PBC: Prefetched Blocks Compaction

Raghavendra K., *Student Member, IEEE,* Biswabandan Panda, *Student Member, IEEE,* and Madhu Mutyam, *Senior Member, IEEE*

Abstract—Cache compression improves the performance of a multi-core system by being able to store more cache blocks in a compressed format. Compression is achieved by exploiting data patterns present within a block. For a given cache space, compression increases the effective cache capacity. However, this increase is limited by the number of tags that can be accommodated at the cache. Prefetching is another technique that improves system performance by fetching the cache blocks ahead of time into the cache and hiding the off-chip latency. Commonly used hardware prefetchers, such as stream and stride, fetch multiple contiguous blocks into the cache. In this paper we propose prefetched blocks compaction (PBC) wherein we exploit the data patterns present across these prefetched blocks. PBC compacts the prefetched blocks into a single block with a single tag, effectively increasing the cache capacity. We also modify the cache organization to access these multiple cache blocks residing in a single block without any need for extra tag look-ups. PBC improves the system performance by 11.1% with a maximum of 43.4% on a 4-core system.

Index Terms—Memory Structures, Cache Memories, Compression, Compaction, Cache Design, Prefetching.

1 Introduction

Last-level-cache (LLC) is one of the critical resources in a multi-core system. Effective management of the LLC plays a key role in improving the system performance. Micro-architectural techniques that provide more cache hits at the LLC are the key. Two such techniques are cache compression and hardware prefetching.

A cache compression technique increases the *effective* cache capacity[1] without increasing the cache size. The increase in the cache capacity provides additional cache space that can hold a larger working set resulting in performance improvement. Cache compression is based on the observation that significant amount of data accessed/generated by a program share common patterns that can be represented (stored) using fewer number of bits. Some of the common patterns are: repeated values [7], zeros [6], narrow values[2] [5], and the recently proposed base-delta immediate [1] that exploits the narrow differences between the data values stored within a cache block.

Hardware prefetching, on the other hand, hides the off-chip DRAM latency. Instead of waiting for a cache miss to fetch data from the DRAM, a prefetcher trains itself to identify such misses ahead of time and prefetches data into the cache, thereby converting the cache miss into a potential cache hit. One of the key features of the addresses generated by commonly used prefetchers, such as stride and stream, is that these addresses correlate in space due to spatial locality, i.e, multiple contiguous cache blocks are fetched from the DRAM by the prefetcher.

Our Goal: We observe that the key features of prefetching can be used for compression, which can further increase the effective cache capacity. Our goal in this work is to design a mechanism that can exploit the features of hardware prefetching for compressing multiple prefetched blocks into a single block.

Our Approach: We treat multiple contiguous prefetched cache blocks that share common higher-order address bits as one single cache block, and exploit the data patterns present across these blocks. In effect, multiple prefetched blocks are compacted into a single block, which we refer to as *compact block*. In a generic compressed/uncompressed LLC, these prefetched cache blocks would spread across different cache sets. We modify the LLC organization to access these prefetched blocks (having different set indices, but compacted into a single cache block), *without incurring any additional tag look-ups*. Note that we use the word *compaction* and not *compression* as the later is used in prior techniques which compress data *within* a block and *not across* the blocks. A compact block is also different from a super block [4], [8] because in the latter only blocks that share a common tag are put together to reduce tag store overhead.

To the best of our knowledge, for multi-core systems, this is the first work that compacts multiple prefetched cache blocks into a single cache block at the shared LLC.

The contributions of this paper are as follows:

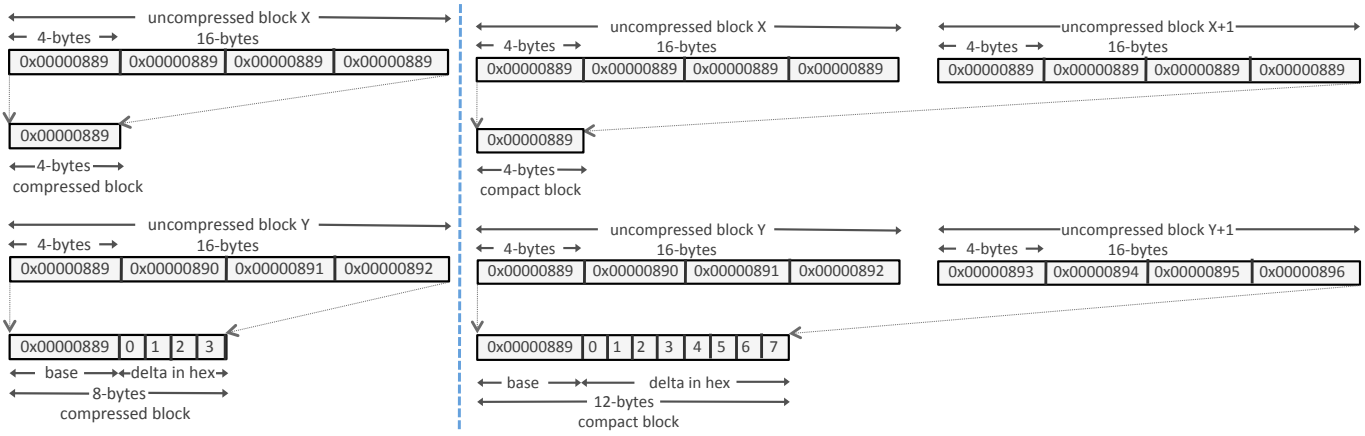
(i) We propose prefetched blocks compaction (PBC), a low-cost and practical compaction technique that compacts multiple prefetched blocks into a single cache block called compact block, by taking into account the data present across these blocks. (Section 3.2)

(ii) We implement PBC by modifying the cache organization and using different hash functions to index into a cache set such that the compacted prefetched blocks share a common set index. Through this effective mechanism, PBC does not incur any additional delay when accessing a cache set. PBC

- *Raghavendra K. is with the Department Computer Science and Engineering, Indian Institute of Technology Madras, Chennai, India. E-mail: raghu@cse.iitm.ac.in*
- *Biswabandan Panda is with the Department Computer Science and Engineering, Indian Institute of Technology Madras, Chennai, India. E-mail: biswa@cse.iitm.ac.in*
- *Madhu Mutyam is with the Department Computer Science and Engineering, Indian Institute of Technology Madras, Chennai, India. E-mail: madhu@cse.iitm.ac.in*

1. Effective cache capacity = #valid compressed cache blocks $\times$ block size.

2. small value stored in a large size data type. Example - 1 bit/byte stored in a 4 byte `integer` data type.



(a) **Compression** (b) **Compaction**

Figure 1: **An example illustrating the benefits of compaction.**

also does not require additional tags. (Section 3.1)

(iii) We evaluate PBC across a wide variety of workloads and compare its performance to a baseline system with no compaction. On an average (geomean), PBC improves the system performance (in terms of harmonic speedup) by 11.1% and 13.6% for 4- and 8-core systems across seventy and twenty five workloads, respectively. (Section 4)

2 Background and Motivation

Hardware Prefetching: In this work, we consider three different hardware prefetching techniques: stream [3], stride [19], and one of the state-of-the-art techniques, Spatial Memory Streaming (SMS) [18].

Stream: A stream prefetcher keeps track of multiple access streams. Once trained, it issues k prefetch requests at a time, where k is the prefetch degree[3]. Stream prefetchers are employed in many commercial processors such as IBM's POWER series [3] and Intel's Nehalem [21].

Stride: A stride prefetcher attempts to learn simple stride (distance between the memory addresses referenced by a program counter) based memory accesses, on the basis of the past behavior of the program counter. It also stores the last-address referenced by the program counter. In future, on a miss, the prefetcher issues prefetch requests to $last\text{-}address + k \times stride$, where k is the prefetch degree. Similar to stream prefetching, stride prefetching is also widely used for its low hardware overhead and simplicity in design.

SMS: SMS is also one of the widely used hardware prefetchers because of its high-performance, practically implementable design and low hardware overhead. SMS leverages the spatial locality present over a large memory region (spatial region). It exploits code based correlation to exploit the data locality by using the program counter to train the accesses within the spatial region. For a given spatial region, SMS predicts and prefetches the future access pattern based on the history of access patterns triggered by that program counter in the past.

The prefetch addresses generated by the above prefetchers being contiguous provide an opportunity to compact multiple prefetched blocks into a single block.

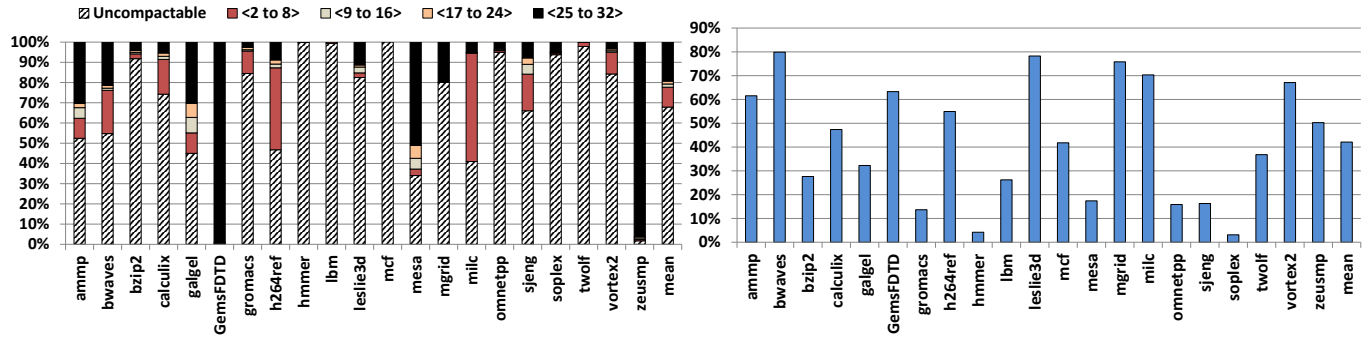
3. Determines the number of prefetches to be issued in one instant.

Cache Compression: Prior works on cache compression, such as [1], [5], [6], and [4] are successful in doubling/quadrupling cache capacity. The basic premise of these techniques is that a significant amount of data values generated/accessed by a program share a common pattern. For example, if all the words in a cache block store a value of zero or the words have the same value, then a single bit/word is enough to represent the entire block depending on the compression algorithm.

BDI: Base-delta-immediate (BDI) [1] is one such compression technique that not only compresses zero and repeated values but also exploits the case where the relative difference between the values stored in a cache block is small. For the ease of illustration, we consider a 16-byte cache block Y as shown in Figure 1a. The cache block is logically divided into a set of fixed size words, in this case four 4-byte words. The relative difference in values (deltas) between a chosen base and the words are stored along with the base. For block Y, 0x00000889 is the base and 0x00, 0x01, 0x02, and 0x03 are the deltas. Additional bits, referred to as *encoding bits*, which represent the compression pattern exploited, the size of the base, and the size of the words are stored along with the compressed block. It follows that the position of each word within a compressed block (which can be inferred using encoding bits) is independent of other words, which makes the decompression process parallel, leading to low decompression latency. When a processor requests for the block, the encoding bits are used to decompress the block and the request is serviced. Table 1 shows the encoding bits used in BDI for a 64 byte cache block.

S.No.	Name	Base	Delta	Size	Encoding bits
1	Zeros	1	0	1	0000
2	Rep Values	8	0	8	0001
3	B8-D1	8	1	16	0010
4	B8-D2	8	2	24	0011
5	B8-D4	8	4	40	0100
6	B4-D1	4	1	20	0101
7	B4-D2	4	2	36	0110
8	B2-D1	2	1	34	0111
9	NoCompre.	N/A	N/A	64	1111

Table 1: **BDI encoding for a 64 byte cache block. B: Base size and D: Delta size. All sizes are in bytes.**



(a) **Distribution of prefetched blocks at the LLC. <a to b> signifies that a to b number of blocks are compactable into a single block.**

(b) **Percentage of prefetched blocks at the LLC.**

Figure 2: **Behavior of prefetched blocks at the LLC across a wide variety of SPEC CPU 2000/2006 benchmarks.**

2.1 Motivating Example

Cache compression techniques exploit data patterns *within* a block to reduce the effective cache space needed to store the block. Cache block size is an architecture parameter and not a program characteristic. Hence, data patterns across significant number of contiguous blocks also share a common pattern. In this work, we try to exploit data patterns *across* multiple contiguous prefetched blocks. To motivate, we consider two cache blocks (block X and block Y), as shown in Figure 1a. Block X contains 4 words, each of 4 bytes, containing 0x00000889 as the data. In this case, BDI represents the entire cache block using a single 4 byte compressed block, saving 12 bytes of cache space. For cache block Y, the relative difference between the 4-byte data values is very small. In this case, BDI stores a 4-byte base, which is 0x00000889 along with the relative difference (delta) between the base and the data values. This reduces the cache space required to store a block, from an uncompressed block of size 16 bytes to a compressed block of size 8 bytes.

Next, we try to identify the patterns across the cache blocks. To illustrate, we extend the previous example in Figure 1b. Figure 1b shows blocks $X+1$ and $Y+1$ that are contiguous to blocks X and Y, respectively. All the words in X and $X+1$ contain the data 0x00000889. In the case of BDI two compressed blocks (one for each uncompressed block), both containing 0x00000889, are stored. On the other hand, compaction stores only a single data value 0x00000889 representing two cache blocks. Similarly, when contiguous blocks Y and $Y+1$ are compressed separately, each compressed block would occupy 8-bytes. Compaction, on the other hand, uses a single base to compact both the blocks, thus saving upon space (12-bytes instead of 16-bytes). In the case of compression, every compressed block in the cache has a set of encoding bits, whereas compaction stores only a single set of encoding bits for the entire compact block (which holds multiple cache blocks).

One way to naively exploit this opportunity is to increase the block size and then apply cache compression. However, blindly increasing the block size may affect the effective utilization of cache capacity, and may lead to performance degradation, if the percentage of unused words in a block is high. Also, it creates more pressure on the DRAM bandwidth to fetch the oversized cache blocks.

Opportunity: Figure 2a shows the fraction of prefetched blocks that are *compactable* with a stream prefetcher (with prefetch degree 8) enabled at the 512KB LLC. For this experiment, we try to compact multiple prefetched blocks (from 2 blocks to 32 blocks) into a single block. On an average, across 21 SPEC CPU 2006 benchmarks, 32.1% of the prefetched blocks can be compacted into a single block. For benchmarks such as *GemsFDTD* and *zeusmp*, more than 97% of the prefetched blocks can be compacted into a single block. On the other hand, there is limited opportunity for benchmarks such as *hmmer* and *lbm* in which almost all the prefetched blocks are uncompactable. Figure 2b shows the occupancy of prefetched blocks at the LLC. On an average, the occupancy is 42%. For benchmarks such as *milc*, *GemsFDTD*, *h264ref*, and *zeusmp*, more than 50% of the blocks at the LLC are prefetched, out of which more than 50% of the blocks can be compacted into a single block. This shows there is an opportunity to exploit this trend, and this forms the motivation for our work. The next section describes our technique (PBC) in detail.

3 Prefetch Blocks Compaction

This section describes three key parts of PBC:
(i) Indexing into a compact block through a simple modification to the cache organization. The change in organization provides access to multiple blocks that have different set indices but residing in a single compact block without incurring any additional latency, **(ii)** Compacting multiple contiguous prefetched blocks into a single block, and **(iii)** Decompacting a compact block.

3.1 Indexing into a Compact Block

Generic Cache Organization: To access the LLC, depending on the number of sets and the block size of the LLC, memory address is split into three fields: block offset, index and tag (from LSB to MSB). In this organization, two contiguous cache blocks get indexed to sets with consecutive indices. For example, for an LLC with a cache block size of 64 bytes, block X (with address A) and X+1 (with address A + 64) would be indexed into set Y and Y+1, respectively. When a processor generates a request, the index bits from the address select the set where the data might be cached. Then the tag part of the address is matched with the tags present in the selected set. If there is a hit, the data corresponding to

the tag is serviced.

In the generic cache, if we compact blocks X and X+1, and place it in set Y then when a processor generates a request for block X+1, set Y+1 would be searched for a matching tag, resulting in a cache miss even though block X+1 is actually residing in the cache in set Y.

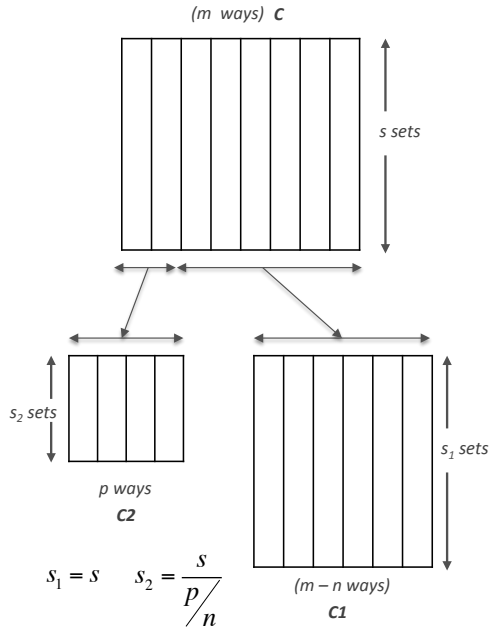


Figure 3: **Splitting of an *m*-way set associative cache having *s sets* into *C1* and *C2*.**

PBC Organization: PBC splits a generic cache (C) with *s sets* and *m ways* into two parts: *C1* and *C2*. Figure 3 shows this splitting. *C1* stores the non-compact blocks and *C2* stores the compact blocks. *C1* has s_1 *(s_1 = s) sets* with *m - n* ways (a reduction from *m* ways) and *C2* has s_2 sets. To minimize the number of conflict misses in *C2*, we increase the number of ways from *n* to *p* by reducing the number of sets from *s* to s_2. Together, the capacity of *C1* and *C2* is same as the generic cache (C).

Figure 4 shows the memory address split for a non-banked cache to access *C1* and *C2* using hash functions *h1* and *h2* respectively. *C1* is indexed using the same bits as in a generic cache. *C2* on the other hand is indexed using a combination of bits from the tag and the actual index bits. It is a concatenation of lower order bits from the tag and

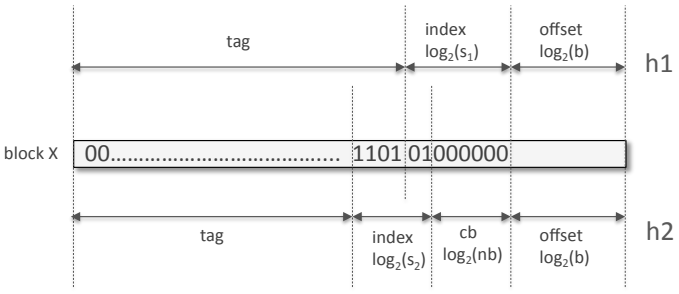


Figure 4: **Hash functions *h1* and *h2* when applied on block *X*. s_1: number of sets in *C1*, s_2: number of sets in *C2*, *nb*: number of blocks a compact block can hold, *b*: block size.**

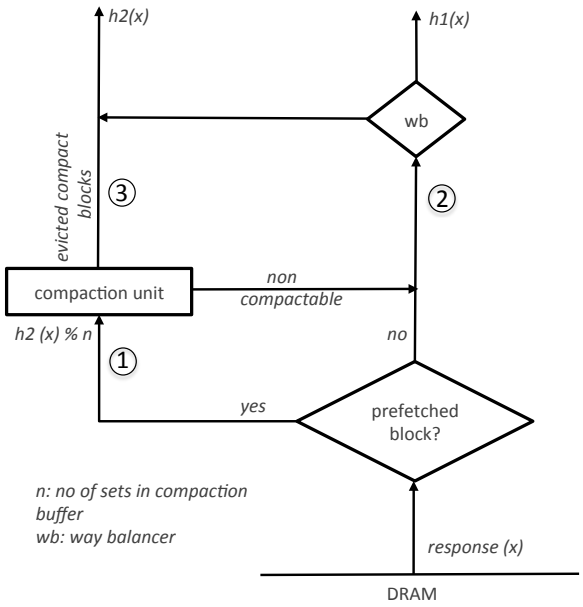


Figure 5: **Compaction process using PBC.**

higher order bits of what would have been the index bits in a generic cache. We refer to the unused lower order bits when *h2* is applied on an address, as *compact block bits* (*cb*) and the number of blocks a compact block can hold as *nb*. An entry in the tag store for a compact block consists of a single tag, *cb* of the first cache block, a valid bit, and a coherence bit for each block in the compact block.

Maximum number of blocks that can be compacted into a single block is limited by *cb*. However there is no lower limit on the number of blocks a compact block can hold. If at least two blocks can be compacted together, the resultant compact block is placed in *C2* and the other uncompact blocks are placed in *C1*.

Upon a request to the LLC, the LLC controller searches both *C1* and *C2* in parallel. In *C2* apart from matching the tag, the controller matches the *cb* bits that are stored along with the compact block's tag. If the *cb* from the request falls in the range of (*cb* : *cb* + $\log_2$ *(nb) - 1*) and if the individual cache block in the compact block is valid, then the LLC controller reports a hit. The range *cb* (*first block*) : *cb* + $\log_2$ *(nb) - 1* (*last block*) corresponds to the *cb* of all the blocks that are present within a compact block. A case where both *C1* and *C2* report a hit never arises.

3.2 Compacting and Decompacting Multiple Blocks

As noted in Section 2, compacting multiple prefetched blocks provides us with an opportunity to effectively utilize cache space. We use BDI to do so and though multiple blocks are compacted into a single block, the complexity and hardware involved is no more than compressing/decompressing a single block of data using BDI. PBC is *not* a compression technique. It can be used with any compression technique proposed in literature such as BDI and decoupled compressed cache (DCC) [4] to improve the system performance even further.

Compaction: In compression techniques, data responded back from the DRAM passes through the compressor unit and from thereon, the compressed/uncompressed

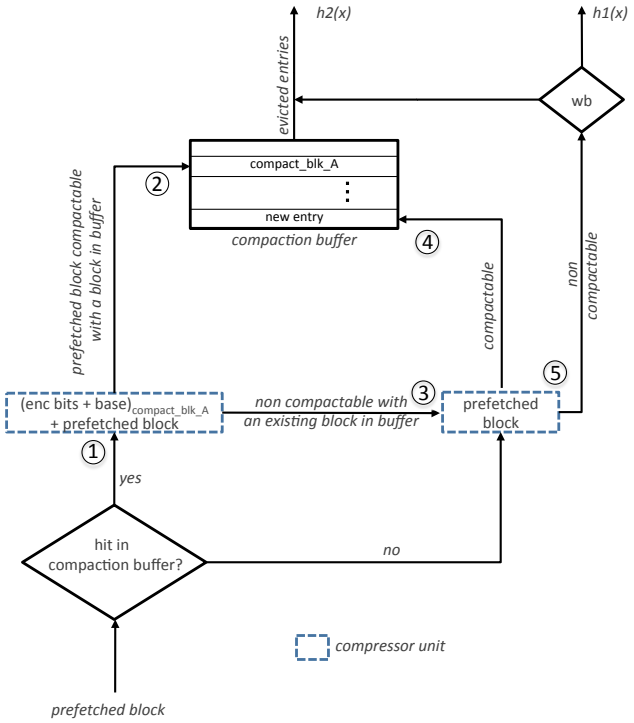


Figure 6: **Internals of a compaction unit.**

block is placed in the LLC. In PBC, when a prefetched block is responded back from the DRAM, it cannot be directly placed into the LLC after compression, as this block can potentially be compacted with other blocks and stored as a single block.

To achieve compaction, we use a small 2-way set associative buffer, referred to as *compaction buffer*, indexed using *h2(x)* index bits, where *x* is the address of the prefetched block. The tag portion of the compaction buffer has four fields: *h2(x)* tag, *h2(x)* index, *h2(x)* cb and *encoding* bits. Multiple non-contiguous cache blocks with different *h2(x)* indices, which share a common *h2(x)* tag, might get compacted into a single compact block (which should hold only those blocks that share a common *h2(x)* index). To avoid this scenario, *h2(x)* index is also stored along with the *h2(x)* tag.

Figure 5 shows the overall compaction process. A prefetched block responded from the DRAM is fed into the compaction unit (①). Non-compactable cache blocks and demand blocks are placed in *C1* or *C2* depending on the outcome of the way balancer (②). Evicted compact blocks from the compaction unit are placed in *C2* (③).

Figure 6 shows the compaction unit in detail. The compaction unit contains a compressor and a compaction buffer. An incoming prefetched block is indexed into the compaction buffer using *h2(x)*.

A hit at the compaction buffer would imply that an opportunity to compact this incoming prefetched block (*x*) with an existing compact block exists. (For example, *compact_blk_A*, which might already be holding several other blocks, all sharing the same *h2(x)* tag and index.) The base and the encoding bits from the compact block are read and passed into the compressor unit along with the prefetched block (①). The compaction unit tries to compress the prefetched block using the same pattern as that of the compact block. If the prefetched block fits into the compact block, it is compacted and placed within the compact block *compact_blk_A*. (②)

If the prefetched block does not fit into the compact block, the compact block is evicted from the compaction buffer. The prefetched block is sent again to the compressor unit (this time, to exploit the data pattern present within the block alone) (③) and from thereon to the compaction buffer (④) or the way balancer (⑤) depending on whether or not it is compressible.

On a miss at the compaction buffer, if the incoming prefetched block is compressible, a victim compact block is evicted (FIFO) from the buffer and the compressed prefetched block is allocated in its position. If the prefetched block is incompressible, it is sent to the way balancer.

Figure 7 illustrates the compaction process further, by considering an example of an empty compaction buffer and three prefetched blocks (Y, Y+1, and Y+2), responded from the DRAM. The blocks share a common tag and index when *h2* is applied. Block Y, being a miss at the compaction buffer, is compacted by determining the pattern present within the block. The encoding bits are set correspondingly. Block Y+1 and Y+2 result in a hit at the buffer. They are compacted using the base and the encoding bits present in the compact block Y. In future, any request to a block in the compact block (example Y+1, either in *C2* or compaction buffer) is responded by decompacting using the base, encoding bits and the corresponding delta present in the compact block.

Decompaction: On a hit in *C2*, PBC decompacts the block using the encoding bits stored along with the tag and responds with the data. In BDI, each word occupies a fixed location in a compressed block. In PBC this translates to each cache block occupying a fixed location in the compact block. PBC reads the base of the compact block and the words that are needed for reconstructing the block, and sends it to the decompresser unit to get the decompressed cache block. Note that though multiple blocks reside in a compact block, the decompaction latency is no more than

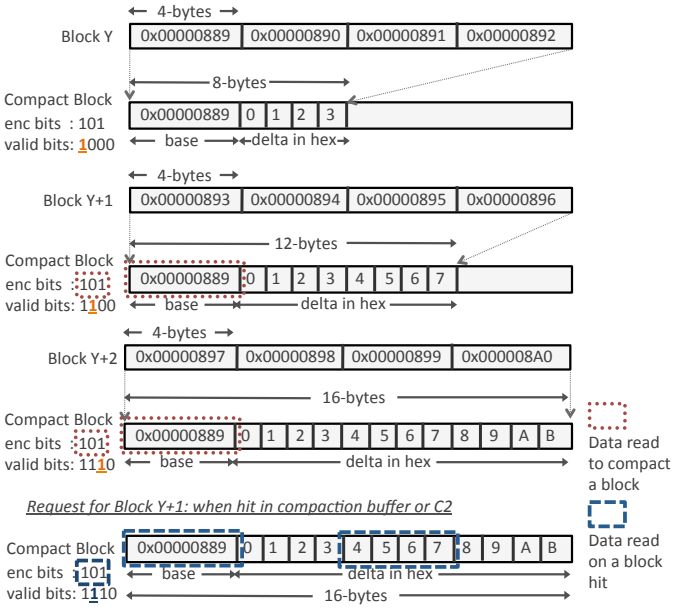


Figure 7: **An example illustrating compaction and decompaction in PBC.**

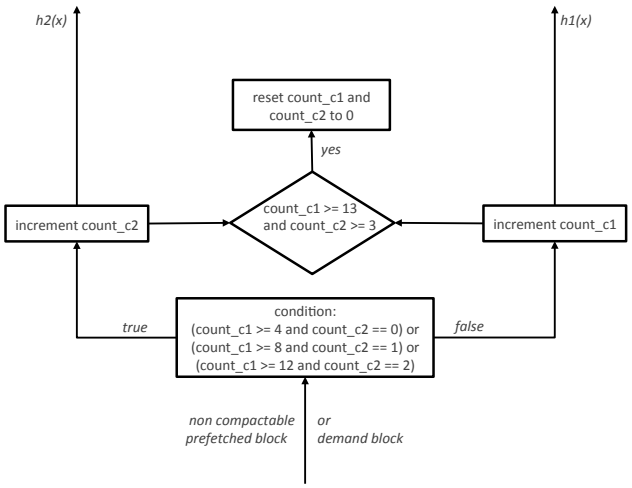


Figure 8: **Internals of a way balancer. The counter values shown are for a cache (*C*) with 16 ways (*m*) being split into *C1* with 13 ways (*m - n*) and *C2* using those 3 ways (*n*).**

the decompression latency of a single cache block. With PBC, the access latency increases by one cycle, which comes from the tag search (the number of parallel tag searches are increased when compared to a generic cache). If there is a cache hit, to service the demand requests, the latency increases by one more cycle (a total of two cycles, which includes the tag search and the decompression latency). In PBC, we don't store dirty blocks inside a compact block, and therefore a write hit does not require re-compaction.

Handling Writebacks: Upon a writeback, an evicted dirty block from an upper level cache writes its data back to the LLC. A writeback hit for a block in a compact block at *C2* necessitates a need for re-compaction of the entire compact block since the data contained in the block has been modified. To avoid this, in PBC, any writeback hit at *C2* invalidates that particular block in the compact block by resetting the corresponding *valid* bit, and the dirty block is allocated in *C1*. *Note, since we don't store dirty blocks inside a compact block, we need only a single coherence bit to distinguish between shared and exclusive coherence states per block.*

Encoding bits in PBC: In PBC, a cache block is either stored as part of a compact block or as an uncompressed block. If a 64-byte cache block is compressible to 34, 36 or 40 byte block, using B2-D1, B4-D2 or B8-D4 respectively (Table 1), the size of the resultant compact block, when a new block is compacted with this compressed block, would be greater than 64 bytes. Hence, we do not use compress patterns 5, 7 or 8 while compacting blocks using PBC and would require only three bits to encode the rest of the patterns.

Way Balancer: In PBC, the cache space will be under-utilized if an application either generates very few prefetch requests or the percentage of prefetched blocks that can be compacted is very less. In such scenarios, very few blocks get placed in *C2*. This leads to performance degradation. To avoid this, we incorporate a way balancer (wb) – a control logic that keeps track of the number of blocks that get placed in *C1* and *C2* using two saturating counters. If for *m - n* blocks that are allocated in *C1*, *n* compact blocks are allocated in *C2*, the counters are reset. If fewer than *n* compact blocks are allocated in *C2*, the way balancer allocates the demand blocks or the uncompressed prefetched blocks to *C2* such that for every *m - n* blocks allocated in *C1*, *n* blocks are allocated in *C2*. Note that *C2* can also hold a non-compacted block. (The encoding bits will be set to represent the same).

In general, for a given m and n ($n < m$), we find x such that:

$$x = \lfloor \frac{m}{n} + \frac{1}{2} \rfloor$$

In other words, x is m/n rounded off to the nearest integer. Using this x, the decision to allocate the block in either *C1* or *C2* is made as follows:
For some $i \in [1, n]$,
$if\ (count\_c1 \geq i * x\ \&\&\ count\_c2 == i - 1)\ then$ allocate to *C2* $else$ allocate to *C1*, and update the counters accordingly. The counters are reset when $count\_c1 \geq m$ and $count\_c2 \geq n$.

Figure 8 shows the internals of a way balancer with *m* = 16 and *n* = 3. The counter values are selected in such a way that, the way balancer tries to allocate n blocks to *C2* over a phase of *m - n* blocks being allocated to *C1*.

3.3 Design Choices

Placement of the compaction buffer: The compaction buffer can be placed at the DRAM. This results in off-chip memory bandwidth savings, as fewer bytes will be transferred from DRAM to LLC due to compaction. On the flip side, a demand request generated for a block cached at the compaction buffer will end up in the miss status handling register (MSHR) of LLC and will continue to remain so until the compacted block is serviced back from the DRAM controller. So even though the data is read from the DRAM, the demand request for a block can end up being a miss at the LLC, which degrades performance. To tackle this issue, hints from the MSHR can be sent to the compaction unit placed at the DRAM, at regular intervals. The hints notify the compaction unit about those misses which were initially prefetch requests, but are now requested by the processor (demand misses). The compaction unit would then service these requests immediately. The other place for the compaction buffer is beside the LLC, where it is searched in parallel with the LLC. In this work, for the evaluation of PBC we take the latter approach.

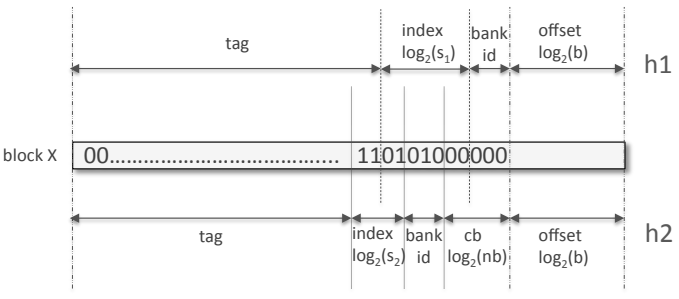


Figure 9: **Memory address split for a banked cache.**

LLC size	s	s_1	s_2	m	n	p
2MB	2048	2048	1024	16	3	6
4MB	4096	4096	2048	16	3	6
8MB	8192	8192	4096	16	3	6

Table 2: **Various cache parameters used in PBC.**

C2 cache	1024 sets, 6 ways
#compact blocks in C2 (NC)	6144
#blocks within a compact block (NB)	32
compact block size at the compaction buffer	64 bytes
Metadata per compact block	3/1/1 enc/v/coh bits
Entries in the Compaction buffer (NE)	256 entries
***h2* tag, *h2* index, cb**	11, 10, and 5 bits
Access latency of Compaction buffer	3 cycles
Decompaction latency	2 cycles

Table 3: **Parameters specific to PBC for a 2MB LLC. enc - encoding bits, v - valid bit, coh - coherence bits.**

PBC for a banked cache organization: In a generic banked cache organization, consecutive cache blocks are mapped to consecutive banks. For example, two consecutive cache blocks X and X+1 are mapped to consecutive banks B0 and B1, respectively. In PBC, each bank is split into *C1* and *C2*, similar to a non-banked cache as illustrated in Section 3. Figure 9 shows the memory address split for a banked cache when *h1* and *h2* is applied to incorporate PBC. For indexing into *C1*, we retain the hash function of a generic cache organization (*h1*). Since *C2* contains compact blocks, higher order bits (after cb bits from LSB) are used to determine the bank id and the set index as shown in *h2*. When a cache block is requested at the LLC, both *C1* and *C2* are searched in parallel (It might so happen that *C1* from one bank and *C2* from a different bank get searched in parallel). A case where two banks report a hit does not arise because a block can be stored only in *C1* or *C2*, irrespective of the bank chosen. Note that with this organization, multiple consecutive blocks mapping to different banks in a generic cache can be compacted as a single block and mapped to a bank chosen by *h2(x)*. To be able to handle multiple cache blocks, which span across multiple banks, the compaction unit is placed at the DRAM controller. Depending on the outcome of the compaction unit, the cache blocks are sent to *C1* or *C2* of a particular bank. In this work for the evaluation of the proposed idea, we consider a non-banked cache where we place the compaction buffer at the LLC.

LLC Parameters specific to PBC: Table 2 shows the values of s_2, n, and p for the fixed values of s, s_1, and m for various sizes of LLC. In our simulations, we find that these values are best suited (determined empirically by sweeping through the various values of n and p), for a wide range of workloads. We consider a 256 entry, 2 way set associative compaction buffer. The decompaction latency of PBC is same as the decompression latency of BDI.

3.4 PBC with cache compression

As noted in Section 3.2, PBC uses BDI internally to compact prefetched blocks. Any cache compression technique can be incorporated both at *C1* and *C2*. *C1*, storing demand blocks and uncompacted prefetched blocks is like any other generic cache that can be compressed. In *C2*, a compact block, even after holding several other blocks, might not span the entire space allocated for a block. Also, the demand blocks and uncompacted prefetched blocks sent to *C2* by the way balancer can be compressed. By doubling the tag array at both *C1* and *C2*, as done in BDI, for a given cache space a maximum of double the number of compacted/uncompacted blocks can be allocated. Instead of BDI, if any other compression technique is used to compress multiple cache blocks (i.e., for compaction), then the decompression latency increases and is variable. For example, if the third block in a compressed block is requested then both the first and second blocks need to decompressed. Note that it is feasible to use a compression algorithm at *C1* that is different from the one used for compaction of the blocks at *C2*.

3.5 Hardware overhead

Table 3 shows the parameters specific to PBC. Table 4 shows the hardware overhead for a 2MB LLC. The total hardware overhead of PBC is 70.76kB, 3.4% of the baseline LLC of 2MB. In terms of tag storage, *C2* uses a tag store of 58.5KB (including the *C2* overhead), that is around 0.95 times the tag store of the baseline cache (61.4kB). PBC is a low-cost technique that provides significant performance improvement.

C2	NC × (((v + coh) × NB) + enc)	51.4 kB
Compaction Buffer	NE × (*h2* tag + *h2* index + cb + enc bits) + NE × (compact block size + 8) bytes	19.36 kB
Total Overhead		70.76kB

Table 4: **Hardware Overhead of PBC on a 4-core system with 2MB LLC. For NB, NC, and NE, refer Table 3.**

4 Evaluation Methodology

We use gem5 [12] simulator to simulate 4- and 8-core system running multi-programmed workload mixes. We create seventy 4-core and twenty-five 8-core workload mixes by mixing benchmarks from SPEC CPU 2000 and 2006 benchmark suites [9] based on the characteristics as shown in Table 5. We evaluate the effectiveness of PBC over a baseline system with no compaction with a per-core stream prefetcher ON. Next we evaluate the benefits of using PBC along with the state-of-the-art cache compression technique, BDI. We also show the effectiveness of a system that uses PBC with stride prefetcher and the state-of-the-art SMS [18] prefetcher. We measure the system performance in terms of weighted speedup (WS) [11] (higher the better) and harmonic mean of speedups (HS) [10] (higher the better), and system fairness using an unfairness (lower the better) metric [22]. We also report the off-chip bandwidth consumption in terms of GB/sec. The metrics of interest are defined as follows:

$$IS_i = \frac{CPI_i^{together}}{CPI_i^{alone}}, \quad WS = \sum_{i=0}^{N-1} \frac{IPC_i^{together}}{IPC_i^{alone}}$$
$$HS = \frac{N}{\sum_{i=0}^{N-1} \frac{IPC_i^{alone}}{IPC_i^{together}}} \tag{1}$$
$$Unfairness = \frac{MAX\{IS_0, IS_1, ..., IS_{N-1}\}}{MIN\{IS_0, IS_1, ..., IS_{N-1}\}}$$

HS is the reciprocal of the average normalized turn-around time and WS is equivalent to system throughput [17]. HS balances both performance and fairness. $\text{IPC}_i^{together}$ is the IPC of core i when it runs along with other N-1 applications on a multi-core system of N cores. IPC_i^{alone} is the IPC of core i when it runs alone on a multi-core system of N cores. We fast-forward 10 billion instructions, warm-up the system for

Compaction Friendly (CF)	bwaves, GemsFDTD, h264ref, mesa, milc, and zeusmp
Cache Sensitive (CS)	bzip2, mgrid, soplex, twolf, vortex2, omnetpp, and leslie3d
CS and CF	ammp and galgel
Neither CS nor CF	gromacs, hmmer, lbm, mcf, sjeng, and calculix

Table 5: **Benchmark characteristics. Compaction friendly: If more than 40% of the prefetched blocks are compactable. Cache sensitive: If the ratio of improvement in performance (IPC) by going from 1MB to 2MB LLC is greater than 10%.**

Processor	4-core and 8-core CMP, 4.7 GHz, Out of Order
Fetch/Commit width	8
ROB/LQ/SQ/Issue Queue	192/96/64/64 entries
L1 D/I Cache	32 kB, 4 way, 3 cycle latency
LLC	2/4 MB for 4/8 cores, 16 way, 26 cycle latency
MSHR entries	16, 64/128 at L1, LLC with 4/8 cores,
Cache Line Size	64B
Prefetchers at LLC	Stream Prefetcher, with degree = 8 and distance = 64
	Stride Prefetcher, with degree = 8 and distance = 64
	SMS Prefetcher, with region size of 2KB
DRAM Controller	On-chip, Open Row, 64 read & write queue entries, FR-FCFS scheduler
DRAM Bus	split-transaction, 800 MHz, BL=8
DRAM	DDR3 1600 MHz (11-11-11) 1/2 channels for 4/8 cores, Peak Bandwidth = 12.8 GB/s

Table 6: **Parameters of the simulated system.**

250 million instructions, and collect statistics for the next 250 million instructions, which is similar to earlier works [15] [16]. Table 6 provides the details of system parameters used in our evaluation.

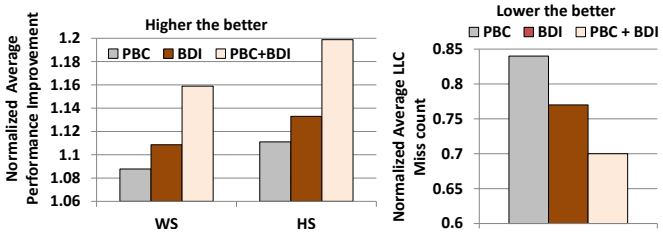


Figure 10: **Normalized average WS and HS comparison and normalized average LLC miss count comparison, of seventy 4-core workloads on a 4 core system.**

4.1 Results and Analysis

This section describes the effectiveness (in terms of HS and WS) of PBC and PBC + BDI for 4-core and 8-core systems. PBC+BDI is the combination of PBC, which compacts prefetched blocks, and BDI, which compresses a single cache block. In case of PBC, we do not use any compression techniques that compresses single cache blocks. We compare PBC and PBC+BDI with a baseline system with no compression, and with a stream prefetcher ON. We create seventy and twenty-five 4-core and 8-core workload mixes, respectively. We create these mixes randomly by mixing the benchmarks from each category in roughly equal proportions (which we mention in Table 5). Figure 10 shows the normalized average (geomean) of HS and WS and LLC miss counts of PBC, BDI and PBC+BDI, on a 4-core system. The average is across seventy workloads with a stream prefetcher ON. PBC improves the HS and WS by 11.1% and 8.7% respectively. PBC delivers this performance by reducing the average LLC miss count by 16%. The maximum improvement (in terms of HS) of 43.4% comes from a workload that consists of *ammp-leslie3d-vortex2-zeusmp*. The workload that consists of *mcf-soplex-omnetpp-gromacs* shows the minimum improvement of 0.1%.

BDI delivers performance improvement of 13.29% (HS) and 10.8% (WS) with an average reduction of LLC miss count by 23.6%. PBC, when combined with BDI (PBC+BDI) delivers 15.8% and 19.8% improvement in the average WS and HS, respectively, with an average reduction of LLC miss count by 29.2%. PBC+BDI provides a maximum improvement of 71.2% (in *ammp-soplex-twolf-zeusmp*) and minimum improvement of 3.24% (in *mcf-soplex-omnetpp-gromacs*).

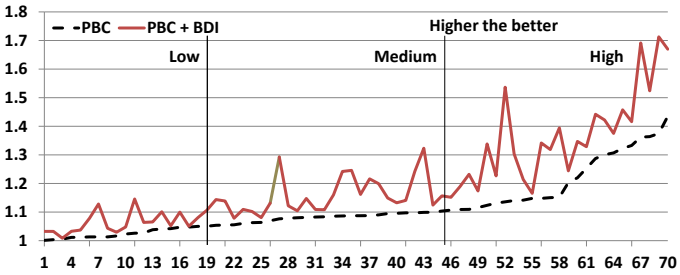


Figure 11: **Performance improvement of PBC (in terms of HS) across seventy 4-core workloads on a 4 core system.**

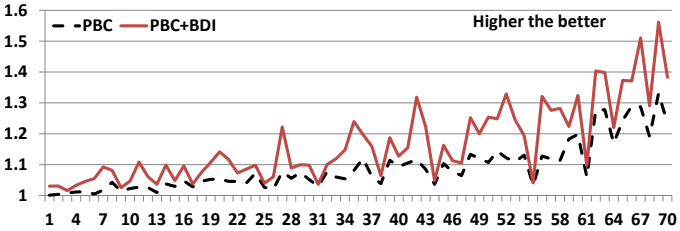


Figure 12: **Normalized WS for 4-core workloads. The workloads are ordered as per the order shown in Figure 11.**

Figure 11 shows the detailed improvement in HS delivered by PBC and PBC+BDI, for all 70 4-core mixes. We show the mixes in the increasing order of their HS improvement with PBC and divide them into low, medium, and high performance zones based on the performance improvement (in terms of HS). An improvement of less than 5% is termed as *low*, improvement between 5 to 10% is *medium*, and the mixes with more than 10% improvement with PBC are termed as *high*. Across all the workload mixes, PBC improves the performance. Out of seventy 4-core workload mixes, with PBC, 18 mixes show an improvement of less

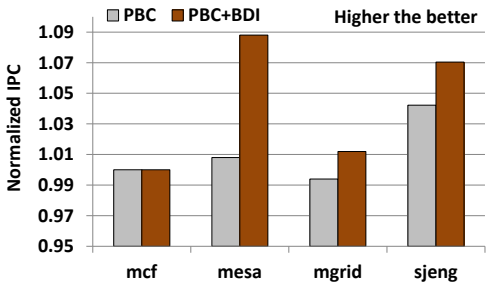


Figure 13: **Workload Mix 5 - Individual IPC**

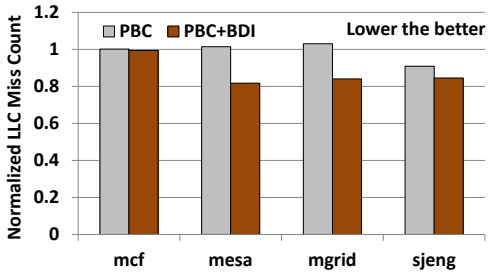


Figure 14: **Workload Mix 5 - Individual LLC Miss Count**

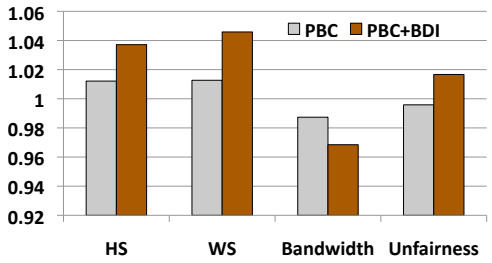


Figure 15: **Workload Mix 5 - Normalized HS, WS, bandwidth consumption, and unfairness.**

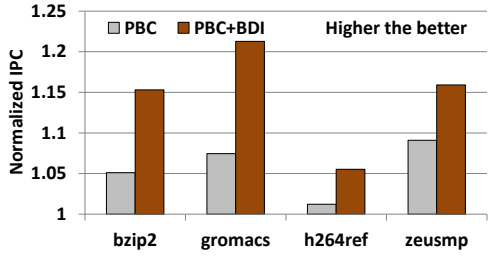


Figure 16: **Workload Mix 20 - Individual IPC**

than 5% (low performance zone), 26 mixes show an improvement in between 5-10% (medium performance zone), and rest (26 workload mixes) show an improvement of greater than 10% (high performance zone). The HS curve looks encouraging as 52 out of 70 mixes (74.2%) show an improvement of more than 5%. Figure 12 shows the improvement in WS for the same seventy 4-core workloads. Next, we analyze two workload mixes from each of the three performance zones. Our observations are as follows:

Case I: Low performance zone: Workload mix 4 (*gromacs-mcf-milc-vortex2*) shows very low performance improvement (1.01%) with PBC. The primary reason for this behavior is that all the applications of this mix contain very few compactable blocks, which limits the opportunity for PBC.

Next, we consider workload mix 5 that contains *mcf*, *mesa*, *sjeng*, and *mgrid*. In terms of HS, PBC improves the performance by 1.2% and BDI improves the performance by 2.6%, and the combination of PBC and BDI (PBC+BDI) provides a performance improvement of 3.7%. Figure 13 and Figure 14 show the improvement in individual IPCs and reduction in the LLC miss counts for PBC and PBC+BDI, respectively. As *mesa* is compaction-friendly and compressible, the combination of PBC+BDI provides an LLC miss count reduction of 22.3% resulting in 8.8% improvement in individual IPC. This behavior of *mesa* helps *sjeng*. PBC+BDI provides additional cache space for *sjeng*, which results in 18.3% reduction in LLC miss count and 7% improvement in individual IPC. Figure 15 shows the improvement in HS and WS. It also shows the reduction in bandwidth consumption and reduction in unfairness for workload mix 5. Both PBC and PBC+BDI reduce the bandwidth consumption, and there is slight increase (less than 2%) in terms of unfairness.

Case II: Medium performance zone: Next, workload mix 19 (*mcf-soplex-vortex-zeusmp*) shows an improvement of 7.6%. It contains *mcf*, that is neither cache sensitive nor compaction friendly, and *soplex*, that is cache sensitive but its prefetched blocks occupancy is less than 2% at the LLC. On the other hand, for benchmarks, such as *vortex2* and *zeusmp*, PBC improves their respective IPCs by 7.4% and 7.9% by reducing their LLC miss rate by 12.3% and 23.3%.

We consider workload mix 20, which contains *bzip2*, *gromacs*, *h264ref*, and *zeusmp*. In terms of HS, PBC and PBC+BDI provide an improvement of 5.3% and 14.3%, respectively, and in terms of WS, PBC and PBC+BDI provide an improvement of 5.3% and 14.1%, respectively. Figure 16 and Figure 17 show the improvement in individual IPCs and reduction in the LLC miss counts for PBC and PBC+BDI, respectively. All the benchmarks of this workload mix except *h264ref* are the benefactors of PBC and PBC+BDI as PBC+BDI improves the individual IPCs by 15.3%, 21.2%, and 15.9% for *bzip2*, *gromacs*, and *zeusmp*, respectively. In terms of LLC miss counts, PBC provides an LLC miss count reduction of 24% and 27% for *gromacs* and *zeusmp*,

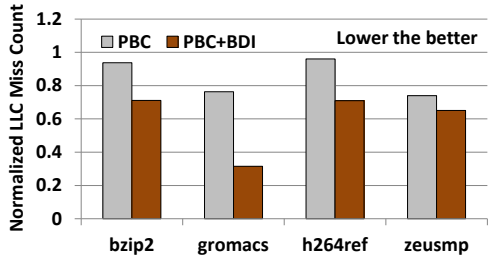


Figure 17: **Workload Mix 20 - Individual LLC Miss Count**

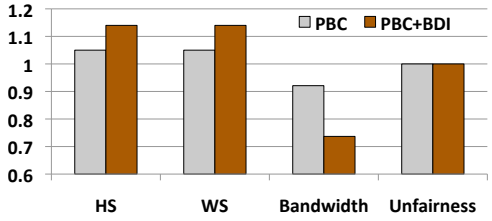


Figure 18: **Workload Mix 20 - Normalized HS, WS, bandwidth consumption and unfairness.**

respectively. The reduction goes further with PBC+BDI as it reduces the individual LLC miss-counts by 69.5% and 35% for *gromacs* and *zeusmp*, respectively. This reduction in LLC miss count helps *bzip2* in improving its own performance (15.3%). *h264ref*, also gets benefit from the additional space, which results in 1.2% and 5.5% improvement in its IPC for PBC and PBC+BDI, respectively. Figure 18 shows the improvement in HS, WS, bandwidth consumption, and unfairness. In terms of unfairness, both PBC and PBC+BDI are equally fair as compared to the baseline. PBC+BDI improves both HS and WS by at-least 10% with 25% reduction in the bandwidth consumption.

Case III: High performance zone: We consider workload mix 49 (*galgel-mesa-sjeng-soplex*), which has two benchmarks that are cache sensitive (*galgel* and *soplex*) and two that are compaction friendly (*mesa* and *galgel*), *galgel* being both cache sensitive and compaction friendly. *sjeng* is neither cache sensitive nor compaction friendly, but there is an overall improvement in the system (by 14%) because of the other three benchmarks. PBC reduces the miss-rate of *galgel*, *mesa*, *sjeng*, and soplex by 34%, 12.4%, 14%, and 10.8%, respectively, which results in respective IPC improvements of 33%, 2.1%, 2.9%, and 11.5%.

Workload mix 70 (*ammp-leslie3d-vortex2-zeusmp*) has both cache sensitive (*leslie3d*, *vortex2* and *ammp*) and compaction friendly (*ammp* and *zeusmp*) benchmarks resulting in 43.4% and 66.9% improvement in HS with PBC and PBC+BDI, respectively. In terms of WS, PBC and PBC+BDI provide an improvement of 23.4% and 38.3%, respectively. Figure 19 and Figure 20 show the improvement in individual IPCs and reduction in the LLC miss counts for PBC and PBC+BDI, respectively. PBC results in significant improvement in the IPC of *ammp* and *leslie3d* by 81.6% and 27.9% respectively. This improvement in IPC comes from their reduction in the individual LLC miss count, which is in the magnitude of 2.4X and 1.6X. When combined with BDI, PBC+BDI provides a reduction of LLC miss count by

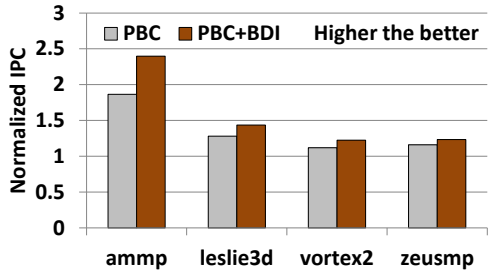


Figure 19: **Workload mix 70 - Individual IPC**

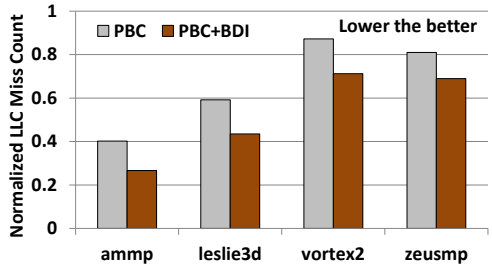


Figure 20: **Workload mix 70 - Individual LLC Miss Count**

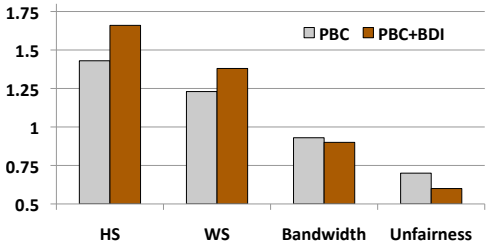


Figure 21: **Workload mix 70 - Normalized HS, WS, bandwidth consumption, and Unfairness.**

3.7X and 2.2X, respectively. This mix is a good example that shows the potential of PBC. Figure 21 shows the improvement in HS and WS, and reduction in bandwidth and system unfairness.

In workload mix 1 (*mcf-soplex-omnetpp-gromacs*), *omnetpp* is cache sensitive but not compaction friendly and the rest are neither cache sensitive nor compaction friendly. In this mix, we do not expect PBC to deliver noticeable performance improvement as there is no opportunity and PBC improves HS by a marginal 0.1%.

We conclude that PBC delivers high performance gain in general if the workload mix has more than one cache sensitive benchmark and at-least one compaction friendly benchmark.

PBC and BDI complement each other across all the workloads, which results in significant performance improvement of more than 20% (in terms of HS) in 27 out of 70 workload mixes, and 53 out of 70 workload mixes show a performance improvement of more than 10%.

Bandwidth Consumption: On an average across 70 workloads, PBC reduces bandwidth consumption (GB/sec) by 6% and the combination of PBC and BDI (PBC+BDI) reduces the bandwidth consumption by 11.8%. Both PBC and PBC+BDI provide a good tradeoff between performance and bandwidth consumption as PBC provides a performance improvement of 11.1% with a reduction in bandwidth consumption of 6%, and PBC+BDI provides an improvement of 19.8% with a reduction in bandwidth consumption of 11.2%. Figure 22 shows the normalized bandwidth consumption for 70 workload mixes. Out of 70 workloads, only one workload (workload mix 6) consumes additional bandwidth, which is marginal (1.2%).

System Fairness: The primary goal of PBC is to improve

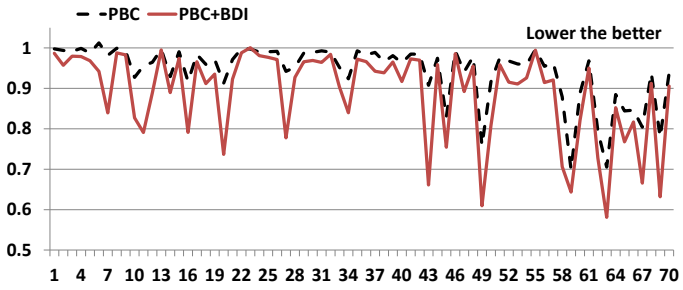


Figure 22: **Normalized Bandwidth Consumption for 4-core workloads in GB/sec. The workloads are the sorted workloads, which are sorted as per their improvement in HS with PBC as shown in Figure 11.**

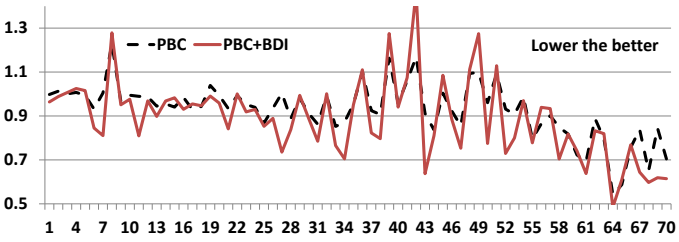


Figure 23: **Normalized system unfairness for 4-core workloads. The workloads are the sorted workloads, which are sorted as per their improvement in HS with PBC as shown in Figure 11.**

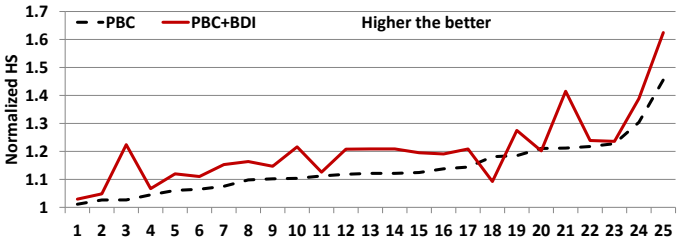


Figure 24: **Normalized HS for 8-core workloads.**

the system performance but it also improves the system fairness. On an average, across 70 4-core workloads, PBC reduces the unfairness by 9.2%. The primary reason behind this improvement is that the additional cache space provided by compaction friendly applications help cache sensitive ones. PBC+BDI goes further in improving the fairness by reducing unfairness by 12.7%. Out of 70 workloads, only 12 workloads show increase in unfairness. Figure 23 shows the trend in terms of unfairness for 70 4-core workloads.

8-core results: For an 8-core system, on an average (geomean), when compared to the baseline, PBC improves the performance by 13.6% (HS) and 11.8% (WS), with a

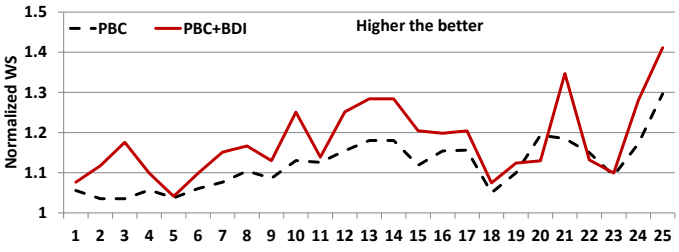


Figure 25: **Normalized WS for 8-core workloads.**

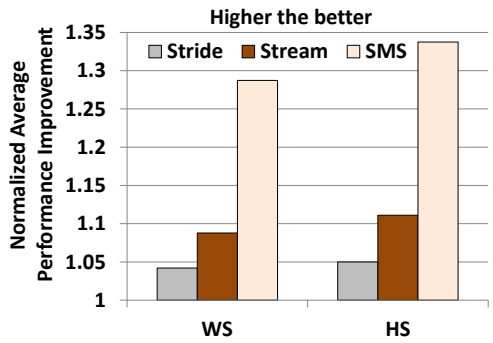


Figure 26: **PBC with stride, stream, and SMS prefetcher.**

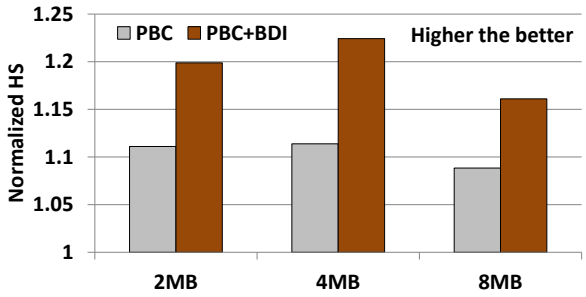


Figure 27: **Effectiveness of PBC with different LLC size. The improvement shown is over the baseline cache of the same size.**

maximum HS improvement of 45.5% (in *ammp-galgel-h264ref-leslie3d-mcf-soplex-vortex2-zeusmp*) and a *minimum* HS improvement of 1.1% (in *leslie3d-mcf-mesa-sjeng-soplex-twolf-vortex2-zeusmp*). Hence, we find that PBC is effective even with the increase in core count. Figure 24 and Figure 25 show the normalized HS and normalized WS for 25 workloads. The workloads are sorted as per the performance improvement in terms of HS, with PBC.

4.2 Other Results

PBC with Different Prefetching Techniques: Till now, we showed the effectiveness of PBC with a stream prefetcher. Now we show the performance improvement of PBC with simple *stride* [19] and state-of-the-art SMS [18] prefetcher in Figure 26. For our evaluation, we use a stride prefetcher with prefetch degree of 8, which is same as the prefetch degree that we use for evaluating PBC with *stream* prefetchers. For SMS, we use spatial regions of size 2KB (thirty two 64-byte cache blocks), which is optimal for SPEC based applications. So on a miss, an SMS prefetcher can prefetch up to 32 cache blocks in one instance.

Compared to the baseline with a *stride* prefetcher, on an average across seventy 4-core workload mixes, PBC improves the WS and HS by 4.2% and 5%, respectively. This is due to the low prefetch issue rate, as a simple stride prefetcher fails to exploit the access patterns that are non-strided. On the other hand, PBC with an SMS prefetcher outperforms the baseline by an average improvement of 28.7% in WS and 33.7% in HS. The primary reason for this improvement is the high prefetch issue rate, which results in high compaction for compaction friendly benchmarks, and this provides additional LLC space for the cache sensitive benchmarks to exploit.

We conclude that PBC is effective across a wide range of prefetching techniques. The effectiveness increases with the increase in the prefetch issue rate.

LLC sensitivity: Figure 27 shows the effect of LLC size on the performance of PBC with a stream prefetcher. For a 4-core system, we consider 2MB, 4MB and 8MB LLC. PBC improves the performance (HS) by 11.3% and 8.8% and PBC+BDI provides an improvement (HS) by 22.4% and 16% for 4MB and 8MB LLC respectively. Compared to 4MB, the effectiveness of PBC goes down with an 8MB LLC as larger working set of most of the applications fit at the LLC.

PBC with FDP: FDP [20] is a prefetcher aggressiveness controller, which tries to improve the performance by

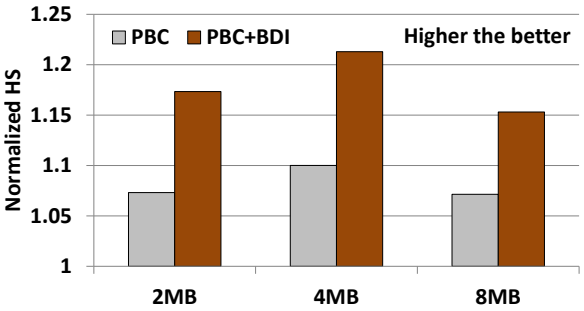


Figure 28: **Performance improvements of PBC and PBC+BDI with a 3 level cache hierarchy. L2: 256KB and LLC: 2MB/4MB/8MB. The improvement shown is over the baseline cache of the same size.**

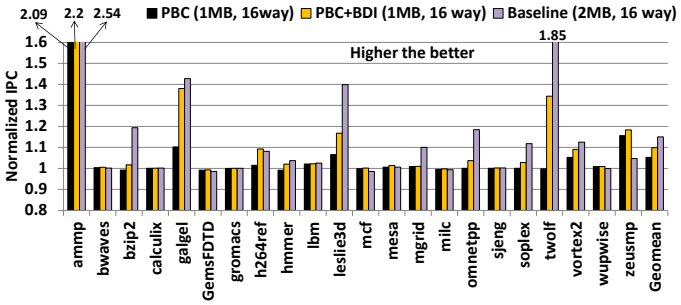


Figure 29: **Performance improvement in terms of IPC with PBC and PBC+BDI for single-core workloads. Note that the baseline contains a 1MB cache with 16 ways.**

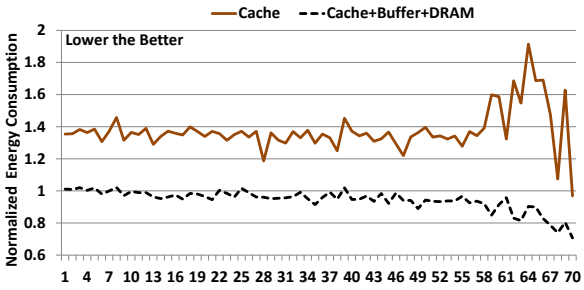


Figure 30: **Normalized energy consumption for 4-core workloads. Workloads are ordered as per the order in Figure 11.**

throttling the prefetchers, based on prefetch metrics such as prefetch-accuracy, timeliness, and cache pollution. It uses five throttling levels: [1,4], [2,8], [2,16], [4,32] and [4,64] where [x,y] is a throttling level with *prefetch-degree* of x and *prefetch-distance* of y. Throttling up/down corresponds to increase/decrease in the aggressiveness by one level. For multi-core systems that employ FDP, PBC delivers performance improvements, in terms of WS and HS by 5.3% and 9.5% with a maximum of 27.4% and 40% of WS and HS. This shows that PBC is still effective in the presence of prefetch throttling techniques, such as FDP. PBC provides better performance for those workload mixes that contain applications with high prefetch accuracy, which stay at the highest throttling level. However, for workloads that contain applications with low prefetch accuracy, there is lesser opportunity to compact multiple prefetched blocks as the prefetchers stay at the lowest throttling levels.

PBC for 3 levels of cache: We evaluate PBC for a 3 level cache hierarchy with PBC applied at the last-level-cache (LLC) and private L2s (prefetcher turned on at both L2 and LLC). PBC remains effective for a 3 level cache, with an average performance improvement in terms of HS is 10% (256kB L2 cache and 4MB LLC). Figure 28 shows the performance improvements of PBC and PBC+BDI. Similar to figure 27, when compared to 4MB, the effectiveness of PBC goes down with an 8MB L3 as larger working set of most of the applications fit at the L3 cache.

PBC for 1-core system: Figure 29 shows the improvement in IPC for 22 SPEC benchmarks. We also report the performance improvement that can be gained by doubling the cache size from 1MB to 2MB in a baseline system. On an average (geomean), doubling the cache capacity results in 15% performance improvement. Compared to a baseline system with 1 MB cache, PBC and PBC+BDI provide performance improvements of 5.2% and 9.8%, respectively. Benchmarks, such as *bwaves*, *calculix*, *GemsFDTD*, *gromacs*, *mcf*, *mesa*, *milc*, *sjeng*, and *wupwise* that show no improvement, are the cache insensitive ones and do not gain performance even with double the cache size.

Hit rates of C1 and C2: The average hit rates of *C1* and *C2* are 58% and 11% with a maximum of 77% *C1* hit rate for a workload mix that contains *gromacs-soplex-sjeng-vortex2* and minimum of 20% for the workload mix that contains *GemsFDTD-mcf-omnetpp-galgel*. For *C2*, the maximum and minimum hit rates are 23% and 4% for workload mixes that contain *h264ref-GemsFDTD-omnetpp-milc* and *soplex-milc-bzip2-hmmer*, respectively. For a cache with 16 ways being split into *C1* with 13 ways and *C2* using the other 3 ways, we expect the ratio of the hit rates (*C2* to *C1*) to be roughly around 3/13 and found the same in our evaluations. High hit rates at *C2* would imply more number of cache blocks being compacted into a single block.

Effectiveness of the way balancer: To quantify the effectiveness of the way balancer, we run PBC without the way balancer. Compared to the baseline, PBC without the way balancer provides 7.4% improvement in terms of HS. In contrast, PBC with the way balancer provides 11.1% improvement in terms of HS. Also workload mixes like *mcf-soplex-omnetpp-gromacs*, *bzip2-omnetpp-sjeng-vortex2*, lose out on performance (by around 2%). In the absence of the way balancer, the cache space is not effectively utilized, which results in lesser performance improvement.

Impact of parallel look-up on power: We compare the power consumed by PBC for parallel lookup in both *C1* and *C2* with the baseline cache. We use CACTI 6.0 [23] to get the power numbers. The generic cache (*C*) that contains 2048 sets and 16 ways with 15 bits of tag consumes 0.25nJ per access. On the other hand, PBC contains *C1* and *C2*. *C1* has 2048 sets and 13 ways with 15 bits of tag, and *C2* has 1024 sets and 6 ways with 11 bits of tag. The power values are not optimized in CACTI, if the number of ways is not multiples of 2, hence we report the values for higher number of ways. *C1*, in the worst case (assuming a 16-way, 2048 sets cache) will consume 0.25nJ per access and *C2* will consume 0.09nJ per access (assuming an 8-way, 1024 sets cache). Note that, in our technique we search 19 ways, 13 at *C1* and 6 at *C2* in parallel, when compared to 16 ways in *C*.

Figure 30 shows the energy savings obtained by PBC over a generic cache. For each workload, two data points

are presented, one for the savings obtained in the LLC alone, and second, representing the savings obtained when all the three, LLC, compaction buffer and the DRAM are considered together. When energy consumption of LLC alone is considered, PBC on an average across 70 workloads consumes 36.8% of additional energy because of the extra tag lookup for each access. Upon considering all the three components together, since the memory power consumption dominates the overall energy consumption, PBC obtains 5.8% of energy savings on an average, across 70 workloads. The energy consumption at the memory is reduced because of the reduced number of read/write accesses to the memory.

5 Related Work

Sardashti et al. propose a decoupled compressed cache (DCC) [4] that exploits spatial locality to increase the compressed cache capacity without significant increase in tag overhead. To achieve this, DCC uses super blocks (aligned contiguous cache blocks that share a single address tag). By using a single tag, DCC stores higher number of compressed blocks in a cache set when compared to the previous cache compression techniques. In DCC, up to four aligned contiguous cache blocks can share a single address tag and hence, at the maximum, can quadruple the cache capacity. However with PBC, we observe that it is beneficial to have blocks which can hold more than four blocks, especially in the case of prefetched blocks, where the addresses exhibit high spatial locality. Also, PBC exploits the relation between the data patterns present across these contiguous cache blocks. DCC can be implemented along with PBC (at *C2*) to further improve the system performance.

Dusser et al. propose an augmented, Zero-Content Augmented Caches (ZCA) [14], in which null blocks (blocks containing all zeros) are associated with a single tag and accessed in parallel with the cache. ZCA targets only null blocks and can be augmented with PBC. Tabatabai et al. propose to compress companion cache blocks (cache blocks whose addresses differ by a single bit) together [2]. They propose two different cache mapping policies to index into the companion block. They also propose to proactively prefetch the companion block. Alameldeen et al. [13] study the interaction between cache compression and hardware prefetching and they observe the interaction to be positive.

Zhang et al. propose a technique to enable partial cache line prefetching through data compression [24], without the addition of prefetch buffers or increasing the memory traffic. The compression algorithm is based on the characteristics exhibited by dynamic values observed across various applications. For every cache block in memory, another block is associated as a prefetch candidate. When a cache block is prefetched, the compressor checks if the i^{th} word of the block and the i^{th} word from the associated block is compressible. If they are compressible, the two words are compressed together and sent as one word, which is part of the prefetched block, requiring no extra memory bandwidth. By prefetching partial or full cache blocks in advance, without them occupying extra space in cache, the technique improves the system performance when compared to a baseline system that does not support prefetching. PBC on the other hand works with a prefetcher enabled system. It does not decide which of the blocks are to be prefetched (which is solely decided by the prefetcher).

Yang et al. propose Frequent Value Compression (FVC) [25], based on the observation that significant amount of values accessed by cache, belong to a small set of frequently occuring values. FVC achieves compression by encoding these values using lesser number of bits. In FVC, each cache block can hold either one uncompressed block or two compressed block, compressed to at least half their size. FVC is limited to compressing only values that are frequent and fails to identify other compressible patters. It also incurs the overhead of profiling which is needed to identify the frequent values.

Alameldeen et al. identify seven frequently occurring compressible patterns [26], such as all the bytes in a 32-bit word being the same, 8-bit or 4-bit sign-extended data stored in a 32-bit word etc. and propose Frequent Pattern Compression (FPC) [27], which exploits these patterns. Each cache block is compressed word by word (32-bit), by storing the pattern encoding bits if the word belongs to any of the seven patterns. Since the position of the word in a cache block is not fixed (every word can either be in compressed or decompressed format), the decompression happens in a serialized fashion. A five-stage pipeline is proposed to mitigate the decompression latency. Alameldeen et al. also propose a technique to dynamically assess the costs and benefits of compression and predict whether to compress a cache block or not.

Arelakis et al. propose using of huffman based statistical compression technique, SC^2 [28], to aggressively increase the effective cache capacity (upto 4X). Though huffman based algorithms need sampling of data to generate codes, they note that there is little variation in data locality over time and across applications. They propose a pipelined decompression engine to keep the decompression latency to ten cycles. Chen et al. propose C-Pack [29], which compresses frequently appearing words by using compact encodings, that allows for parallel compression of multiple words. C-Pack also tries to combine pairs of compressed blocks into a single block, with each compressed block having its own tag and size fields associated with it.

6 Conclusion

We proposed an effective mechanism to compact multiple prefetched blocks into a single block. We also showed the effectiveness of taking into account, data patterns present across contiguous prefetched cache blocks. On 4- and 8-core system, PBC improves the performance by 11.10% and 13.67%, respectively. The effectiveness of our technique is independent of the underlying prefetching technique and prefetch throttling technique. PBC is effective for both single and multicore systems. We conclude that PBC provides a promising way to further increase the effective LLC capacity.

Acknowledgements

We would like to thank the reviewers and our colleagues for their very useful and detailed comments that helped us significantly improve the quality of the paper.

REFERENCES

[1] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd 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)*, pp. 377-388, 2012

[2] Ali-Reza Adl-Tabatabai, Anwar M. Ghuloum, and Shobhit O Kanaujia., "Compression in cache design". In *Proceedings of the 21st annual international conference on Supercomputing (ICS '07)*, pp. 190-201, 2007

[3] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy., "POWER4 system microarchitecture", *IBM J. Res. Dev.* 46, 1, pp. 5-25, 2002

[4] Somayeh Sardashti and David 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)*, pp. 62-73, 2013

[5] Alaa R. Alameldeen and David A. Wood., "Adaptive Cache Compression for High-Performance Processors". In *Proceedings of the 31st annual international symposium on Computer architecture (ISCA '04)*, pp. 212-223, 2004

[6] Magnus Ekman and Per Stenstrom, "A Robust Main-Memory Compression Scheme", In *Proceedings of the 32nd annual international symposium on Computer Architecture (ISCA '05)*, pp. 74-85, 2005

[7] Yiannakis Sazeides and James E. Smith, "The predictability of data values". In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture (MICRO 30)*. pp. 248-258, 1997

[8] A. Seznec, "Decoupled sectored caches: conciliating low tag implementation cost". In *Proceedings of the 21st annual international symposium on Computer architecture (ISCA '94)*, pp. 384-393, 1994.

[9] SPEC: www.spec.org

[10] K. Luo, Jayanth Gummaraju, and Manoj Franklin, "Balancing throughput and fairness in SMT processors", In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '01)*, pp. 164-171, 2001

[11] Allan Snavely and Dean M. Tullsen, Symbiotic jobscheduling for a simultaneous multithreaded processor, In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems (ASPLOS IX)*, pp. 234-244, 2000.

[12] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood., The gem5 simulator, *SIGARCH Comput. Archit. News*, 39, 2, pp. 1-7, 2011

[13] Alaa R. Alameldeen and David A. Wood.,"Interactions Between Compression and Prefetching in Chip Multiprocessors", In *Proceedings of the thirteenth international symposium on high performance computer architecture (HPCA)*, pp. 228-239, 2007

[14] Julien Dusser, Thomas Piquet, and Andr Seznec, "Zero-content augmented caches", In *Proceedings of the 23rd international conference on Supercomputing (ICS '09)*, pp. 46-55, 2009.

[15] R Manikantan, Kaushik Rajan, and R Govindarajan, "Probabilistic shared cache management (PriSM)", In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*, pp. 428-439, 2012

[16] Daniel Sanchez and Christos Kozyrakis, "Vantage: scalable and efficient fine-grain cache partitioning", In *Proceedings of the 38th annual International Symposium on Computer architecture (ISCA '11)*, pp. 57-68, 2011

[17] S. Eyerman and L. Eeckhout, "System-Level Performance Metrics for Multiprogram Workloads", in *IEEE Micro*, pp. 42-53, 2008

[18] Stephen Somogyi, Thomas F. Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos, "Spatial Memory Streaming", In *Proceedings of the 33rd annual International Symposium on Computer Architecture (ISCA '06)*, pp. 252-263, 2006.

[19] Jean-Loup Baer and Tien-Fu Chen, "An effective on-chip preloading scheme to reduce data access penalty", In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing (Supercomputing '91)*, pp 176-186, 1991

[20] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt., "Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers". In *Proceedings of the IEEE 13th International Symposium on High Performance Computer Architecture (HPCA '07)*, pp. 63-74, 2007

[21] Intel 64 and ia32 architecture software developer's manuals. http://www.intel.com/products/processor/manuals/

[22] Onur Mutlu and Thomas Moscibroda, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors", In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '07)*, pp. 146-160, 2007

[23] N. Muralimanohar, R. Balasubramonian and N. P. Jouppi., "Cacti 6.0: A tool to understand large caches. Technical report", University of Utah and Hewlett Packard Laboratories, 2007

[24] Youtao Zhang and Gupta, R., "Enabling partial cache line prefetching through data compression", *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, pp. 277-285, 2003

[25] Youtao Zhang and Gupta, R., "Frequent value compression in data caches", In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture (MICRO 33)*. pp. 258-265, 2000

[26] A. R. Alameldeen and D. A. Wood. "Frequent pattern compression: A significance-based compression scheme for L2 caches" *Tech. Rep., University of Wisconsin-Madison*, 2004.

[27] 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)*, pp.212223, 2004.

[28] Angelos Arelakis and Per Stenstrom, "SC^2: A Statistical Compression Cache Scheme". In *Proceedings of the 41st annual International Symposium on Computer Architecture (ISCA '14)*, pp.145-156, 2014.

[29] X. Chen, L. Yang, R. Dick, L. Shang, and H. Lekatsas." C-pack: A high-performance microprocessor cache compression algorithm" In *IEEE Transactions on VLSI Systems (TVLSI)*, Aug. 2010.

Raghavendra K is a Ph.D. candidate in the department of computer science and engineering at IIT Madras. His current research interests are cache and DRAM compression techniques.

Biswabandan Panda is a Ph.D. candidate in the department of computer science and engineering at IIT Madras. His current research interests are hardware prefetching and shared resource management techniques for CMPs.

Madhu Mutyam is a professor in the department of computer science and engineering at IIT Madras. His current research focus is on multicore architectures, specifically issues related to memory system design and networks-on-chip. He is a senior member of IEEE and ACM. His Erdos number is 3.