Synergistic Cache Layout For Reuse and Compression

Biswabandan Panda, André Seznec

Synergistic Cache Layout For Reuse and Compression

Biswabandan Panda
Indian Institute of Technology Kanpur, India
biswap@cse.iitk.ac.in

André Seznec
INRIA Rennes, France
andre.seznec@inria.fr

ABSTRACT

Recent advances in research on compressed caches make them an attractive design point for effective hardware implementation for last-level caches. For instance, the yet another compressed cache (YACC) layout leverages both spatial and compression factor localities to pack compressed contiguous memory blocks from a 4-block super-block in a single cache block location. YACC requires less than 2% extra storage over a conventional uncompressed cache.

Performance of LLC is also highly dependent on its cache block replacement management. This includes allocation and bypass decision on a miss as well as replacement target selection which is guided by priority insertion policy on allocation and priority promotion policy on a hit. YACC uses the same cache layout as a conventional set-associative uncompressed cache Therefore the LLC cache management policies that were introduced during the past decade can be transposed to YACC. However, YACC features super-block tags instead of block tags. For uncompressed block, these super-block tags can be used to monitor the reuse behavior of blocks from the same super-block. We introduce the First In Then First Use Bypass (FITFUB) allocation policy for YACC. With FITFUB, a missing uncompressed block that belongs to a super-block that is already partially valid in the cache is not stored in the cache on its first use, but only on its first reuse if any. FITFUB can be associated with any priority insertion/promotion policy.

YACC+FITFUB with compression turned off, achieves an average 6.5%/8% additional performance over a conventional LLC, for single-core/multi-core workloads, respectively. When compression is enabled, the performance benefits associated with compression and FITFUB are almost additive reaching 12.7%/17%. This leads us to call this design the Synergistic cache layout for Reuse and Compression (SRC). SRC reaches the performance benefit that would be obtained with a 4X larger cache, but with less than 2% extra storage.

ACM Reference Format:
Biswabandan Panda and André Seznec. 2018. Synergistic Cache Layout For Reuse and Compression. In *International conference on Parallel Architectures and Compilation Techniques (PACT '18), November 1–4, 2018, Limassol, Cyprus.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3243176.3243178

1 INTRODUCTION

Compressed caches have become an attractive design point for effective hardware implementation for last-level caches (LLCs). Recently proposed compressed cache layouts, such as decoupled compressed cache (DCC) [25], skewed compressed cache (SCC) [23], and yet another compressed cache (YACC) [24] leverage super-block tags[1] to implement compressed caches with a limited storage overhead (less than 2% for SCC and YACC). On YACC and SCC, the data array is organized in fixed size data entries (typically 64B the size of an uncompressed block), each data entry being associated with a unique and fixed tag. However, this tag is a super-block tag and the data entry can contain either multiple compressed memory blocks or a single uncompressed memory block. The read access time of YACC or SCC is similar as the ones of uncompressed caches apart for the decompression. YACC and SCC are agnostic to the cache compression algorithm. However, YACC and SCC ensures that the allocation unit in the LLC is a complete 64B data entry. The DIctionary SHaring (DISH) compression scheme [19] leverages this particularity. With DISH, a dictionary is shared among the memory blocks represented in the data entry. This allows more efficient compression than traditional compression schemes [3, 10, 21] that compress every memory block independently. As a result, YACC or SCC coupled with DISH performs generally better than a 2X uncompressed cache[19].

LLC replacement policies for compressed caches has received little attention so far, apart from [7, 11, 20]. Before of the introduction of YACC, the adaptation of the state-of-art replacement policies for compressed caches was very challenging. For instance, on DCC [25], the replacement policy had to manage the super-block replacements (when the super-block tag is missing) and the block replacements (when the super-block tag is present, but the block is missing). As the allocated data size is known only after compression, a single cache miss could lead to several super-blocks evictions and several cache block evictions. In contrast, YACC has a very similar layout as a conventional cache, apart that a super-block tag is associated with each data array entry instead of a block tag (Figure 1), thus allowing to map either an uncompressed data block or up to four co-compressed blocks

[1] A super-block is an aligned and contiguous group of blocks (compressed and uncompressed) that use a single tag called super-block tag for all the blocks. For example, a 4-block super-block that contains four cache blocks share a single super-block tag.

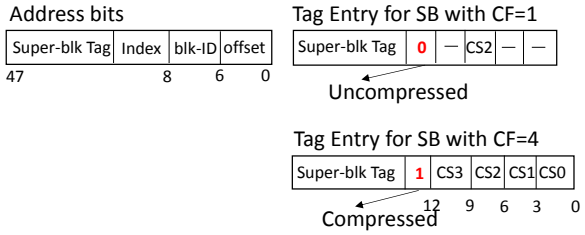

Figure 1: YACC tag entries with different compression factors (CFs). CS:coherence states/V/I bits. SB: Super-block.

of the super-block in the data entry. On a cache allocation, a single data entry is replaced. Most LLC management policies that were proposed for conventional caches [14], [22], [13], [31], [26], [15], [17], [12] can be directly transposed to the YACC cache.

However YACC uses super-block tags. As conventional tags, super-block tags can be used to monitor activities/status of the blocks present in the cache. They can also be used to monitor the activities/status of blocks belonging to the super-block, but absent from the cache. Many fetched blocks are not reused before eviction (e.g. [5]). The super-block tag can be used to (partially) exploit this property on super-blocks that are uncompressed or not co-compressible. The **F**irst **I**n **T**hen **F**irst **U**se **B**ypass (FITFUB) allocation policy leverages the super-block tag in YACC as follows.*On an LLC miss, on a block Bi within a missing super-block B, a data entry and its super-block associated tag are allocated. On a subsequent miss on Bj in the same super-block B, if the block cannot be co-compressed with Bi, then the validity status associated with Bj is updated in the super-block tag as First-Use, but the data entry is not allocated. An additional data entry and its associated super-block tag location are allocated for Bj only if it gets reused before super-block B is evicted from the cache.* On YACC, FITFUB can be implemented on top of a conventional priority insertion/promotion policy.

Interestingly, FITFUB is an efficient LLC allocation policy even when compression is turned off and makes the YACC layout attractive even for uncompressed caches. In our experiments, with compression turned off, YACC with FITFUB achieves an average 6.5% and 8% higher performance than a conventional LLC that uses SHiP++ [31] (an extended version of SHiP as per cache replacement championship held in ISCA '17) for single-core and multi-core workloads, respectively. When the DISH compression is turned on, YACC+FITFUB achieves 12.7% and 17% higher performance than a conventional uncompressed LLC for single-core and multi-core workloads, respectively while YACC only achieves 8.3% and 10%, respectively. As the benefits of cache compression through the YACC layout and the benefits of the FITFUB policy are nearly additive, and as FITFUB essentially identifies the reused cache blocks when uncompressed, we call it *Synergistic cache layout for Reuse and Compression (SRC)*.

The remainder of the paper is organized as follows. Section 2 presents the background on compressed caches. Section 3 introduces our experimental framework. The motivation and opportunity behind SRC is in Section 4. Section 5 introduces the FITFUB allocation policy for YACC. We evaluate the overall SRC proposal in Section 6 while Section 8 concludes this study.

2 BACKGROUND

This section provides the background on the state-of-the-art compressed cache layouts and cache compression schemes.
Compressed Cache Layouts: Compressed caches use a compression technique that compresses cache blocks and a compaction technique that compacts the compressed blocks in the data array. Till the proposition of **DCC**[25], the proposed designs were considering the use of more tags than data entries.

In DCC[25], super-block tags limit the tag volume, the main observation being that often adjacent contiguous cache blocks co-reside at the LLC. DCC compacts up to 16 four 64B block super-block in 1024B. The **SCC**[23] and the YACC cache [24] further simplify the cache layout design through associating a single super-block tag with a fixed 64 bytes data entry.

SCC[23] is a compressed cache layout that compacts multiple (in power of two, such as 1, 2, and 4) compressed blocks from the same super-block and stores them in one data entry. It uses super-block tags and skewed associative mapping [27]. It maintains an one-to-one mapping between a tag entry and a data entry, and it compacts cache blocks based on their compression factors (CFs). For a data entry of 64 bytes, the CF of a cache block is: four/two/one if it is compressed to <16B/between 16B to 32B/>32B.

YACC [24] simplifies the design aspects of SCC, and maintains the one-to-one mapping between a tag entry and a data entry. It uses a cache layout, which is similar to a regular set-associative uncompressed cache. It removes skewing from SCC, and achieves the same performance level. However, similar to SCC, it compacts cache blocks by taking their CFs into account. YACC tracks from one to four cache blocks with the help of a super-block tag. An important property of YACC is that all the blocks belonging to a 4-block super-blocks are stored in the same cache set. *That is: YACC has a set associative structure, apart that it uses super-block tags instead of block tags, and that a data entry-aka a cache line location-can store either an uncompressed cache block, or several compressed cache blocks from the same super-block. This mapping*

provides an average performance reduction of less than 1.5% as it reduces the effective associativity of a set. In this paper, we consider a version of YACC, which considers only two possible compression factors, CF=1 (1 uncompressed block) and CF=4 (1 to 4 compressed blocks). In that context, Figure 1 represents the super-block tag. Compared to the tag of a conventional cache and assuming a 3-bit coherence/validity state per block, the extra tag storage is only 8 bits per tag ((three 3-bits + one bit compressed/uncompressed) - (two address bits that YACC saves from the address splitting)), i.e. 1.5% storage overhead on the LLC.

Cache Compression Techniques: State-of-the-art compressed cache layouts are independent of the underlying compression techniques such as CPACK [10], FPC [3], BDI [21], SC^2 [6] and DISH [19], out of which BDI and DISH provide compression ratios of 1.7X and 2.3X when associated with a YACC layout, respectively [19]. On a compressed cache, decompression is on the critical path. BDI and DISH decompress a cache block in a single cycle.

BDI [21] 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).

DISH [19] is a compression technique that was specifically designed for the compaction schemes such as SCC and YACC that always allocate a full data block in the LLC and that associate it to a super-block tag. DISH uses a dictionary that is shared by up to four cache blocks. In this way, DISH manages both compression and compaction together. Both BDI and DISH can be used with a YACC layout. DISH is more efficient than BDI when associated with YACC [19]. It even simplifies the YACC layout design, retaining only two cases CF=4 and CF=1. Therefore, apart when explicitly mentioned, in the remainder of the paper, DISH is used as the compression scheme.

3 EXPERIMENTAL METHODOLOGY

This section describes the experimental methodology that we use throughout the paper. We use the x86-based gem5 [9] simulator to evaluate the effectiveness of SRC cache at the LLC. Table 1 illustrates the baseline configuration for the simulated systems. We simulate both single-core and multi-core (16- and 32-core) systems, and we estimate the cache latencies with CACTI 6.5 [18]. To calculate the power consumption of DRAM, we use the Micron Power Calculator. We measure the energy consumption of LLC and DRAM, off-chip transfers between LLC and DRAM, as well as the energy consumed by the compressor and de-compressor of DISH. For single-core evaluations, we collect the statistics from the region of interest for 2B instructions.

Processor	1/16/32-cores, 3.7 GHz, out of order
L1 D/I, L2	32 KB (4 way), 256KB (8 way)
Banked Shared L3	2/32/64 MB for 1/16/32 cores with 16 ways, non-inclusive
MSHRs	16/16/16 per core MSHRs at L1/L2/L3
Line size	64B in L1, L2 and L3
Cache Replacement policy	SHiP++ [31]
Compressed Cache Layout	YACC [24]
Compression Technique	DISH [19]
L2 prefetcher	Stream based [30], 32 streams with degree = 4 and distance = 32
On-chip interconnect	Crossbar
DRAM controller	1/4/8 controllers for 1/16/32-cores, Open Row, 64 read/write queues, FR-FCFS
DRAM bus	split-transaction, 800 MHz, BL=8
DRAM	DDR3 1600 MHz (11-11-11), Max bandwidth/channel - 12.8 GB/sec

Table 1: Parameters of the simulated system

	Benchmarks	Types
SPEC CPU 2006 [29]	xalancbmk, bzip2, hmmer, soplex, mcf, omnetpp, h264ref, cactusADM, lbm, gromacs, zeusmp	Sensitive-positive (SP)
	libquantum, milc, bwaves, namd, leslie3d, sjeng	Insensitive (IN)
PARSEC [8]	canneal, dedup, ferret, fluidanimate, freqmine, vips	Sensitive-positive (SP)
	blackscholes, bodytrack, swaptions, facesim, streamcluster, x264	Insensitive (IN)
CRONO [1]	sssp, apsp	Sensitive-positive (SP)
	pagerank, community detection (community), bfs, betweenness centrality (bc)	Insensitive (IN)
Machine Learning	sparse matrix vector multiplication (spmv), symmetric Gauss-seidel smoother (symgs)	Sensitive-positive (SP)
	locality sensitive hashing (lsh). stochastic gradient descent (sgd), K nearest neighbor (knn)	Insensitive (IN)

Table 2: Classification of benchmarks.

Workload selection: Table 2 classifies 40 benchmarks from SPEC CPU 2006 [29] [2], PARSEC [8], CRONO [1], and from the world of graph analytics and machine learning, into 2 classes (SP, and IN), based on their sensitivity to cache capacity. A benchmark is *sensitive-positive* (SP) if there is an improvement in performance 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. IN also includes the benchmarks that are *sensitive-negative*, for which performance decreases when the LLC size doubles (this occurs when LLC misses do not drop since we model a longer access latency for a larger LLC).

[2]In our gem5 set-up, only 17 SPEC applications run correctly. Seven applications do not run correctly with gem5 and the rest of the applications provide incorrect outputs.

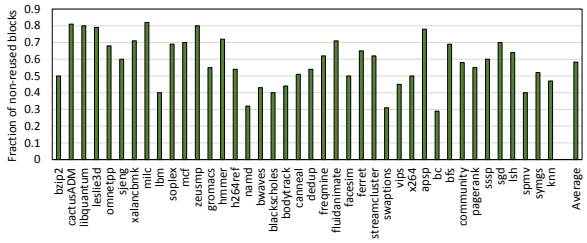

Figure 2: Fraction of non-reused LLC blocks.

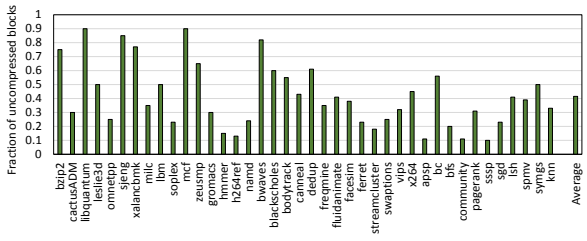

Figure 3: Fraction of uncompressed LLC blocks.

We created multi-programmed workload mixes from single-threaded SPEC CPU 2006, PARSEC, CRONO, and other benchmarks. We created these workload mixes based on their LLC sensitivity and fraction of non-reused cache blocks. At the LLC, if a workload gets a performance increase over 8% when the LLC size doubles, then we classify it as high (*H*) in terms of LLC sensitivity otherwise, the workload is termed as low (*L*). Similarly, in terms of non-reused blocks, if a workload exhibits more than 50% of LLC blocks that are non-reused, we termed it high (*h*); otherwise, it is termed as low (*l*). We select 200 16-core and 32-core workloads into four categories such as *Hh, Hl, Lh, and Ll* where each category consists of 50 workloads. For multi-core evaluations, we evaluate both multi-programmed and multi-threaded workloads. For multi-threaded workloads (such as PARSEC and CRONO), we use the region of interest for 2B instructions. For multi-programmed workloads, we simulate the multi-programmed mixes by running each benchmark for 2B instructions from the region of interest. A workload terminates when the slowest benchmark completes 2B instructions. For all the evaluations, we warm-up the cache for 1B instructions before running the region of interest.

Metrics: We use data block misses per kilo instruction (MPKI) to measure the behavior of the cache itself. For single-core simulations and for multi-threaded workloads, we use speedup as the metric, i.e., $\frac{Exectime_{baseline}}{Exectime_{technique}}$. For multi programmed mixes, we use normalized weighted speedup (WS) [28] and the normalized fair speedup (FS) [16] where $WS = \sum_{i=0}^{N-1} \frac{IPC_i^{together}}{IPC_i^{alone}}$, $FS = \frac{N}{\sum_{i=0}^{N-1} \frac{IPC_i^{alone}}{IPC_i^{together}}}$, $\text{IPC}_i^{together}$ 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 N-core multi-core system. *WS* tends to be a good measure for system throughput improvement while *FS* balances system throughput improvement with maintenance of system fairness.

4 MOTIVATION AND OPPORTUNITY

Albericcio et al. showed that a significant fraction of the blocks missing in the LLC do not get any reuse before eviction from LLC [5]. Our experiments show that this property still holds for compressed caches. Fig. 2 illustrates the fraction of LLC blocks that are not reused on YACC in our experiments for single core. On average, more than 55% of blocks (both compressed and uncompressed) do not get a reuse at the LLC, with the maximum of 82%. In particular, streaming applications such as `libquantum` have a significant number of LLC blocks getting no reuse. The Reuse cache [5] was proposed to exploit this property on an uncompressed cache. The Reuse cache layout is radically different from the one of a conventional cache. It implements 4X more tags than the number of data cache blocks; the tag array and data array are decoupled (pointers and back pointers are needed). On a miss on a memory block, only the tag is allocated and the data is bypassed to the core. On the first (tag) reuse, i.e., hit on the tag array but miss on the data array, the data block is allocated on data array. Our experiments confirm that, in most cases, the Reuse cache achieves performance in the same range or higher as a 2X larger conventional cache. Unfortunately the Reuse cache also incurs very significant hardware overhead; ignoring the extra logic, the Reuse cache necessitates about 20% of extra storage than a conventional cache. With the Reuse cache, data blocks have three possible presence status, *tag-only*, *tag+data*, and *invalid*. The *tag-only* status allows to track the first use blocks that are not present in the data array.

In our experiments, with YACC, on average, 42% of the blocks sitting in the LLC are uncompressed, with the maximum being 90% (Fig. 3). This means that 42% of the super-block tags (that can map up to four contiguous blocks) represent one *tag+data* block state and 3 *invalid* block states. For a super-block with super-block tag (say SA) that contains one uncompressed block A2, the introduction of the presence status *tag-only* in YACC will allow to track the reuse property of its companion cache blocks (A0, A1 and A3).

In the next section, we present the FITFUB allocation policy for YACC. FITFUB exploits this extra presence status to (partially) filter the allocation of single use uncompressed blocks in the YACC cache.

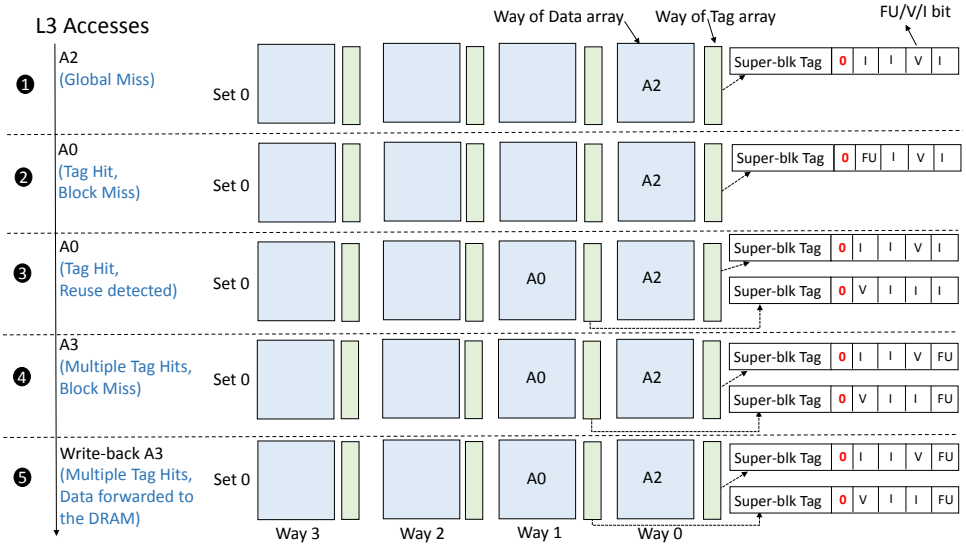

Figure 4: Different access scenarios with the U-SRC cache. FU is the *first-use* bit. A0 to A3 are the blocks of a super-block SA. For each access, we show the updated tag entries (after the access is serviced).

5 FIRST IN THEN FIRST USE BYPASS ALLOCATION POLICY

We describe FITFUB policy as follows: *On an LLC miss for a block Bi within a missing super-block B, a data entry and its associated super-block tag are allocated. On a subsequent miss on Bj in the same super-block B, and if the block cannot be co-compressed with Bi, the validity status associated with Bj is updated in the super-block tag as First-Use, but the data entry is not allocated. An additional data entry and its associated super-block tag location are allocated for this second block Bj, only if it gets reused before super-block B is evicted from the cache.*

We detailed below the various access scenarios induced by the FITFUB policy on the YACC cache. For the sake of simplicity, we first describe the scenarios when all blocks are uncompressed (or not co-compressible), and then we generalize to the mix of uncompressed and compressed cache blocks.

5.1 Scenarios with uncompressed blocks

We consider in this section that all blocks in the super-block are uncompressible (or that they are not co-compressible). On YACC, four contiguous cache blocks A0, A1, A2 and A3 belonging to a 4-block super-block (say SA) are mapped to the same cache set. A super-block tag is associated with each data entry. Below are the different access cases (refer Figure 4):

Global miss on SA (①): On the access, if the super-block englobing the block, e.g. block A2 in super-block SA is missing then a super-block tag is allocated in the tag array. The data block is placed in the associated data entry. In the tag, the CS field (as mentioned in Figure 1) associated with the loaded block is marked as *valid*. The CS fields associated with the other blocks in the super-block are marked as *invalid*.

Hit on super-block tag, but requested block is *invalid* (②): When block A0 is accessed and if tag SA with block A2 valid is present, a hit is encountered on the tag array, but a miss on the data array (CS0 is invalid). Block A0 is marked as *first-use* in CS0 (meaning this is the first access to block A0), the block is forwarded to L2 cache and then to the processor. But it is not allocated in the data array.

Hit(s) on super-block tag, but request block is marked as *first use* block (③): On a subsequent access to block A0, a new super-block tag and its associated data block is allocated in the cache for A0. Its CS0 field is marked as *valid*. In the first tag associated with SA (that was allocated on the access of A2), the CS field is marked as *invalid*.

Multiple super-block tag hits, but request block is *invalid* (④): On an access to A3, the same process is repeated apart that on the first access, the CS3 field in both tags associated with A2 and A0 are set to *first use*.

Writeback at the LLC (⑤): On a writeback at the LLC, if the block is valid in the LLC, it is updated. Otherwise, on a full miss or a *first use* hit, the block is directly forwarded to the DRAM without modifying the LLC. The rationale of this policy is that multiple uses by the processor are, in practice read uses. In practice, we leverage the use of super-block tags to record the recent first use of blocks belonging to super-blocks that have been recently touched: a single tag is able to record this information for four blocks of the super-block.

An extra coherence/validity state: FITFUB needs an additional state for cache blocks: the *first-use* state. For a non-inclusive cache hierarchy, depending on whether the number of possible states allowed by the coherence protocol is power of two or not, this additional state can lead to an additional bit per block in the super-block.

5.2 Scenarios with compressible blocks

So far we have only considered un-compressible blocks. In this section, we describe few extensions for accommodating compressed blocks. The first type of modification corresponds to the cases 1 and 3 of Section 5.1: On a cache block allocation, if the block is compressible then it is stored as compressed block and the compressed bit is set in the super-block tag (refer ① of Figure 5). The second type of modification corresponds to cases 2 and 4 of Section 5.1 where one needs to take into account that a data entry that contains one or more compressed blocks can still be completed with a missing block.

Hit on super-block tag, compressed bit is set, however block is present in *invalid* **state**: This scenario corresponds to the access to A0 with super-block tag SA when a valid block A2 is present in compressed form. In this case, if A0 is co-compressible[3] with A2 then A0 is packed in the same data entry as A2 and marked as valid (refer ② of Figure 5). However if the missing block is in-compressible (or not co-compressible) then it would be marked as *first use*, as illustrated for block A1 (refer ③ of Figure 5).

Hit on super-block tag, compressed bit is set, however block is in *first use* **state**: This corresponds to the case ④ in Figure 5. A1 was in *first-use* state and is reused so a dedicated data entry is allocated for it (general case). In the case where A1 has been modified since its first use (e.g a Write-Back) and has become co-compressible with A2, one can pack it in the already allocated entry.

Multiple super-block tag hits, at least one super-block tag has compressed bit set, however block is in *invalid* **state**. In this scenario, the block could potentially be co-compressible with several cache entries, one has to ensure that at most one copy of the block (in compressed format) is stored in the cache and the corresponding super-block tag correctly updated (refer ⑤a of Figure 5). If it can not be compacted then it is marked as *first use* in all super-block hitting tags (refer ⑤b of Figure 5).

The third type of modification corresponds to the write-backs. For most writebacks, the scenario is directly derived from the scenario on uncompressed blocks, i.e., we update the block in the cache if the data block is present and is co-compressible with the other co-located blocks in the LLC. If the block is marked as *first use* or is *invalid* then we directly write back the block into the DRAM.

However particular attention must be given to the case where the compressibility status of the block changes and becomes incompatible with the previous state. This arises when the block was compressible and co-located with one or more other blocks, and becomes in-compressible or not co-compressible with its companion blocks. In this case, the block is directly written back to the DRAM and invalidated in the LLC (refer ⑥ of Figure 5). The rationale is that, we find, in these cases more than 80% of the time the written back cache block do not get a reuse before its replacement. Techniques such as dead-write predictor [2] can be used to improve the decision process for this event.

5.3 FITFUB, YACC and inclusive Cache Hierarchies

For inclusive cache hierarchies, the FITFUB allocation policy violates the inclusiveness by not allocating data entries at the LLC. To resolve this issue, we rely on the solution proposed for the Reuse cache, the tag-only (TO) status. So at the LLC, a coherence state can have two different variations: one in which data is present in the data array and one where only the tag is valid (corresponding to a first use). A coherence transaction on a block in TO state should be propagated to the caches closer to the processor. With the addition of a new state, there are two new coherence transitions that are possible: *tag-only* to *tag+data* and *tag+data* to *tag-only*. This incurs an additional overhead of 1 bit per cache block.

6 PERFORMANCE EVALUATION

We first evaluate the FITFUB allocation policy on YACC layout with compression turned off. *For convenience, we refer this configuration as U-SRC*. Then we evaluate the FITFUB allocation policy with YACC and DISH compression turned on. We refer to this configuration as the SRC cache. Our evaluation addresses the FITFUB allocation policy. Therefore, for all simulated caches on an insertion or a block hit, the block priority is updated as in SHiP++ [31].

6.1 Evaluation: Compression turned off

In this section, we evaluate the effectiveness of U-SRC and compare it with a baseline cache as well as the Reuse Cache [5]. The Reuse cache is simulated as a reference point since it was essentially designed to capture the potential of bypassing first use prefetch blocks.

Due to space limitation, we illustrate the results for single core only. Figure 6 and Figure 7 illustrate the improvement

[3]The DISH compression scheme proposes two encoding schemes, and two blocks of a super-block can be stored in the same data entry only if they can be compressed with same encoding scheme [19]

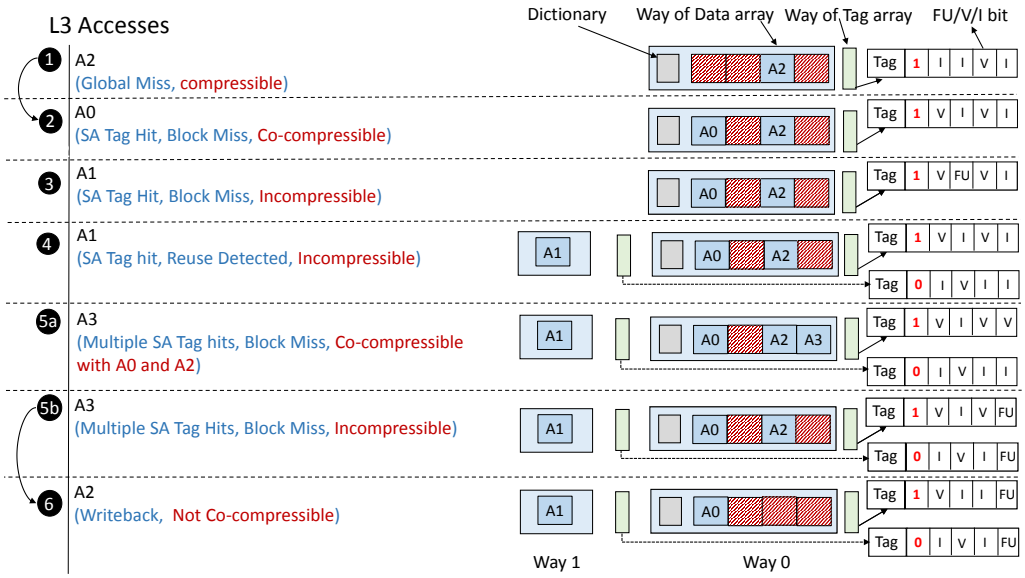

Figure 5: FITFUB: Access scenarios for compressible blocks..

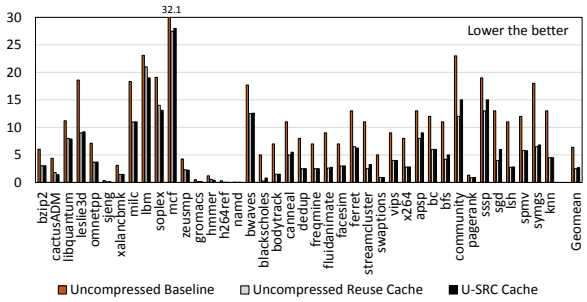

Figure 6: LLC misses in terms of absolute MPKI.

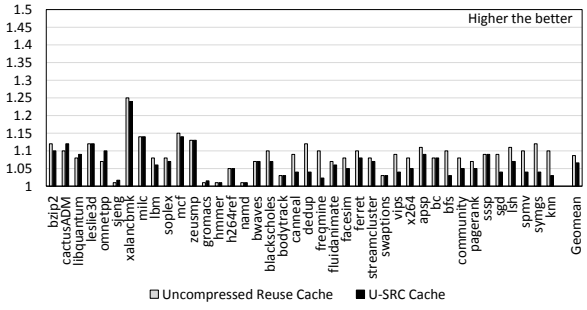

Figure 7: Speedup normalized to an uncompressed baseline LLC.

on LLC misses and execution time, respectively. U-SRC simulation results for multicores are illustrated in Figure 10 of Section 6.

The miss rates of the Reuse cache and U-SRC are very similar and this for all applications in our benchmark set. The miss rates are in many cases significantly lower than on the conventional cache confirming than bypassing first use blocks allows to keep useful multi-used blocks in the cache. Both the Reuse cache and U-SRC keeps information on multiple uses on a 4X larger size than the effective data array. The SHiP++ policy applied on the baseline does not have such information.

Note that, the improvement in miss rates does not translate in the same performance improvement on U-SRC as on the Reuse cache. Compared to the baseline uncompressed LLC, the Reuse cache improves the performance by 8.6% and U-SRC only captures $\frac{3}{4}$ th of this potential, i.e, 6.5% performance improvement.

In practice, the misses encountered by U-SRC and the Reuse cache are not similar. U-SRC allocates a data entry on the first miss on 4-block super-block while the Reuse cache treats all blocks as equal. This translates in missed opportunities of bypassing for U-SRC, but also in extra misses for reused blocks on the Reuse cache. In our experiments, in most cases these two phenomena compensate approximately each other in terms of miss numbers. While the miss rates on the caches are similar, the access patterns on the main memory and the cost of the misses are different. The average cost of a miss on U-SRC is higher than on the Reuse cache. For instance, for a fully used super-block encountering multiple uses, the Reuse cache suffers two bursts of four misses, each burst generating 4 contiguous DRAM accesses, thus potentially benefiting from row locality. On the same sequence, U-SRC saves a low cost miss, but unfortunately statistically this low cost miss is replaced by a higher cost miss.

However, given the higher design complexity of the Reuse cache compared to a conventional cache design and U-SRC,

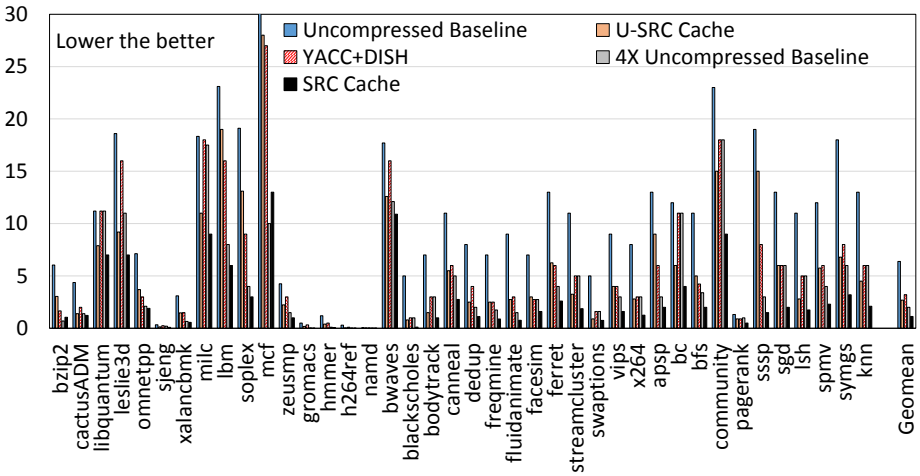

Figure 8: LLC misses in terms of absolute MPKI.

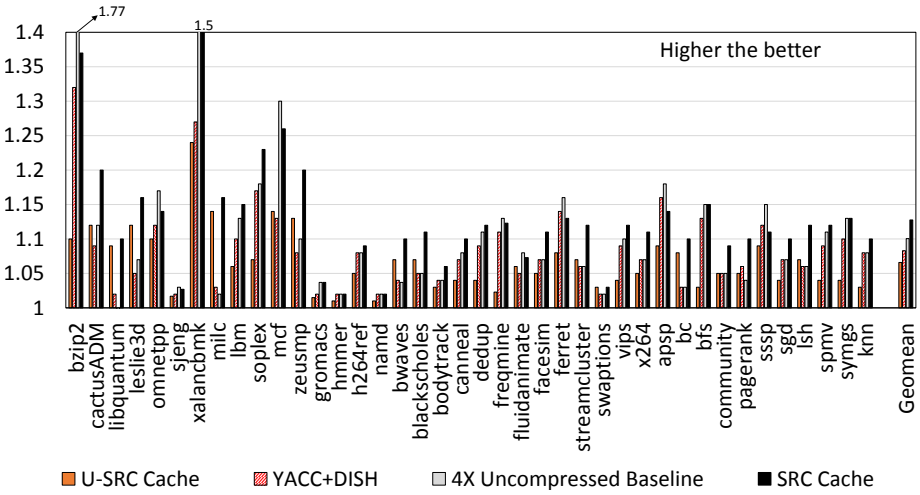

Figure 9: Speedup normalized to an uncompressed baseline LLC.

in terms of storage overhead (about 20% storage overhead), and design complexity (tag array and data array are decoupled, and pointers and back pointers are needed), U-SRC (i.e YACC layout and FITFUB policy) appears as a cost-effective design point even for uncompressed LLCs.

6.2 Compression turned on

In this section, we evaluate the effectiveness of the SRC cache i.e YACC layout + FITFUB allocation policy with cache compression turned on. We compare it with the baseline compressed cache YACC. In both cases, we assume that the DISH compression scheme is used. We also compare SRC cache with an uncompressed Reuse cache, with U-SRC, with the conventional cache and a 4X sized conventional cache. DISH takes 24 and 1 cycles for compression and decompression, respectively. The access time to the cache is modelled through CACTI 6.5, therefore the access time of a 4X larger cache is longer than the one on the baseline cache.

Single-core Results: Fig. 8 illustrates the absolute MPKI numbers for an uncompressed baseline, U-SRC, YACC+DISH, 4X uncompressed LLC, and SRC. The SRC cache always encounters fewer misses than YACC and U-SRC. It even encounters fewer LLC misses than the 4X uncompressed baseline cache, except for `mcf` and `bzip2`. For many memory-intensive (high MPKI) applications such as `mcf` and `lbm`, the reduction is very significant with number of LLC misses reduced by more than 3X.

In several cases, the benefit comes essentially from compression e.g. `sssp`, `apsp`, and `bzip2`. In other cases, the benefit comes from FITFUB allocation policy e.g. `libquantum`, `leslie3d`, and `bwaves`. In many cases, the reduction in LLC misses comes from both the block allocation policy and the cache compression e.g `lbm`, `soplex`, and `knn`. Compared to

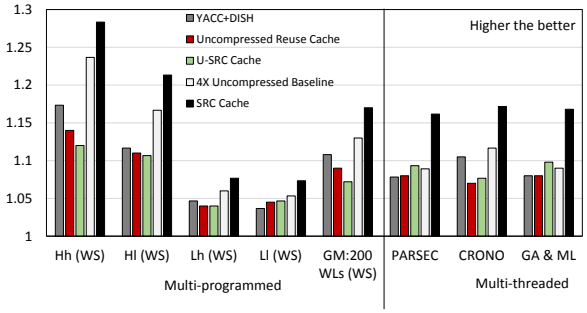

Figure 10: Normalized system performance (weighted speedup) over an uncompressed baseline.

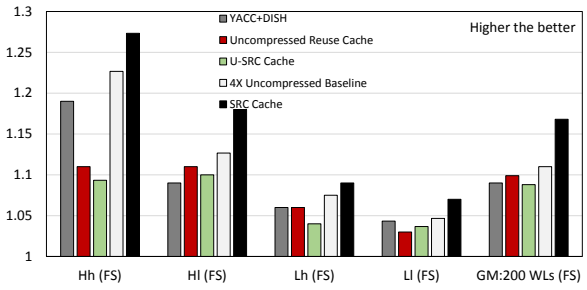

Figure 11: Normalized system performance (fair speedup) over an uncompressed baseline.

YACC+DISH, SRC reduces the average miss rate (in terms of MPKI) from 3.2 to 1.1. This reduction induces a performance improvement as illustrated in Figure 9. On average, compared to an uncompressed baseline, SRC cache provides 12.7% of speedup (a maximum of 50% for `xalancbmk`) whereas a 4X uncompressed baseline, YACC+DISH, and U-SRC provide speedups of 10%, 8.3% and 6.5% respectively. SRC always outperforms both YACC+DISH and U-SRC, and on many benchmarks, the performance benefits of compression and allocation policy are nearly additive. Moreover, SRC cache outperforms 4X uncompressed baselines on most of the applications. A few points can be underlined. i) In a few cases, the 4X larger cache performs worse than other configurations despite similar (`libquantum` with U-SRC) or lower miss ratios (`zeusmp` with U-SRC). This was expected since our simulations assume a longer cache access latency for 4X larger cache.

ii) On average U-SRC provides less performance benefit than YACC+DISH despite a larger miss rate reduction. This can attributed to the difference of miss patterns that are induced by the FITFUB allocation policy, leading to less DRAM row access locality.

iii) For several benchmarks on which U-SRC is lagging in performance against the Reuse cache despite a similar or smaller miss rate, e.g `freqmine`, `bfs`, and `ferret` (see Figure 6 and 7), the same phenomenon occurs with SRC against the 4X cache, but with a smaller amplitude. Once again, this can attributed to the patterns of misses which result in less locality on DRAM row accesses in memory. However, the phenomenon is less pronounced than on the uncompressed cache since compressed blocks do no generate extra misses on the second use.

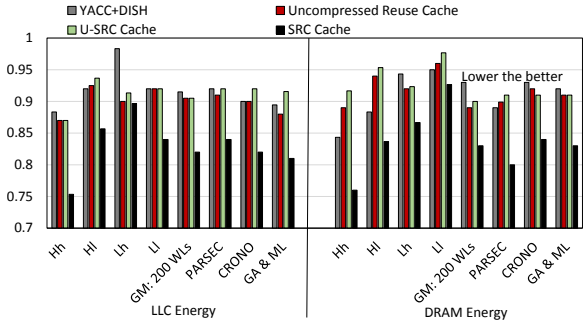

Figure 12: Improvement in memory sub-system energy normalized to uncompressed baseline.

iv) Among our benchmarks, `bzip2` is a very particular case with the 4X cache reaching 77% of performance improvement. With the 4X cache, the working set of bzip2 almost fits in at the LLC. Moreover, the stream of misses exhibits poor locality, DRAM row hit rate is very low (less than 50 %), therefore the cost of each individual miss is very high.

Multi-core results: For 16- and 32-core systems, we use an LLC of 32MB and 64MB, respectively. All other shared resources are scaled as mentioned in Table 1. We divide the multi-core results into two types - multi-programmed and multi-threaded. For multi-programmed workloads, we provide the performance improvement for each of the classes (*Hh, Hl, Lh, Ll*) defined according to their sensitivity to cache size and their ratio of reused blocks (see Section 3).

Fig. 10 illustrates the performance improvement for 200 multi-programmed workloads that span across four categories, and multi-threaded workloads. On average (geomean of weighted speedup) across 200 workloads, compared to an uncompressed baseline, SRC cache provides 17% improvement whereas uncompressed Reuse cache, U-SRC cache and 4X uncompressed baseline provide improvements of 9%, 8%, and 13%, respectively. Fair speedups are in the same range (refer Figure 11). We observe similar trends for multi-threaded applications.

In general SRC cache provides additional LLC space by not allocating data entries to non-reused blocks through helping the cache-sensitive applications of a workload mix. As expected, workload mixes of category *Hh* that contain cache sensitive applications and high proportions of single use blocks are the greatest benefactors of SRC cache. As also expected mixes such as *Ll* get more marginal benefits with

SRC cache as the individual applications are less sensitive to an increase of the cache size.

SRC on inclusive Cache Hierarchies: So far, the evaluation has considered non-inclusive cache hierarchy. We also simulated a relaxed inclusive cache hierarchy using the *tag-only* state described in Section 5.3. On a relaxed inclusive cache, the effectiveness of SRC remains similar as observed for non-inclusive caches.

6.3 Energy consumption

Through reducing execution time, SRC reduces the energy consumption both in the cores and in the memory system. We focus our evaluation on the shared memory system including LLC, DRAM and compressor and de-compressor. Figure 12 illustrates the reduction in the energy consumption in the memory components (LLC and DRAM) allowed by the SRC cache for the multicore workloads. On average, SRC reduces the energy consumption by 17 % on both the DRAM and the LLC. However these similar average percentages cover different scenarios. For instance, on the PARSEC workloads, the benefit at the LLC is lower than on DRAM while on L1 workloads the energy benefit at the DRAM is much lower than the energy benefit at the cache. At the LLC, the additional static power required by SRC over the baseline configuration is marginal (roughly 2%), corresponding to the storage overhead. Therefore, on our benchmarks, the static energy consumption of SRC is about 13 % lower than the one of the conventional cache ($1 - \frac{1.02}{1.17} \equiv 13\%$). The savings on dynamic energy are slightly higher. They come from the substantial reduction of the number of misses, and also from the reduction of writes on the cache for data blocks that are bypassed to the L2 cache.

For DRAM energy, the savings are also slightly higher on dynamic consumption than on static consumption. The static energy reduction is proportional to the reduction of the execution time ($1 - \frac{1}{1.17} \equiv 15\%$). For the dynamic energy, the reduction comes essentially from the reduction of the number of accesses, and also on the change in the access patterns on the DRAM which improves the row buffer hit rate.

Compression/Decompression: While energy is saved at the memory components (LLC and DRAM), additional energy is spent on compression/decompression. From our evaluation, energy for compressing a block with the DISH compression scheme is about 15% of the energy of a read access on the cache. Energy for decompressing a block is 3% of the energy for a read access. This leads to an overall compression/decompression energy in the 4% range of the overall LLC energy since i) all blocks in the LLC are not compressed and therefore not decompressed, ii) the blocks bypassed with the FITFUB policy do not flow through the compression hardware. Overall, the extra energy spent on compression/decompression is much smaller than energy saved at the memory system.

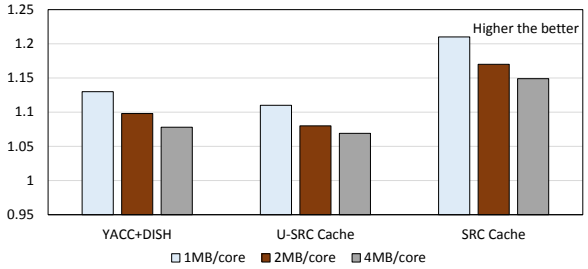

Figure 13: Effect of LLC size: Performance normalized to an uncompressed baseline for multi-core systems.

6.4 Sensitivity Studies

LLC size: To understand the effect of LLC size on SRC cache, we simulate multi-core systems with 1MB, 2MB and 4MB per core. Fig. 13 illustrates the average performance for different cache sizes for multicores. As expected, the performance benefit is higher for a small cache (1MB per core), but is still significant for the larger configuration (4MB per core).

Cache Replacement Policies: In our evaluation, SRC cache uses SHiP++ [31] as its underlying cache replacement policy. We study the effectiveness of SRC cache with different cache insertion/promotion priority policies such as not recently reused (NRR) [4], hawkeye [12], and sampling dead-block predictor (SDBP) [15]. Figure 14 illustrates the performance difference when we change the cache replacement policy. That is the column (SRC, NRR) illustrates the performance improvement of SRC with NRR over SRC with SHiP++. Similarly the column (baseline, hawkeye) illustrates the performance ratio of baseline with hawkeye over baseline with SHiP++. Overall, on our experimental set, the average performance difference between these replacement policies remains very small ($< 2\%$) for all the cache structures. That is for all the tested policies, SRC outperforms the baseline configuration by 17% to 18%. This makes a strong case for using FITFUB as a block allocation policy. However LLC management policies answer two different different questions i) on a miss, whether the block will be allocated in the LLC? ii) which is the target for replacement ? The FITFUB allocation policy only answers to the first question and can be associated directly with most of the previously proposed LLC management policies to manage the YACC cache. There are many possible optimizations around FITFUB and optimizing the replacement target policy that could be explored. For instance, one could explore the priority level at which compressed blocks and uncompressed blocks should be inserted, one could explore the priority level at which compressed

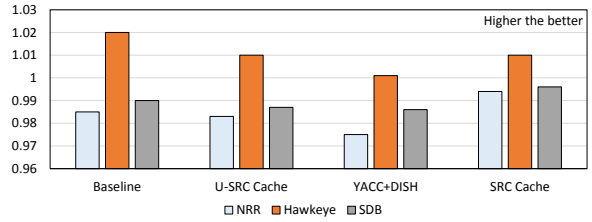

Figure 14: Effect of cache management policies on SRC Cache: Normalized performance to the same cache structure that uses SHiP++. Results are for multi-core systems.

blocks and uncompressed blocks should be promoted on a hit, one can predict when not to bypass. All these optimizations might bring some additional performance gains on SRC, but are left for future work.

Compression Technique: So far, we have evaluated SRC cache with DISH technique. However, as SRC cache is independent of the underlying compression schemes, we evaluate SRC cache with BDI, CPACK+Z, and FPC compression techniques that use YACC layout. Compared to the baseline uncompressed cache, SRC cache that uses BDI, CPACK+Z and FPC provide performance improvements of 13.8%, 11.2% and 10.3%, respectively, on multi-core systems. Therefore, SRC cache is effective across different compression techniques providing high performance gain and energy savings. Unsurprisingly, DISH that was designed especially for the YACC cache layout, achieves the best performance improvement (17%).

7 RELATED WORK

To the best of our knowledge, this is the first study on replacement/insertion policy for compressed caches that associate a single super-block tag to a fixed data entry, i.e., YACC or SCC. LLC replacement policies for uncompressed caches and their possible adaptation to the SRC context have been previously discussed. On the other hand, there has been a limited number of studies on optimizing insertion/replacement policies for compressed caches. A few studies [7, 11, 20] address cache designs where a tag is not associated with a fixed data entry, or several tags are associated with a data entry.

ECM [7] proposes size aware cache replacement policies by taking the compressed size of the cache blocks into account for evicting LLC blocks. CAMP [20] outperforms size aware cache replacement policies by dynamically prioritizing cache blocks using their compressed block sizes. In the same direction, on SRC, one could prioritize promotion/insertion of data entries based on their compression status and/or on the number of valid compressed blocks in the data entry.

Gaur et al. [11] acknowledge that replacement decision is a nightmare on compressed caches with multiple tags associated with the same data entry. For a compressed cache allowing a maximum of two compressed blocks per data entry, they propose the base-victim compression cache. On the base-victim compression cache, a data entry stores a main block and possibly a victim block. The replacement policy only considers the main block. On a miss on block B as main block (either a real miss or a hit on a victim block), one allocates a data entry for block B, and one tries to keep the evicted main block (if compressible) as companion victim block with some main block. This elegant policy guarantees that the sequence of *misses as main blocks* is the same as the sequence of misses on an uncompressed cache with the same replacement policy.

8 CONCLUSION

To accommodate cache compression, the YACC [24] layout appears as a very promising solution, particularly when associated with an adapted compression scheme such as DISH [19]. It achieves significant performance improvement compared with a conventional cache design at a very limited storage overhead (1.5 %).

The structure of the YACC layout is very similar to the one of a conventional set-associative cache apart that that it associates a super-block tag with each data entry (aka line in the data array) instead of a block tag in order to map the co-compressible of a super-block in a single data entry. The performance of a cache also depends on its management and particularly on the decision of allocating in the LLC or bypassing blocks fetched from the memory. On YACC, the super-block tag offers us the opportunity to implement the FITFUB allocation and bypass policy. After a first uncompressed block from a super-block has been allocated in the cache, its super-block tag can be used to monitor the reuse usage of its companion blocks in the super-block. This allows to bypass these companion blocks on their first use, and store them (and allocate a data entry for them) only on the second use of the block. FITFUB can be implemented on top of any priority insertion/promotion policy for the cache.

Our experiments show that, the benefit from cache compression plus FITFUB is in the order of 12.7% and 17% for single core and multi-core workloads, respectively. These benefits also translate in energy savings on the memory system (LLC and DRAM).

As a global proposition, SRC appears as very cost effective design leveraging cache compression and an effective allocation and bypass policy at very limited hardware overhead: the compression/decompression logic and 1.5% storage overhead.

REFERENCES

[1] M. Ahmad, F. Hijaz, Qingchuan Shi, and O. Khan. 2015. CRONO: A Benchmark Suite for Multithreaded Graph Algorithms Executing on Futuristic Multicores. In *Workload Characterization (IISWC), 2015 IEEE International Symposium on*. 44–55. https://doi.org/10.1109/IISWC.2015.11

[2] J. Ahn, S. Yoo, and K. Choi. 2014. DASCA: Dead Write Prediction Assisted STT-RAM Cache Architecture. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 25–36. https://doi.org/10.1109/HPCA.2014.6835944

[3] A. R. Alameldeen and D. A. Wood. 2004. Frequent Pattern Compression: A Significance-Based Compression Scheme for L2 Caches.. In *Technical Report 1500, Computer Sciences Department, University of Wisconsin-Madison*.

[4] Jorge Albericio, Pablo Ibáñez, Víctor Viñals, and Jose María Llabería. 2013. Exploiting Reuse Locality on Inclusive Shared Last-level Caches. *ACM Trans. Archit. Code Optim.* 9, 4, Article 38 (Jan. 2013), 19 pages. https://doi.org/10.1145/2400682.2400697

[5] Jorge Albericio, Pablo Ibáñez, Víctor Viñals, and José M. Llabería. 2013. The Reuse Cache: Downsizing the Shared Last-level Cache. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 310–321. https://doi.org/10.1145/2540708.2540735

[6] Angelos Arelakis and Per Stenstrom. 2014. SC2: A Statistical Compression Cache Scheme. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 145–156. http://dl.acm.org/citation.cfm?id=2665671.2665696

[7] S. Baek, H. G. Lee, C. Nicopoulos, J. Lee, and J. Kim. 2013. ECM: Effective Capacity Maximizer for high-performance compressed caching. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*. 131–142. https://doi.org/10.1109/HPCA.2013.6522313

[8] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*. ACM, New York, NY, USA, 72–81. https://doi.org/10.1145/1454115.1454128

[9] 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. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[10] Xi Chen, Lei Yang, Robert P. Dick, Li Shang, and Haris Lekatsas. 2010. C-PACK: A High-performance Microprocessor Cache Compression Algorithm. *IEEE Trans. Very Large Scale Integr. Syst.* 18, 8, 1196–1208. https://doi.org/10.1109/TVLSI.2009.2020989

[11] Jayesh Gaur, Alaa R. Alameldeen, and Sreenivas Subramoney. 2016. Base-victim Compression: An Opportunistic Cache Compression Architecture. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. IEEE Press, Piscataway, NJ, USA, 317–328. https://doi.org/10.1109/ISCA.2016.36

[12] Akanksha Jain and Calvin Lin. 2016. Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. IEEE Press, Piscataway, NJ, USA, 78–89. https://doi.org/10.1109/ISCA.2016.17

[13] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely, Jr., and Joel Emer. 2008. Adaptive Insertion Policies for Managing Shared Caches. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*. ACM, New York, NY, USA, 208–219. https://doi.org/10.1145/1454115.1454145

[14] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. 2010. High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP). In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. ACM, New York, NY, USA, 60–71. https://doi.org/10.1145/1815961.1815971

[15] Samira Manabi Khan, Yingying Tian, and Daniel A. Jimenez. 2010. Sampling Dead Block Prediction for Last-Level Caches. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '43)*. IEEE Computer Society, Washington, DC, USA, 175–186. https://doi.org/10.1109/MICRO.2010.24

[16] Kun Luo, J. Gummaraju, and M. Franklin. 2001. Balancing thoughput and fairness in SMT processors. In *Performance Analysis of Systems and Software, 2001. ISPASS. 2001 IEEE International Symposium on*. 164–171. https://doi.org/10.1109/ISPASS.2001.990695

[17] R Manikantan, Kaushik Rajan, and R Govindarajan. 2011. NUcache: An Efficient Multicore Cache Organization Based on Next-Use Distance. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA '11)*. IEEE Computer Society, Washington, DC, USA, 243–253. http://dl.acm.org/citation.cfm?id=2014698.2014862

[18] Naveen Muralimanohar and Rajeev Balasubramonian. [n. d.]. CACTI 6.0: A Tool to Understand Large Caches. ([n. d.]).

[19] Biswabandan Panda and André Seznec. 2016. Dictionary Sharing: An Efficient Cache Compression Scheme for Compressed Caches. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, 2016*. IEEE/ACM, Taipei, Taiwan. https://hal.archives-ouvertes.fr/hal-01354246

[20] G. Pekhimenko, T. Huberty, R. Cai, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. 2015. Exploiting compressed block size as an indicator of future reuse. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. 51–63. https://doi.org/10.1109/HPCA.2015.7056021

[21] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2012. 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)*. ACM, New York, NY, USA, 377–388. https://doi.org/10.1145/2370816.2370870

[22] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. 2007. Adaptive Insertion Policies for High Performance Caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA '07)*. ACM, New York, NY, USA, 381–391. https://doi.org/10.1145/1250662.1250709

[23] Somayeh Sardashti, André Seznec, and David A. Wood. 2014. Skewed Compressed Caches. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE Computer Society, Washington, DC, USA, 331–342. https://doi.org/10.1109/MICRO.2014.41

[24] Somayeh Sardashti, Andre Seznec, and David A. Wood. 2016. Yet Another Compressed Cache: A Low-Cost Yet Effective Compressed Cache. *ACM Trans. Archit. Code Optim.* 13, 3, Article 27 (Sept. 2016), 25 pages. https://doi.org/10.1145/2976740

[25] Somayeh Sardashti and David A. Wood. 2013. 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)*. 62–73.

[26] Vivek Seshadri, Onur Mutlu, Michael A. Kozuch, and Todd C. Mowry. 2012. The Evicted-address Filter: A Unified Mechanism to Address Both Cache Pollution and Thrashing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT*

'12). ACM, New York, NY, USA, 355–366. https://doi.org/10.1145/2370816.2370868

[27] André Seznec. 1993. A Case for Two-way Skewed-associative Caches. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA '93)*. ACM, New York, NY, USA, 169–178. https://doi.org/10.1145/165123.165152

[28] Allan Snavely and Dean M. Tullsen. 2000. 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)*. ACM, New York, NY, USA, 234–244. https://doi.org/10.1145/378993.379244

[29] Cloyce D. Spradling. 2007. SPEC CPU2006 Benchmark Tools. *SIGARCH Computer Architecture News* 35 (March 2007). Issue 1.

[30] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. 2007. 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)*. IEEE Computer Society, Washington, DC, USA, 63–74. https://doi.org/10.1109/HPCA.2007.346185

[31] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C. Steely, Jr., and Joel Emer. 2011. SHiP: Signature-based Hit Predictor for High Performance Caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. ACM, New York, NY, USA, 430–441. https://doi.org/10.1145/2155620.2155671