Conciliating Speed and Efficiency on Cache Compressors

Daniel Rodrigues Carvalho, André Seznec

To cite this version:

Daniel Rodrigues Carvalho, André Seznec. Conciliating Speed and Efficiency on Cache Compressors. ICCD 2021 - 39th IEEE International Conference on Computer Design, Oct 2021, Virtual, United States. pp.1-5. hal-03354883

HAL Id: hal-03354883
https://inria.hal.science/hal-03354883

Submitted on 26 Sep 2021

Conciliating Speed and Efficiency on Cache Compressors

Daniel Rodrigues Carvalho
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
odanrc@yahoo.com.br

André Seznec
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
andre.seznec@inria.fr

***Abstract*—Cache compression algorithms must abide by hardware constraints; thus, their efficiency ends up being low, and most cache lines end up barely compressed. Moreover, schemes that compress relatively well often decompress slowly, and vice versa. This paper proposes a compression scheme achieving high (good) compaction ratio and fast decompression latency. The key observation is that by further subdividing the chunks of data being compressed one can tailor the algorithms. This concept is orthogonal to most existent compressors, and results in a reduction of their average compressed size. In particular, we leverage this concept to boost a single-cycle-decompression compressor to reach a compressibility level competitive to state-of-the-art proposals. When normalized against the best long decompression latency state-of-the-art compressors, the proposed ideas further enhance the average cache capacity by 2.7% (geometric mean), while featuring short decompression latency.**

***Index Terms*—Cache memories, Data compaction and compression, Compression technologies**

I. Introduction

Cache compression tends to heavily rely on the spatial and temporal localities of data; in essence, it expects that previously seen values will be perfectly or partially repeated. Hence, there is a predominance of dictionary-based pattern compressors — compressors that use the earliest values in a line as references for the following values, applying patterns to compare and match values, generally at a byte level. These references are then used to remove repeated bits in the following values (*value deduplication*) [1]–[3].

Deduplication usually assumes that a single basic data type (*e.g.*, 32 bits) is persistently used; however, this assumption does not hold for all workloads. To cope with that, compressors can add patterns to emulate smaller data types (*e.g.*, two consecutive 16-bit values whose *msb* match previously seen values). This means that to be able to compress all basic data types, compressors would need to provide patterns to cover all possible permutations of matching/non-matching bytes, which is expensive. With the rising use of 64-bit values, comprising all permutations becomes even more prohibitive.

In addition, although having more patterns improves compression effectiveness, but complicates decompression hardware, it increases the latency overhead. For instance, BDI [4] can achieve 1-cycle decompression by covering only two patterns, but its average *compression ratio (ratio between the compressed and uncompressed sizes — lower is better)* on SPEC 2017 benchmarks is high, at 86.3%. Lower ratios can be achieved by proposals with more patterns - *e.g.*, C-Pack [2] (70.5%), FPC (76.7%) [5], FPC-D [3] (69%), X-Match (77.5%), and X-RL (77.3%) [1]; however, their decompression can be as slow as a word per cycle.

This paper presents a *new perspective on the pattern-matching problem* that helps increase pattern coverage of existing algorithms. The following contributions are made:

- We propose **Region-Chunk (RC) compression**, a *concept* that explores the granularity of cache compression to favor data deduplication and improve compressors' efficiency.
- We formalize the definition of a generic base-delta compressor comprising any number of bases. We also describe optimizations to its representation, increasing its effectiveness when compared to the naive approach.
- We design various compressors built upon Region-Chunk and the generalized base-delta compressor. These attain high efficiency and fast decompression.

The term **compaction ratio** is the number of valid blocks per data entry, and measures the efficacy of the compressed system — higher is better.

II. Exploring Compression Granularity

Cache lines are composed of basic data types: 8, 16, 32 and 64 bits. There is often some regularity among elements of a line, and cache compressors try to capture it by parsing the lines in fixed-sized **chunks**. Figure 1 shows this process: a line is split into 2-byte chunks (step A), which are passed to the compressor (step B), which generates compressed data (step C). While small chunks cannot capture correlation of larger data types, big chunks are harder to compress, as they may encompass various types. A typical compromise is 32-bit chunks [6].

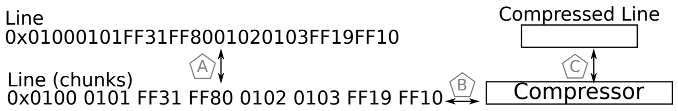
Line
0x01000101FF31FF8001020103FF19FF10
A
Line (chunks)
0x0100 0101 FF31 FF80 0102 0103 FF19 FF10
B
Compressed Line
C
Compressor

Fig. 1. A cache line is split into chunks to generate compressed lines under conventional compression.

Most state-of-the-art compressors tend to search for partial or full pattern matches, comparing chunks either against fixed values or previously seen chunks to reduce duplication. It is expected that some bits will consistently have more matches than others; notably, that the MSBs of chunks tend to vary less than their LSB counterpart. Hence, compressors tend to use patterns that deduplicate the MSB but copy the LSB [2]–[4].

Given that bits compress differently based on their position, it might be more advantageous to group and compress them separately. We hereby propose further dividing chunks into equally-sized **regions**, which are compressed independently — a concept we call **Region-Chunk compression**.

A. The Region-Chunk Concept

Region-Chunk compression rewires each line into multiple ”region lines” which are assigned as the input of their respective compressor, which work as they would normally do. Figure 2 shows the slight differences in the process: the line is split into chunks, and the chunks into regions to generate a line per region (step A’); then each region’s compressor processes its line (step B) to generate its respective compressed line (step C_1). The lines are then concatenated (step C_2).

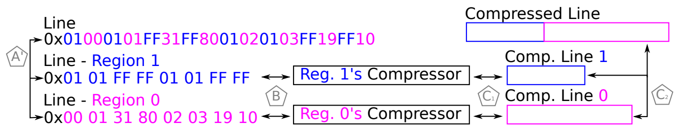

Fig. 2. A Region-Chunk compressor with two regions (MSB and LSB). Compressing and decompressing regions works by using only the regions of interest of the cache line as input to their respective compressors.

The main advantage of regions is that **each compressor can be tailored to the region it compresses**. For instance, if MSB regions have less variability than LSB regions then then MSB region’s compressor can have less patterns or a lower maximum number of different dictionary entries — thus reducing the number of metadata bits needed and compressor complexity. In addition, the number of non-matching bits could be modified to increase the likelihood of deduplications, or reduce the size of compressed data. Finally, regions allow using simpler compressors, yet still attaining high pattern coverage due to all the possible combinations of each region compressors’ output.

We will refer to compressors where a cache line is divided into w-bit chunks, each with x-bit regions as a $\mathbf{R}_x\mathbf{C}_w$ **compressor**. Notice that *conventional compression is a subset of* R_xC_w *compression* ($x = w$).

B. Latency of a Region-Chunk-based Compressor

The Region-Chunk concept is orthogonal to the compression algorithm. The latency of the compression and decompression steps are equal to the latency of the slowest region compressor. There are two extra processes applied on top of these steps. Before compressing, the cache line must be split into ”region cache lines”, which are fed into the respective region compressors. This is a simple data rewiring; thus, its latency is negligible. After the region’s compressed data is generated, an extra cycle may be needed to calculate and shift it to its position in the compressed line. This requires a few adders (to sum the previous regions’ compressed data’s sizes) and shifters. An analogous reverse process with similar costs is needed on decompression.

It is important to notice that *this second extra process can be done in parallel with the data fetch if the position can be calculated in advance* — *e.g.*, when the region’s compressed line’s size can be extracted from the tags — so that the extra cycle does not impact the decompression latency [4].

To minimize latency we propose using a *multi-compressor* — a compressor that selects the best compression option among its multiple sub-compressors — based on base-delta compressors [4] as each region’s compressor.

Multi-compressors need extra bits to identify which sub-compressor is responsible for the decompression of some compressed data. These bits can be stored in either the tag entry — preferred when the region and chunk sizes are similar — or in the data entry — preferred when there are multiple regions in a chunk. The former configuration, when performing sequential accesses, allows hiding any extra delay needed by the multi-compressor or the Region-Chunk concept: sub-compressor decoding and regions’ position extraction can be done in parallel with tag checks. The latter generates less metadata overhead but adds 1-2 extra cycles before the decompression — the time to decode and select the sub-compressor. Consequently, multi-compressors whose sub-compressors are based on base-delta compressors have a decompression latency of 1 and 3-4 cycles for these configurations, respectively.

III. Attuning Base-Delta Compressors

The Base-Delta-Immediate (BDI) compressor [4] was proposed as a quick multi-compressor. The compressed data of its base-delta sub-compressors contain three fields: a *base*, which is the first unique non-zero chunk seen when parsing a line; *deltas*, which are the differences of each chunk from the existing bases; and a *bitmask* per delta to inform to which base it refers. This last field is needed because BDI allows two bases: one stored explicitly in the base field, and the other used implicitly; that is, one dictionary value is reserved to a fixed known value — an all-zeros chunk. When a delta is calculated relative to the zero base, the zero-base index is used normally in the bitmask field, but the zero-base itself is never stored.

BDI, however, compresses poorly. In SPEC 2017, the average percentage of cache lines that are compressible with each of BDI’s sub-compressors — Zeros, Repeated Values, $B8\Delta4$, $B8\Delta2$, $B8\Delta1$, $B4\Delta2$, $B4\Delta1$, $B2\Delta1$ — is, respectively, 11.3%, 12.7%, 15.0%, 17.1%, 42%, 14.6%, 19.3%, 15.1%. That is, most lines would have needed more than two bases to compress. We hereby generalize the concept of a base-delta compressor to extend its support to any number of bases: *a base-delta compressor that parses a cache line in w-bit chunks to store up to x explicit and y implicit w-bit bases, and whose deltas have z bits is represented as* $\boldsymbol{C_w I_x E_y D_z}$.

A. Optimizing Base-Delta Compressors

Ideally, bases would be selected based on the range of values in the line. In practice, arbitrarily picking the first occurrence of a new value is simpler, yet only marginally degrades performance [4]; thus, assuming a uniform distribution of values, the probability of being able to compress the next values after the base is set is the same, *regardless of the contents of the base's last z bits*. This fact can be leveraged to assume that **the arbitrary base's z last bits are always fixed at a value — *e.g.*, zero — and are, thus, not stored**.

The mapping on the left of Figure 3 represents this idea. In the example, a naive $\mathrm{C}_{32}\mathrm{I}_1\mathrm{E}_1\mathrm{D}_8$ compressor would parse the 64-bit cache line *0x0123456701234568* as: base=*0x01234567*, deltas=*0x00*, *0x01*; however, its optimized version would parse the line as: base=*0x012345*, deltas=*0x67*, *0x68*. That is, deltas are relative to the base's implicit extended value, *0x01234500*.

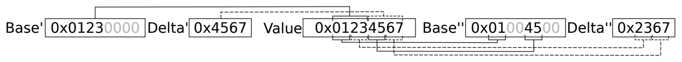

Fig. 3. Mapping deltas to different positions. The left mapping's deltas are relative to the last two bytes, while the right mapping associates deltas to the first and third least-significant bytes. Grayed out bits are not stored.

A minor optimization can also be made to the bitmask/-pointer field representation of pattern compressors in general. When compressing a cache line, chunks are parsed sequentially, and the dictionary of bases is populated orderly. As the dictionary is not initially full, **the bitmask fields of the first chunks parsed do not need to be able to index all the dictionary entries**. The number of bits needed by the bitmask of the C_{th} chunk of a $\mathrm{C}_w\mathrm{I}_x\mathrm{E}_y\mathrm{D}_z$ compressor is given by $numBitmaskBits_C = \log_2(w + \min(C, y))$.

For instance, a $\mathrm{C}_w\mathrm{I}_1\mathrm{E}_3\mathrm{D}_z$ will have its dictionary initially populated with one value: the implicit base (IB); thus, the first chunk can refer to two possible values - IB or a newly added base (B_1) - requiring only 1 bitmask bit. The second chunk assumes a worst case scenario where the first chunk added a B_1; so it can refer to three possible values - IB, B_1, or another new base (B_2) - requiring 2 bitmask bits. Although in a regular pattern compressor this optimization would increase hardware complexity, in a $\mathrm{C}_w\mathrm{I}_x\mathrm{E}_y\mathrm{D}_z$ compressor it does not: the width of the bitmask is defined at design time, based on the number of explicit and implicit bases of the compressor.

Unless otherwise stated, $\mathrm{C}_w\mathrm{I}_x\mathrm{E}_y\mathrm{D}_z$ compressors use these optimizations by default.

B. Dissociating Base Size from Parsing Type

Conceptually, $\mathrm{C}_w\mathrm{I}_x\mathrm{E}_y\mathrm{D}_z$ behaves similarly to a dictionary-based compressor with two patterns — "no match", and "match the $w - z$ MSBs". Consequently, *the major difference between regular pattern compressors and $C_wI_xE_yD_z$ is that $C_wI_xE_yD_z$ makes the number of occurrences of each pattern fixed.* For example, when compressing a 64B cache line with a $\mathrm{C}_{32}\mathrm{I}_1\mathrm{E}_2\mathrm{D}_8$ compressor, the "no-match" and "match the three MSBs" patterns occur exactly 2 and 14 times, respectively.

Hence, as in pattern compressors, other patterns can be used to loosen the relationship between the base size and its type-processing capabilities — *i.e.*, deltas can be remapped so that they refer to the least-significant portions of the desired type. For instance, the default $\mathrm{C}_{32}\mathrm{I}_x\mathrm{E}_y\mathrm{D}_{16}$ compressor focuses on compressing 32-bit data types, so its delta bits correspond to the two least-significant bytes of the chunks. To cover 16-bit data types instead, the delta bytes can be remapped to be extracted from the chunks' first and third bytes (right mapping in Figure 3).

Unless otherwise specified, $\mathrm{C}_w\mathrm{I}_x\mathrm{E}_y\mathrm{D}_z$ compressors extract their delta bytes from the chunk's LSBs.

C. Stride Compressor

Sometimes, the deltas in a base-delta compressor present a certain characteristic that can be described by a mathematical equation. In this case there is no need to store all delta instances; they can be represented by such equation's variables instead. A common instance of this situation is the arithmetic sequence: an iterated value being assigned to an array of variables has a fixed constant difference between them (a stride). Therefore, we also propose a compressor that covers this particular case, which is still simple enough to keep a 1-cycle decompression latency. The **Stride Compressor** compresses data as a single base-delta pair, and the decompression of any chunk is given by $C_n = base + (n-1) \cdot delta$. For example, the sequence of 8-bit values *0x31*, *0x32*, *0x33*, *0x34*, *0x35*, *0x36*, *0x37*, *0x38* has a fixed stride of *0x01* between its deltas, and thus it is represented by the base *0x31*, and the stride *0x1*.

IV. Related Work

X-Match reorders the dictionary to take advantage of Huffman code on the most recently seen value, and X-RL expands it with an encoding for runs of zeros [1]; C-Pack [2] simplifies X-Match by removing reordering and using fixed-size encoding; BDI [4] is a base-delta multi-compressor covering multiple chunk sizes, while providing a single partial-match pattern to achieve minimal decompression latency; FPC-D [3] partially covers both 32 and 64-bit data types with a 32-bit region granularity by using a 2-entry FIFO as its dictionary to reduce decompression latency. FPC [5] is a pattern-only scheme (*i.e.*, no dictionary), and SC^2 [7] builds a global dictionary using probabilistic models of the workload's data. *The Region-Chunk concept is orthogonal to these algorithms, so it can be applied to any of these line-based compressors to improve their efficiency*.

Compressors must be associated with a compaction scheme to increase the effective cache capacity. Older layouts double the number of tags to allow any pair of lines to co-allocate, while modern proposals tend to reduce this overhead by associating multiple neighbor blocks to a single shared tag [6], [8]. Recently, some proposals moved the tag information to the data entry [9].

V. Methodology

We simulated using gem5 [10]. The baseline model executes out-of-order (OOO), as detailed in Table I. The compaction layout of the compressed caches was PSS [8]. Compression-related statistics are averaged across all (de)compressions.

Compaction-related statistics are averages of snapshots (taken every 100K ticks) of the cache's contents.

SimPoints was used [11] to take multiple checkpoints per benchmark of SPEC 2017 [12]. The warm-up period was 100M instructions; workloads were then run for 200M instructions. The average of each benchmark's statistics was calculated through the arithmetic mean of its checkpoints. The total geometric mean of the benchmarks was normalized to a non-compressed baseline system. The number of Misses Per Kilo-Instruction (MPKI) was used to discard benchmarks that would barely benefit from larger caches (*i.e.*, compression is hardly useful when $MPKI < 1$). Qflow [13] was used for VLSI synthesis (35nm).

TABLE I: Baseline system configuration.

Processor	1 core, OOO, 8-issue
Cache line size	64B
L1 I/D	32KB, 4-ways, 4 cycles, LRU
L2	256KB, 8-ways, 12 cycles, RRIP [14]
Shared L3	1MB, 8-ways, 34 cycles, RRIP, compressed
MSHRs and write buffers	64
DRAM	DDR4 2400MHz 17-17-17, tRFC=350ns, 4GB
Architecture	ARM 64 bits
Clock	4GHz
Image	Ubuntu Trusty, Little Endian

A. Selecting Base-Delta Sub-Compressors

An R_xC_w compressor contains $\frac{w}{x}$ region compressors. These region compressors can be instances of any compressor; but *we propose a set of multi-compressors based on the $C_wI_xE_yD_z$ compressors (**SubR_xC_w**) to combine fast decompression and good compressibility*. These configurations are defined once, **at design time** (*i.e.*, no runtime overhead). Each SubR_xC_w has $S = 2^k - 1$, $k \in \mathbb{N}$ sub-compressors: one encoding is reserved for when all sub-compressors fail and data is left uncompressed. The process taken to select which sub-compressors worked best for each SubR_xC_w, and their final configurations are described elsewhere [15].

VI. Results

When suitable the compressors are compared against FPC-D [3], the best performing state-of-the-art compressor in our tests. We assume FPC-D's level of parallelization achieves 4-cycle decompression. Although not shown due to space constraints, as expected, the more optimizations are applied to the base-delta compressors, the more effective the compressor becomes.

A. Granularity Exploration

Figure 4 shows the average compaction ratio of R_xC_w using the proposed SubR_xC_w compressors, and a Sub$R_{32}C_{64}$ without the Region-Chunk concept. We also apply Region-Chunk to FPC-D — which parses lines in 32-bit chunks — by splitting the input into two 32-bit regions ($R_{32}C_{64}$FPC-D). R_xC_w using the proposed SubR_xC_w compressors perform comparably to prior work. In particular, most R_xC_{64} compressors outperform them.

Big chunks capture all basic types, and small regions reduce the amount of duplication. In addition, the more regions exist, the larger the metadata overhead. A good trade-off is achieved with 16-bit regions. *Further dividing chunks into regions is slightly beneficial to dictionary-based cache compressors in general, but is great for base-delta compressors in particular.* Results for Sub$R_{32}C_{64}$ show that *the main improvements of R_xC_w are due to the region abstraction, **not** the new selection of base-delta compressors.*

B. Single-Cycle Decompression

All previous results assume that the regions' encodings are stored within the data entry; thus, each compressor's decompression takes 3 cycles. However, as in BDI, decompression can be partially done in parallel with the data access. If the encodings are stored in the tags, and the cache performs sequential accesses, the decompression latency becomes a single cycle instead. Besides, by removing a few bits, some sub-compressors co-allocate better, notably the ones whose compressed sizes are close to half the block pair (BP)'s size.

This significantly improves compaction ratio, further increasing the benefits of the faster decompression. Figure 5 shows the IPC (top) and compaction ratio (bottom) when storing the encoding in the tag and in the data entry. Due to space constraints only R_XC_{64} compressors' results are shown, comparing them to BDI and Frequent Pattern Compression with limited Dictionary support (FPC-D), as well as a twice larger uncompressed cache.

Despite the benefits of storing the encoding in the tags, the tag overhead can be quite high for configurations with multiple regions, so it is better to use this latency improvement in configurations with up to two regions. Nonetheless, similar compaction ratio improvements can be achieved by removing a single delta bit from a few key sub-compressors (*e.g.*, $C_{64}I_1E_0D_{32} \rightarrow C_{64}I_1E_0D_{31}$).

C. Compressor Area overhead

The circuit of any individual $C_wI_xE_yD_z$ is analogous to any base-delta sub-compressor of BDI — *e.g.*, Sub$R_{32}C_{64}$ (no Region-Chunk) has 2x BDI's area.

Region-Chunk compression only adds on top of the underlying compressors a few adders and shifters to compose and decompose the compressed line — a minor cost compared to the compression algorithm's area itself. The compressor-related area is approximately equal to the sum of the areas of the region compressors. Yet, since the input size is smaller, and the natural parallelization provided by Region-Chunk allows reducing each region compressor's level of parallelization, region compressors are simpler than their non-Region-Chunk counterpart. For example, a $R_{32}C_{64}$ containing two Sub$R_{32}C_{64}$ has 2.5x BDI's area, instead of a naively expected 4x.

VII. Conclusion

This paper explores the granularity of cache compressors to increase their efficiency. By focusing on the possible data types contained in cache lines, it is possible to define regions that likely hold similar values and reduce duplication (Region-Chunk compression). This concept is compressor-independent,

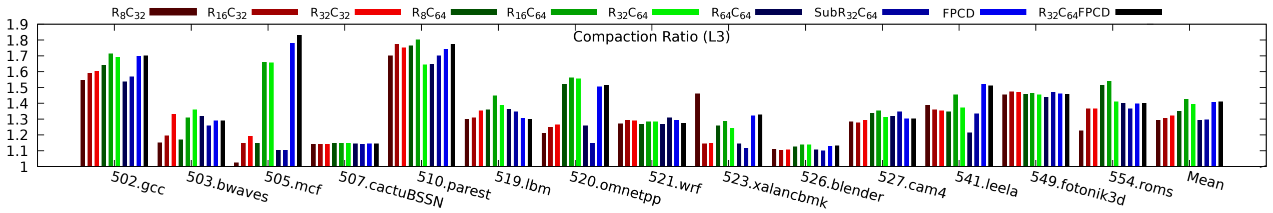

Fig. 4. Comparison of the compaction ratio of the R_YC_X compressors — higher is better.

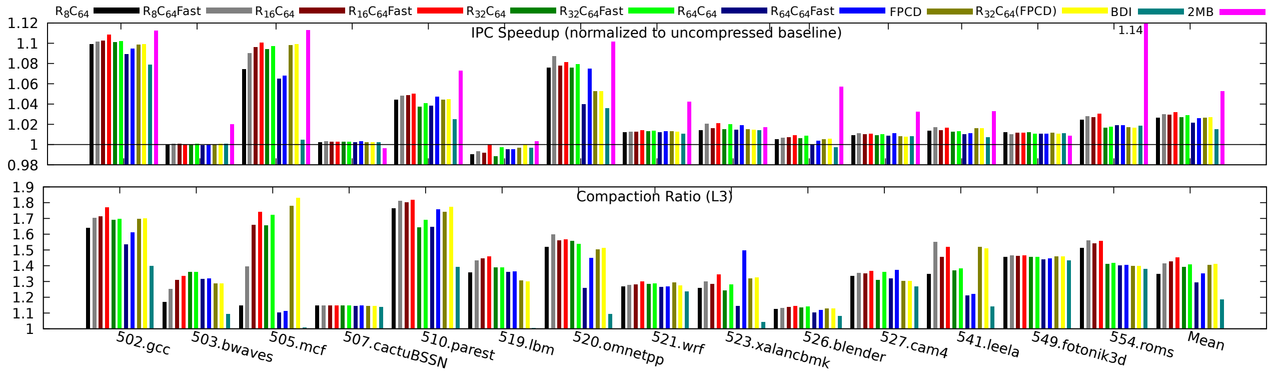

Fig. 5. Comparison of the R_YC_X compressors with the encoding stored in the data entry versus in the tags. Top plot is the IPC speedup, normalized to an uncompressed baseline. Bottom plot is the compaction ratio. Higher is better for both plots.

but highly advantageous for base-delta compressors; thus, we generalize and optimize the definition of base-delta compressors, and use them as region compressors.

The proposed configurations compress comparably to complex compressors while still maintaining a low decompression latency: they reach an average effective cache capacity of 1.42x — greatly improving from BDI's 1.18x. The Region-Chunk concept is orthogonal to existing compressors; when applied to the previous best state-of-the-art compressor its compaction factor improves from 1.38x to 1.39x without any region-compressor tailoring. Future work can address per-region compressor tailoring, and improving the stride compressor's coverage with more bases and deltas.

References

[1] M. Kjelso, M. Gooch, and S. Jones, "Design and performance of a main memory hardware data compressor," in *EUROMICRO 96. Beyond 2000: Hardware and Software Design Strategies., Proceedings of the 22nd EUROMICRO Conference*, IEEE. Prague, Czech Republic: IEEE Computer Society, 1996, pp. 423–430. [Online]. Available: https://doi.org/10.1109/EURMIC.1996.546466

[2] X. Chen, L. Yang, R. P. Dick, L. Shang, and H. Lekatsas, "C-pack: A high-performance microprocessor cache compression algorithm," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 18, no. 8, pp. 1196–1208, 2010. [Online]. Available: https://doi.org/10.1109/TVLSI.2009.2020989

[3] A. R. Alameldeen and R. Agarwal, "Opportunistic compression for direct-mapped dram caches," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '18. Alexandria, Virginia, USA: Association for Computing Machinery, 2018, p. 129–136. [Online]. Available: https://doi.org/10.1145/3240302.3240429

[4] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-delta-immediate compression: Practical data compression for on-chip caches," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12. Minneapolis, Minnesota, USA: Association for Computing Machinery, 2012, p. 377–388. [Online]. Available: https://doi.org/10.1145/2370816.2370870

[5] A. R. Alameldeen and D. A. Wood, "Frequent pattern compression: A significance-based compression scheme for l2 caches," *Dept. Comp. Scie., Univ. Wisconsin-Madison, Tech. Rep*, vol. 1500, 2004.

[6] D. R. Carvalho and A. Seznec, "Understanding cache compression," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 18, no. 3, pp. 1–27, 2021.

[7] A. Arelakis and P. Stenstrom, "Sc2: A statistical compression cache scheme," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. Minneapolis, Minnesota, USA: IEEE Press, 2014, p. 145–156. [Online]. Available: https://doi.org/10.1109/ISCA.2014.6853231

[8] D. R. Carvalho and A. Seznec, "A case for partial co-allocation constraints in compressed caches," in *Proceeding of the 21st Annual International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS XXI)*, ser. SAMOS '21, 2021.

[9] S. Hong, B. Abali, A. Buyuktosunoglu, M. B. Healy, and P. J. Nair, "Touché: Towards ideal and efficient cache compression by mitigating tag area overheads," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. Columbus, OH, USA: Association for Computing Machinery, 2019, p. 453–465. [Online]. Available: https://doi.org/10.1145/3352460.3358281

[10] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, Aug. 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718

[11] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS X. San Jose, California: Association for Computing Machinery, 2002, p. 45–57. [Online]. Available: https://doi.org/10.1145/605397.605403

[12] S. P. E. Corporation, "Spec cpu 2017," https://www.spec.org/cpu2017/, 2017, accessed: 2019-10-10.

[13] R. T. Edwards, "Qflow," http://opencircuitdesign.com/qflow/reference.html, 2019, accessed: 2020-10-07.

[14] A. Jaleel, K. B. Theobald, S. C. Steely, and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. Saint-Malo, France: Association for Computing Machinery, 2010, p. 60–71. [Online]. Available: https://doi.org/10.1145/1815961.1815971

[15] D. Rodrigues Carvalho, "Towards compression at all levels in the memory hierarchy," Ph.D. dissertation, 2021. [Online]. Available: http://www.theses.fr/2021REN1S011