2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)

Data Locality Exploitation in Cache Compression

Qi Zeng, Rakesh Jha, Shigang Chen, and Jih-Kwon Peir
Department of CISE, University of Florida
Gainesville, Florida, USA
e-mail: {qizeng, rakesh, sgchen, peir}@cise.ufl.edu

***Abstract*--State-of-the-art cache compression methods compress multiple neighboring blocks often called as a sector into a single 64-byte block to effectively enlarge the cache capacity. A compressed block is created by storing 4-byte data patterns as dictionary entries and pointers to them for compressing multiple blocks. Furthermore, sector-based tag array maintains one-to-one mapping between tag and data arrays in order to preserve conventional cache access mechanism. We present a dual-block compression method which uses an entire uncompressed block as dictionary and compresses multiple neighboring blocks in a separate companion block to provide a larger dictionary for better compression ratios. Furthermore, we introduce the concept of buddy-set which expands the compressible candidate blocks across two adjacent cache sets to enlarge the scope of compression. Performance evaluations for the last-level cache show that the proposed dual-block compression with expansion of compressible candidates in the buddy-set can enlarge the cache by an average of 60% while current state-of-art compression proposal has only 29% improvement. The proposed scheme demonstrates 8.9% speedup over caches without compression.**

Keywords — Cache compression, Shared dictionary compression, Data locality, Sector cache.

I. INTRODUCTION

New applications and emerging technologies are driving the advancement of microarchitecture innovations and designs. Emerging big data, cloud computing, and deep learning applications demand bigger storage and present profound challenges to future processors. Caches continue playing a critical role in hiding memory latency and alleviating constraints due to memory bandwidth in modern multicore processors. In this paper, we introduce a new cache compression method which can effectively enlarge the last-level cache capacity without changing conventional cache design and its access mechanism.

It is well-known that memory references exhibit temporal and spatial *address locality*. After a data element is referenced, the data item (temporal) and its nearby data (spatial) tend to be referenced in the near future. In addition, memory references also exhibit *data locality*. A repeated instruction has a good chance to produce data with constant or regular (stride) values in nearby memory locations [1]. Existing dictionary-based cache compression solutions [2][3] capture the repeated data patterns in a dictionary and compress a cache block by using pointers to link to these data patterns.

The traditional approach of compressing multiple neighboring data blocks into a single block with conventional block size (e.g. 64 bytes) maintains the mapping between cache tags and data arrays and helps preserving conventional cache access mechanism. A recent cache compression proposal, dictionary-sharing (DISH) [2], compresses up to four neighboring blocks in a sector and compacts them into a single 64-byte block with shared dictionary of frequent data patterns. However, we discover that confining the compressed blocks and compacting them at 64-byte granularity limits the number of frequent data patterns and lowers the compression ratio. We also realize that data locality exists across neighboring blocks such that the content of one block varies slightly to the content of its neighboring blocks. By using the entire content of a block as dictionary, more repeated data patterns can be recorded for improving the compression ratio. We refer this approach as *dual-block* compression.

We further learn that the compression ratio suffers when the number of compressible candidates is limited. In DISH, for example, only four neighboring blocks are considered as the candidates for compression. In this paper, we introduce a new concept of *buddy* cache sets which are a pair of adjacent cache sets where blocks of adjacent 4-block sectors are allocated. By expanding the compression candidates from 4 sector blocks to 8 blocks across the buddy sets, the compression ratio can further be improved.

Performance studies show that dual-block compression using an entire block as dictionary can achieve 1.43 average compression ratio which effectively enlarges the cache size by 43%. By expanding the compressible blocks in the buddy set, the compression ratio reaches to 1.60, a 60% cache enlargement. In comparison with the state-of-the-art DISH compression method, the proposed schemes improve the compression ratios by 11% and 24% respectively. We also evaluate the speedups and results show that about 5.6% and 8.9% average speedups can be achieved comparing with caches without compression.

This paper makes three key contributions. First, to the best of our knowledge, this is the first work to explore block content locality among neighboring blocks and to use an uncompressed block as the dictionary for cache compression. Second, we recognize the importance to enlarge the scope of finding compressible blocks and compact them into a single 64-byte block. We introduce the buddy sets concept and include blocks from adjacent cache set as the candidates for compression. Third, performance evaluation shows significant improvement of the

978-1-5386-7308-9/18/$31.00 ©2018 IEEE
DOI 10.1109/ICPADS.2018.00054

IEEE computer society

compression ratio of our approach over the state-of-the-art DISH compression method. Although our studies focus on fundamental locality behavior on data content, it is applicable to many-core parallel systems since the target of compression is for the shared last-level cache.

The paper is organized as follows. We first present the motivation and show content similarity among neighboring cache blocks in section 2. Section 3 describes the proposed dual-block compression and compaction schemes. We also demonstrate a compression example using real block contents extracted from soplex. Section 4 provides the performance evaluation methodology. Performance results are given in Section 5. This is followed by the related work in Section 6 and conclusion in Section 7.

II. Motivation using Block Content Similarity

Cache compression has been researched in recent years [2][4][5][6][7][8][9][10][11]. Besides a high compression ratio, modern approaches can access a compressed cache the same way as accessing an uncompressed cache with the following three key features:

- To maintain one-to-one mapping of cache tag and data, each tag in the tag array is associated with a corresponding 64-byte data block in the data array, either compressed or uncompressed.
- To save tag space and exploit content-similarity of neighboring blocks, sector-cache design [12][13] has been adopted. Four blocks in a sector are allocated in the same cache set as the target for compression.
- To adopt frequent pattern based compression, each compressed 64-byte block consists of several 4-byte data patterns (as the dictionary) and uses remaining space to record up to four pointer groups corresponding to each compressed original block, where pointers in each group link to the dictionary entries.

One recent approach, DISH [2], demonstrates the above features and achieves decent compression ratios. DISH records up to eight 4-byte data in the dictionary and uses the remaining space to record pointer groups for up to four 64-byte blocks. Each compressed block requires sixteen 3-bit pointers where each pointer refers to a 4-byte data entry in the dictionary. Sector-based tag array records the sector tags with individual block IDs to determine cache hit/miss. Each sector has 4 consecutive 64-byte blocks allocated in the same cache set.

Based on cache block contents from SPECCPU 2006 benchmarks [14], we observe two fundamental shortcomings in the DISH compression approach as described below. We also present innovative solutions to overcome such shortcomings.

- First, we observe that many benchmark programs contain a large number of common data patterns (referred as *keys*) in the dictionary, the combined size of which may be beyond 64-byte capacity for compressing blocks in a sector. We also observe that these workloads reveal high inter-block content similarity such that the contents of neighboring blocks differ only by a small number of keys.

  In Table 1, we pick four snapshot examples from libquantum, mcf, perlbench, and soplex to demonstrate the existence of such inter-block content similarity among four sector blocks. For example, in soplex, the four sector blocks consist of 10, 9, 10, and 10 4-byte keys respectively and cannot be compressed. However, if we select the first block (*bf80*) as the base, the other three blocks have only 2, 3, and 3 different keys (marked in bold). Such behavior motivates us to pick one uncompressed block as the *master* block and compress other content-similar blocks in a separate *companion* block. The companion block records a few different keys and pointer groups to link to 16 4-byte data in the master block as well as the keys in the companion block. We call this approach as a *dual-block* compression, expanding the keys in two blocks.
- Second, we observe that spatial locality among blocks in a sector is weak in some benchmark programs. As a result, not all blocks in a sector are present in cache during the life time of a sector, limiting the blocks that can be compressed. It is challenging to increase the sector size and allocate all sector blocks in the same cache set since it can cause thrashing among sectors and degrade cache performance. We present a novel idea which uses a pair of adjacent cache sets as *buddy sets*. The least-significant index bits of the buddy sets are complement of each other. In the dual-block compression described previously, the master block and its companion block can come from two adjacent 4-block sectors located in a pair of buddy sets. Furthermore, each master block can serve more than one companion block located in the buddy sets. Compressing blocks across two adjacent sectors doubles the neighboring compression candidates without altering cache placement of sector blocks.

We explore block content similarity behavior by measuring the number of 4-byte shared keys required to cover the contents of blocks in a sector. This experiment is performed using the GEM5 whole system simulation tool [15] running SPECCPU 2006 benchmarks. Detailed description of the simulation environment will be given in Section 4. For a high compression ratio, the number distinct keys must be limited. In DISH, for example, when the number of keys is beyond 8, it cannot compress the cached blocks in a sector into a single 64-byte block.

In Figure 1, we separate the sectors into 4 groups based on the average number of keys: 1-8, 9-16, 17-24, and 25-64 measured during the life time of a sector. We can observe that although the percentages of 1-8 keys are low in several benchmarks, the percentages of 9-16 and 17-24 keys are high in these benchmarks. Therefore, the compression ratio can be improved by increasing the number of sharing keys. These results provide justification to use the dual-block compression to supply more keys.

In the second experiment, we measure the average number of cached blocks in a sector. As observed in Figure 2, astar, gobmk, gcc, GemsFDTD, omnetpp, perlbench show poor spatial locality with an average of 1.6 – 2.8 blocks presented in cache during the life time of a 4-block sector, which limits the candidate blocks for compression. In this paper, we expand candidate blocks in adjacent sectors located in a pair of buddy sets to take advantage of the proposed dual-block compression.

III. New Dual-block Compression

To simplify our presentation without loss of generality, we introduce our proposal on an 8MB, 16-way sector-based LLC with 64-byte blocks. A conventional sector cache design is used with 4 blocks per sector [12]. The physical address has 48 bits. The addressing scheme and the tag array layout are given in Figure 3. Each tag entry consists of a 3-bit function code (FC), a 29-bit sector tag, 4 valid bits for presence of sector blocks, and 4 coherence states (CS) for the 4 blocks. The tag entry for a companion block records individual block ID and the coherence state for three compressed blocks. The definition of the FC and the content of a companion block will be given later.

3.1. Dual-Block Compression

Dual-block compression is attempted when multiple blocks in the same sector are not compressible into a single block. When there are three uncompressed sector blocks, dual-block compression is tested to select a master block and two other blocks which are compressible in a separate companion block using shared keys in both blocks. Since the master block is in an uncompressed form, we select the one which contains the most keys. With 4-block sectors, a 3-block companion can achieve a compression ratio of 2. With a 2-block companion, the ratio is 1.5.

TABLE 1. EXAMPLES OF 4-BYTE HEX CONTENTS IN 4-BLOCK SECTORS FROM 4 WORKLOADS (***libquantum*, *mcf*, *perlbench*** and ***soplex***).

4-byte	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
libquantum																
(92214)	**3a3504e8**	**0**	**01004008**	**001b2278**	3a3504e8	0	**01000000**	**001b2280**	3a3504e8	0	01004008	**001b2288**	3a3504e8	0	01000000	**001b2290**
(92215)	3a3504e8	0	01004008	**001b2298**	3a3504e8	0	01000000	**001b22a0**	3a3504e8	0	01004008	**001b22a8**	3a3504e8	0	01000000	**001b22b0**
(92216)	3a3504e8	0	01004008	**001b22b8**	3a3504e8	0	01000000	**001b22c0**	3a3504e8	0	01004008	**001b22c8**	3a3504e8	0	01000000	**001b22d0**
(92217)	3a3504e8	0	01004008	**001b22d8**	3a3504e8	0	01000000	**001b22e0**	3a3504e8	0	01004008	**001b22e8**	3a3504e8	0	01000000	**001b22f0**
mcf																
(8d397c)	**0**	0	**1e**	0	1e	0	**003ac708**	0	**00155a70**	0	**1**	0	**23313410**	0	**2342fed0**	0
(8d397d)	0	0	1e	0	1e	0	**003ac638**	0	00155a70	0	1	0	**233c3a10**	0	**2342ff10**	0
(8d397e)	0	0	1e	0	1e	0	**003ac3c8**	0	00155a70	0	1	0	**233ddbd0**	0	**2342ff50**	0
(8d397f)	0	0	1e	0	1e	0	**003ac228**	0	00155a70	0	1	0	**233ee450**	0	**2342ff90**	0
perlbench																
(60604)	**0**	0	**450ba8b6**	0	**2030fbe8**	**1**	**6**	0	0	0	**aefcbc10**	0	**20f182e8**	1	**c**	0
(60605)	0	0	**ef94b794**	0	**20416008**	1	**4**	0	0	0	450ba8b6	0	2030fbe8	1	6	0
(60606)	0	0	**aefcbc10**	0	**20c49c78**	1	**12**	0	0	0	**21534396**	0	20416008	1	4	0
(60607)	0	0	**450ba8b6**	0	2030fbe8	1	6	0	0	0	aefcbc10	0	210a2fa8	1	12	0
soplex																
(bf80)	**0**	**bff00000**	**00001bb5**	**00000b00**	0	**40000000**	**2**	0	0	**3ff00000**	**0000375e**	**00000700**	0	bff00000	**0000372c**	00000b00
(bf81)	0	40000000	2	0	0	3ff00000	**000052d5**	00000700	0	bff00000	**000052a3**	00000b00	0	40000000	2	0
(bf82)	0	3ff00000	**00006e4c**	00000700	0	bff00000	**00006e1a**	00000b00	0	40000000	2	0	0	3ff00000	**000089c3**	00000700
(bf83)	0	bff00000	**00008991**	00000b00	0	40000000	2	0	0	3ff00000	**0000a53a**	00000700	0	bff00000	**0000a508**	00000b00

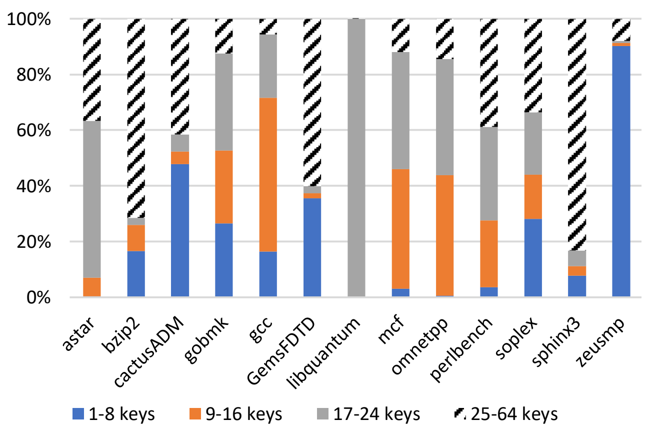

Figure 1. Distribution of sectors on number of keys

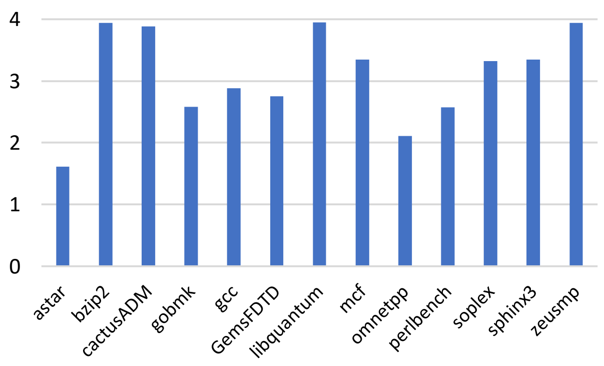

Figure 2. Average cached blocks in a 4-block sector

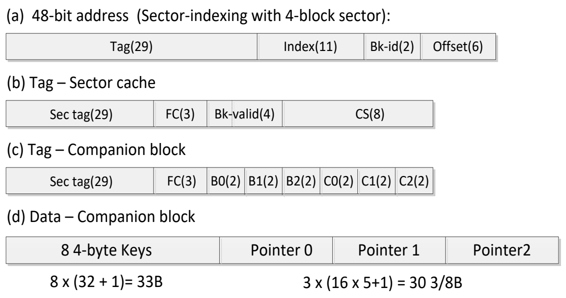

Figure 3. (a) 48-bit address; (b) tag for 4-block sector cache; (c) tag for companion block; (d) data for companion block

The detailed bit layout of a companion block is given in Figure 3(d), in which we can record 8 different keys and use them as extra dictionary for the companion block. Up to 3 pointer groups are used to compress 3 blocks, which is sufficient to cover the remaining 3 blocks in a sector. Since there are a total of 24 keys (16 in the master block and 8 in the companion block), 16 pointers of 5 bits each are needed for each pointer group to record a compressed block. The total companion block size is 63 and 3/8 bytes. Note that for the companion block, as shown in Figure 3(c), three block IDs and their coherence states are encoded in the tag array, thus free from recording as a part of the pointer groups in the companion block. When a master block is replaced from the cache, its companion block must also be invalidated. Therefore, when a companion block is referenced, both the master and the companion are promoted to the MRU positions.

3.2. Buddy Sets

Buddy sets are two adjacent cache sets, whose indices are identical except for the least-significant bit being 1 and 0 respectively. For simplicity, consider a 4-block sector cache with only 6 index bits in 10-bit block address in Figure 3(a). Two block addresses, *aa-000000-xx* and *aa-000001-yy*, represent block *xx* of sector *aa* in set *000000* and block *yy* of sector *aa* in set *000001* respectively. These two sets, indexed by *000000* and *000001*, are formed the buddy sets. Sectors *aa* in both sets are two consecutive 4-block sectors located in the pair of buddy sets, henceforth referred as *buddy sectors*. Note that the buddy sectors have identical tags located in the buddy sets.

To enlarge the compressible blocks beyond blocks in a 4-block sector, we can double the candidates from both buddy sectors. When a missed block is uncompressible in the original cache set, a search in the buddy set is carried out if another uncompressible sector block already exists in the original set. All blocks in the buddy sectors are tested for dual-block compression. If successful, the block with the most keys among all blocks in the buddy sectors is selected as master block. The remaining blocks are compressed into a companion block. The master block can be in one set, while two or more blocks from the buddy sectors can be in a companion block placed in the other set.

Each master can serve one or more companions. When searching for dual-block compression, an existing master may be found for serving the new companion. Therefore, ideally, a master block can serve two 3-block companions, one in each set to achieve a compression ratio of 7/3=2.33. A master block can also serve one 3-block companion in the same set and two other 2-block companions in the buddy set to achieve the compression ratio of 8/4=2.

3.3. Putting It All Together

With different compression options, a 3-bit function code (FC) is associated with each tag in the sector tag array. The definition of FC is given in Table 2 where five FCs are defined for master and companion block identification as well as the locations of their counterparts.

TABLE 2. FUNCTION CODE DEFINITION

Function Code (FC)	Description
000	Uncompressed
001	64-byte compression
010	Master, companion in same set
011	Master, companion in buddy set
100	Master, companions in both sets
101	Companion, master in same set
110	Companion, master in buddy set

During cache access, in case of a tag match, the corresponding data block is fetched from the data array. For an uncompressed block with FC = 000, 010, 011, or 100, the data block is accessed normally. For single compressed block with FC = 001, the data block is fetched and decompressed using the correct pointer group and the keys in the dictionary. In case that the hit is to a companion block, the master block must be fetched. The master block has the same tag as the companion block and located either in the same set, FC=101, or in the buddy set, FC=110. The data block is then decompressed using the keys in both blocks and the correct pointers in the companion block.

In summary, when a cache miss occurs, a new block is moved into cache. The first attempt is to compress it with an existing compressed block either a single compressed block or a companion block. If unsuccessful, DISH-like search to find compressible blocks in another sector and tries to compress them into a single 64-byte block. In case that this still fails, dual-block compression is tested, first in the same set, and then in the buddy set. The missed block is allocated into cache as an uncompressed block to replace the LRU block if none of the above attempts is successful.

Upon receiving a data writeback that updates a compressed block, the compressed block could become uncompressible. When it happens, we need to invalidate the single compressed block or the companion block and allocate new cache frames in the set for holding the decompressed blocks. If the writeback is performed at a master block, we need check if the companion block can still be compressed with the updated master using the same compression process. Note that decompression and re-compaction in response to cache writebacks is a necessary step for cache compression methods. Fortunately, a dirty block writeback is not on the performance critical path.

3.4. Example

In this section, we use a snapshot of four block contents in a sector taken from ***soplex*** to illustrate how dual-block compression works. As shown in Table 1, four blocks in a sector with block addresses: *bf80, bf81, bf82,* and *bf83* have 10, 9, 10, and 10 distinct keys. No pair of blocks can be compressed into a 64-byte block using the DISH compression scheme. However, if we select the first block *bf80* which has the highest number of keys as the master block, the other three blocks can be compressed in a companion block since only 2, 3, and 3 additional keys are required for these three blocks. Figure 5 shows the content of the uncompressed master block and the compressed companion block. Each pointer group has 16 5-bit pointers to link either to the 16 4-byte dictionary in the master block, or to the 8 4-byte dictionary in the companion block. We use the most significant bit to differentiate the keys in the master ('0') or in the companion ('1').

We further examine the buddy sector content in this example to illustrate the benefit of expanding compression candidates in the buddy set. As shown in Table 3, four blocks, *bf84, bf85, bf86,* and *bf87*, in the buddy sector have 9, 10, 10, and 9 distinct keys. Again, no pair of the blocks can be compressed into a single 64-byte block. When any two blocks arrive in cache, a search in the buddy set may find the existing master block *bf80.* This master block can serve both the companion block containing *bf81, bf82,* and *bf83* in the same set as well as another companion block consisting of any three blocks of *bf84, bf85, bf86,* and *bf87* in the buddy set. Given *bf80* as the master, the four blocks in the buddy sector require additional 2, 3, 3, and 2 extra keys, as are marked bold in Table 3. Since the companion block provides three pointer groups, only three blocks can be compacted into the companion block with additional 7 or 8 keys. Therefore, seven blocks can be compressed and compacted into three blocks, with one master and two companions in this real example.

The function code in the tag directory differentiates the master and the companion blocks (Table 2). In the above example, let's assume that the master *bf80* and three sector blocks *bf81, bf82*, and *bf83* arrive in cache first and satisfy dual-block compression. Both the master and the companion are in the same set with identical address tag. They differ only in the block ids. The master and the companion block are coded FC=010 and FC=101 indicating that the respective master and companion blocks are located in the same set. When the companion block in the buddy set is compressed successfully, the FC for the master is changed to 100 indicating existence of two companions located in both sets. In this case, the companion in the buddy set is coded FC=110 denoting where the master is.

Upon a hit to a companion, proper FC provides where to fetch the master and to adjust master's replacement position. Similarly, with proper FC, the master knows where to invalidate the companion when master is evicted. It is important to note that the master and the companions always have identical address tag regardless whether they are in the same set or in the buddy sets. There is no need for extra links between the master and the companions.

3.5. Overhead Analysis

With one-to-one tag and data mapping, only 3-bit function code is added and there is no other storage overhead for storing the compression related meta-data. We estimate the area and latency overhead using CACTI 6.5 [16]. With a 32-nm technology, for a 4MB cache that occupies $34mm^2$ (both tag and data array), the area used by the compressor and de-compressor is less than $0.7mm^2$, which counts for only 2% extra in terms of space overhead.

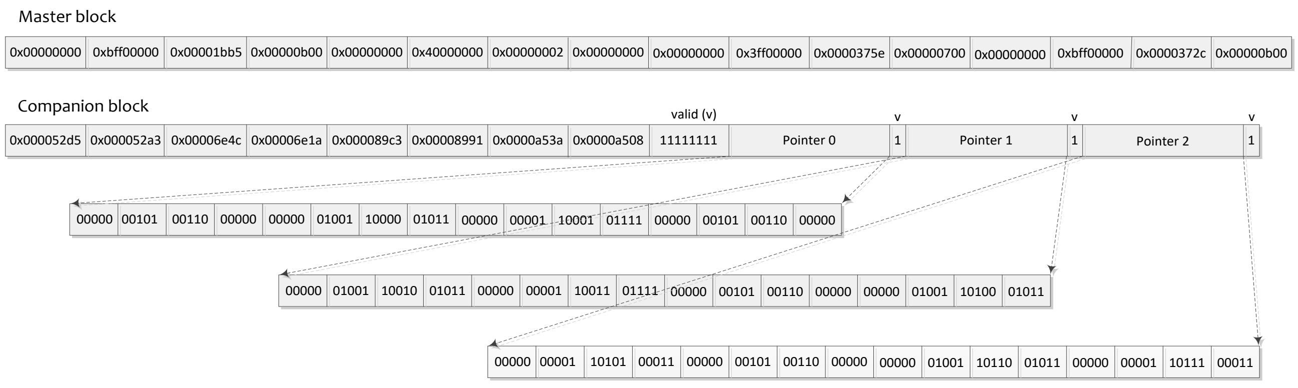

Figure 5. Dual-block compression for 4-block sector in ***soplex***

TABLE 3. BUDDY SECTOR CONTENT FROM ***soplex***

4-byte	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
(bf80) *master*	**0**	**bff00000**	**00001bb5**	**00000b00**	0	**40000000**	**2**	0	0	**3ff00000**	**0000375e**	**00000700**	0	bff00000	**0000372c**	00000b00
(bf84)	0	40000000	2	0	0	3ff00000	***0000c0b1***	00000700	0	bff00000	***0000c07f***	00000b00	0	40000000	2	0
(bf85)	0	3ff00000	***0000dc28***	00000700	0	bff00000	***0000dbf6***	00000b00	0	40000000	2	0	0	3ff00000	***0000f79f***	00000700
(bf86)	0	bff00000	***0000f76d***	00000b00	0	40000000	2	0	0	3ff00000	***00011316***	00000700	0	bff00000	***01120000***	00000b00
(bf87)	0	40000000	2	0	0	3ff00000	***73***	00000700	0	bff00000	***3***	00000b00	0	40000000	2	0

According to CACTI, a read or write to 4MB SRAM cache with 32-nm technology takes 40 cycles. The main latency overhead is from accessing two blocks during a hit to a compressed companion block. If both the master and the companion blocks are in the same set, the normal tag array search can locate both the companion block and its master. Fetching the companion and master blocks can be overlapped with multiple banks in the data array. We estimate it takes 1 and 25 cycles for decompression and compression, respectively. In the worst case when the master is from the buddy set, an immediate accessing to the master block in the buddy set is triggered. It takes 2 cycles in decompression as a result and the compression takes 27 cycles due to buddy set operations. This compression process is off the cache access critical path. Upon a L3 miss, data from DRAM is sent back to L1/L2 to satisfy CPU requests before compression operations. Obviously, there is no extra latency for uncompressed master block.

IV. EVALUATION METHODOLOGY

Gem5 [15], a cycle accurate system simulator, is employed to evaluate different compression schemes. The cache model of Gem5 is modified to support the compression and decompression with extra latency. The processor model is configured as a processor with a three-level cache hierarchy. Cache compression techniques are applied to the third and last level cache. The architecture parameters of the CPU and memory are in Table 4.

TABLE 4. PARAMETERS OF THE SIMULATED SYSTEM

Components	Configurations
Processor	3GHz, out-of-order, 8-issue
L1 D/I	private, 32 KB each, 4-way, 64B blocks
L2, L3	private 256 KB, 8-way L2, and shared 4MB, 16-way, non-inclusive L3
MSHRS	16, 16, 32 MSHRs at L1, L2, L3
DRAM	8GB, DDR4-2400, 64bit I/O, 8 banks, 2KB row buffer tCL-tRCD-tRP-tWR: 13-13-13-14
DISH	8 4-byte keys, 4 16×3-bit pointer arrays
Companion	8 4-byte keys, 3 16×5-bit pointer arrays

The cache compression helps retaining more blocks in cache, which is more useful for applications with high MPKI and sensitive to the cache size. From SPECCPU 2006 [14] benchmark, we test and select workloads with relatively high MPKI, along with gobmk and perlbench for comparison purpose. These two applications demonstrate interesting inter-block locality. A set of 13 benchmarks are used in our experiment. To locate the most representative phase of whole program execution, we use *SimPoint* [17]. We collect statistics for 500M instructions after a warm-up of 1B instructions from the checkpoints. Without the help from *SimPoint*, DISH simulated a different phase of the programs by fast forward 20 billion instructions for each application. As a result, their compression performance is different from what we obtained using *Simpoint.*

We also notice that L2 prefetcher is used in modern cache design. Stream-based prefetcher [18] can bring in more neighboring blocks and help the compression ratio. However, in Figure 2, for workloads such as astar and omnetpp with weak spatial locality, the prefetcher may fetch useless neighbors and inflate the compression ratio. So, we exclude the prefetcher from our simulation model.

V. PERFORMANCE RESULTS

5.1. Compression Ratio

We first compare the compression ratio, which is defined as the effective cache size compared to a cache without compression. We calculate the compression ratio at the set level and take the average across all the cache sets [2]. Figure 6 shows the compression ratios achieved by compression with buddy-set (*Buddy-set*), dual-block for home set only (*Dual-block*), and DISH. The compression ratios are 1.60, 1.43, and 1.29 respectively for the three schemes. Buddy-set and Dual-block, show about 24%, and 11% improvement over DISH.

The compression ratio varies quite significantly among workloads using three compression schemes. Astar, gcc, libquantum, mcf, omnetpp, perlbench, soplex have substantial improvement from Buddy-set and for these workloads, the effective cache size improvements are ranging 25-86% over that of DISH. At the same time, we notice that some of these workloads, such as astar, libquantum, mcf, omnetpp, and perlbench, have poor compression ratios for DISH due to lack of sufficient space for 9 or more keys to be recorded in a single block. Zeusmp and cactusADM show high compression ratios (>2) with DISH alone by efficiently compacting multiple blocks into one with only 8 keys, so there is no much room for improvement for our proposed scheme.

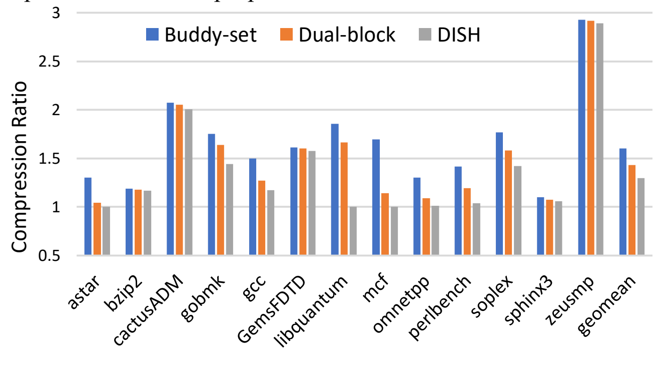

Figure 6. Compression ratios for different schemes

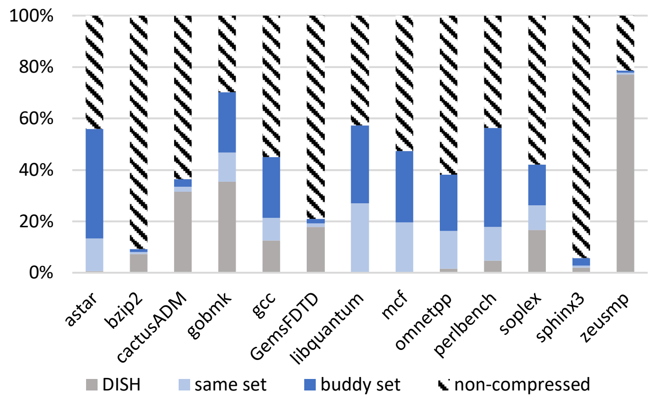

Figure 7. Distribution of techniques used in compression

For Buddy-set based algorithm, the improvement also comes from additional compressible blocks in the same set or from the buddy set, while the Dual-block scheme can benefit only from compressible ones within same set. Figure 7 shows the distribution of the compression techniques when Buddy-set is used. As expected, workloads with significant amount of dual-block compressions show higher compression ratios. Dual-block has little impact on workloads which is dominated by single 64-byte compression and/or uncompressed blocks, such as bzip2, cactusADM, GemsFDTD and zeusmp.

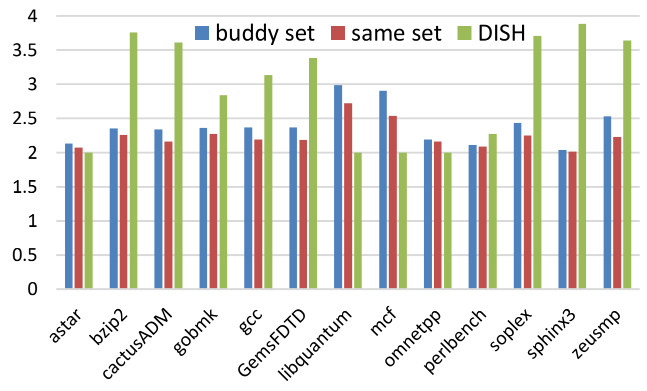

Figure 8. Average number of blocks compacted in a block

We collect the average number of blocks compressed into a single 64-block or compressed into a companion block. As shown in Figure 8, the average number of blocks is ranging from 2 to 3.9. Workloads such as astar, libquantum, mcf, and omnetpp display inefficient single 64-byte block compression with an average close to two out of four blocks. For companion blocks, libquantum and mcf have an average of 3 and 2.9 respectively, which are close to the maximum 3 compressed blocks using Buddy-set and show higher compression ratio in Figure 6 too. Among all other workloads, the average compacted blocks range from 2.2 to 2.5 per companion block.

5.2. Speedup

Next, we measure speedups of execution time by effectively increasing the LLC size with different cache compression schemes. Figure 9 shows that the average speedups of 8.9%, 5.6% and 3.6% can be achieved for Buddy-set, Dual-block, and DISH respectively, normalized to an uncompressed baseline 4MB cache. Again, the speedup varies widely from different workloads and different compression methods. Astar and soplex display the highest speedup about 34-39% for Buddy-set compression, zeusmp has about 18%, while most others only experience 1-9% improvement. Besides, DISH is especially worse since many workloads have insignificant improvement on compression ratios. Mcf shows negative improvement due to overheads from compression and decompression.

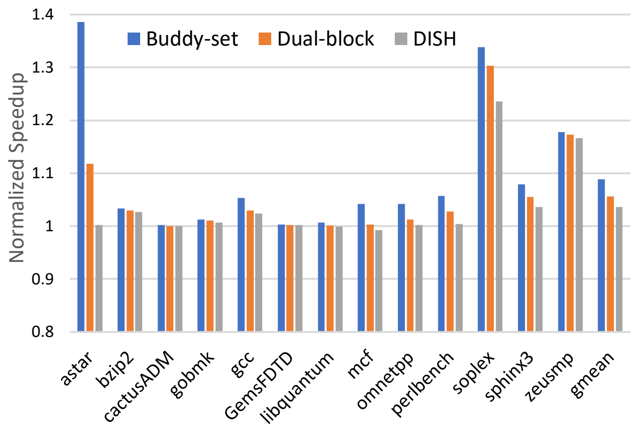

Figure 9. Speedup over a 4MB LLC

VI. Related Work

IBM MXT [19] uses real-time main memory compression to effectively double the main memory capacity. Another efficient memory compaction scheme is also reported [20] which handles zero value compaction as well as avoids indirect memory mappings. X-Match [20] is a dictionary-based compression algorithm. It uses content-addressable memory and allows partial matching with dictionary entries for variable-size encoded data. C-Pack [4] uses a single shared dictionary of 64 bytes for cache compression. It compresses frequently appearing words into few bits. It can pack multiple compressed blocks into a single uncompressed block frame. CPACK+Z [10] is a variation of C-PACK with feature that detects zero blocks.

Prior work has proposed compression algorithms for last-level cache (LLC) with variable block sizes [7]. Instead of recording frequent data patterns, Base-Delta-Immediate [21] uses one base value for a cache block, and replaces the other data values of the block in terms of their respective delta (differences) from the base value. It compresses a cache block by exploiting the data value correlation property with minimum decompression latency.

Decoupled Compressed Cache (DCC) [10] proposes tracking compressed blocks at super-block (sector) level. DCC compresses each 64-byte block into zero to four 16-byte sub-blocks and uses a decoupled tag-data mapping [13] to allow blocks to be stored anywhere in a cache set. Skewed

Compressed Cache (SCC) [5] stores neighboring compressed blocks in a power-of-two number of 8-byte sub-blocks in a compressed 64-byte block. It only allows the blocks with the same compressibility to be compressed into a 64-byte block. It uses sparse super-block tags and a skewed associative mapping [22] that preserves a one-to-one direct mapping between tags and data.

Yet Another Compressed Cache (YACC) [6] is similar to SCC while eliminates skewing. It uses sector tag array and an unmodified data array with direct tag and data mapping. YACC allows the use of any replacement policy and is able to store neighboring blocks in one data entry if they have similar compressibility to fit into a single data block. The latest Dictionary Sharing (DISH) is similar to SCC. It can compress multiple sector blocks into a single 64-byte block. Each compressed 64-byte block records up to eight 4-byte frequent data patterns in the dictionary. It uses the remaining space to build 4 pointer groups pointing to the 4-byte data blocks in the dictionary.

VII. Conclusion

We propose an efficient cache compression method which compresses and compacts neighboring blocks into a single 64-byte cache block. As in other compression approaches, it records multiple 4-byte data (keys) in a dictionary and uses pointers to compress multiple cache blocks in a sector. However, it is different from other approaches as it proposes scheme that can use two blocks for compaction when the number of distinct dictionary keys are too big to be compacted into a single block. This dual-block compression uses one uncompressed block as the master and compacts other sector blocks in a separate companion block. Furthermore, we propose the idea of buddy sets where two adjacent sectors are allocated. By expanding the compressible candidates across the buddy sets, it helps the compression ratio. The performance evaluation using SPECCPU 2006 workloads demonstrates the proposed compression method can effectively increase the cache capacity by 60%.

Acknowledgement

The authors would like to acknowledge anonymous reviewers for their constructive comments. This work is supported in part by NSF project CNS-1719222.

References

[1] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value locality and load value prediction," *Proc. seventh Int. Conf. Archit. Support Program. Lang. Oper. Syst. - ASPLOS-VII*, no. October, pp. 138–147, 1996.

[2] B. Panda and A. Seznec, "Dictionary sharing: An efficient cache compression scheme for compressed caches," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.

[3] Jun Yang, Youtao Zhang, and R. Gupta, "Frequent value compression in data caches," in *Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000*, no. Cc, pp. 258–265.

[4] X. Chen, L. Yang, R. P. Dick, L. Shang, and H. Lekatsas, "C-pack: A high-performance microprocessor cache compression algorithm," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 18, no. 8, pp. 1196–1208, 2010.

[5] S. Sardashti, A. Seznec, and D. a. Wood, "Skewed Compressed Caches," *2014 47th Annu. IEEE/ACM Int. Symp. Microarchitecture*, pp. 331–342, 2014.

[6] S. Sardashti, A. Seznec, and D. A. Wood, "Yet Another Compressed Cache," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 3, pp. 1–25, Sep. 2016.

[7] A. R. Alameldeen and D. A. Wood, "Adaptive Cache Compression for High-Performance Processors," *ACM SIGARCH Comput. Archit. News*, vol. 32, no. 2, p. 212, Mar. 2004.

[8] A. Arelakis and P. Stenstrom, "SC2: A statistical compression cache scheme," *Proc. - Int. Symp. Comput. Archit.*, pp. 145–156, 2014.

[9] A. Arelakis, F. Dahlgren, and P. Stenstrom, "HyComp: A Hybrid Cache Compression Method for Selection of Data-type-specific Compression Methods," *Proc. 48th Int. Symp. Microarchitecture*, pp. 38–49, 2015.

[10] S. Sardashti and D. A. Wood, "Decoupled compressed cache," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO-46*, 2013, pp. 62–73.

[11] G. Pekhimenko, T. Huberty, R. Cai, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Exploiting compressed block size as an indicator of future reuse," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 51–63.

[12] J. S. Liptay, "Structural aspects of the System/360 Model 85, II: The cache," *IBM Syst. J.*, vol. 7, no. 1, pp. 15–21, 1968.

[13] A. Seznec, "Decoupled sectored caches: conciliating low tag implementation cost and low miss ratio," *Proc. 21 Int. Symp. Comput. Archit.*, pp. 384–393, 1994.

[14] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, 2006.

[15] N. Binkert, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. Hill, and D. Wood, "The gem5 Simulator," *Comput. Archit. News*, vol. 39, no. 2, p. 1, 2011.

[16] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A Tool to Model Large Caches," *Symp. A Q. J. Mod. Foreign Lit.*, no. HPL-2009-85, pp. 0–24, 2009.

[17] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," *ACM SIGOPS Oper. Syst. Rev.*, vol. 36, no. 5, p. 45, 2002.

[18] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," *Proc. - Int. Symp. High-Performance Comput. Archit.*, pp. 63–74, 2007.

[19] R. B. Tremaine, P. a. Franaszek, J. T. Robinson, C. O. Schulz, T. B. Smith, M. E. Wazlowski, and P. M. Bland, "IBM Memory Expansion Technology (MXT)," *IBM J. Res. Dev.*, vol. 45, no. 2, pp. 271–285, 2001.

[20] J. L. Núñez and S. Jones, "Gbit/s lossless data compression hardware," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 11, no. 3, pp. 499–510, 2003.

[21] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-delta-immediate compression," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques - PACT '12*, 2012, p. 377.

[22] A. Seznec, "A case for two-way skewed-associative caches," in *Proceedings of the 20th annual international symposium on Computer architecture - ISCA '93*, 1993, vol. 21, no. 2, pp. 169–178.