Dictionary Sharing: An Efficient Cache Compression Scheme for Compressed Caches

Biswabandan Panda, André Seznec

Dictionary Sharing: An Efficient Cache Compression Scheme for Compressed Caches

Biswabandan Panda
INRIA
Campus de Beaulieu, Rennes, France
Email: biswabandan.panda@inria.fr

André Seznec
INRIA
Campus de Beaulieu, Rennes, France
Email: andre.seznec@inria.fr

***Abstract*—The effectiveness of a compressed cache depends on three features: i) the compression scheme, ii) the compaction scheme, and iii) the cache layout of the compressed cache. Skewed compressed cache (SCC) and yet another compressed cache (YACC) are two recently proposed compressed cache layouts that feature minimal storage and latency overheads, while achieving comparable performance over more complex compressed cache layouts. Both SCC and YACC use compression techniques to compress individual cache blocks, and then a compaction technique to compact multiple contiguous compressed blocks into a single data entry. The primary attribute used by these techniques for compaction is the compression factor of the cache blocks, and in this process, they waste cache space.**

In this paper, we propose dictionary sharing (DISH), a dictionary based cache compression scheme that reduces this wastage. DISH compresses a cache block by keeping in mind that the block is a potential candidate for the compaction process. DISH encodes a cache block with a dictionary that stores the distinct 4-byte chunks of a cache block and the dictionary is shared among multiple neighboring cache blocks. The simple encoding scheme of DISH also provides a single cycle decompression latency and it does not change the cache layout of compressed caches.

Compressed cache layouts that use DISH outperforms the compression schemes, such as BDI and CPACK+Z, in terms of compression ratio (from 1.7X to 2.3X), system performance (from 7.2% to 14.3%), and energy efficiency (from 6% to 16%).

I. INTRODUCTION

Effective management of last-level-cache (LLC) capacity is key to system performance. Increase in the cache capacity improves system performance but at the cost of energy and power. Compressed caches such as decoupled compressed cache (DCC) [1], skewed compressed cache (SCC) [2], and yet another compressed cache (YACC) [3] use compression and compaction techniques to improve the cache capacity, with marginal overhead in hardware, energy, and power. When a cache block is responded from the DRAM, a compression technique compresses the cache block and then a compaction technique takes multiple compressed blocks that are compressed independently, and compacts them to fit in a fixed cache space (a data entry)[1]. A compressed cache also provides a cache layout [2] that helps in accessing physical data entries with permissible latencies.

State-of-the-art compressed caches provide physical layouts that address some of the fundamental issues of compressed caches, which are: (i) mapping of tag and data entries, (ii) low tag and meta-data overhead, and (iii) supporting flexible cache replacement policies. Among the recent works, YACC is the simplest compressed cache that has addressed the above issues, and is equally effective as SCC. It simplifies the cache layout of SCC without sacrificing the benefits of SCC. Also the cache layout is agnostic to the compression techniques. *Note that we use the term compressed cache for techniques, such as DCC, SCC, and YACC that provide both compression and compaction (and not only compression).*

Compressed caches, such as SCC and YACC, use the notion of super-block, which is an *aligned and contiguous* group of blocks (compressed and uncompressed), and use a single tag called *super-block tag* for all the blocks that are part of the super-block. For example, a 4-block super-block that contains 4 cache blocks share a single super-block tag. The tag entry of a super-block, also contains additional information about the compression factor (CF) of the blocks that are part of the super-block. With YACC and SCC, for a data entry of 64 bytes, CF of a cache block is: four if it is compressed to 16 or less than 16 bytes (4-compressible), two if it is compressed to a size between 16 to 32 bytes (2-compressible), and one if it is incompressible or compressible to more than 32 bytes.

The Problem. As SCC and YACC use CF as the main attribute for compaction, *on a demand miss at the LLC, these techniques either compress and compact the new cache block with one of the cache blocks of an incomplete super-block that has the same CF or it allocates a new 64B data entry (if the block is not compactable with its neighbors).* This potentially leads to wastage of LLC space. *One of the reasons for this wastage is that compression techniques, when compress cache blocks are oblivious to the compaction process.* There are three major cases where SCC and YACC waste LLC space. Fig. 1 shows the three cases in which super-blocks waste one data entry because of CF based compaction. For the three cases (see Fig. 1), YACC uses 3, 4, and 3 64B data entries, whereas the compressed blocks could fit in 2, 3, and 2 data entries, respectively. For example, in case-1, a 32B compressed cache block that is allocated in data entry 2 (B2) can be compacted with B0 and B1 in the 32B that is left in data entry 1,

[1]A data entry (usually of size 64B) of a cache is the fundamental unit of the data store that contains cache block(s). In an uncompressed cache, it stores only one cache block. However, in a compressed cache, a data entry can contain multiple compressed cache blocks.

[2]Layout of tag/data store and the mapping of a tag entry with a data entry.

978-1-5090-3508-3/16/$31.00 2016 European Union

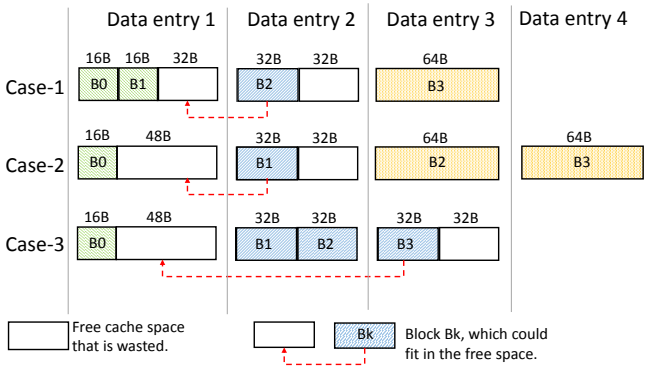

Fig. 1: Wastage of cache space (in terms of 64B data entries) in SCC and YACC layouts because of CF based compaction. B0 to B3 are the contiguous cache blocks of a 4-block super-block.

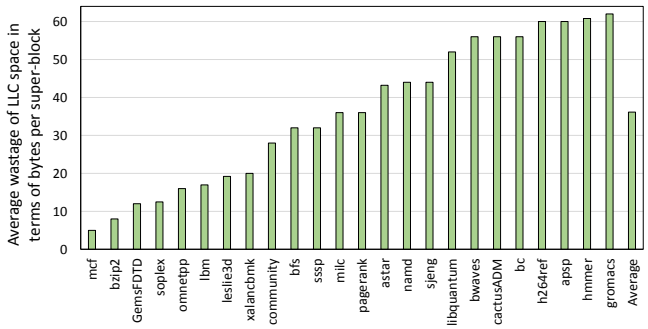

Fig. 2: Wastage of LLC space.

potentially saving a data entry (in this case data entry 2).

Motivation. Fig. 2 shows the wastage of LLC space in terms of bytes per super-block with YACC. For a 2MB LLC, across 24 single-threaded CRONO [4] and SPEC CPU 2006 [5] applications, on average, more than 32B is wasted per super-block because of the CF based compaction. Applications such as `gromacs` contains pointers that results in contiguous cache blocks of different CFs, whereas `GemsFDTD` contains mostly zeros that results in contiguous cache blocks with same CF.

Our goal is to propose a simple yet efficient cache compression scheme, which can also drive the compaction process leading to better utilization of data entries. With this goal, we aim to improve the effectiveness of the state-of-the-art compressed cache layouts, such as YACC, without changing its cache layout.

Our approach *is to compress a single cache block by keeping in mind that the corresponding block will be compacted along with its neighboring blocks.* The compaction process compacts multiple blocks based on their data contents, and not based on their CFs. In this way, both compression and compaction go hand in hand in a synergistic manner, and it improves the effective utilization of data entries. To the best of our knowledge, this is the first work on cache compression that compresses cache blocks in a way that helps in improving the effectiveness of the state-of-the-art compressed cache layouts. We make the following contributions:

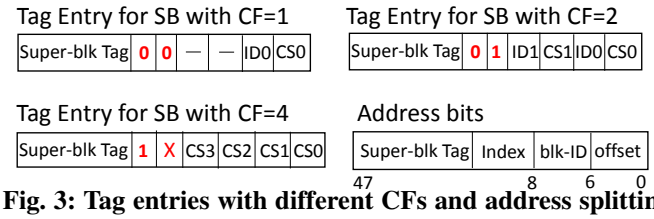

Fig. 3: Tag entries with different CFs and address splitting of a 48-bit physical address. ID, CS, and SB stand for blk-ID, coherence states, and super-block respectively. CFs of 1, 2, and 4 are encoded as 00, 01, and 1X, respectively.

(i) We propose dictionary sharing (DISH), a compression scheme that uses two encoding schemes to exploit the data that is shared across multiple blocks. The schemes help in compressing cache blocks by keeping cache compaction in mind (Section III).

(ii) We propose a simple and practical design of the compression, compaction, and decompression process (Section IV).

(iii) We evaluate DISH with YACC based layout for a wide variety of workloads. On average (geomean), across 156 workloads, DISH outperforms compression techniques, such as base-delta-immediate (BDI) [6] and CPACK+Z [7] in terms of compression ratio (from 1.7X to 2.3X), performance (from 7.2% to 14.3%), and energy efficiency (from 6% to 16%) (Section V).

II. Background

This section provides background on the state-of-the-art compressed caches and cache compression schemes.

State-of-the-art Compressed Caches. Compressed caches use a compression technique that compresses cache blocks and a compaction technique compacts the compressed blocks into one data entry. The main observation that drives the compaction process is that adjacent contiguous cache blocks co-reside at the LLC.

(i) SCC [2] is a compressed cache design that compacts multiple (in power of two, such as 1, 2, and 4) compressed blocks and stores them in one data entry. It uses super-block tags and skewed associative mapping that helps in maintaining an one-to-one mapping between a tag-entry and a data-entry. However, it compacts cache blocks based on their CFs.

(ii) YACC [3] simplifies the design aspects of SCC and uses a cache layout, which is similar to a regular uncompressed cache. It removes skewing from SCC. However, similar to SCC, it compacts cache blocks by taking their CFs into account. Throughout the paper, we consider the cache layout of YACC as the underlying compressed cache layout. YACC tracks from one to four cache blocks with the help of a super-block tag. Fig. 3 shows the splitting of address bits and the contents of tag entries (in addition to valid and dirty bits) for various CFs. In Fig. 3, the address bits are split to form a super-block tag, set index, blk-ID, and byte offset. *The blk-ID helps in finding out one out of the four blocks that are part of the super-block. Note that, with this mapping, four contiguous cache blocks share the super-block tag and are mapped to one cache set.* In tag entries, CFs are also encoded. *Note that*

the compaction process of SCC and YACC is unaware of the underlying compression techniques that compress cache blocks for it. The compaction process just leverages the CFs of the blocks. Both SCC and YACC achieve high performance with limited tag overhead ($<$ 2% of additional storage) and fast data entry access (no indirection).

Cache Compression Techniques. Although, state-of-the-art compressed caches are independent of the underlying compression techniques, compressed caches such as SCC and YACC are evaluated using CPACK+Z [7].

(i) CPACK+Z is a compression technique that detects and compresses the frequently appearing data words, such as 0X00000000 into few bits. It also extracts other kinds of data patterns and compresses cache blocks by detecting and storing the other frequently appearing 4-bytes in a dictionary. CPACK+Z is a variation of C-PACK with a feature that detects zero blocks.

(ii) BDI [6] is a recent compression technique that provides similar compressibility as CPACK+Z with *much lower* decompression latency. BDI compresses a cache block by exploiting the data correlation property. It uses one base value for a cache block, and replaces the other data values of the block in terms of their respective deltas (differences) from the base value. BDI tries different granularities of base (2 bytes to 8 bytes) and deltas (1 byte to 4 bytes). The highlight of BDI is its *low* decompression latency of one cycle.

Both CPACK+Z and BDI are proposed for compressing cache blocks and these techniques are unaware of the compaction process that compacts the compressed cache blocks. For the rest of the paper, we use the layout of YACC [3] as our baseline compressed cache layout as experimental results show that with a simple cache layout, YACC exhibits similar behavior to SCC that uses a more complex layout.

III. DISH:Encoding Schemes

This section provides an overview of DISH and explains the data encoding schemes.

A. Overview

The main idea that we develop in this paper is to leverage the fact that independent of the compression factor, a 64B data entry is allocated in the LLC, which either stores an uncompressed block or it stores several valid (up to four) compressed blocks of a 4-block super-block. With DISH, a data entry that stores four compressed blocks does not put a restriction that the size of the compressed blocks should be less than or equal to 16 bytes. DISH treats a 64-byte cache block as 16 4-byte chunks. Compressibility is the main reason behind choosing 4-byte chunks instead of 1, 2, or 8. On average, across 24 applications, we find the compression ratio of 4-byte chunks the highest.

DISH extracts the distinct 4-byte chunks of a cache block and uses encoding schemes to compress them. Next, it compacts multiple compressed blocks into one data entry if they have same set of distinct 4-byte chunks. In this way, even if the compressed block size is greater than 16 bytes, a data entry can hold four of them by storing only one set of distinct 4-byte chunks (a.k.a. dictionary) that is shared by all the four compressed blocks. This kind of compaction is possible because of two dominant *inter-block data localities* (data shared across multiple blocks), which are as follows:

Data content locality corresponds to the occurrence of same or similar data contents across contiguous cache blocks. For example, initialization of a large array with a constant value or zero can lead to the presence of same data content across contiguous cache blocks. We quantify the content locality by comparing[3] the distinct 4-byte chunks of a cache block with the distinct 4-byte chunks of its three neighboring blocks. For a 2MB LLC, on average across 24 applications, we find 32% of the LLC blocks have the same set of distinct 4-byte chunks as their neighbors.

Upper data bits locality corresponds to a scenario, where the upper data bits[4] of a 4-byte chunk are the same across multiple contiguous cache blocks however the lower bits differ. For example, pointers that spread over a large memory region. We perform studies to find out the scope of upper data bits locality. On average, across 24 applications, we find that 37% of the LLC blocks have upper data bits locality that is spread across four blocks. Based on these observations, we propose two encoding schemes (scheme-I and scheme-II) to exploit inter-block data content and upper data bits localities.

Note that with DISH, a compressed cache block necessitate 39 bytes of cache space but it is able to compact up to four compressed blocks in 57 bytes (i.e. less than a 64B data entry) due to dictionary sharing (see Fig. 4). Section III-B explains how DISH makes it possible.

B. Scheme-I

To compress a 64B cache block that contains 16 4-byte chunks, DISH encodes the cache block with a dictionary of n entries that contains n distinct 4-byte chunks. *if the cache block contains more than n distinct 4-byte chunks, then DISH treats the block as incompressible*. DISH parses the 4-byte chunks of an uncompressed block from MSB to LSB and inserts a 4-byte chunk into the dictionary, whenever it encounters a distinct 4-byte chunk that is not present in the dictionary. Each entry of the dictionary also contains a validity bit (V/I) that becomes V when a distinct 4-byte chunk is allocated into an entry. For each 4-byte chunk of an uncompressed cache block, DISH uses a *fixed-width* pointer that points to one of the n dictionary entries. So, a cache block is encoded as 16 *fixed-width* pointers of $log_2(n)$ bits along with its dictionary. For example, if a cache block contains eight distinct 4-byte chunks then the dictionary contains eight entries, and DISH encodes the block with a dictionary of eight entries and 16 3-bit pointers.

[3]We restrict this comparison to three neighboring blocks of a 4-block super-block as the maximum CF supported by YACC is 4. Also for comparison, the ordering of the 4-byte chunks within the cache blocks is irrelevant.

[4]We sweep through different widths of upper bits and find that 28-bits provide maximum opportunity for compaction.

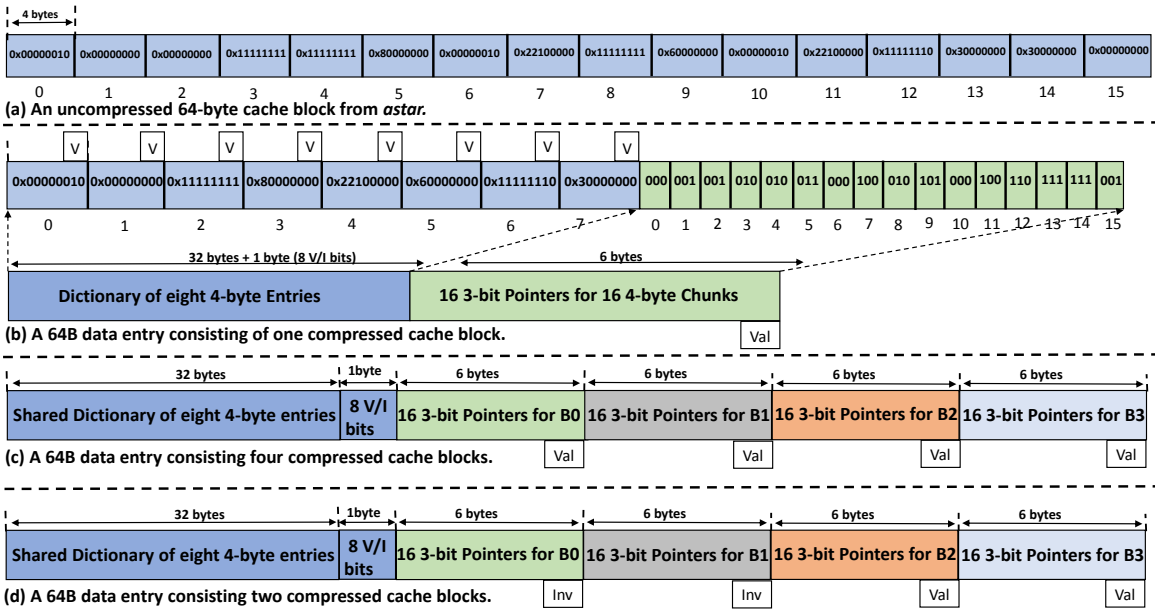

Fig. 4: Encoding of a cache block using scheme-I that exploits inter-block data content locality.

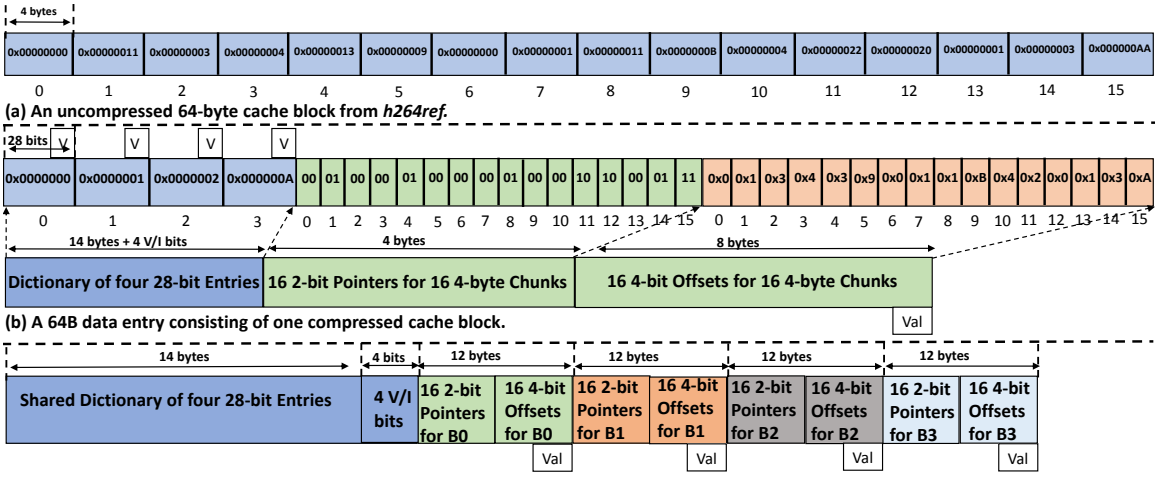

Fig. 5: Encoding of a cache block using scheme-II that exploits inter-block upper data bits locality.

Fig. 4 shows a 64B cache block from `astar` that is encoded with a dictionary of eight entries (entry-ID 0 to 7) and compressed using scheme-I. Fig. 4(a) shows an uncompressed cache block B0 in terms of 16 4-byte chunks (chunk-ID 0 to 15). Fig. 4(b) shows the contents of the eight entry dictionary and 16 3-bit pointers for each of the 16 4-byte chunks of an uncompressed block. For example, in Fig. 4(a), the 4-byte chunk with chunk-ID 7 of the uncompressed block that contains `0x22100000` is present in the entry-ID 4 of the dictionary (see Fig. 4(b)). The pointer field of the 4-byte chunk with chunk-ID 7 is encoded as $(100)_2$, which means that the data, which is present in the chunk-ID 7 of the uncompressed block, currently resides at the entry-ID 4 of the dictionary. In the next paragraph, we explain how this encoding scheme helps in an efficient compaction process.

As the basic principle of DISH is to pack maximum of four blocks in one 64B data entry, there are two kinds of data entries that can stay at the LLC: i) a data entry with CF=1 that contains an uncompressed block and ii) a data entry with CF=4 that can contain 1 to 4 valid compressed blocks. On subsequent LLC misses to the neighboring cache blocks B1, B2, and B3 that share the super-block tag, DISH encodes them individually, and if the dictionary entries of B1, B2, and B3 are *same*[5] as the dictionary entries of B0 then cache blocks B0 to B3, share and use only one dictionary (dictionary of B0), and DISH compacts them within one single data entry of 64B. The data entry contains a dictionary and four (one for each compressed block) sets of 16 3-bit pointers. DISH uses a

[5]Same set of distinct eight 4-byte chunks. The ordering of the 4-byte chunks within the cache blocks is irrelevant.

Val/Inv bit for each set of 16 3-bit pointers that says about their validity. In this way, a single data entry of 64B becomes a 4-block super-block with CF=4 that consists of four compressed cache blocks. Fig. 4(c) shows the encoding of a data entry that holds the data of four cache blocks (B0 to B3). Similarly, if only two cache blocks share their dictionary entries (say B2 and B3) then a single data entry would occupy eight 4-byte dictionary entries and 2 sets of *valid* 16 3-bit pointers, resulting in a super-block that contains two valid cache blocks. Fig. 4(d) illustrates the same. Note that with DISH, a data entry can compact two non-contiguous blocks that share the same super-block tag. For example blocks B0 and B3 can be compacted and allocated in one data entry. Similarly, B0 and B2, and B1 and B3, can be compacted into one data entry.

Incompressibility: DISH considers a cache block incompressible if it contains more than n (# entries in the dictionary) distinct 4-byte chunks. For our studies, we consider a dictionary of eight entries. The primary reason for this decision is that if a data entry contains more than eight (say nine) distinct 4-byte chunks then the dictionary should contain nine entries and there should be 16 4-bit pointers (instead of 3-bit pointers) to point one of the nine dictionary entries. In this way, if a data entry has to compact four compressed blocks then it would need 9×4 bytes (for dictionary) + four sets of 16 4-bit pointers (4×16×4-bit) = 36 bytes + 32 bytes = 68 bytes, which is greater than 64 bytes (size of a data entry). Based on the experimental studies, we find that a dictionary of eight 4-byte entries outperforms dictionaries of sixteen 2-byte entries and four 8-byte entries. We find that the potential for sharing the dictionary decreases with the increase in the width of the chunk (8-byte chunks). With 2-byte chunks, the scope for compression decreases.

Incomplete Dictionary. There are cache blocks that contain only one or two distinct 4-byte chunks. For example, a block containing `0x00000000` in all of its 4-byte chunks. While encoding and compressing these kinds of blocks, the dictionary of the first block B0 would contain fewer than eight entries. So, if the neighboring blocks contain chunks of data that are not present in the dictionary of B0 then the remaining entries of the dictionary of B0 get filled by the distinct 4-byte chunks of B1, B2, or B3.

C. Scheme-II

Scheme-I can not compress a cache block if it contains more than eight distinct 4-byte chunks. However, there are cache blocks that are still compressible even if they contain more than eight distinct 4-byte chunks. Scheme-II addresses this. Similar to scheme-I, scheme-II also treats an uncompressed block as 16 4-byte chunks. As this scheme explores the inter-block upper data bits locality, it extracts the upper 28 bits of each 4-byte chunk. Similar to scheme-I, scheme-II uses a dictionary of n entries that stores n distinct 28-bit chunks. So, with a dictionary of n entries, if an uncompressed block has more than n distinct 28-bit chunks then DISH treats the block as *incompressible*. For this scheme, DISH uses a dictionary of four entries. Note that a dictionary of more than four entries does not provide opportunity to compact four blocks in a single 64B data entry. Similar to scheme-I, each of the 16 4-byte chunks uses 2-bit pointers that points to one of the four dictionary entries (entry-ID 0 to 3). Note that the dictionary contains 28-bit entries, hence the scheme also has to store the lower 4-bits of each of the 4-byte chunks as offsets from the four 28-bit dictionary entries. Fig. 5(a) shows the contents of an uncompressed cache block B0 from `h264ref` and Fig. 5(b) shows the encoding scheme used by the scheme-II. For example, for the 4-byte chunk with chunk-ID 2 that contains `0x00000003` in Fig. 5(a), the corresponding upper 28-bit is `0x0000000`, which is stored in the dictionary with entry-ID 0 in Fig. 5(b). Thus, the pointer of chunk-ID 2 contains $(00)_2$ and the offset of chunk-ID 2 contains `0x3` in Fig. 5(b).

If neighboring blocks B1, B2, and B3, contain the same four or fewer distinct 28-bit chunks then a single data entry can hold B0, B1, B2, and B3 by storing only one dictionary of four entries, four sets of 16 2-bit pointers and four sets of 16 4-bit offsets (refer Fig. 5(c)). Similar to scheme-I, DISH uses a *Val/Inv* bit for each set of 16 2-bit pointers and 16 4-bit offsets that says about their validity. In this way, a single data entry of size 64 bytes accommodates the data of four cache blocks through scheme-II. Similar to scheme-I, we perform studies on various chunk sizes (2 byte to 8-byte), width of the offsets (2 bits to 16 bits), and the corresponding number of pointers. On average across 24 applications, we find, less than 31% of LLC blocks are incompressible with the encoding schemes of DISH.

Adaptive DISH. DISH proposes scheme-I and scheme-II, two data encoding schemes for two different kinds of inter-block data localities. However, there are uncompressed cache blocks that satisfy the conditions of compressibility of both scheme-I and scheme-II. For those blocks, we use set-dueling-monitors (SDMs) [8] to choose between scheme-I and scheme-II. We dedicate 32 *leader* cache sets that use scheme-I and similarly 32 *leader* cache sets that use scheme-II. In case of tie, the leader cache sets use their pre-defined schemes, whereas the rest of the cache sets (*follower sets*), follow the winning scheme that is selected by the SDMs. For a multi-core system with n cores that run n applications, DISH uses n SDMs, where each SDM uses a 7-bit saturating counter for the leader sets. The counter gets incremented/decremented whenever scheme-I/scheme-II is chosen by an application in its leader sets. For the follower sets, DISH chooses the winning scheme by monitoring the most significant bit of the counter.

IV. DESIGN AND IMPLEMENTATION

This section explains the compression, compaction and the decompression process of DISH. DISH takes a 64-byte uncompressed block[6] and concurrently evaluates the conditions of compressibility of both scheme-I and scheme-II. If a block is not compressible by any of the two schemes then the block is allocated in the LLC with CF=1. If one of the

[6]For our study, we have considered the industry standard of 64B block size. However, DISH can be applied 128B blocks. For scheme-I, DISH can encode a 128B block with a dictionary of 8 4-byte chunks and 32 3-bit pointers.

Algorithm 1 DISH Compressor based on Scheme-I

```
1: Input: Blk[16], a 64B block as 16 4-byte chunks.
2: Output: If compressible then a compressed block (CBlk) else Blk.
3: DReg[8]:8 32-bit registers holding 8 distinct 4-byte chunks.
4: VReg[8]:Validity-bit/register. Initially set to I.
5: PArray[16]:Array that store pointers for 16 4-byte chunks.
6: Iflag=FALSE: Incompressible if TRUE.
7: for each i, where i is the chunk-id and 0≤i≤15 do
8:   for each j, where j is the register-id and 0≤j≤7 do
9:     if (VReg[j]==I) then
10: // A distinct 4-byte chunk is allocated DReg[j]
11:       DReg[j]=Blk[i]; PArray[i]=j; VReg[j]=V;
12:       break;
13:     if (VReg[j]==V && Blk[i]==DReg[j]) then
14: // A 4-byte chunk that is already present in DReg[j]
15:       PArray[i]=j; break;
16:     if (j==7 && VReg[j]==V && DReg[j]≠Blk[i]) then
17: // Incompressible block
18:       Iflag=TRUE; return Blk;
19: if (Iflag==FALSE) then
20:   Allocate CBlk; Copy DReg and PArray into the CBlk;
21:   return CBlk;
```

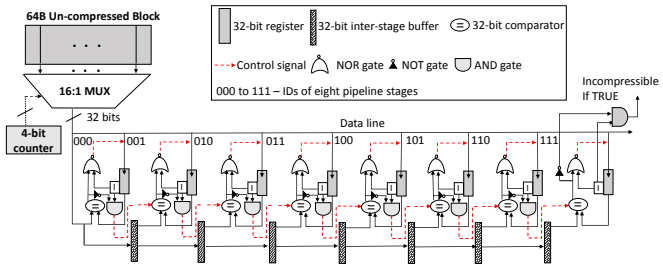

Fig. 6: Compressor circuit for scheme-I.

schemes succeeds then the block is compressed using the corresponding scheme. When both schemes succeed, DISH uses SDMs to decide which scheme to use. Note that the process of compression does not affect the response latency (not in the critical path of the program in execution) of an LLC as DISH compresses a block after responding to the upper level L2 cache.

A. Compression

Algorithm 1 illustrates the compression process in terms of a pseudo-code. Algorithm 1 takes an uncompressed block and checks for the occurrence of eight or fewer distinct 4-byte chunks. To store the distinct eight 4-byte chunks, DISH uses an array of eight 32-bit registers called DReg. DISH also uses an array called PArray that stores 16 pointers for 16 4-byte chunks. Each register in the DReg has a validity bit VReg, initially set to invalid (I), becomes valid (V) when the corresponding register contains a 4-byte chunk. DISH stops this compressibility test when it does not find any invalid register to allocate a distinct 4-byte chunk, and allocates an uncompressed block at the LLC. Note that the 4-byte chunks are allocated to the 32-bit registers in an incremental fashion (at any point in time, if VReg[i] is V and VReg[$i+1$] is I then it means DReg contains $i+1$ distinct 4-byte chunks). If a block becomes compressible then the contents of DReg becomes the content of the dictionary and the content of the PArray become the pointers of a compressed block. In the worst case, the process of evaluating compressibility takes 24 cycles (last 4-byte chunk take 8 cycles) and in the best case it takes 9 cycles (if first nine 4-byte chunks are distinct then the block becomes incompressible). We convert Algorithm 1 into a verilog code that synthesize an eight-stage pipelined circuit (See Fig. 6). Due to space constraints, we do not explain the circuit. We show this circuit to make a case for practical design. Faster but more complex compression circuit could be implemented if compression latency becomes an issue.

In terms of functionality, the circuit for scheme-II is similar to the circuit of scheme-I, with differences such as 4-stage pipelined circuit (instead of 8), 28-bit registers and buffers (instead of 32 bits), an additional circuit for extracting upper 28-bit from 4-byte chunks, and additional storage in terms of an array of offsets that store the offsets for each of the 16 4-byte chunks. Based on the outcome of the scheme-I and scheme-II, DISH transfers the contents of the registers into the cache block, along with the array of offsets (for scheme-II) and array of pointers (for both scheme-I and scheme-II).

B. Compaction

To explain the compaction process in a simple way, we assume that the blocks B0, B1, B2, and B3 of a 4-block super-block are responded from the DRAM in that order. We also assume that B0 is already compressed and encoded in a compressed form through one of the proposed encoding schemes, and the super-block tag entry of B0 contains `1X` (meaning a super-block with CF=4 that can potentially compact 4 cache blocks into one data entry. Please refer Fig. 3). With DISH, we use the don't care bit (`X`) provided by YACC layout to identify the encoding scheme used by DISH (0 for scheme-I and 1 for scheme-II). When block B1 is responded from the DRAM, DISH checks whether B1 maps to the same super-block tag as B0. Note that with YACC, a maximum 4 cache blocks share a super-block tag. A hit in a super-block tag with the same index (refer to address bits of Fig. 3), means that there is an adjacent block B0 that is already present in the cache.

DISH performs the compressibility test on B1 for both the encoding schemes. At the same time, the dictionary entries of B0 is compared with the potential dictionary of B1. If B1 is compressible with the scheme that is used by B0 and with the same dictionary entries as B0 then B1 is allocated into the same data entry as B0. The corresponding validity/coherence bits are updated in the tag and data entries of B0. Note that the updated data entry of B0 will now contain two sets of pointers (if encoded using scheme-I or scheme-II) and two sets of offsets (if encoded using scheme-II). Moreover, if the dictionary for B0 is incomplete, the dictionary might become

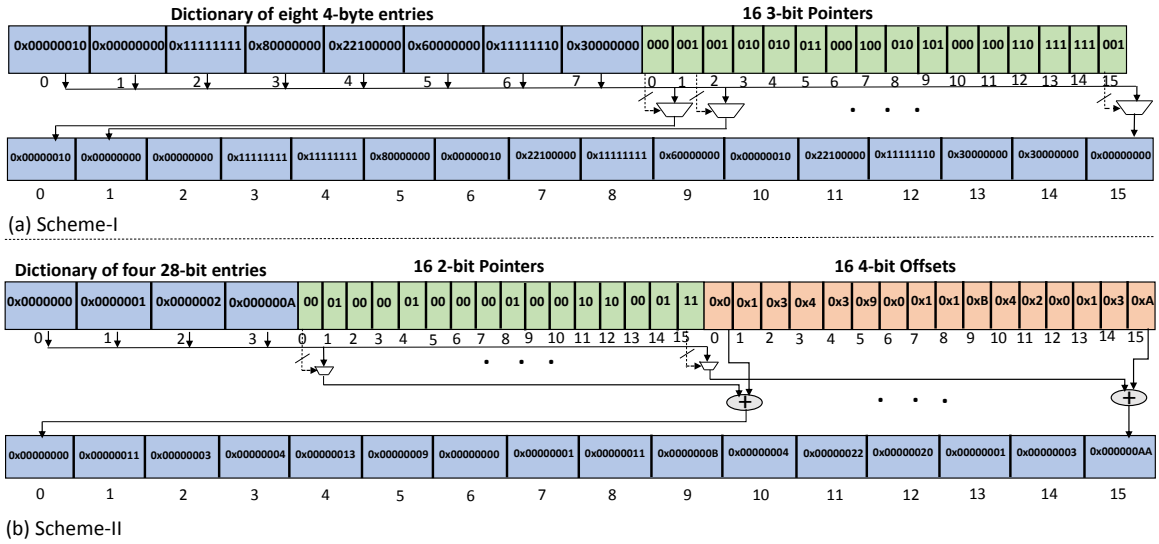

Fig. 7: Decompression hardware of DISH. (+) is a concatenation operator and not an addition.

complete with additional distinct 4-byte chunks from B1. If B1 fails the compressibility test then DISH allocates a new 64B data entry with CF=1 for an uncompressed B1. To compare the dictionaries of blocks B0 and B1, we replicate the compression hardware and preset the dictionary associated with block B0 in one of the compressors. In this way, three tests of compression can be performed in parallel for B1: one for each encoding scheme that tests the compressibility of B1 and one with preset B0 dictionary that checks whether the distinct 4-byte chunks of B1 would result in same dictionary entries as B0.

When B2 is responded from the DRAM the same compaction process (as for B1) is used. For B2, in the worst case, up to four parallel tests of compressions (one with preset B0, one with preset B1, and two for two encoding schemes with B2) will be required before compacting it with B1 and B0, B0 only, or B1 only. Also when B3 comes, DISH repeats the same process. This process of compaction consumes additional dynamic energy as DISH has to retrieve the dictionaries of the neighboring blocks.

Writebacks are also handled accordingly. In case of a write-back, a data entry that contains four compressed blocks, may become non-compactable because of a single compressed block, and when it happens DISH invalidates the corresponding compressed block and re-allocates it with the same compression process. However, if a write-back does not affect the compressibility, DISH just updates its pointers and the offsets. Note that there is no additional latency for compaction as all the comparisons of dictionary entries happen concurrently along with the compressibility test of a block. Also, for cache replacement, DISH just follow the underlying YACC layout and uses meta-data information per super-blocks for LRU and RRIP [9] based techniques.

With DISH, determining a hit/miss for a given block A3, requires an access to super-block-tag. Through super-block-tag, DISH checks the presence/absence of the neighboring blocks (say A0 to A2) and their nature (compressed or not). Note that all the neighboring blocks of a super-block are mapped to one cache set. In case of a miss, and if neighboring block(s) is present and compressed, the dictionar(ies) is read from the data array (based on the Blk-IDs) using the exact same access-slot that would have been used in case of a hit, and this does not incur any additional accesses on the cache. Note that Our cache accesses to data and tag array happen parallely (same to the fast lookup mode of CACTI [10]) and it happens irrespective of a hit or a miss. Once the dictionaries are retrieved from the data entry, DISH transfers them to the compressors (in this case, the three compressors are used). These compressors then store the dictionary entries of A0 to A2 in their respective registers. Note that the entire process happens while the data block A3 is getting retrieved from the DRAM. So this does not add to access-latency of A3. Also, during this process, incoming requests that come to the L3 request queue do not wait for additional cycles because of DISH.

For writebacks, DISH incurs two accesses: (i) retrieving the dictionaries and (ii) updating them after compression. On a writeback, DISH first reads the super-block tags. Then in the case when the block or one of its neighbors is compressed, DISH reads them to allow the combined compression/compaction. Thus a writeback consumes two access slots on the data array (instead of one access) and this leads to some extra contention at the LLC request queue for the incoming read requests.

C. Decompression

The decompression process happens when a demand request results in an LLC hit. Based on the splitting of the address bits and block indexing as mentioned in Section II, DISH indexes the particular block using the block-ID and decompresses it. Based on the encoding scheme, DISH finds out the corresponding pointers and offsets of the compressed block. Fig. 7 shows the decompression process for scheme-I and scheme-II.

Processor	1/8/16-cores, 3.7 GHz, out of order
L1 D/I, L2	32 KB (4 way), 256KB (8 way)
Shared L3	1/16/32 MB for 1/8/16 cores with 8/16/16 ways, non-inclusive
MSHRs	16, 16, 16/128/256 MSHRs at L1, L2, L3 with 1/8/16 cores
Cache line size	64B in L1, L2 and L3
Replacement policy	RRIP [9]
L2 prefetchers	Stream based [11], 32 streams with degree = 4 and distance = 32
On-chip interconnect	Crossbar, transfer latency - 4 clock cycles, arbitration latency - 5 clock cycles
DRAM controller	1/2/4 controllers for 1/8/16-cores, Open Row, 64 read/write queues, FR-FCFS, drain-when-full
DRAM bus	split-transaction, 800 MHz, BL=8
DRAM	DDR3 1600 MHz (11-11-11) Max bandwidth/channel - 12.8 GB/sec

TABLE I: Parameters of the simulated system.

	Benchmarks and their Types
LLC-capacity-sensitive-pos	SP: xalancbmk, bzip2, hmmer, GemsFDTD, soplex, mcf, omnetpp, h264ref, cactusADM, gromacs, sssp, apsp, astar
LLC-capacity-sensitive-neg	SN: libquantum, lbm, milc, bfs, bwaves, betweenness centrality (bc)
LLC-capacity-insensitive	IN: pagerank, namd, leslie3d, sjeng, community detection (community)

TABLE II: Classification of applications.

C1, C2, C3,	All-SP, All-SN, All-IN,
C4, C5, C6, C7, C8, C9,	0.75SP-0.25SN, 0.75SP-0.25IN, 0.75SN-0.25IN, 0.75IN-0.25SN, 0.75IN-0.25SP, 0.75SN-0.25SP,
C10, C11, C12,	0.5SP-0.5SN, 0.5SP-0.5IN, 0.5SN-0.5IN,
C13	Random

TABLE III: 13 categories of multi-programmed workload mixes. xSP-ySN corresponds to a mix that contains x fraction of SP benchmarks and y fraction of SN benchmarks.

For scheme-I, DISH uses 16 multiplexers for 16 4-byte chunks, and fetches 16 4-byte chunks from the dictionary based on the 16 pointers. *The crux of our scheme is its decompression latency, which is one cycle*. So a demand request is serviced with an additional latency of one cycle when compared with an uncompressed cache. In case of scheme-II, a compressed block goes through a set of multiplexers (similar to scheme-I), and DISH concatenates the offsets of each of the 4-byte chunks with the output of their respective multiplexers.

D. Overhead Analysis

This section provides the area, latency, and energy overheads that comes with DISH. DISH does not incur any additional storage for storing the meta-data. However, in terms of additional circuits, DISH uses seven 32-bit buffers, eight 32-bit registers, eight 32 bit comparators, 16 MUXs of size 8:1 (for scheme-I), and 16 MUXs of size 4:1 (for scheme-II). It also uses a circuit to extract the offsets. We estimate the area, latency, and energy overheads by using CACTI 6.5 [10] and Synopsys Design Compiler. For a 32-nm technology, and for a 2MB cache that occupies 22mm^2 (tag array + data array), the area occupied by the compressor and de-compressor is less than 0.3mm^2. For multi-core systems with 16MB (54mm^2) and 32MB (93mm^2) LLCs, the compressor and the de-compressor circuit is replicated for each cache bank, resulting in an area overhead of 2.4mm^2. As mentioned earlier, in the worst case, DISH takes 24 cycles and 1 cycle for compression and decompression, respectively. In terms of dynamic energy per access, the compressor and the de-compressor of DISH consume 0.09nJ and 0.02nJ, respectively, whereas an uncompressed 2MB cache consumes 0.51nJ. In terms of static power per cache bank, DISH dissipates 0.061mW, which is negligible compared to the static power consumed (18.4mW) by a 2MB LLC.

V. Experimental Evaluation

In this section, we evaluate the effectiveness of DISH and compare it with BDI and CPACK+Z that use YACC based layout and CF based compaction. We compare these techniques in terms of compression ratio, performance improvement, and improvement in energy consumption. Note that CPACK+Z and BDI takes 16/9 and 2/1 cycles for compression/decompression, respectively. We do not show the results for SCC based cache layout as the effectiveness (in terms of performance and compression ratio) of SCC is almost the same as YACC. However, YACC is simpler and practical to implement.

Evaluation Methodology. We use the gem5 [12] simulator to evaluate the effectiveness of DISH at the LLC. Table I shows the baseline configuration of our simulated system. We simulate both single-core and multi-core (8- and 16-core) systems, and we estimate the cache latencies by using CACTI 6.5 [10]. We collect the statistics for workloads by running each benchmark in a workload for 500M instructions after a fast-forward of 20B instructions and warm-up of 500M instructions, which is similar to the methodology used in [13], [14], and [15]. A workload terminates when the slowest benchmark completes 500M instructions.

Metrics of Interest. We use the following metrics to evaluate the effectiveness of DISH: compression ratio, misses per kilo instruction (MPKI), bus accesses per kilo instruction (BPKI) [11], and speedup. We calculate the compression ratio at the set level granularity and take the average across all the cache sets. Misses per kilo instruction and bus accesses per kilo instruction provide effect of cache capacity on the LLC misses and off-chip DRAM traffic, respectively. For single-core simulations, we use speedup as $\frac{Execution-time_{baseline}}{Execution-time_{technique}}$, whereas for multi-core systems, we use weighted speedup [16], which is $\sum_{i=0}^{N-1} \frac{IPC_i^{together}}{IPC_i^{alone}}$, where $\text{IPC}_i^{together}$ is the IPC of core i when it runs along with other N-1 applications and IPC_i^{alone} is the IPC of core i when it runs alone on a multi-core system of N cores.

Workload selection. Table II classifies 24 benchmarks (from SPEC CPU 2006 and CRONO benchmark suites) into 3 classes (SP, SN, and IN), based on their sensitiveness to the cache capacity. A benchmark is *sensitive-positive* (SP) if there is an improvement in performance with the increase in the LLC size, *sensitive-negative* (SN) if there is performance degradation with the increase in the LLC size (applications for which LLC misses do not drop but access latency increases

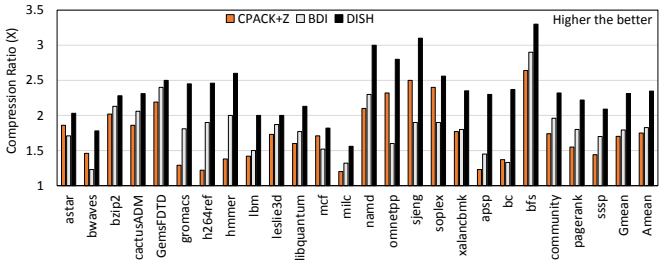

Fig. 8: Compression ratios for YACC.

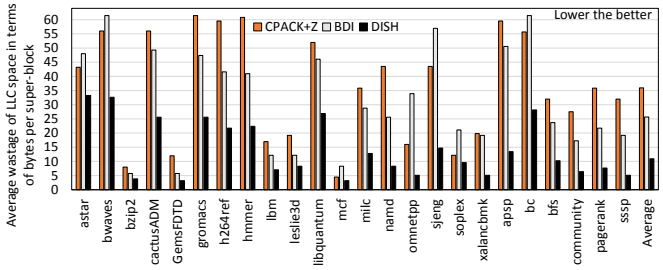

Fig. 9: Reduction in the LLC wastage.

with the increase in the LLC size), and *insensitive* (IN) if the increase in the cache size does not affect performance or affect it negligibly. Table III shows the 13 categories of multi-programmed workload mixes that we create from the 24 benchmarks. Note that, while creating multi-programmed workload mixes, we use single-threaded CRONO benchmarks. We create 104 8-core and 52 16-core workload mixes that are distributed equally across 13 categories.

A. Single-core Results

Compression ratio. Fig. 8 shows the compression ratios achieved by CPACK+Z, BDI, and DISH with a YACC based compressed cache layout. On average (geomean), DISH provides a compression ratio of 2.34X, whereas CPACK+Z and BDI provide compression ratios of just above 1.6X. With CPACK+Z and BDI, none of applications provide compression ratios that are higher than 3X. However, DISH provides compression ratios of up to 3.3X. For applications such as `gromacs`, `hmmer`, `h264ref`, and `apsp`, scheme-II of DISH plays an important role in improving the compressibility. These applications, mostly contain pointers and offsets that spread across multiple cache blocks, and as Fig. 2 shows, these applications also have more wastage of LLC space. For other applications, adaptive DISH helps as both the schemes play an important role. Our simple compression scheme achieves more than 85% accuracy on average, when compared to an ideal DISH (a trace based compression scheme that always chooses the correct encoding scheme). However, there are applications such as `soplex` and `bfs` where SDMs are accurate for just over 70% of the time. Fig. 9 shows the reduction in the wastage of LLC space. Compared to CPACK+Z and BDI, DISH reduces the LLC wastage by 3.2X and 2.3X, respectively. Applications such as `sssp` shows the highest reduction of 6.2X over CPACK+Z. Reduction in the wastage of LLC space allows more data to be packed in data entries,

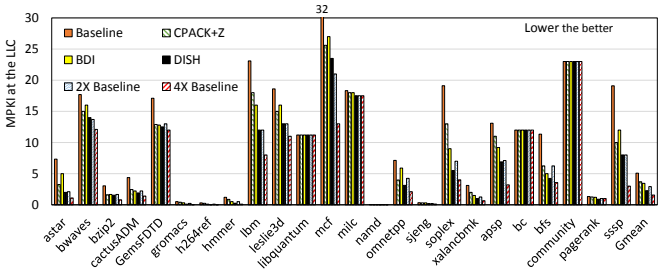

Fig. 10: LLC misses in terms of MPKI.

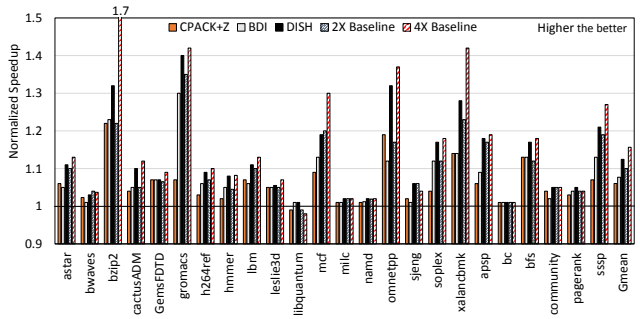

Fig. 11: Speedup over a 2MB LLC.

which improves the compression ratio.

Observations. We find three kinds of super-blocks at the LLC: i) both CPACK+Z and BDI with CF based compaction, and DISH provide the same compression ratio. This happens when all the blocks of a super-block have the same or similar data content, which leads to same CF. So, compared to CF based compaction, DISH provides no improvement in compression ratio. ii) CPACK+Z or BDI with CF based compaction compact more blocks than DISH. This is a *rare* case (on average, across all the applications, only 16% of the LLC blocks fall into this case). This happens when cache blocks fail to share the dictionary, which provides no opportunity for compaction whereas CPACK+Z and BDI are able to compress the blocks and CF based compaction compacts more blocks than DISH. Applications such as, `lbm` and `mcf` contain this kind of super-blocks. iii) DISH performs better than BDI and CPACK+Z because of the reasons mentioned in the Fig. 1.

LLC Misses. The higher compression ratio of DISH helps in reducing the LLC misses. DISH reduces LLC misses of all the cache sensitive applications. Moreover, for most of the applications, LLC misses of a 2MB DISH is lower than a 4MB uncompressed baseline cache. At the same time, it does not affect the cache in-sensitive applications, and the LLC misses of those applications remain the same with DISH. On average, compared to a baseline cache, CPACK+Z, BDI, DISH, and 2X baseline cache, provide 23%, 29%, 50%, and 39.2% reductions in the LLC misses, respectively. Fig. 10 shows the LLC misses (in terms of MPKI) for all the 24 applications. Note that improvement in compression ratio does not guarantee reduction in LLC misses as some of the applications do not benefit from the increased cache space. For example, LLC misses of `libquantum` does not change till an LLC of size

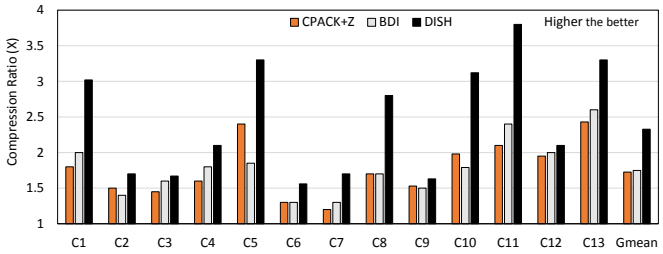

Fig. 12: Per category compression ratios.

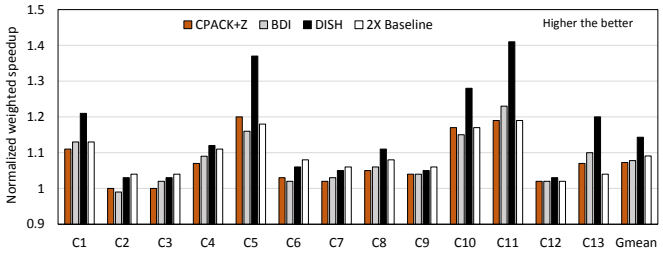

Fig. 13: Per category weighted-speedups.

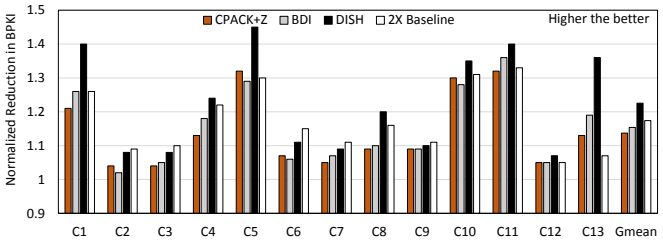

Fig. 14: Reduction in off-chip traffic.

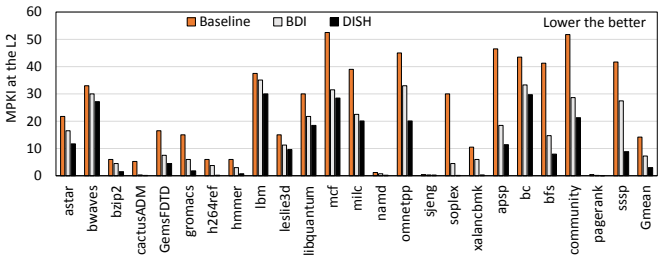

Fig. 15: L2 cache misses in terms of MPKI.

32MB, and beyond 32MB, it drops to zero as its working set fits in. We observe a similar trend in `bc` and `community`.

Speedup. Fig. 11 shows the speedup that is achieved by DISH normalized to an uncompressed baseline cache. We also show the performance that can be achieved by doubling (2X) and quadrupling (4X) the LLC capacity. Note that our experiments assume an increase in the cache access latency by 7 and 14 cycles for 2X and 4X baseline caches (based on CACTI 6.5), respectively. DISH outperforms CPACK+Z and BDI for all the cache sensitive applications. For cache insensitive applications, DISH provides marginal improvement over CPACK+Z and BDI. On average, DISH provides 12.4% of speedup (a maximum of 40% for `gromacs`) whereas a 2X baseline cache provides a speedup of 10.1%. BDI and CPACK+Z provide improvements of 6% and 7.6%, respectively. *DISH provides better speedup compared to CPACK+Z and BDI as the reduction the LLC wastage helps the cache-sensitive applications significantly. Also, as the decompression latency of DISH is only one cycle, it has no effect on the performance of the cache insensitive applications.*

B. Results for Multi-core Systems

For multi-core systems, we use 2MB/core and use 16MB and 32MB LLCs for 8-core, and 16-core simulated systems, respectively. The access latencies are also updated based on CACTI 6.5 [10]. We also scale all other shared resources as mentioned in Table I.

Compression ratio. Fig. 12 shows the average of compression ratios for 156 (104 8-core and 52 16-core) multi-programmed workload mixes based on the 13 categories as mentioned in Table III. Note that we simulate an equal number of workloads from each category. On average, DISH provides a compression ratio of 2.32X, with a maximum compression ratio of 3.9X for a workload that belongs to C10 category. Out of 156 workloads, only 51 workloads show a compression ratio of less than 2X, which makes a very strong case for DISH. On the other hand, CPACK+Z and BDI provide compression ratios of 1.72X and 1.74X respectively. *For workload categories, such as C1, C5, C10, and C11, where at-least 50% of the applications are cache-sensitive-positive, and there is under-utilization of data entries (LLC wastage of more than 32B per super-block), DISH provides compression ratios of more than 3X.*

Weighted Speedup. Fig. 13 shows the performance improvement in terms of weighted speedup for each of 13 categories. On average (geomean) across 156 workloads, DISH provides 14.3% improvement whereas BDI, CPACK+Z, and 2X baseline cache provide improvements of 7.7%, 7.2%, and 9%, respectively. DISH outperforms CPACK+Z and BDI for all the categories. For categories that contain only cache-insensitive applications, DISH marginally performs better CPACK+Z. One of the reasons for this trend is, the decompression latency of DISH is eight cycles lower compared to CPACK+Z. DISH provides a maximum improvement of 51.2% for an 8-core workload that contains `gromacs-mcf-h264ref-namd-leslie3d-hmmer-astar-sjeng`, in which the data entries that are wasted by cache-insensitive applications are utilized by cache-sensitive applications. This results in significant performance improvements for applications such as `astar`, `sjeng`, `hmmer`, and `h264ref`. *If a multi-programmed workload mix contains at-least 25% cache sensitive applications, and its super-blocks contain more than 40 bytes of average wastage then DISH provides significant performance improvement (at-least 15% improvement over the baseline).*

Off-chip traffic. In multi-core systems, off-chip bandwidth consumption is also an essential component, and improvement in the compression ratio at the LLC helps in reducing the off-chip bandwidth consumption as the number of data transfers between the LLC and the DRAM get reduced. Fig. 14 shows the reduction in the traffic. On average, compared to the baseline, DISH provides 22.5% reduction in the traffic

whereas CPACK+Z, BDI, and 2X baseline provide reductions of 13.6%, 15.3%, and 17.3%, respectively. DISH provides better reductions for workloads that contain applications that are cache sensitive with higher LLC MPKIs.

Energy consumption. As performance is not the only metric that is important for the modern systems, we quantify the energy consumption of the memory subsystem (LLC+DRAM), ignoring the energy benefit that comes because of reduction in the execution time. We quantify the static and dynamic energy consumed by LLC, DRAM, compressors/decompressors of DISH, and memory transfers. On average across 156 workloads, compared to a baseline LLC, DISH is 22% more energy efficient, and compared to CPACK+Z and BDI, DISH is 16% more energy efficient. Note that the energy overheads of DISH is marginal compared to the energy gain.

C. Further Evaluations

Effect of additional access: To quantify the effect of additional access due to writeback, we simulate an ideal but unrealistic DISH that does not incur additional accesses because of writebacks, and compare the performance with the proposed DISH. For single core systems, on average, the unrealistic DISH would provide an additional improvement of 1.8% when compared to the proposed DISH. For multi-core systems, in terms of weighted speedup, compared to the proposed DISH, the unrealistic DISH provides additional improvement of 2.2%. Applications such as `lbm`, `libquantum`, and `mcf` that have high LLC WPKCs (writebacks per kilo cycles) of 18, 12, and 9.6 are affected the most because of the additional access. Overall, the contention at the LLC request queue has marginal effect on the system performance.

DISH at the L2. As the decompression latency of DISH is just one cycle, there is an opportunity to extend DISH at the L2. Among CPACK+Z and BDI, BDI is the compression technique with permissible decompression latency. We simulate BDI and DISH at the L2 with a YACC based layout. On a demand miss at the L2, we send a compacted 64B data entry (that contains the data of multiple cache blocks) to the L2 cache controller. Fig. 15 shows the reduction in the L2 MPKI, with this enhancement. Also, DISH at the L2 along with DISH at the LLC provides an off-chip traffic reduction of 3.8X. For multi-core systems, this optimization improves the combined cache capacity (per core L2s + LLC) by 2.78X, which results in 19.2% improvement in weighted speedup. Note that our baseline cache hierarchy is non-inclusive. For inclusive/exclusive cache hierarchies, additional care need to be taken to maintain inclusiveness/exclusiveness.

Effect of LLC size. We simulate 8-core systems with 8/16/32/64MB LLC and we find, none of the compression schemes are effective for a 64MB LLC as the working sets of most of the applications fit in the LLC. When we migrate from 16MB to 32MB, there is slight degradation in the performance improvement (from 14.3% to 11.1%), whereas when we migrate to 8MB, there is further increase in the performance improvement (from 14.3% to 19.6%).

DRAM bandwidth. We study the sensitivity of DRAM bandwidth by changing the bandwidth per DRAM controller from 12.8GB/sec to 6.4GB/sec and 25.6GB/sec. The compression process takes 24 cycles. So if DRAM bandwidth is significantly high then the compression process can be a bottleneck. We find for some 16-core workloads, DRAM bandwidth of more than 32 GB/sec creates a bottleneck. For those workloads we can use multiple DISH compressors. Overall, the effectiveness of DISH changes marginally with the change in the available off-chip bandwidth.

Prefetching. To understand the sensitiveness of hardware prefetchers, we simulate DISH with no prefetchers. The effectiveness of DISH remains the same with no hardware prefetchers, with performance improvement of 10.8% for single-core workloads. In case of multi-programmed workloads, the average improvement over the baseline is 12.1%.

VI. Related Work

To the best of our knowledge, this is the first work on cache compression that helps in cache compaction by exploiting the feature of compressed cache layouts, such as SCC and YACC. Several cache compression techniques have been proposed that exploit the inter-block data localities to compress a cache block. ZCA [17], and a technique proposed by Ekman and Stenstrom [18] compress the zero blocks, whereas Alameldeen and Wood compress cache blocks that have narrow values (a small value stored in large size data type, for example a value of 1 that needs only one bit is stored with a `long int` data type) [19]. Arelakis and Stenstrom propose a statistical compression technique called SC^2 [20] that uses Huffman encoding. Although SC^2 provides better compression ratio than all other techniques, it needs software support for sampling of data to generate the Huffman encoding and in this process it demands storage of 18KB. Also, its decompression latency can go up to 14 cycles, which is high compared to 1 cycle decompression latency of BDI and DISH. Recently, a technique called Hycomp [21] is proposed that switches between different compression techniques, such as BDI, SC^2, and ZCA as none of the compression schemes works well across all applications.

Apart from the above mentioned techniques, there are techniques such as frequent value compression (FVC) [22] that encodes the frequently occurring values. It also needs profiling to identify the frequent values. On the other hand, Alameldeen and Wood identify seven frequently occurring compressible data patterns in frequent pattern compression (FPC) [23]. Tian et al. [24] improve the cache capacity by detecting duplicate blocks and replacing it with only one copy of the data. Baek et al. [25] propose size aware cache replacement policies that takes the compressed size of the cache blocks into account for replacing cache blocks. Pekhimenko et al. propose CAMP [26] that outperforms size aware cache replacement policies. DISH is orthogonal to these replacement schemes.

VII. Conclusion

This paper proposed DISH, a simple, yet efficient cache compression technique that exploits the layout of the state-of-

the-art compressed caches. In contrast to the prior cache compression schemes, such as CPACK+Z and BDI that compress a cache block independently, DISH compresses a cache block, anticipating that the block might be compacted with other neighboring blocks. In this way, more blocks get compacted in one data entry, which helps in reducing the wastage of LLC space, and improving the effective utilization of data entries.

DISH exploits two kinds of data patterns that are shared across multiple blocks and provides a simple and practical design with a decompression latency of only one cycle. This makes DISH easily employable in modern systems. Based on our experiments with YACC, DISH outperforms the state-of-the-art compression schemes in terms of compression ratio (2.3X from 1.7X) and performance improvement (14.3% from 7.2%). We conclude that DISH is a low-latency compression technique that is suitable for state-of-the-art compressed caches, such as SCC and YACC.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers and the members of the ALF team for their suggestions. The authors also thank Somayeh Sardashti and David Wood for their help. This work is partially supported by ERC Advanced Grant DAL No. 267175. This work is also partially supported by an Intel research gift.

REFERENCES

[1] S. Sardashti and D. A. Wood, "Decoupled compressed cache: Exploiting spatial locality for energy-optimized compressed caching," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, (New York, NY, USA), pp. 62–73, ACM, 2013.

[2] S. Sardashti, A. Seznec, and D. A. Wood, "Skewed compressed caches," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, (Washington, DC, USA), pp. 331–342, IEEE Computer Society, 2014.

[3] S. Sardashti, A. Seznec, and D. A. Wood, "Yet Another Compressed Cache: a Low Cost Yet Effective Compressed Cache," in *ACM Transactions on Architecture and Code Optimization*, 2016.

[4] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan, "CRONO: a benchmark suite for multithreaded graph algorithms executing on futuristic multicores," in *Workload Characterization (IISWC), 2015 IEEE International Symposium on*, pp. 44–55, Oct 2015.

[5] C. D. Spradling, "SPEC CPU2006 Benchmark Tools," *SIGARCH Computer Architecture News*, vol. 35, March 2007.

[6] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-delta-immediate compression: Practical data compression for on-chip caches," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, (New York, NY, USA), pp. 377–388, ACM, 2012.

[7] X. Chen, L. Yang, R. P. Dick, L. Shang, and H. Lekatsas, "C-PACK: a high-performance microprocessor cache compression algorithm," vol. 18, (Piscataway, NJ, USA), pp. 1196–1208, IEEE Educational Activities Department, Aug. 2010.

[8] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer, "Adaptive insertion policies for managing shared caches," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, (New York, NY, USA), pp. 208–219, ACM, 2008.

[9] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, (New York, NY, USA), pp. 60–71, ACM, 2010.

[10] N. Muralimanohar and R. Balasubramonian, "CACTI 6.0: A tool to understand large caches."

[11] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, (Washington, DC, USA), pp. 63–74, IEEE Computer Society, 2007.

[12] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, pp. 1–7, Aug. 2011.

[13] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and efficient fine-grain cache partitioning," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, (New York, NY, USA), pp. 57–68, ACM, 2011.

[14] R. Manikantan, K. Rajan, and R. Govindarajan, "Probabilistic shared cache management (PriSM)," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, (Washington, DC, USA), pp. 428–439, IEEE Computer Society, 2012.

[15] D. Kadjo, J. Kim, P. Sharma, R. Panda, P. Gratz, and D. Jimenez, "B-Fetch: branch prediction directed prefetching for chip-multiprocessors," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, (Washington, DC, USA), pp. 623–634, IEEE Computer Society, 2014.

[16] A. Snavely and D. M. Tullsen, "Symbiotic job scheduling for a simultaneous multithreaded processor," in *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, (New York, NY, USA), pp. 234–244, ACM, 2000.

[17] J. Dusser, T. Piquet, and A. Seznec, "Zero-content augmented caches," in *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, (New York, NY, USA), pp. 46–55, ACM, 2009.

[18] M. Ekman and P. Stenstrom, "A robust main-memory compression scheme," in *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, (Washington, DC, USA), pp. 74–85, IEEE Computer Society, 2005.

[19] A. R. Alameldeen and D. A. Wood, "Adaptive cache compression for high-performance processors," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, (Washington, DC, USA), pp. 212–, IEEE Computer Society, 2004.

[20] A. Arelakis and P. Stenstrom, "Sc2: A statistical compression cache scheme," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, (Piscataway, NJ, USA), pp. 145–156, IEEE Press, 2014.

[21] A. Arelakis, F. Dahlgren, and P. Stenstrom, "Hycomp: A hybrid cache compression method for selection of data-type-specific compression methods," in *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, (New York, NY, USA), pp. 38–49, ACM, 2015.

[22] J. Yang, Y. Zhang, and R. Gupta, "Frequent value compression in data caches," in *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 33, (New York, NY, USA), pp. 258–265, ACM, 2000.

[23] A. R. Alameldeen and D. A. Wood, "Frequent pattern compression: A significance-based compression scheme for l2 caches.," in *Technical Report 1500, Computer Sciences Department, University of Wisconsin-Madison*, 2004.

[24] Y. Tian, S. M. Khan, D. A. Jiménez, and G. H. Loh, "Last-level cache deduplication," in *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, (New York, NY, USA), pp. 53–62, ACM, 2014.

[25] S. Baek, H. G. Lee, C. Nicopoulos, J. Lee, and J. Kim, "Ecm: Effective capacity maximizer for high-performance compressed caching," in *High Performance Computer Architecture (HPCA), 2013 IEEE 19th International Symposium on*, pp. 131–142, Feb 2013.

[26] 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 *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pp. 51–63, Feb 2015.