Like all skewed associative caches, SCC tends to have fewer conflicts than a conventional set-associative cache with the same number of ways.

A. SCC Functionality

Figure 2 illustrates SCC functionality using some examples. This figure shows a 16-way cache with 8 cache sets. The 16 cache ways are divided into four way groups, each including four cache ways. For the sake of clarity, Figure 2 only illustrates super-blocks that are stored in the first way of each way group. This example assumes 64-byte cache blocks, 8-block super-blocks, and 8-byte sub-blocks, but other configurations are possible. A 64-byte cache block can compress to any power-of-two number of 8-byte sub-blocks (i.e., 1, 2, 4, or 8 sub-blocks). Eight aligned neighbors form an 8-block super-block. For example, blocks *I—P* belongs to *SB2*.

SCC associates one sparse super-block tag with each data entry in the data array. Each tag can effectively map (1) a single uncompressed cache block, (2) two adjacent compressed blocks, each compressed to 32 bytes, (3) four adjacent compressed blocks, each compressed to 16 bytes, or (4) eight adjacent compressed blocks, each compressed to 8 bytes. A tag keeps appropriate per-block metadata (e.g., valid and coherence) bits, so it may not be fully populated. If all eight neighbors exist and are compressible to one 8-byte sub-block each, SCC will compact them in one data entry, tracking them with one tag. For example, all blocks of *SB2* are compacted in one data entry in set #7 of way #1. SCC tracks them with the corresponding tag entry with the states of all blocks set as valid (V in Figure 2). If all cache blocks were similarly compressible, SCC would be able to fit eight times more blocks in the cache compared to a conventional uncompressed cache. On the other hand, in the worst-case



SB1: A_ _ _ _ _ _ _; SB2: IJKLMNOP; SB3: QRSTUVWX; SB4: YZ_ _ _ _ _ _

Figure 2. Skewed Compressed Cache

scenario when there is no spatial locality (i.e., only one out of eight neighbors exists in the cache) or blocks are not compressible, SCC can still utilize all cache space by allocating each block separately. For example, there are only blocks *Y* and *Z* from *SB4* present in the cache, and neither are compressible. Thus, SCC stores them separately in two different sets in the same way group, tracking them separately with their corresponding tags.

SCC uses a block's compressibility or compression factor (CF) and a few address bits to determine in which way group to place the block. A block's compression factor is zero if the block is not compressible, one if compressible to 32 bytes, two if compressible to 16 bytes, and three if compressible to 8 bytes. For instance, in Figure 2, block *A* maps to a different set in each cache way depending on its compressibility, shown in hatched (red) entries. SCC allocates *A* in way group #0, #1, #2, or #3 if *A* is compressible to 32 bytes (4 sub-blocks), 64 bytes (8 sub-blocks), 8 bytes (1 sub-block), or 16 bytes (2 sub-blocks), respectively. These mappings would change for a different address, so that each cache way would have a mix of blocks with different compression ratios. For instance, SCC allocates block *A* and block *I* in cache way #1, if *A* is uncompressible and *I* is compressed to 8 bytes (1 sub-block). Using this mapping technique, for a given block, its location determines its compression ratio. This eliminates the need for extra metadata to record block compressibility.

Although SCC separately compresses blocks, it maps and packs neighbors with similar compressibility into one physical data entry. For example, SCC compacts blocks *I* to *P* (*SB2*) into a single physical data entry (set #7 of way #1) as each block is compressed to 8 bytes. However, when neighboring blocks have different compressibility, SCC packs them separately into different physical data entries. For instance, blocks of *SB3* (blocks *Q* to *X*) have three different compression ratios. SCC allocates blocks *R*, *U*, *V*, and *X*, which are compressible to one sub-block each, in one physical data entry (set #3 of way #0). It tracks them with the corresponding tag entry (also shown in Figure 2) with valid states for these blocks. It stores adjacent blocks *S* and *T* in a different physical entry since each one is compressed to four sub-blocks. It also stores block *Q* in way #2 as it is compressible to 32B. Finally, it allocates block *W* separately as it is not compressible, tracking it with a separate sparse super-block tag shown in Figure 2.

Within a physical data entry, a block offset directly corresponds to the block position in its encompassing super-block. In Figure 2, for example block *X* is the first block of *SB3,* similarly its position in the physical data entry in cache way #0 is fixed in the first sub-block. In this way, unlike previous work, SCC does not require any extra metadata (e.g., backward pointers [26] or forward pointers [13]) to locate a block in the data array. By eliminating the need for extra pointers, SCC simplifies data paths, provides fast lookups, lowers area overhead and design complexity, while still allowing variable compressed sizes.

While eliminating extra metadata simplifies SCC's design, it has the potential to hurt cache performance by increasing conflict misses and lowering effective cache associativity. A conventional 16-way set-associative cache can allocate a block in any cache way, but SCC restricts a block to a 4-way way group based on the block's compression factor. For example, when storing block *A* with compressed size of 16B, SCC can store it only in one of the four cache ways (including way #3) grouped together in Figure 2. To mitigate the effect of this restriction, SCC employs skewing inside way groups, indexing each cache way with a different hash function to spread out accesses.

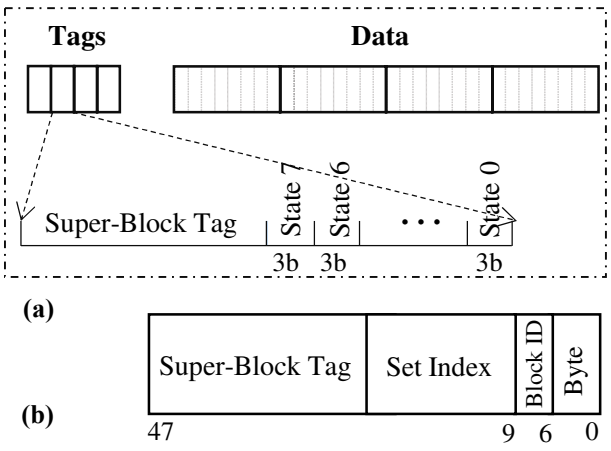


Figure 3. (a) One set of SCC (b) Address

This helps to reduce conflict misses and increases effective associativity.

B. SCC Structure

Structurally, SCC shares many common elements with previously proposed compressed caches [26][1] and the multi-page size skewed-associative TLB [3]. Figure 3 (a) shows one set of SCC tag array and its corresponding data set for a 4-way associative cache. Similar to a regular cache, SCC keeps the same number of tags as physical data entries in a cache set (e.g., 4 tags and 4 data entries per set in Figure 3). However, unlike a regular cache, which tracks exactly one single block per tag entry, SCC tag entries track a super-block containing 8 adjacent blocks. Figure 3 illustrates that each tag entry includes the super-block tag address and per-block coherency/valid states (e.g., eight states for 8-block super-blocks). Figure 2 also shows some examples of tag entries for block *W* in set #4 of way #3, blocks *I—P* in set #7 of way #1, and blocks *R,U,V,X* in set #3 of way #0. The data array is largely similar to a conventional cache data array, except it is organized at sub-blocks (e.g., 8 bytes).

$$W_1W_0 = A_{10}A_9 \wedge CF_1CF_0 \quad (1)$$

Unlike a regular cache that can allocate a block in any cache way, SCC takes into account block compressibility. Equation (1) shows the way selection logic that SCC uses when allocating a cache block. It uses the block compression factor (CF_1CF_0) and two address bits ($A_{10}A_9$) to select the appropriate way group (W_1W_0). The block compression factor (CF_1CF_0) is zero if the block is not compressible, one if compressible to 32 bytes, two if compressible to 16 bytes, and three if compressible to 8 bytes. SCC maps neighboring blocks with similar compressibility to the same data entry. Thus, the way selection logic uses address bits $A_{10}A_9$, which are above the super-block offset. Note that since SCC uses address bits in way selection, even if all cache blocks are uncompressible (CF == 0), they will spread out among all cache ways.

SCC uses different set index functions to prevent conflicts between blocks in the same super-block. Just using bit selection, e.g., the consecutive bits beginning with A_{11}, would result in all blocks in the same super-block mapping to the same set in a way group, resulting in unnecessary conflicts. For example, if none of the blocks were compressible, then all eight uncompressed blocks would compete for the four entries in the selected way group (in Figure 2). To prevent this, SCC uses the index hash functions shown in (2), which draw address bits from the Block ID for the less compressible blocks. These functions map neighboring blocks to the same set only if they can share a data entry (based on their compression factor). SCC also uses different hash functions [3] for different ways in the same way group, to further reduce the possibility of conflicts.

$$\text{Set Index} = \begin{cases} h_0(\{A_{47}\text{—}A_{11}, A_8A_7A_6\}) & \text{if CF==0} \\ h_1(\{A_{47}\text{—}A_{11}, A_8A_7\}) & \text{if CF==1} \\ h_2(\{A_{47}\text{—}A_{11}, A_8\}) & \text{if CF==2} \\ h_3(A_{47}\text{—}A_{11}) & \text{if CF==3} \end{cases} \quad (2)$$

Within a 64-byte data entry, a compressed blocks location depends only on its compression factor and address, eliminating the need for extra metadata. Equation 3 shows the function to compute the byte offset for a compressed block within a data entry.

$$\text{Byte Offset} = \begin{cases} \text{none} & \text{if CF==0} \\ A_6 << 5 & \text{if CF==1} \\ A_7A_6 << 4 & \text{if CF==2} \\ A_8A_7A_6 << 3 & \text{if CF==3} \end{cases} \quad (3)$$

C. SCC Cache Operations

Figure 4 illustrates how SCC operates for the main cache operations. On a cache lookup, since the accessing block's compressibility is not known, SCC must check the block's corresponding positions in all cache ways. To determine which index hash function to use for each way, SCC uses (4), the inverse of (1).

$$CF_1CF_0 = A_{10}A_9 \wedge W_1W_0 \quad (4)$$

For example, in Figure 2, when accessing block *A*, the tag entries in set #1 of way #3, set #5 of way #2, set #2 of way #1, and set #6 of way #0 (i.e., all hatched red tag entries) are checked for a possible match. A cache hit occurs if its encompassing super-block is present (i.e., a sparse super-block tag match), and the block state is valid. On a read hit, SCC uses the compression factor and appropriate address bits (using (3)) to determine which of the corresponding sub-blocks should be read from the data array.

On a write hit (e.g., a write-back to an inclusive last-level cache), the block's compressibility might change. If the block can still fit in the same place as before (i.e., its new

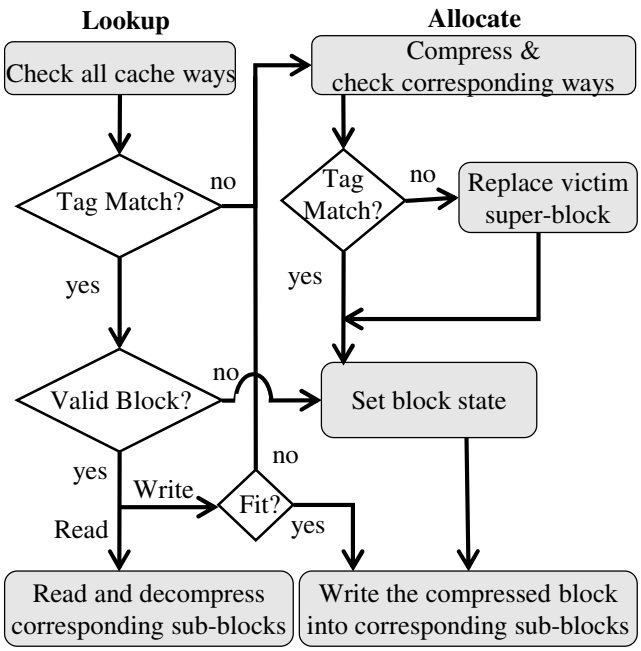


Figure 4. SCC Operations

size is less than or equal to the old one), SCC will update the block in place. Otherwise, SCC invalidates the current version of the block first by setting its corresponding state to invalid. Note that neighboring blocks that share the data entry are not affected. SCC then allocates a new entry as described below for a cache miss. Fortunately, this case does not arise very frequently; simulation results show that on average 97% of updated blocks fit in their previously allocated space.

SCC handles cache misses and write hits that do not fit in their previous space the same way. SCC first uses the block's (new) compression factor and address to search whether an existing sparse super-block of the right size has already been allocated for a neighboring block. For example, consider a write to block *R* in Figure 2 that changes the compression factor from 3 (8 bytes) to 1 (32 bytes). SCC would invalidate the old copy of R in set #7 of way #0 and write the new data in set #3 in way #2.

Detecting a sparse super-block hit is more complex than a normal tag match for two reasons. First, the size of the sparse super-block—and hence the number of tag bits that must be checked—depends upon the compression factor. For example, to detect that block R can be reallocated to the sparse super-block in set #3 of way #2, SCC must make sure that not only the super-block tag bits match, but that bits A_8 and A_7 also match, since the compression factor is 1 (32 bytes). Second, since SCC does not store bits $A_8A_7A_6$ in the tag entry, it must infer them from the coherence states. For example, SCC can infer that both A_8 and A_7 are one in set #3 of way #2 by testing if either $State_7$ or $State_6$ are valid (in this example $State_7$ is valid because block Q is valid).

If no matching sparse super-block tag with the right compression factor exists, SCC needs to select and evict a victim to make room. SCC selects the least-recently-used super-block tag within the way group (e.g., one of 4 ways in Figure 2). It then evicts all blocks that map to that tag's corresponding data entry. For example, if SCC needs to allocate a new block in set #0 of way #2, it would free that data entry by evicting blocks *S* and *T* (i.e., both cache lines in that data entry). Note that the rest of blocks from *SB3* will stay in the cache in set #4 of way #3 (block *W)* and set #3 of way #0 (blocks *R,U,V,X)*. For victim blocks, SCC can determine their compression factor based on the cache way and tag address using (4). After evicting the victim blocks, SCC updates the sparse super-block tag and inserts the new compressed block into the appropriate sub-blocks of the data entry.

SCC's replacement mechanism is much simpler than that needed by DCC. In DCC, allocating space for a block can trigger the eviction of several blocks, sometimes belonging to different super-blocks. In case of a super-block miss, all blocks associated with the victim super-block tag must be evicted, unlike SCC that evicts only blocks belonging to a particular data entry. In addition, in DCC, blocks belonging to other super-blocks may need to be evicted too. Thus, determining which block or super-block is best to replace in DCC is very complex.

SCC also never needs to evict a block on a super-block hit, while DCC may. SCC will allocate the missing block in its corresponding data entry, which is guaranteed to have enough space since the compression factor is used as part of the search criteria. In DCC, a super-block hit does not guarantee that there is any free space in the data array.

V. METHODOLOGY

Our target machine is an 8-core multicore system (Table 2) with OOO cores, per-core private L1 and L2 caches, and one shared last level cache (L3). We implement SCC and other compressed caches at the L3. We evaluate SCC using full-system cycle-accurate GEMS simulator [24]. We use CACTI 6.5 [10] to model area and power at 32nm. We report total energy of cores, caches, on-chip network, and main memory.

We simulate different applications from SPEC OMP [30], PARSEC [11], commercial workloads [5], and SPEC CPU 2006. Table 3 shows the list of our applications. We run mixes of multi-programmed workloads from memory-bound and compute-bound SPEC CPU 2006 benchmarks. For example, for astar-bwaves, we run four copies of each benchmark. In Table 3, we show our applications in increasing LLC MPKI (Misses per Kilo executed Instructions) order for the Baseline configuration. We

TABLE 2: SIMULATION PARAMETERS

Processors	8, 3.2 GHz, 4-wide issue, out-of-order
L1 Caches	32 KB 8-way split, 2 cycles
L2 Caches	256 KB 8-way, 10 cycles
L3 Cache	8 MB 16-way, 8 banks, 27 cycles
Memory	4GB, 16 Banks, 800 MHz DDR3.

TABLE 3: APPLICATIONS

	Application	LLC MPKI
Low Mem Intensive	ammp	0.01
	blackscholes	0.13
	canneal	0.51
	freqmine	0.65
Medium Mem Intensive	bzip2 (mix1)	1.7
	equake	2.2
	oltp	2.3
	jbb	2.7
	wupwise	4.3
High Mem Intensive	gcc-omnetpp-mcf-bwaves-lbm-milc-cactus-bzip (mix7)	8.4
	libquantum-bzip2 (mix2)	9.3
	astar-bwaves (mix5)	9.3
	zeus	9.3
	gcc-166 (mix4)	10.1
	apache	10.6
	omnetpp-4-lbm-4(mix8)	11.2
	cactus-mcf-milc-bwaves (mix6)	13.4
	applu	25.9
	libquantum(mix3)	43.9

classify these workloads into: low memory intensive (L), medium memory intensive (M), and high memory intensive (H) if their LLC MPKI is lower than one, between one and five, and over five respectively. We run each workload for approximately 500M instructions with warmed up caches. To address workload variability, we simulate each workload for a fixed number of work units (e.g., transactions) and report the average over multiple runs [6].

We study the following configurations at LLC:

- **Baseline** is a conventional 16-way 8MB LLC.
- **2X Baseline** is a conventional 32-way 16MB LLC.
- **FixedC** doubles the number of tags (i.e., 32 tags per set) compared to Baseline. Each cache block is compressed to half if compressible, otherwise stored as uncompressed.
- **VSC** doubles the number of tags compared to Baseline. A block is compressed and compacted into 0-4 contiguous 16-byte sub-blocks.
- **DCC_4_16** has same number of tags per set (i.e., 16 tags per set) as the Baseline, but each tracks up to 4 neighboring blocks (4-block super-blocks). In DCC, one tag tracks all blocks belonging to a super-block. A block is compressed to 0-4 16-byte sub-blocks, compacted in order but not necessarily in contiguous space in a set. This is the configuration analyzed by Sardashti and Wood [26].
- **DCC_8_8** is similar to DCC_4_16, but it tracks up to 8 neighboring blocks (8-block super-blocks). A block is compressed to 0-8 8-byte sub-blocks.
- **SCC_8_8** has same number of tags per set (i.e., 16 tags per set) as the Baseline, but each tracks up to 8 neighboring blocks (8-block super-blocks). Unlike DCC, SCC might use multiple sparse super-block tags to track blocks of a super-block in case all cannot fit in one data entry. A block is compressed to 1-8 8-byte sub-blocks. A given block can be mapped to a group of four cache ways (out of 16 ways) based on block address and compressibility.
- **SCC_4_16** is similar to SCC_8_8, but it tracks 4-block super-blocks. A block is compressed to 1-4 16-byte sub-blocks. For a given address, we divide the cache into three way groups containing 4 ways, 4 ways, and 8 ways, respectively. We map a block to these groups if the block is uncompressed, compressed to 32-bytes, or compressed to 16-bytes, respectively.
- **Skewed Base** models a 4-way skewed associative cache with conventional tags (no super-blocks) and no compression.

TABLE 4: COMPRESSED CACHES AREA OVERHEAD

	Tags	Coherence Metadata	Compression Metadata	Total LLC Overhead
FixedC	5.3%	0.6%	0.3%	6.2%
VSC	5.3%	0.6%	1.1%	7.0%
DCC 4_16	-0.1%	1.7%	5.2%	6.8%
DCC 8_8	-0.3%	3.8%	11.8%	15.3%
SCC 4_16	-0.2%	1.7%	0	1.5%
SCC 8_8	-0.4%	3.9%	0	3.5%

VI. DESIGN COMPLEXITIES

Compressed caches effectively increase cache capacity at the cost of more metadata. Table 4 shows the area breakdown of different compressed caches compared to Baseline. We assume a 48-bit physical address space. These compressed caches differ in the way they provide needed tags to track compressed blocks, and their tag-data mapping. In Table 4, we separate their area overhead caused by more tags (including tag addresses and LRU information), extra metadata for coherence information, and extra metadata for compression (including any compression flag, compressed block size, etc.).

The earlier FixedC and VSC designs double the number of tags, which increases the LLC area by about 6%. FixedC requires no additional metadata for tag-data mapping, since it retains a one-to-one tag-data relationship. It only stores a 1-bit flag per block to represent if a block is compressed or not. VSC allows variable-size compressed blocks, requiring three bits of additional metadata per block to store its compressed size. VSC uses this modest additional metadata to determine the location of a compressed block.

DCC uses the same number of tags as Baseline, but each tag tracks a 4- or 8-block super-block. The tags use fewer bits for the matching address, thus compared to a regular cache tags are smaller. On the other hand, DCC needs additional coherence state for each block. DCC_4_16, with 4-block super-blocks, increases LLC area by 1.7% due to more coherence states. By doubling the super-block size, DCC_8_8 can track twice as many blocks but increases the

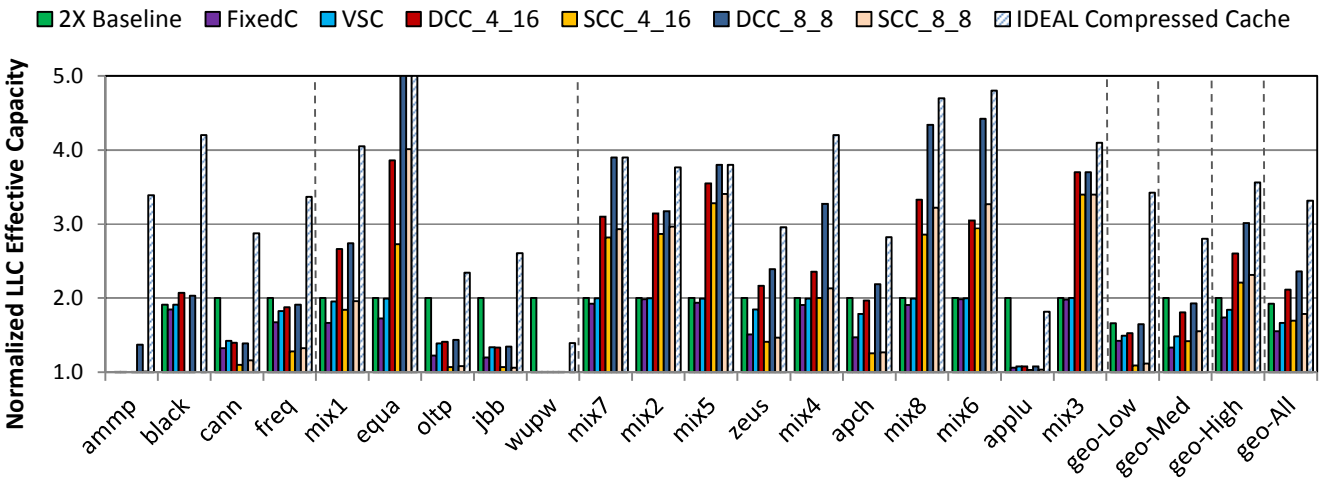


Figure 5. Normalized LLC effective capacity

additional area overhead to 3.8%. DCC decouples tag-data mapping, requiring extra metadata to hold the backward pointers that identify a block's location. DCC keeps one backward pointer entry (BPE) per sub-block in a set. In DCC_4_16, backward pointer entries incur 5.2% area overhead. Smaller sub-block sizes can reduce internal fragmentation, and so improve cache utilization, but at the cost of more BPEs. DCC_8_8 uses 8-block sub-blocks, has 16*8 BPEs per set, resulting in 11.8% extra area overhead for the metadata.

SCC also tracks super-blocks, and thus has tag overhead lower than a conventional cache, but differs from DCC in two ways. First, SCC only needs (pseudo-)LRU state for the tags, while DCC maintains additional state for the decoupled sub-blocks. Second, SCC does not require extra metadata to track a block's location because of its direct tag-data mapping. SCC keeps only the tag address, LRU state and per-block coherence states. SCC_4_16 incurs 1.5% area overhead, more than a factor of 4 lower overhead than DCC_4_16. Similarly, SCC_8_8 incurs 3.5% area overhead, 78% less area overhead than DCC_8_8.

VII. Evaluation

A. Cache Utilization

Figure 5 shows the effective capacity of the alternative cache designs normalized to Baseline for our workloads. We calculate the effective capacity of a cache by periodically counting the number of valid blocks. An ideal compressed cache would have a normalized effective capacity that is the same as the application's compression ratio. Practical compressed caches trade off effective capacity for lower overheads and lower complexity. In addition, some low memory intensive workloads, such as ammp, have small working sets, which fit in a small cache even though they have highly compressible data.

Figure 5 also shows that compressed caches can achieve much of the benefit of doubling the cache size, despite their low area overheads. 2X Baseline, which doubles the area used by the LLC, can hold on average 1.9 times more blocks

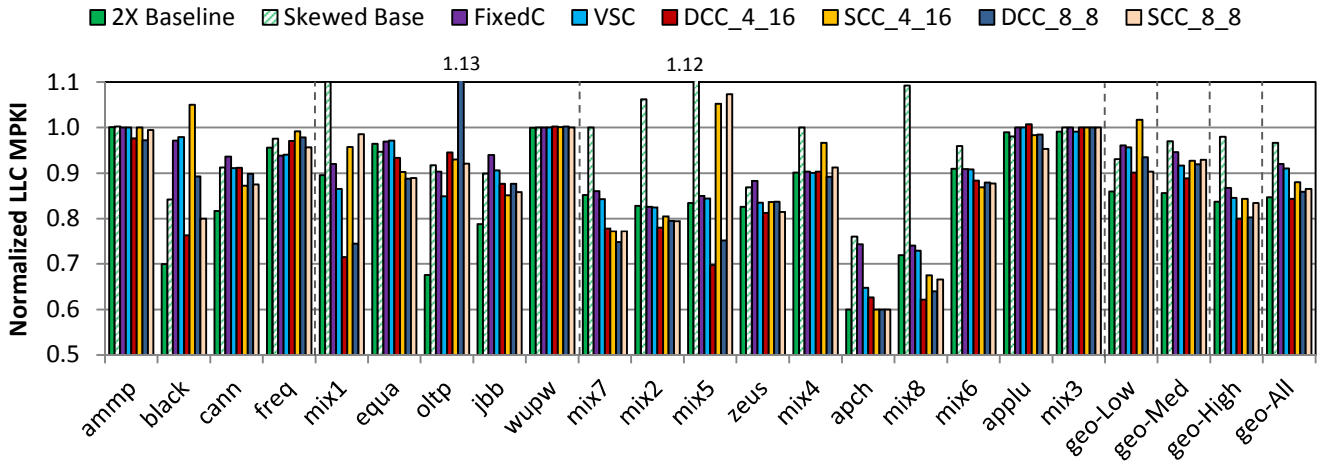


Figure 6. Normalized LLC MPKI

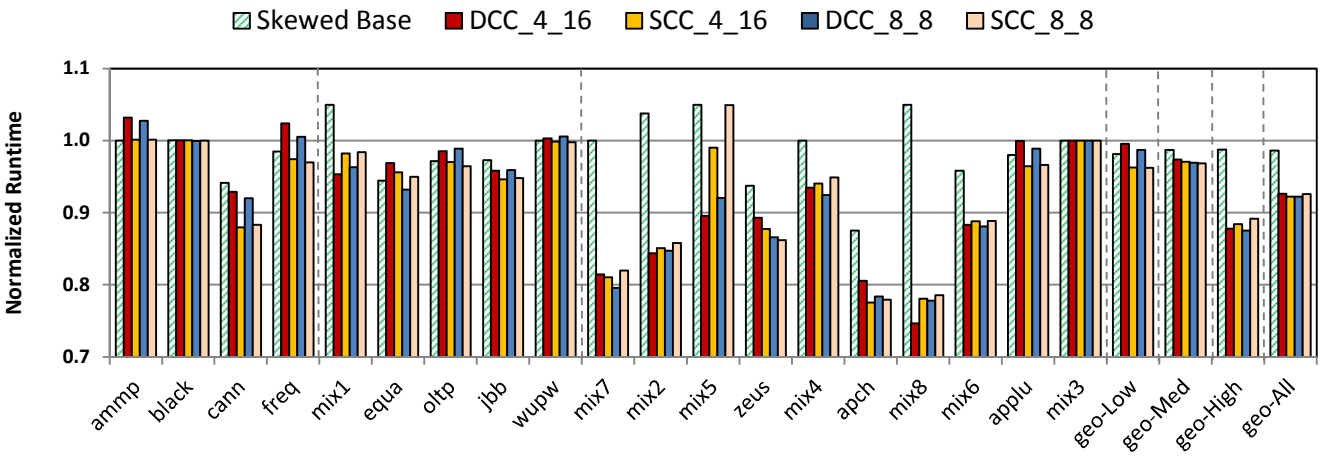


Figure 7. Normalized performance of different SCC and DCC configurations

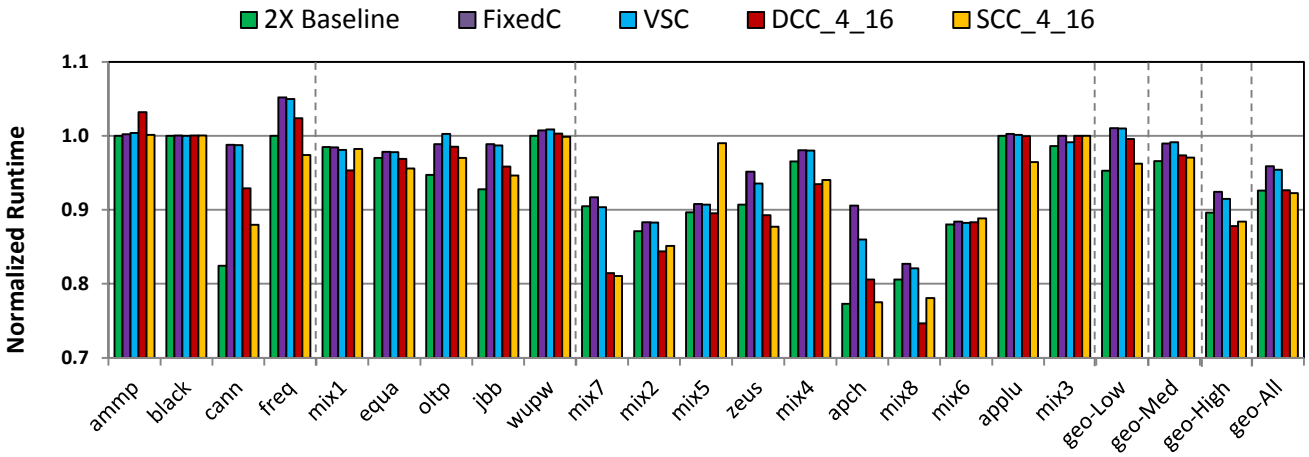


Figure 8. Normalized performance of different cache designs

(at most 2x and only 1.6x for the low memory intensive applications). FixedC and VSC provide, on average, 1.5x and 1.6x the normalized effective capacity, respectively. Like 2X Baseline, they can hold at most twice as many blocks since they have exactly twice as many (regular) tags.

SCC and DCC can further increase effective capacity because tracking super-blocks allow a maximum effective capacity equal to the super-block size (e.g., 4x and 8x). SCC_4_16 and SCC_8_8 provide, on average, normalized effective capacities of 1.7 and 1.8. SCC achieves the highest effective capacity for memory intensive workloads (on average ~2.3), outperforming 2X Baseline.

DCC achieves a greater normalized effective capacity than SCC because its decoupled tag-data mapping reduces internal fragmentation and eliminates the need to ever store more than one tag for the same super-block. In DCC, non-neighboring blocks can share adjacent sub-blocks, while in SCC only neighboring blocks can share a data entry and only if they are similarly compressible. In addition, DCC does not store zero blocks in the data array, while SCC must allocate a block with compression factor of 3 (i.e., 8 bytes). DCC_4_16 and DCC_8_8 achieve, on average, normalized effective capacities of 2.1 and 2.4, respectively. Of course, this comes at more than four times the area overhead and higher design complexity compared to SCC.

B. Cache Miss Rate

Figure 6 shows the LLC MPKI (Misses per Kilo executed Instructions) for different cache designs. Doubling cache size (2X Baseline) improves LLC MPKI by 15%, on average, but at significant area and power costs. Compressed caches, on the other hand, increase effective capacity and reduce cache miss rate with smaller overheads.

SCC improves LLC miss rate, achieving most of the benefits of 2x Baseline. On average, SCC provides about 13% lower LLC MPKI than Baseline. It achieves the greatest improvements for memory intensive workloads (on average %16). SCC's improvements come from two sources: reduced capacity misses and reduced conflict misses. By increasing effective capacity using compression, SCC obviously tends to reduce capacity misses. But SCC also reduces conflict misses as a result of its skewed-associative tag mapping. SCC primarily uses skewing to map different size compressed blocks to one of four way groups, while preserving a direct, one-to-one tag-data mapping. SCC

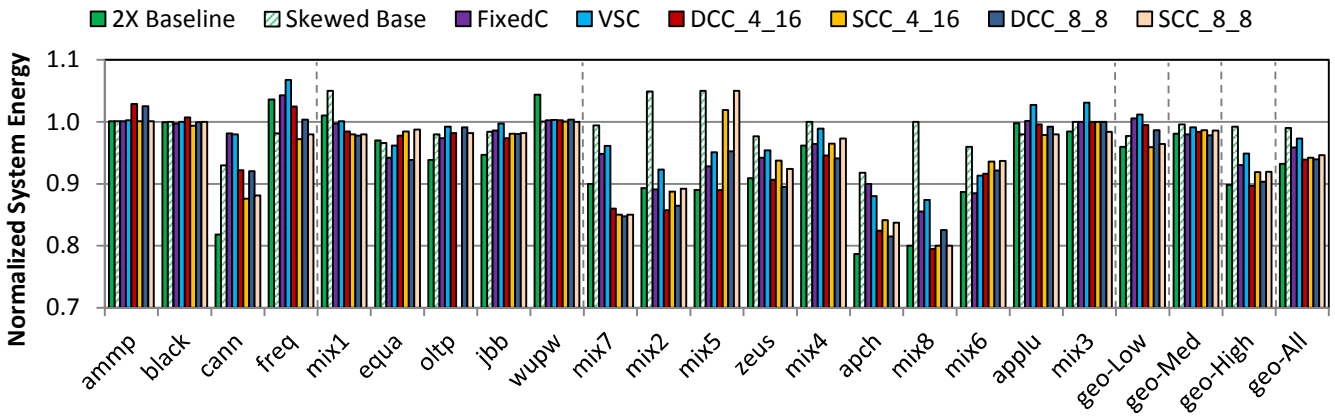


Figure 9. Normalized system energy

further uses skewing to reduce conflicts between blocks within a 4-way way group.

To show the impacts of skewing on miss rate, Skewed Base in Figure 6 models a 4-way skewed cache. On average, Skewed Base performs in the same range as the 16-way Baseline (about 4% lower MPKI). For some workloads, such as Apache and Zeus, skewing reduces conflict misses significantly by spreading out the accesses. In SCC, this results to even lower miss rate of these workloads due to compression. On the other hand, for few workloads (mix2, mix5, and mix8), skewing cannot compensate the negative impacts of lowering the associativity in Skewed Base. For those workloads, SCC shows lower miss rate improvements, and even 7% LLC miss rate increase for mix5.

Compared to DCC, SCC provides similar improvements with a factor of 4 lower area overheads. By tracking super-blocks, both DCC and SCC perform better than FixedC and VSC. Although DCC_8_8 achieves higher effective capacity than DCC_4_16 and SCC, it performs on average similar to DCC_4_16. For oltp, DCC_8_8 even increases LLC MPKI by about 13% as mapping 8 neighbors to the same set can increase conflict misses.

For completeness we also analyzed a design that combines skewed associativity with DCC. Due to the decoupled tag-data mapping of DCC, adding skewing to DCC results in a more complicated design and replacement policy that we do not consider practical to implement. DCC stores sub-blocks of a block anywhere in a cache set. When applying skewing, this means the sub-blocks of a block can be indexed to different sets. Thus, a BPE needs to store set index as well resulting to high area overheads (~15% area overhead for a configuration similar to DCC_4_16). In addition, skewing can significantly complicate replacement policy in DCC. A block allocation can trigger multiple block evictions as a block can be allocated across different sets. Our results (not shown here) show that adding skewing to DCC improves it marginally.

C. *System Performance and Energy*

Figure 7, Figure 8, and Figure 9 show system performance and energy of different cache designs. Our reported system energy includes both leakage and dynamic energy of cores, caches, on-chip network, and off-chip main memory.

By increasing cache efficiency and reducing accesses to the main memory, compressed caches can improve system performance and energy, achieving the benefits of larger caches. SCC improves system performance and energy by up to 22% and 20% respectively, and on average 8% and 6% respectively. SCC achieves comparable benefits as previous work DCC with a factor of four lower area overheads.

SCC benefits differ per application. It provides the highest improvements for memory intensive workloads (on average 11% and maximum of 23% faster runtime for apache). On the other hand, it has the smallest gains for low memory intensive workloads (on average 4%). For cache insensitive workloads, such as ammp, blackscholes and libquantum (mix3), SCC does not impact their performance and energy.

Figure 8 also shows the performance of Skewed Base, which basically separates skewing impacts on SCC performance. In Skewed Base, a block can be mapped to a group of 4 ways based on its address. Each of those ways is hashed differently. For some workloads, such as apache, Skewed Base improves their performance and energy by spreading out accesses. For these workloads, although SCC has smaller effective capacity than previous work DCC, SCC overall miss rate improvement is comparable to DCC and 2X Baseline. On the other hand, for few workloads (mix1, mix5, and mix8), skewing cannot compensate the effect of lower effective associativity in Skewed Base. For these workloads, SCC achieves lower performance and energy improvements. For mix5 (astar-bwave), SCC has about 5% increase in runtime and energy.

VIII. RELATED WORK

Compression. Data compression has been widely used to increase storage capacity at disks, main memory, and caches. IBM MXT [9] uses real-time main memory compression to effectively double the main memory capacity. Ekman et al. [23] propose a compressed memory design that extends page table entries to keep compressed block sizes. The decoupled zero-compressed memory [19] detects null blocks using a

design based on decoupled sectored set-associative caches. Pekhimenko, et al. [14] propose a new compressed memory design that exploits compression locality to simplify address mapping.

There are many proposals on cache compression. Arelakis et al. [8] utilize statistical compression to improve cache performance and power. Lee's proposed compressed cache [16] uses a direct tag-data mapping, packing adjacent, aligned blocks into a single line if both blocks are compressed by at least a factor of two. Alameldeen and Wood [1] propose a variable-size compressed cache that uses extra metadata (block size) to locate a block. They also propose an adaptive technique to turn off compression whenever not beneficial, which could potentially be used with SCC. Similarly, IIC-C [13] supports variable compressed blocks, but uses forward pointers to associate a tag with its sub-blocks resulting in high cache area overhead (24%). Dusser et al. [19] augment the conventional cache with a specialized cache to store zero blocks (ZCA). A recent compressed cache design, DCC [26], tracks super-blocks to track more blocks with low area overhead. Unlike SCC, DCC decouples tag-data mapping using backward pointers to locate sub-blocks of a compressed block in a set.

Compression has also been used to reduce cache power consumption. Residue cache [29] reduces L2 cache area and power by compressing and storing blocks in half size if compressible. DZC [22] also reduces power by only storing non-zero bytes of cache blocks. Similarly, FVC [21] and significance compression [25] can reduce power by accessing half a block if it is compressible. Similarly SCC can also reduce LLC dynamic power due to accessing fewer bytes for compressed blocks.

S. Baek, et al. [27] improves the performance of compressed caches using a size-aware cache replacement mechanism. Their proposal is orthogonal to ours.

Skewing. Skew-associative caches [2][3] index each way with a different hash function, resulting in lower conflict misses. ZCache uses skewing, but further increases associativity by increasing the number of replacement candidates for a given block. In the context of TLB management, Seznec [3] exploits skewed associativity to handle multiple page size granularity at the cost of reduced associativity for each page size.

Super-block caches. This work builds on earlier work on super-block caches. Super-blocks, originally called sectors, have long used to reduce tag overhead [18][7][17]. A conventional super-block cache uses one tag to track multiple adjacent, aligned blocks, reducing tag area overhead at the cost of lower cache utilization due to internal fragmentation. Seznec's [7] Decoupled Sector Cache reduces internal fragmentation by decoupling the tags from the data blocks using backward pointers.

IX. CONCLUSIONS

In this paper, we propose Skewed Compressed Cache, a new low-overhead hardware compressed cache. SCC compacts compressed blocks in the last-level cache in such a way that it can find them quickly, and minimize the storage overhead and design complexity. To do so, SCC uses sparse super-block tags to track more compressed blocks, compact blocks into a variable number of sub-blocks to reduce internal fragmentation, but retain a direct tag-data mapping to find blocks quickly and eliminate the extra metadata.

SCC proposes a direct tag-data mapping by exploiting compression locality. It compresses blocks to variable sizes, and at the same time eliminates the need for extra metadata (e.g., backward pointers). It dynamically packs neighboring blocks with similar compressibility in the same space tracking them with one sparse super-block tag. SCC further uses skewing to spread out blocks for lower conflicts. Like previous work DCC, SCC achieves performance comparable to that of a conventional cache with twice the capacity and associativity. But SCC does this with less area overhead (1.5% vs. 6.8%).

X. ACKNOWLEDGEMENTS

This work is supported in part by the European Research Council Advanced Grant DAL No 267175, 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 significant financial interests in AMD, Google, and Panasas. The authors would like to acknowledge Jason Power, Hamid Reza Ghasemi, members of the Multifacet research group, our shepherd Mike O'Connor and our anonymous reviewers for their comments.

REFERENCES

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

[2] A. Seznec, "A case for two-way skewed-associative caches," in Proc. of the 20th annual Intl. Symp. on Computer Architecture, 1993.

[3] A. Seznec, "Concurrent Support of Multiple Page Sizes on a Skewed Associative TLB,"IEEE Transactions on Computers, 2004.

[4] A. Seznec, F. Bodin, ``Skewed-associative caches'', Proceedings of PARLE' 93, Munich, June 1993, also available as INRIA Research Report 1655. http://hal.inria.fr/docs/00/07/49/02/PDF/RR-1655.pdf.

[5] Alaa R. Alameldeen, Milo M. K. Martin, Carl J. Mauer, Kevin E. Moore, Min Xu, Mark D. Hill, David A. Wood, Daniel J. Sorin, "Simulating a $2M Commercial Server on a $2K PC," IEEE Computer, 2003.

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

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

[8] Angelos Arelakis, Per Stenstrom, "SC2: A statistical compression cache scheme," In Proceedings of the 41st Annual International Symposium on Computer Architecture, 2014.

[9] Bulent Abali, Hubertus Franke, Xiaowei Shen, Dan E. Poff, and T. Basil Smith, "Performance of Hardware Compressed Main Memory," In Proceedings of the 7th IEEE Symposium on High-Performance Computer Architecture, 2001.

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

[11] Christian Bienia and Kai Li, "PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors," In Workshop on Modeling, Benchmarking and Simulation, 2009.

[12] Daniel Sanchez, Christos Kozyrakis, "The ZCache: Decoupling Ways and Associativity," in Proceedings of the 43rd annual IEEE/ACM international symposium on Microarchitecture, 2010.

[13] E. Hallnor, S. Reinhardt, "A Unified Compressed Memory Hierarchy," In Proceedings of the 11th International Symposium on High-Performance Computer Architecture, 2005.

[14] Gennady Pekhimenko, Vivek Seshadri, Yoongu Kim, Hongyi Xin, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, Todd C. Mowry, "Linearly Compressed Pages: A Low-Complexity, Low-Latency Main Memory Compression Framework," Annual IEEE/ACM International Symposium on Microarchitecture, 2013.

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

[16] Jang-Soo Lee, Won-Kee Hong, Shin-Dug Kim, "An on-chip cache compression technique to reduce decompression overhead and design complexity," Journal of Systems Architecture, 2000.

[17] Jason Zebchuk, Elham Safi, and Andreas Moshovos, "A Framework for Coarse-Grain Optimizations in the On-Chip Memory Hierarchy," In Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, 2007.

[18] Jeffrey Rothman and Alan Smith, "The Pool of Subsectors Cache Design," International Conference on Supercomputing, 1999.

[19] Julien Dusser, Andre Seznec, "Decoupled Zero-Compressed Memory," In Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers, 2011.

[20] Julien Dusser, Thomas Piquet, André Seznec, "Zero-content augmented caches," In Proceedings of the 23rd international conference on Supercomputing, 2009.

[21] Jun Yang and Rajiv Gupta, "Frequent Value Locality and its Applications," ACM Transactions on Embedded Computing Systems, 2002.

[22] Luis Villa, Michael Zhang, and Krste Asanovic, "Dynamic zero compression for cache energy reduction," In Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture, 2000.

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

[24] M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill, and D. Wood, "Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset," Computer Architecture News, 2005.

[25] Nam Sung Kim, Todd Austin, Trevor Mudge, "Low-Energy Data Cache Using Sign Compression and Cache Line Bisection," Second Annual workshop on Memory Performance Issues, 2002.

[26] omayeh Sardashti and David A. Wood, "Decoupled Compressed Cache: Exploiting Spatial Locality for Energy-Optimized Compressed Caching," Annual IEEE/ACM International Symposium on Microarchitecture, 2013.

[27] Seungcheol Baek, Hyung Gyu Lee, Chrysostomos Nicopoulos, Junghee Lee, and Jongman Kim "ECM: Effective Capacity Maximizer for High-Performance Compressed Caching," In Proceedings of IEEE Symposium on High-Performance Computer Architecture, 2013.

[28] Somayeh Sardashti and David A. Wood, "Decoupled Compressed Cache: Exploiting Spatial Locality for Energy Optimization", in IEEE Micro Top Picks from the 2013 Computer Architecture Conferences.

[29] Soontae Kim, Jesung Kim, Jongmin Lee, Seokin Hong. "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, 2011.

[30] Vishal Aslot, M. Domeika, Rudolf Eigenmann, Greg Gaertner, Wesley B. Jones, and Bodo Parady, "SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance," In Workshop on OpenMP Applications and Tools, 2001.

[31] Xi Chen, Lei Yang, Robert P. Dick, Li Shang, and Haris Lekatsas, "C-pack: a high-performance microprocessor cache compression algorithm," IEEE Transactions on VLSI Systems, 2010.