In the proceedings of the 23rd Conference on Design, Automation and Test in Europe (DATE)

2DCC: Cache Compression in Two Dimensions

Amin Ghasemazar
University of British Columbia
aming@ece.ubc.ca

Mohammad Ewais*
University of Toronto
mewais@ece.utoronto.ca

Prashant Nair
University of British Columbia
prashantnair@ece.ubc.ca

Mieszko Lis
University of British Columbia
mieszko@ece.ubc.ca

***Abstract*—The importance of caches for performance, and their high silicon area cost, have motivated hardware solutions that transparently compress the cached data to increase effective capacity without sacrificing silicon area. To this end, prior work has taken one of two approaches: either (a) deduplicating identical cache blocks across the cache to take advantage of *inter-block* redundancy or (b) compressing common patterns within each cache block to take advantage of *intra-block* redundancy.**

In this paper, we demonstrate that leveraging only one of these redundancy types leads to a significant loss in compression opportunities for several applications: some workloads exhibit either inter-block or intra-block redundancy, while others exhibit both. We propose 2DCC (Two Dimensional Cache Compression), a simple technique that takes advantage of both types of redundancy. Across the SPEC and Parsec benchmark suites, 2DCC results in a 2.12× compression factor (geomean) compared to 1.44–1.49× for best prior techniques on an iso-silicon basis. For the cache-sensitive subset of these benchmarks run in isolation, 2DCC also achieves a 11.7% speedup (geomean).

I. Introduction

Large caches are critical to the performance of many applications: today, top-tier server-class processors from leading manufacturers advertise last-level cache (LLC) capacities in the range of 32MB–64MB. However, caches of these sizes incur significant costs in silicon area, leakage power, and cache access latency. This has resulted in substantial research interest in compressing cache contents to increase the effective cache capacity without paying additional area and performance costs [1–8]. In general, these proposals take advantage of *either* of two different dimensions of redundancy in cached data:

1) *Inter-block redundancy* exists when multiple cache indices store the same blocks of data. This can result from symmetry of some kind (e.g., fluid flow around a symmetric object), when sizable parts of the working set have the same value (e.g., the background of an image), etc.
2) *Intra-block redundancy* exists when a single cache block contains compressible patterns within itself. For example, integers are usually allocated at 32-bit or 64-bit sizes but their values often fit in the least significant byte; similarly, pointers used in a data structure may have been allocated close by and so may have identical most significant bits.

Real workloads, however, exhibit a wide variety of redundancy patterns. To demonstrate this, we estimated intra-block and inter-block entropy in 100 last-level cache images from a range of SPEC and PARSEC benchmarks (see section IV for details) by using Huffman compression [9]. To estimate inter-block entropy, we compressed the entire cache using 64-byte symbols (i.e., one cache block); to estimate intra-block entropy, we compressed each block independently using one-byte symbols.

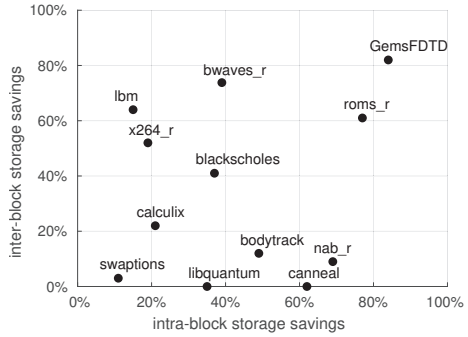

Fig. 1. Space reduction in last-level cache images (100 per benchmark) using entropy encoding (Huffman compression). The x-axis shows encoding within each cache block (symbol size of 1 byte); the y-axis shows encoding across cache blocks (symbol size 64 bytes). 0% indicates that no compression was possible while 100% would indicate that the entire cache was completely compressible. These results provide a motivation for 2D cache compression.

Figure 1 shows how much space can be recovered for each benchmark by taking advantage of inter-block entropy (y-axis) and intra-block entropy (x-axis). Some benchmarks show significant savings by using only one type of redundancy: for example, *lbm* has many identical blocks which are generally not amenable to intra-block compression, while the blocks cached by *canneal* have intra-block value redundancy but most cache blocks are different. Others, such as *bwaves* and *roms*, contain a mixture of identical blocks and some compressible blocks. (The outlier, *GemsFDTD*, has nearly all of its working set filled with zeros and is therefore trivially compressible.)

This paper describes 2DCC, a practical cache compression scheme that works in two dimensions: it allows working sets that contain *either* type of redundancy to be compressed while also enabling compressing working sets that contain *both* types of redundancy. Because this requires decoupling cache structures, replacement policies become a challenge. 2DCC uses separate replacement policies for the tag array and the data array, which optimizes for both reuse and space savings.

Taking advantage of both types of redundancy allows 2DCC to outperform prior state-of-the-art solutions. When applied to the LLC in a server-class CPU, 2DCC achieves 2.12× geomean compression factor across cache sensitive subset of SPEC CPU2017 [10], SPEC CPU2006 [11], and PARSEC [12] — compared to 1.43×–1.49× with best prior methods given the same silicon budget — resulting in a geomean 11.7% speedup.

II. The opportunity for 2D compression

As shown in Figure 1, redundancy in workloads varies widely. Some benchmarks have only intra-block redundancy, some only

*Research performed while a student at the University of British Columbia

978-3-9819263-4-7/DATE20/©2020 EDAA

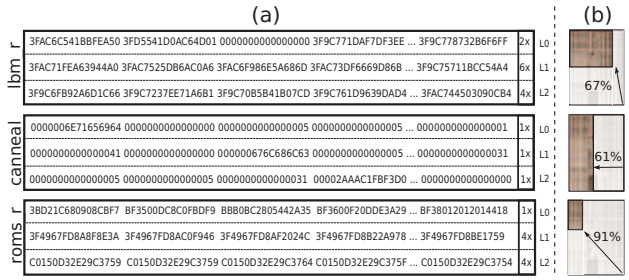

Fig. 2. (a) Redundancy in LLC snapshots of three benchmarks: *lbm_r* shows inter-block redundancy: the three cache lines shown appear twice, 6 times, and 4 times; *canneal* shows different blocks each of which has a compressible 0 prefix; in *roms_r*, blocks appear in multiple copies but words also have similar prefixes. (b) Cache space saved using inter-block compression (y-axis) as well intra-block compression (x-axis) for these LLC samples.

inter-block redundancy, and there are several workloads that showcase both types of redundancy.

For example, Figure 2(a) shows cache block fragments of last-level cache snapshots for three benchmarks, along with the number of exact copies of each block found in the cache.

The top panel shows three blocks of the destination grid written inside *LBM_performStreamCollideTRT()* in *lbm_r*. The cache block is filled with 64-bit floats, which differ enough that intra-block compression (e.g., BΔI [1]) is ineffective. Because of fluid flow symmetry, there are multiple copies of many cache blocks, all of which can potentially be deduplicated, allowing the size of the cache snapshot to be reduced by 67%.

The middle panel shows three blocks addressed by *swap_locations()* in *canneal*. In contrast to *lbm_r*, the working set contains no duplicate blocks. However, there is substantial intra-block redundancy: the data consists mainly of small 32-bit integers (netlist elements and locations). This allows the cache snapshot size to be reduced by 61%.

Finally, the third panel shows cache blocks from *roms_r*, an ocean forecasting model. The locality of behaviour within an surface patch, together with similarities across some patches, creates both intra-block and inter-block redundancy: many cache blocks in the working set are present in several copies, and each contains 64-bit floats that are close to one another. For these cache snapshots, taking advantage of both forms of entropy can potentially save 91% of the cache space.

Figure 2(b) shows the potential cache silicon savings for intra-block (x-axis) and inter-block (y-axis) entropy by using an ideal compression method on the cache snapshots analyzed.

III. 2DCC architecture and operation

Briefly, when a 2DCC cache inserts a new block, it checks whether an identical block is already present; if the block is a duplicate, then a reference to the existing block is inserted instead. If the block is unique, 2DCC attempts to compress it and store it in a part of a line in the cache's data array, with the rest of the line usable by other compressed cache blocks.

This approach presents several challenges. Firstly, duplicate cache blocks must be detected quickly. Secondly, allocating and evicting blocks with different compression factors must not cause fragmentation. Finally, the varying compressibility of workloads means that the cache may be limited by either tag storage or data storage, with each storage structure requiring a separate and different replacement policy.

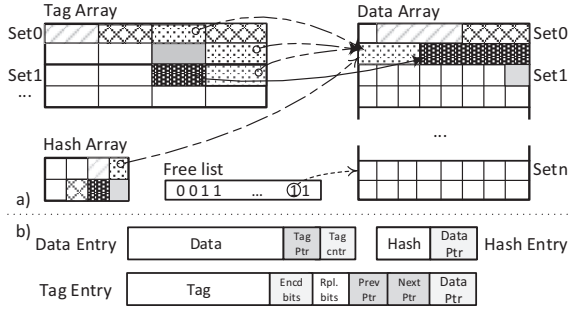

Fig. 3. (a) The three decoupled storage structures that comprise the 2DCC cache: arrows show pointers logically linking the structures. (b) Entry contents.

Storage structures. Unlike conventional caches, which store one full (e.g., 64-byte) data block for every tag, compressed caches can either store multiple blocks in the same space [1, 3, 4, 13] or store only one block for multiple tags [2]; 2DCC similarly decouples the tag and data arrays. In contrast to prior approaches, each tag may point to an 8-byte segment anywhere in the data array rather than to only one index or a few possible locations; this maximizes data array utilization. To avoid storing duplicate blocks, multiple tags may point to the same segment.

To detect inter-block redundancy, 2DCC adds a third structure — the *hash array* — which stores summaries of cached blocks and allows the controller to quickly identify duplicate lines.

Figure 3 illustrates the structure of a 2DCC cache, and shows how the three components are interconnected with pointers.

Data Array. Storage of variable-sized blocks is accomplished by segmenting each set in the data array into eight-byte segments (similar to prior work [1]): a single cache block may occupy from one up to eight contiguous segments depending on the compression factor.

Because the tag array is decoupled from the data array (unlike in [1]) and the cache can store more tags than uncompressed blocks, 2DCC may need to evict blocks when space in the data array runs out even if some tags are still free. To identify the tags that point to a given data segment, 2DCC uses a per-segment back-pointer to one of the corresponding tags. To support the data array replacement policy, each segment also stores a count of tags pointed to it. A free-list bit vector is used to allocate entries and manage free space in the data array.

Tag Array. As in a conventional set-associative tag array, each entry contains the tag itself, tag replacement policy state, and validity/coherence state. Each entry also specifies the compression encoding. The tag entry also contains a "data pointer" to identify the segment(s) storing the cached block.

Finally, multiple tags that point to the same data segment form a doubly-linked list, used to remove all tags associated with an evicted block, and to form a free-list of unused tags.

Hash Array. To detect identical cache blocks, 2DCC needs to compare the *contents* of an incoming block with the contents of blocks that are already cached. Naturally, scanning through the entire cache is not an option. Instead, 2DCC detects candidates for deduplication by storing hashes of block contents in a separate small hash array. Because the common case is that

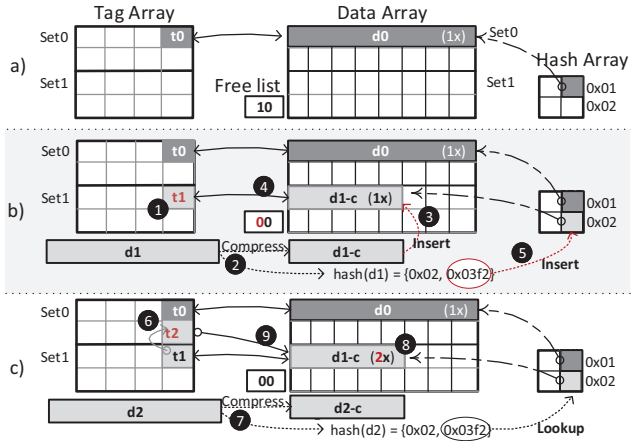

Fig. 4. Example of 2DCC operation on lines from roms_r benchmark: (a) initial state; (b) insertion of unique but compressible block; (c) insertion of a new block whose data is identical to that of panel (b).

incoming lines are unique, the hash array essentially serves to filter out most lines that cannot be deduplicated.

The hash array is a set-associative table. Each entry points to the data array segment where the original block is stored. (This is safe even if the original block is modified or evicted, as hash collisions mean that hash matches must in any case be verified against the full cache block.) Based on our experiments, storing only 1024 hashes is the hash array is sufficient to capture nearly all possible deduplication while reducing the hash collisions to less than 1%.

In operation, each incoming block's contents are hashed using a 64-bit H_3 hash [14]. If the hash is not found, insertion proceeds normally. If the hash matches (i.e., a duplicate block may already exist in the cache), the block it points to is fetched from the data array and the actual data are compared; if the data are identical, then the line is deduplicated, otherwise it is inserted as a new block.

Cache operation example. We begin by tracing "the life of a cache block" on a tiny version of 2DCC in Figure 4 with an example from the roms_r ocean simulation workload, before detailing the rules of operation. We begin with the state in panel (a), with one uncompressible, unduplicated block in the cache with tag t0 and data d0, such as block L0 in Figure 2(a).

In panel (b), a lower-level cache requests an address with tag *t1*, which misses in the tag array ❶ and triggers a backing memory request for its data *d1*. When *d1* arrives, it is compressed to *d1-c*, and, in parallel, hashed to search for duplicates ❷. The hash is then looked up in the hash array to determine whether the block can be deduplicated, but as it is the first occurrence of this data, the lookup fails.

To insert the new block in the cache, the controller consults the freelist to find that set 1 has a free block, and the compressed *d1-c* is inserted there ❸. At the same time, *t1* is inserted in the tag array with its data pointer set to point at *d1-c* and vice versa ❹, and the hash for *d1* is inserted in the hash array ❺.

In panel (c), a lower-level cache requests an address with tag *t2*, which also misses in the tag array ❻; this triggers another memory request. Once data block *d2* arrives, it is compressed (to *d2-c*) and hashed as before ❼. This time, however, the

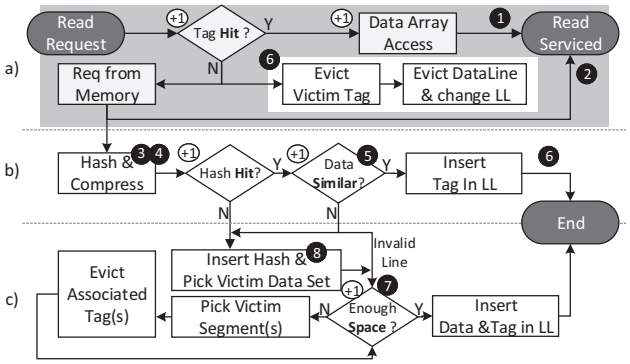

Fig. 5. Read access flowchart in 2DCC. Shaded (a) = on the critical path; unshaded (b, c) = off the critical path. LL = linked list of deduplicated tags.

hash hits in the hash array with the entry pointing to *d1-c*, indicating that *d2* is a possible duplicate of *d1*; to verify this, *d1-c* is retrieved and compared against *d2-c*. An exact match is determined, the deduplication count in *d1-c* is incremented ❽, and *t2* is inserted into the tag array pointing to *d1-c* ❾.

Cache operation details. 2DCC operation is largely similar to that of a conventional cache, with some differences due to tag/data array decoupling and the need to deduplicate and compress stored data. We detail those differences below.

Reads, evictions, and insertions. The operation of these accesses is illustrated in Figure 5. The critical-path portion of read accesses — both hit ❶ and miss ❷ — corresponds to a conventional cache.

The hash and the compressed line size are calculated off the critical path ❸ ❹. If the hash exists in the hash array, the new block is a deduplication candidate, and the existing block is retrieved from the data array and compared against the newly arrived data ❺ to determine if the block is a duplicate.

If the entry is to be deduplicated, an unused tag is obtained either from the tag free list or by evicting an existing tag ❻. If the entry cannot be deduplicated, a data entry is also allocated, possibly following an eviction of some data segments and their associated tags ❼. If the entry was not deduplicated, its hash is also inserted into the hash array to enable future deduplicaton ❽.

Writes. Writes reflect those in a conventional cache: with inclusive write-back caches, which we use in this paper, write requests always hit, and execute off the critical path.

Writes may also change the compressed size. In parallel to the tag access, therefore, the hash of the contents is computed; if this hits in the hash array, the relevant block is fetched and compared to the newly written data. If the newly written block can be deduplicated, the data pointer swings to the existing copy and the redundant segments are freed.

If the written block cannot be deduplicated, it is first re-compressed. If it fits in the same number of segments, the data array entry is overwritten, possibly freeing some segments. If the line is larger, victim segments are selected from the data array before inserting the block as if it were a new insertion.

Intra-block compression/decompression. For compressing individual cache blocks, we use the BΔI compression method [1]. Briefly, BΔI calculates the mean of the words in the block to

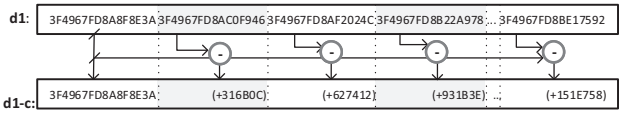

Fig. 6. Example of compressing a 64-byte cache block from roms_r (*d1* from Figure 4). The block consists of 64-bit floating-point numbers whose values are close; they are compressed to a 64-bit base value followed by eight 32-bit offsets, for a total compressed size of 36 bytes.

determine the number of bytes needed to express the distance from this base value or from 0. If all distances can be expressed in fewer bytes than the original value (e.g., 4 bytes), the compressed block consists of the base value followed by a sequence of distance offsets used to reconstruct the original words in the cache block. Decompression consists of adding the offsets to the base, and is completed in one cycle.

Figure 6 shows an example of this process. The block (*d1* from Figure 4) consists of 64-bit floating-point numbers whose values are close. The intra-block compression reduces the block to 40 bytes (an 8-byte base value followed by eight 4-byte offsets), and compacts it to take up 5 segments in the set.

Replacement Policies. As described in earlier, 2DCC has three decoupled structures: a tag array, a data array, and a hash array. Unlike in a conventional cache, the three arrays have different goals and need different replacement policies.

Tag array. The goal of the tag array replacement policy (RP) is to preferentially retain addresses likely to be accessed in the future. The RP should therefore be the same as the equivalent conventional cache RP. In this paper, we use the least-recently-used victim selection with most-recently used insertion (LRU), but other eviction policies may be more appropriate for large caches and specific workloads [15, 16, etc.].

Data array. The data array, on the other hand, provides *storage space* for the blocks identified as likely to be re-referenced. The storage is many-to-one: when several cache blocks contain the same data, one data array entry will be shared among several tags. When evicting a data array entry, all of the tag array entries that point to it must also be evicted. This makes conventional cache victim policies, which do not account for the cost of evicting multiple tags, unsuitable.

Observe that the policy does *not* need to consider which blocks are likely to be re-referenced, as the tag array replacement policy already ensures that only useful blocks are cached. The goal of the data array, therefore, should be to *enable the tag array to store more blocks*. Our policy has three stages:

1) If a set in the data array is free, insert the block there.
2) Otherwise, attempt to find space in a partly occupied block: randomly select four sets, and, if one of them has enough space, insert the new block there.
3) Finally, examine the four blocks from step 2, and select the one that (a) has enough space, and (b) minimizes the number of evicted tags from the tag array.

In effect, this process combines a random sampling process with a selection policy that retains the most deduplicated entries.

Hash array. The main purpose here is to enable deduplication of blocks stored in the data array. Thus, the hash array should identify (a) currently cached blocks whose contents are likely to reappear in other, soon-to-be-accessed blocks, and (b) incoming blocks whose contents are likely to reappear later. We therefore use the LRU policy applied to content hashes.

IV. Results

Methods. We extended ZSim [17] to implement 2DCC and the state-of-the-art hardware compression techniques for intra-block compression (BΔI [1]) and inter-block deduplication (Dedup [2]). We modeled detailed event timing and interconnect congestion for both on- and off-critical-path events. The simulated system is shown in Table I; compression was applied to the L3 level only. We used CACTI 6.0 [18] to estimate silicon area requirements, including all data structures for each compression method. For all performance studies, we normalized the three designs to the same silicon area.

We used an extensive set of integer and floating-point applications from SPEC CPU2017 [10] and PARSEC [12], as well as those applications from SPEC CPU2006 [11] that are not in CPU2017. All were run with large input sizes (native in Parsec and reference in SPEC). Simulations skipped the first 40 billion instructions, and then sampled the last 20% of each 1 billion instructions for a total of 40 billion instructions.

Sizing data structures. Sizing decoupled structures (tag, data, and hash arrays) under a fixed silicon area budget is key to our design. In 2DCC, we must make two sizing decisions: (a) the ratio of tags to raw data blocks (which must exceed 1 to enable compression) and (b) the size of the hash array that captures inter-block redundancy.

Tag array vs. data array. We observed that the compressibility of cache blocks varies not only among applications, but also among different phases within an application, from as low as 1× in *streamcluster* to more than 10× in *fotonik3d_r*. Similar to prior work [2], we allow the cache to store four times more compressed lines than uncompressed lines.

Hash Table. For the hash array, the tradeoff is between, on the one hand, reducing the silicon footprint to make more space for tags and data entries and, on the other hand, making it large enough to capture enough of the inter-block redundancy.

To examine the design space, we compared an oracle hash table — which searches the entire cache for a match — against hash array sizes from 64 to 16,384 entries. In our experiments, 99.2% of the locality was captured with 1,024 entries (64 sets × 16 ways), with a collision rate of < 1%. We use this hash size for the remainder of the experiments.

Silicon area allocation. We configured 2DCC as well as our three baselines to match that of a conventional, uncompressed cache with 1MB of data storage. For the compressed caches,

CPU	i5-750-like: x86-64, 2.6GHz, 4-wide OoO, 80 entry ROB
L1I	32KB, 4-way, 3 Cycle access lat, 64B lines, LRU
L1D	32KB, 8-way, 4 Cycle access lat, 64B lines, LRU
L2	Private, 256KB, 8 way, 11 Cycles lat, 64B lines, LRU
L3	Shared 1 MB, 8-way, 39 Cycles lat, 64B lines, 8 banks
Memory	DDR3-1066, 1GB

TABLE I
Configuration of the simulated system.

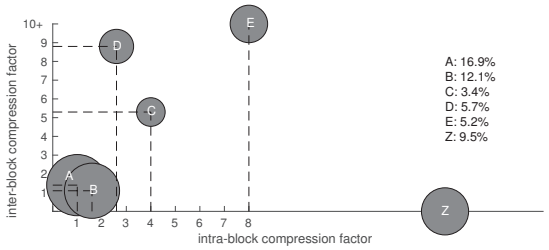

Fig. 7. The bubble sizes represent storage savings due to combined intra- and inter-block compression, plotted against different compression factors. Z is the amount of savings due to all-zero blocks.

the total space available in the data array is less that 1MB because more of the available silicon budget must be dedicated to tags; Table II shows space allocation details.

Metadata for each conventional cache tag entry consists of valid/dirty/LRU bits. 2DCC adds 4 bits for the compression type encoding, 32 bits for the previous and next tag pointers, and 17 bits for the data pointer (11 bits to index the set and 6 bits to index the segment within that set). BΔI adds 10 bits over the conventional cache for compression-type encoding and the segment pointer. Finally, Dedup tag entries add 32 bits for tags pointers and 14 bits for the data pointer. Data array overheads in 2DCC are 16 bits per segment for the tag list pointer, while Dedup has one 16-bit pointer for each cache block. 2DCC also requires a hash array, with each entry consisting of 10 bits for the hash tag and 17 bits for the data segment pointer; Dedup has a similar table but uses only 14 bits for the data pointer.

Effectiveness of 2D compression. Figure 7 shows the inter-block compression factor for each possible intra-block compression factor averaged over all benchmarks; the bubble size shows how much cache area was saved due to a specific combination. The largest savings — 16.9% of cache — come from blocks that cannot be compressed by themselves, but can be deduplicated, on average, 1.4× (bubble A). The next 12.1% is saved by blocks that cannot be deduplicated, but are amenable to intra-block compression with a compression factor of 1.6× on average (B). Significant additional savings (14.3% cache space total) come from blocks that are both compressible within each block but also identical to other blocks (C, D, E). Finally, 9.5% cache space is saved by using a special representation for zero-only blocks.

This validates the 2DCC intuition: both choosing the appropriate compression for each block (A, B) and using both compression methods in the same block (C, D, E) are important.

		Baseline	BΔI [1]	Dedup [2]	2DCC
Tag	#Entries	16384	49152	40960	36864
	Entry Size	39b	49b	85b	93b
	Total Size	78KB	294KB	425KB	414KB
Data	#Entries	16384	12288	10240	9216
	Entry Size	512b	512+0b	512+16b	512+104b
	Freelist				1152b
	Total Size	1024KB	768KB	660KB	694KB
Hash	# Entries	-	-	1024	1024
	Entry Size	-	-	3B	3.375B
	Total Size	-	-	3KB	3.375KB
	Total Size	**1.08MB**	**1.05MB**	**1.07MB**	**1.08MB**

TABLE II
STORAGE ALLOCATION. ALL COMPRESSED CACHES ARE SIZED TO FIT IN THE SAME SILICON SIZE OF A 1MB CONVENTIONAL CACHE WITH 48-BIT ADDRESS SPACE.

Cache	Size(MB)	Dynamic read energy	Leakage power
Conv.	1	0.35 nJ	677.66 mW
BΔI	1	0.37 nJ	679.21 mW
Dedup	1	0.39 nJ	699.70 mW
2DCC	1	0.39 nJ	695.16 mW

TABLE III
DYNAMIC ENERGY AND LEAKAGE POWER OF COMPRESSED CACHES AND CONVENTIONAL OF 1MB (SILICON AREA OF $2.52mm^2$) IN $32nm$ TECHNOLOGY.

Cache footprint. Figure 8(a) shows the cache space needed by different workloads using 2DCC compared to state-of-the-art methods for intra- (BΔI [1]) and inter-block compression (Dedup [2]), normalized to a conventional cache. We report averages over the entire program runtime from an execution-driven simulation (see Section IV). All caches take the same silicon area as a 1MB conventional cache (see Table II).

On average, 2DCC is able to reduce the cache footprint to 47.2% of the original footprint (i.e., 2.1× compression), a substantial improvement over BΔI (67.1%) and Dedup (69.2%).

Performance. We divided the benchmarks into cache sensitive (S) and cache insensitive (NS): we consider a benchmark to be cache insensitive if there is < 3% change in MPKI when the conventional LLC size is doubled (this typically means that their workloads mostly fit in the L2 or even L1D cache).

Figure 8(b) shows that 2DCC reduces cache misses per 1,000 instructions (MPKI) by 1.6× compared to 1.3× for BΔI and 1.2× for Dedup on average for the cache sensitive benchmarks. At the same time, the MPKI impact of cache compression on the cache-insensitive benchmarks is negligible (1.6%)

Figure 8(c) shows that the lower MPKI allows 2DCC to improve performance (IPC) by 11.7% for the cache-sensitive benchmarks, vs. 7.3% for BΔI and 5.2% for Dedup. Cache-insensitive benchmarks can suffer a slight performance degradation (avg. 2.6%): for example, *bwaves* is highly compressible but cache-insensitive, so the compression/decompression latencies are not offset by more frequent cache hits.

We also investigated whether evictions of multiple tags are a significant problem, by measuring the ratio of evicted tags to cache accesses. Because of its better compression, 2DCC has the lowest eviction rate of 0.032 evictions per access, compared to 0.049, 0.042, and 0.041 for the conventional cache, BΔI, and Dedup, respectively. This means that multi-tag evictions are very rare, and do not have any performance impact.

Energy impact. We used CACTI [18] to measure the latency, read energy, and leakage power of 2DCC and the three baselines (see Table III); results show that 2DCC uses 11% more energy for each read, and has a 2.5% leakage power overhead.

V. RELATED WORK

Inter-block deduplication: Data deduplication techniques work well when many cache blocks are either entirely zero [1, 3] or copies of other blocks that concurrently reside in the cache [2, 19]. 2DCC leverages many of the same insights to take advantage of inter-block redundancy, but also takes advantage of intra-block redundancy, which implies substantial differences in the overall structure and operation.

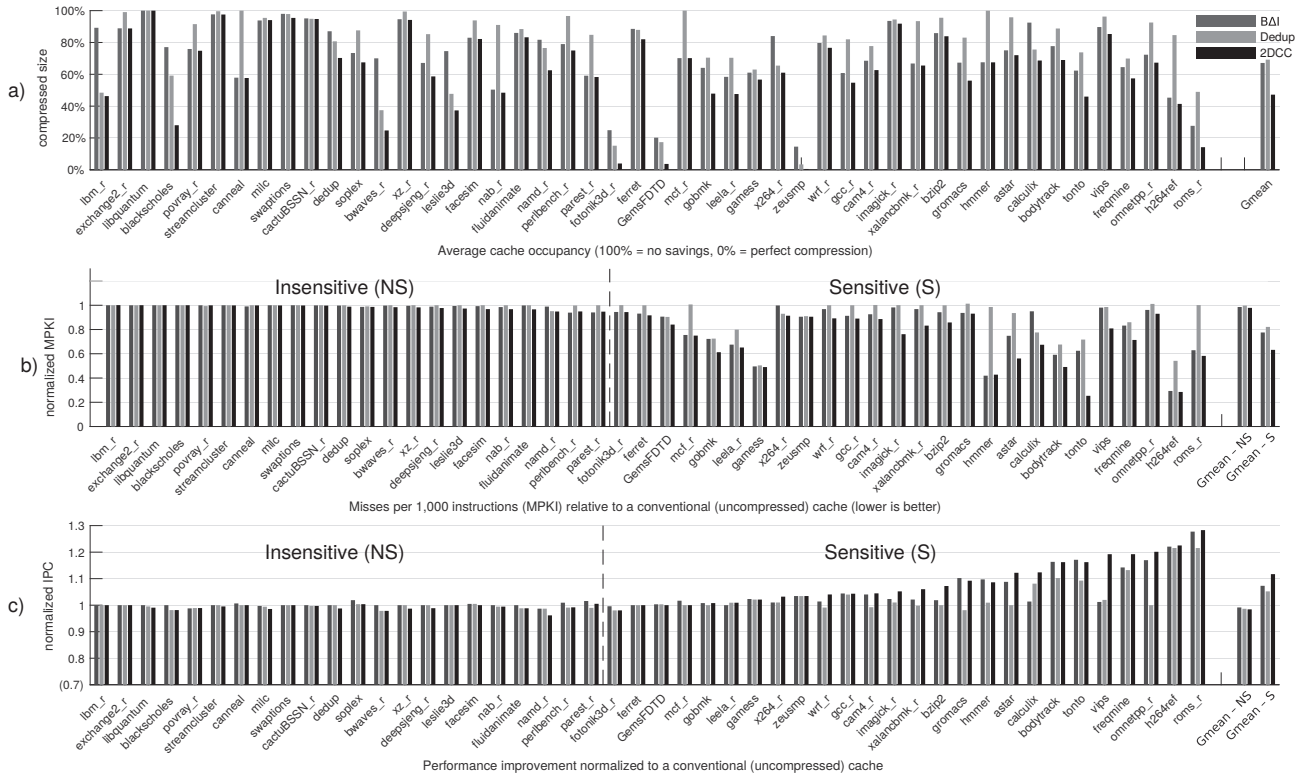

Fig. 8. Cache occupancy, cache miss rates, and performance improvements of 2DCC compared to iso-silicon BΔI (intra-block) and Dedup (inter-block) caches, normalized to an iso-silicon conventional (uncompressed) cache.

Intra-block compression: Data values in a block can also be compressed due to low dynamic range [1, 3]. Prior work categorized these into (a) repeated values, (b) a set of values (especially zeros) repeated in a data block, and (c) near values, which have the same upper bits and different lower bits.

BΔI [1] uses one word-granularity "base" value for each compressed cache block, and replaces the other words in the block with their distances from the base value. 2DCC borrows the BΔI compression method to compress each block; however, because 2DCC also deduplicates identical blocks, the overall cache organization, operation, and replacement policy differ.

In general, intra-block methods are useful in compressing one block (or superblock of several lines, akin to prefetching), but none consider redundancy among different non-contiguous blocks at far-away addresses. 2DCC, in contrast, can store only one copy of duplicate lines, and offers better compression.

VI. Acknowledgements

This research was supported in part by the Natural Sciences and Engineering Research Council of Canada.

References

[1] G. Pekhimenko, V. Seshadri, O. Mutlu, M. A. Kozuch, P. B. Gibbons, and T. C. Mowry, "Base-delta-immediate compression: Practical data compression for on-chip caches," in *PACT*, 2012.

[2] Y. Tian, S. M. Khan, D. A. Jiménez, and G. H. Loh, "Last-level Cache Deduplication," in *ICS*, 2014.

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

[4] J. Yang, Y. Zhang, and R. Gupta, "Frequent value compression in data caches," in *MICRO*, Dec 2000.

[5] E. G. Hallnor and S. K. Reinhardt, "A unified compressed memory hierarchy," in *HPCA*, 2005.

[6] B. Panda and A. Seznec, "Dictionary sharing: An efficient cache compression scheme for compressed caches," in *MICRO*, 2016.

[7] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis, "The Case for Compressed Caching in Virtual Memory Systems," in *USENIX ATC*, 1999.

[8] M. Ekman and P. Stenström, "A Robust Main-Memory Compression Scheme," in *ISCA*, 2005.

[9] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[10] "SPEC releases major new CPU benchmark suite," 2017. [Online]. Available: https://www.spec.org/cpu2017/press/release.html

[11] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *SIGARCH Comput. Archit. News*, vol. 34, pp. 1–17, 2006.

[12] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *PACT*, 2008.

[13] A. R. Alameldeen and D. A. Wood, "Adaptive cache compression for high-performance processors," in *ISCA*, June 2004, pp. 212–223.

[14] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam, "Implementing Signatures for Transactional Memory," in *MICRO*, Dec 2007.

[15] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely Jr, and J. Emer, "SHiP: Signature-based hit predictor for high performance caching," in *MICRO*, 2011.

[16] J. Gaur, A. R. Alameldeen, and S. Subramoney, "Base-Victim Compression: An Opportunistic Cache Compression Architecture," in *ISCA*, 2016.

[17] D. Sanchez and C. Kozyrakis, "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems," in *ISCA*, 2013.

[18] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "CACTI 6.0: A tool to model large caches," *Research report HPL-2009-85, HP Laboratories*, 2009.

[19] T. E. Denehy, W. W. Hsu, T. E. Denehy, and W. W. Hsu, "Duplicate management for reference data," in *IBM Research Report RJ10305*, 2003.