Understanding Cache Compression

Daniel Rodrigues Carvalho, André Seznec

Understanding Cache Compression

DANIEL RODRIGUES CARVALHO and ANDRÉ SEZNEC, Univ Rennes, Inria, CNRS, IRISA, France

Hardware cache compression derives from software-compression research; yet, its implementation is not a straightforward translation, since it must abide by multiple restrictions to comply with area, power, and latency constraints. This study sheds light on the challenges of adopting compression in cache design—from the shrinking of the data until its physical placement. The goal of this article is not to summarize proposals but to put in evidence the solutions they employ to handle those challenges. An in-depth description of the main characteristics of multiple methods is provided, as well as criteria that can be used as a basis for the assessment of such schemes. It is expected that this article will ease the understanding of decisions to be taken for the design of compressed systems and provide directions for future work.

CCS Concepts: • **Computer systems organization** → **Processors and memory architectures**; • **Information systems** → *Data compression*;

Additional Key Words and Phrases: Caches, cache compression

ACM Reference format:
Daniel Rodrigues Carvalho and André Seznec. 2021. Understanding Cache Compression. *ACM Trans. Archit. Code Optim.* 18, 3, Article 36 (June 2021), 27 pages.
https://doi.org/10.1145/3457207

1 INTRODUCTION

Transferring data from and to the main storage has an elevated cost. Because of that, it has become standard for current systems to implement multiple levels of progressively smaller memories (caches) with proportional latencies and energy cost [22, 33, 53, 58]. Unfortunately, whenever an access miss occurs, the next—larger, yet slower—cache level must be accessed, degrading performance. Prior works have addressed increasing cache and memory capacity [5, 72], reducing miss ratio [2, 66], and lowering tag area usage [81, 96]. One branch of research that has had a burst recently is *Cache Compression*. It comprises compressing and compacting data so that the benefits of larger caches are achieved without the main drawbacks of physically increasing the caches themselves; howbeit, this upgrade is not free, and compression and decompression steps must be added to the access process, which increase energy and hit latency.

Cache compression is not merely a matter of making data smaller: data compression by itself cannot increase cache storage—the compression-oblivious conventional cache layouts are not able to fit multiple lines in a single data entry, thus wasting the space saved by compression. Compressed data should be co-allocated with other compressed data, and there must be a

Authors' address: D. R. Carvalho and A. Seznec, Univ Rennes, IRISA/INRIA, Campus de Beaulieu, 35042 Rennes Cedex, France; emails: odanrc@yahoo.com.br, andre.seznec@inria.fr.

1544-3566/2021/06-ART36
https://doi.org/10.1145/3457207

finer-grained way to locate them. This is handled by the *compressed cache organization*, or simply *cache compaction* scheme. Basically, the design of a compressed system encapsulates the compound of decisions of why compression is being applied to the system, when and how to compress data, and where to place data after compression.

In conventional caches, the size of a physical entry in the data array (*data entry*) generally matches the size of the cache line. In compressed caches, however, a single data entry can contain the data of multiple lines [51]. This allows effectively increasing the cache size with limited area overhead: instead of increasing the number of data entries, one adds a compressor, a decompressor, and extra circuitry to handle the organization of compressed blocks. Consequently, the compressor's complexity becomes a crucial trade-off—compression algorithms usually require bigger and more complex hardware as better compression factors are achieved [69]. Moreover, as more data fits in the cache, more metadata is needed to reference it [12]. If the space reserved for this metadata is insufficient, then the compression ratio may be limited by this number; but if it is too high, then area and energy are wasted. Finally, the decompression latency critically affects cache latency, so it should be minimal [4].

Because of that, although, theoretically, compression can be applied to all cache levels, it may be tricky to compress the caches closest to the core; since they are the most frequently accessed, adding a decompression latency to this critical path could greatly degrade performance. Moreover, finding the critical word is not trivial when the data is compressed; thus, decompression must happen before it is located and sent. In spite of that, the closest cache to the core can still be compressed to reduce its miss rate if decompression is immediate [41, 70, 86]. Ultimately, other techniques can be applied to reduce the average decompression time [5, 50, 51].

Previous works [56, 74] group and describe the hardware compression proposals created over the years. They cover proposals on a high-level taxonomy perspective—compression algorithms, and compressed cache organization—and focus on describing each technique isolatedly. This manuscript takes a different approach, dissecting these techniques into their underlying solutions for each of the problems faced by compressed systems. We hereby elucidate the problems and solutions in general terms, and explain what are the typical strategies adopted to handle cache compression, referring to techniques as practical examples.

Throughout this article the terms "cache" and "memory" will be used interchangeably, as well as the terms "cache line," "line," and "block". **Compression ratio** is given by the division of the compressed size by the uncompressed size — i.e., it is a value in the range [0:1]. **Compaction ratio** is the average number of blocks per valid data entry — i.e., it is a value greater or equal to 1. The compression ratio measures the *efficiency* of the compression algorithm, and the compaction ratio quantifies the *efficacy* of the compressed cache. The following sections present the challenges and solutions of compressed memory systems from a conceptual point of view: they describe the motivations to design a compressed system (Section 2); the decisions to be made regarding compressed data (Section 3); the changes that must be made to the general layout to accommodate compression (Section 4); when to use compression (Section 5); ending with an evolutionary study of hardware data-compression algorithms (Section 6). Finally, Section 7 shows how compression interacts with other cache components, and Section 8 concludes this manuscript with an overview and directions for future research.

2 MOTIVATIONS

Data compression is modern day's alchemy. From circuits to web navigation, the goal of turning a big data chunk into a smaller one goes far beyond increasing the density of information; by storing more data in less space, one can potentially *save* area, energy, and bandwidth. One of the most recurrent reasons a computer architect would apply compression is to increase the

effective memory *capacity* at a lower cost than physically augmenting the memory size. Enlarging the cache size increases silicon area, and the static power consumption, in addition to slowing the average access latency down; however, a compressed memory's compression circuitry is only a fraction of its area [21, 63, 69].

Compression can also be used to reduce *energy* consumption: as it requires smaller cache footprint, it leads to lower leakage and driving energy utilization [45, 49, 95, 101]; furthermore, transferring smaller data requires less energy [42]. A third common motivation is to improve *bandwidth*: smaller data transfers can be translated into faster transfers, higher throughput, or even bus-width reduction [32, 45, 68, 82]. Another possible use of compression is to *enhance tolerance* for partially faulty entries—that is, compressing data so that it fits in the non-faulty sub-entries of cache entries [26].

Finally, compression can be used to reduce the *storage overhead of other techniques* for memory enhancement. For instance, techniques to prevent, detect and correct errors in memory units typically require the addition of specialized codes [18]. These extra bits can incur a high area penalty, especially for schemes with higher efficacy [20]. Compression permits to partially or entirely fit these codes within the data storage itself, along with the compressed data [20, 38, 44, 62, 100]. This can be leveraged to reduce the size of the extra storage, as well as its access frequency.

3 HANDLING DATA

Prior to selecting the compression algorithm, multiple decisions must be made regarding the data being compressed; questions such as "What is the granularity of the compressor's input?," "Must the decompressed line perfectly match the original cache line?," and "What should be done in case the compressed size changes?" must be answered first.

3.1 Input Granularity

In compression, in general the larger the data set to compress, the larger its potential to compress. The particularity of cache compression is that the input size is well-defined: the size of a cache line. The larger the cache line size, the more values are likely to repeat, generating more deduplication opportunities for the compressor; yet, the compressor and decompressor's complexities increase with the line size, which means that enlarging the line has a direct negative impact in their latencies and areas.

In addition, cache compressors generally process lines isolatedly, trying to find value similarities within the line itself—a concept called **intra-line compression**; however, there are some techniques that share common data between multiple lines (**inter-line compression**) [29, 63]. More information on how these work can be found in Section 6.1.5. Similar to increasing the cache line size, inter-line compression has the advantage that the probability of seeing similar values increases with the number of samples; therefore, it will likely manage to deduplicate more data, achieving higher compression ratios at the cost of increasing the complexity of the organization scheme [58, 63, 87]. These granularities are not exclusive, so a compactor can use both the intra- and inter-line approaches simultaneously, in a hybrid scheme [29].

3.2 Precision

Computing systems commonly require calculations to be deterministic. As a result, most compression algorithms tackle compression as a **precise** problem, and compression must act as an invertible function—a given line A must be uniquely compressed to line A', and the decompression of A' must regenerate A precisely.

However, a recent field of study called approximate computing proposes consciously deciding whether approximating techniques can be used to reduce power consumption, or speed

calculations up [55]. Its concepts can be also applied to compression under specific contexts and constraints. For example, when compressing lines containing data of images, the **Least-significant Bits (LSB)** of the pixels contain information that, if removed, likely does not affect much the image quality. When applied to compressed caches, **approximate** computing typically uses error/difference thresholds to determine which values should be considered similar, and classifies how similar they are. Then, a data deduplication technique that maps multiple tags into a single data entry is applied (more in that in Section 4). As the contents of lines are not understood by the hardware, the programmer must inform which parts of the address space are approximable; furthermore, since not all applications allow approximate calculations, these compressed cache layouts allow both precise and approximate blocks to co-exist, either by splitting the cache into different areas [54], or by adding metadata to the blocks to inform so [73].

3.3 Overwriting Data

Co-allocating blocks in a data entry is not a one-time problem that is solved after a line has passed through the compressor and has been stored in the cache. Further updates to the line can make its contents change, possibly requiring the line to be relocated. Three cases can arise when data is overwritten: the contents of the (un)compressed line do not change; the new compressed data is smaller than the previous contents (**data contraction**); or the new data is compressed to a size larger than the previous one (**data expansion** or **fat write**).

The first case is trivial: nothing needs to be done, so the line can be kept in its original location. Data expansions, however, are generally undesired: an expanding block may not fit in its current allocated space anymore, requiring special handling. Finally, data contractions are somehow easier to handle: one can either decide to keep the line in its current location, filling the emptied space with padding bits; or handle it analogously to expansions. Handling the resizing follows the compaction scheme's replacement guidelines: one can either remove the overwritten line itself from its current location, re-applying the co-allocation process (*re-insertion*); or move/evict other co-allocated lines in the data entry to *make room* for the expansion to happen. This handling, although inconvenient, is rare, and can be done off the critical path [5, 67, 76].

As a side note, compression schemes that enforce that all co-allocated blocks must fit in a given size may require lines whose size changed to always relocate if the size threshold is crossed, even if there is space available for the block to fit in its current location [75, 76]. For example, in SCC [75] blocks are indexed as a hash function of their address and compressibility; thus, if the compressibility of a block decreases[1], so does its expected location.

4 MAPPING COMPRESSED DATA

Data stored in a cache must be located to be accessed. For that end, conventional cache designs contain both a tag-metadata storage and the data storage itself [83]. This metadata storage encompasses information that is relevant to identify which memory data is stored at a given place in the data storage. Typically, the mapping between these storages is bijective; each cache line fits exactly in a data entry, which is associated with a unique tag identifier, even if not valid. When data is compressed, however, this is not the case anymore, and multiple cache lines can be stored in a data entry. As a result, the mapping between storages must be modified.

The straightforward solution is to remove the injection property by increasing the number of tags. This is the standard approach adopted by most proposals in the literature, being applied by associating multiple tag entries to either a single (many-to-one) or multiple (many-to-many)

[1]When the compressibility of a block increases the extra space can be filled with padding bits to avoid moving the block.

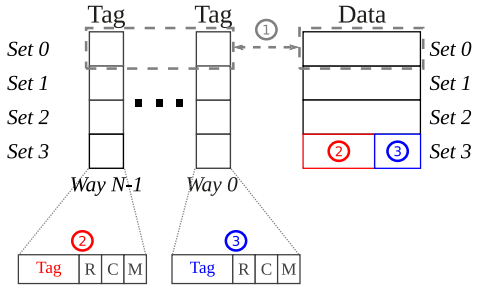

Fig. 1. A generic many-to-one compressed-block mapping.

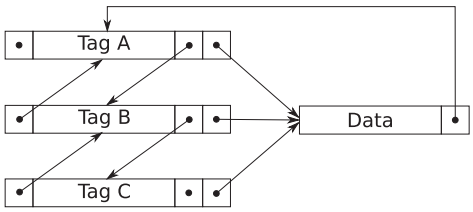

Fig. 2. A generic tag-data doubly linked list. Each tag entry has extra metadata to inform which are the previous entry, next entry, and corresponding data block. Each data entry contains a pointer to the head of its corresponding tag list.

possible data entries. Examples of how state-of-the-art proposals generally approach such designs are presented in the following subsections.

4.1 Many-to-One Mapping

Mapping multiple tags onto a single data entry is the simplest approach; thus, it is the go-to strategy among many compressed cache layouts (e.g., **SCMS** [50, 51] and **YACC** [76]). Although such techniques come in multiple flavors, each with its minor nuances, the general idea can be described as follows (see Figure 1): ① each data entry is uniquely coupled with a few fixed tag entries. ② When a block's data is inserted into a data entry, one of its respective tags is used—its coherence bits are set (C bits), and the respective compression metadata is stored (M bits)—and the block's contents are copied over. ③ If another block is assigned to this data entry, and is able to co-allocate with the existing block, then another tag entry is used, and the new block is stored alongside the previous one. This process is repeated until this data entry has no spare tag entries, a point at which tags would need to be evicted following the replacement policy's guidelines (R bits).

The previously described design works well for compression, in general; however, for the specific case of inter-line granularity (see Section 3.1), one can opt not to compress data entries, but to associate multiple tags with a specific data entry instead—that is, a *variable* number of tags can map to a single data entry. This flexibility can be achieved by connecting tag entries through a doubly linked list and mapping each list to a single data entry [29, 54, 87]. Eviction of a data block requires that each tag entry contains a pointer to the previous and next tags, as well as a pointer to the data entry it is associated with. Finally, since the storages must be effectively fully decoupled, each data entry must hold a pointer to its respective tag list's head. The general design is shown in Figure 2.

To avoid having to perform full storage scans, these techniques typically employ hash tables, which allow quickly locating blocks based on their contents; if two blocks exhibit similar values, they should be mapped to the same data entry. On lookups and writes this map value is generated, and the cache is searched for matches — first in the tag array, and then in the map tag array (hash value) — to find the corresponding data block position. Special care must be taken on replacements, as the removal of a data block implies on the removal of several tag entries. As a result, previous to removing an entry to insert a new one, the cache can be searched for similar values: On matches, the new tag is simply appended to the match's linked list; otherwise, all tag entries referencing the to-be-evicted data block must be properly invalidated.

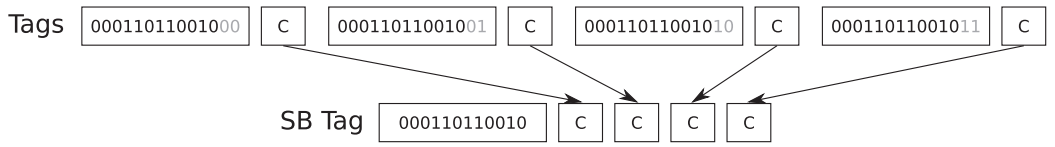

Fig. 3. Tag bits can be deduplicated if the tag field is reduced to accommodate only the **Most-significant Bits (MSB)** of the original tags. Since in this example there are four blocks per superblock, and the tag offset is only two-bits long, the differing tag bits can be stored implicitly though the order of the coherence fields.

4.2 Many-to-Many Mapping

When there is an M to N relationship between the tag and data storages to accommodate for compression, the cache design becomes slightly more complex; yet, the tag and data entries have more flexibility, allowing for better placement decisions. Compressed caches that decouple tag and data are typically subject to at least one level of indirection, and the ideas usually derive from similar schemes conceived for conventional caches, such as the **Indirect-index Cache (IIC)** [31] and the Decoupled Sectored Cache [81]. The main difference between decoupling in uncompressed and compressed caches is the granularity: Since compressed blocks can fit in spaces smaller than the size of data entries, such entries can be partitioned into multiple small *segments* [5, 12, 32, 67, 77] (more on segments in Section 4.4).

For instance, in the **Decoupled Variable-segment Cache (DVSC)** [5] the data entries are divided into eight-bit segments, and despite the fact that each set contains four data entries, the tags assume an eight-way set-associative configuration—i.e., there can be at most eight blocks simultaneously present in a set. Each tag is modified to contain the compression status and a counter of the number of segments used by the data it refers to. On compression, a block is rounded up to a number of segments, and inserted in the set in contiguous address order, that is, at the segment after the last used one—a block's segments can then span over two physical data entries. Due to the compelled contiguity, a block update may require segments to be re-compacted if the new size is different from the old.

4.3 Reducing Tag Overhead

Increasing the number of tags has an exceedingly high cost: the tag storage overhead is increased manyfold. Besides, the usefulness of the extra tags is directly proportional to the workload's compressibility; thus, some approaches group multiple tags into a single tag entry, notably reducing the number of tag bits [1, 75–77]. This can generally be achieved through the exploitation of two remarks: Neighbor values tend to exhibit approximate similarity, and workloads tend to manifest high spatial locality [75–77].

The former remark implies that the contents of neighbor blocks are approximately equal. This can happen, for example, in neighbor pixels of images, which likely contain similar values; and when safeguarding from worst case scenarios—e.g., using data structures larger than the average value [54]. Besides, applications tend to display a high frequency of zeros [24], as it is regularly used to nullify pointers and initialize data. Finally, the second remark states that neighbor blocks are usually brought into the cache within a small time interval. This means that neighboring blocks are usually located in the cache simultaneously, and their compressibility will likely be similar [77].

Therefore, adjacent blocks can be grouped into bigger blocks, called **superblocks (SB)**—blocks that share a single common tag to reduce tag overhead (Figure 3)—in a concept derived from sector caches [61]. This can be done because the individual tags of neighbor blocks differ by very few bits, so the common bits can be shared under a common tag field. Under this configuration, blocks within a **superblock** have minimal tag-storage cost, needing a couple of extra bits to

inform per-block metadata (coherence state, replacement policy, etc.), and possibly some bits to store their tag offsets. This allows significantly reducing the inherent metadata overhead required to track multiple compressed blocks.

There is a major drawback of using superblocks, however: The number of co-allocation opportunities is drastically reduced. While previously any blocks could co-allocate, within a superblock only neighbor blocks can co-allocate. At a high level, superblocks are applying a base-delta compression algorithm (see Section 6) to tags; therefore, this concept can be expanded by changing the compressor being used [58] to find a good trade-off between size reduction and co-allocation opportunities. Alternatively, the maximum number of tags in a superblock does not necessarily need to match the amount of bits in the tag offset (number of neighbours) [104]. This allows sacrificing tag-size reduction and tag-locating complexity to achieve a similar compromise by explicitly storing the tag offset bits of each tag.

Making tags smaller reduces the burden of augmenting the tag array; yet, this overhead still exists. To effectively remove this extra compression cost, some techniques propose storing tags in the data entry [34, 58]. For instance, Touché [34] stores the extra metadata in the data entry itself. When entries contain compressed data the tag array is populated with many short signatures (e.g., one 30-bit tag is replaced by three 9-bit signatures, each referring to a different co-allocated block), instead of storing conventional tags. Since this process effectively reduces the tag size, it increases the number of tag conflicts, which generates false positives when performing lookups; thus, a copy of each original tag must be stored and accessed within the data itself to confirm matches. This reduces the available space to store the compressed data and increases the average hit latency, but it removes the need to increase the tag array's size.

4.4 Finding Segments

Most of these techniques make data smaller with the purpose of fitting more lines into the data entries. This means that, besides the conventional tag-data mapping, they must supply a way to locate the lines at a smaller granularity—i.e., they must inform not only which data entry(ies) contains the data but also where within such entry(ies) the line has been allocated. The simplest way to determine where a block is placed is to have a strictly **constrained** number of placement possibilities and enforcing a **contiguous** storage—the block's contents must not be spread over the data entry. Then, a compressed block's location can be dictated by its compressed size or other location metadata [50, 75, 76]. Figure 4 displays placement under different constraint and contiguity configurations.

Although this approach greatly cuts metadata overhead, it restricts the co-allocation opportunities, since limited placement locations translates to limited size choices (e.g., 8, 16, 32, and 64 bytes). Blocks that are able to compress to sizes smaller than the possible sizes will waste the remaining space with padding bits; furthermore, requiring blocks to be stored contiguously means that if an overwrite to a block happens, and their compressed size changes, blocks may need to be rearranged—a process called *recompaction*.

To make the most out of compression, compressed systems must explicitly keep track of the compressed block's location. This can be achieved through the addition of a field containing the integral (i.e., **unconstrained**) size [5, 21]—e.g., in a cache with 64-byte cache lines this adds up to 7 bits per tag—or pointers to indicate where the block is placed. Pointers can either associate the segments to their tag [29, 77] or the tag to its segments [29, 32]. Additionally, the use of pointers allows relaxing the contiguous property: Multiple pointers can be added to specify the location of the dispersed segments of the compressed blocks, removing the need to recompact data [32, 77]. Figure 5 shows the organization of the Decoupled Compressed Cache (DCC) [77], which is an example of how pointers can be used to map segments to tags.

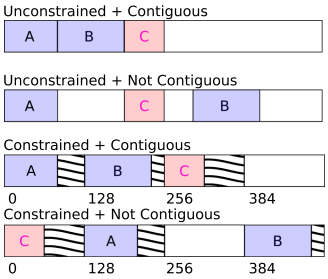

Fig. 4. Placement of compressed block C in a cache entry containing compressed blocks A and B, under different configurations. Stripes are wasted space.

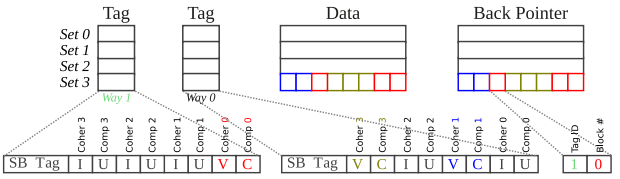

Fig. 5. Locating segments in **Decoupled Compressed Cache (DCC)**. The tag in set 3, way 1 contains a single valid compressed block at sub-block 0 (its respective coherence state is valid, and compression bit is set). The **back pointer (BP)** array contains 3 entries related to this sub-block: 0, 1, and 5. The tag ID of each of these BP entries is set to 1 to match the way at which its respective sub-block can be found, and the block number is set to the number of the sub-block. The corresponding data entries are highlighted with matching colors.

4.5 Specialized Caches

Many authors suggest adding extra smaller caches to store compressed or complement data. These special caches build upon different concepts and characteristics of compressed data; therefore, they are usually orthogonal approaches, and can be simultaneously applied to further enhance the system. The most common idea behind these specialized caches is to provide smaller data representation for frequent values: As shown by Zhang et al. [107], these values account for a significant percentage of accesses, while occurring in nearly half of the different memory locations, and generating a large proportion of caches' misses. In particular, zero values present high predominance over other values [24, 48].

In a frequent-value-based specialized cache some values are defined as frequent (e.g., by static or dynamic profiling) [23, 107]. Lines containing these values can then substitute their occurrences by their respective indices in the frequent-value reference table. In such a system a block's location depends on its contents; thus, special care must be taken to keep coherence, and accesses must perform simultaneous lookups on both conventional and specialized caches.

As in the compression algorithms themselves, one can change the granularity at which frequent values are dealt with. For instance, one can explore the lower variance of MSBs to reduce the width of a data entry. The main cache becomes then a storage of the LSB of cache lines, and each entry is paired with a compressed (e.g., frequent value or sign extension) representation of its MSB. The specialized cache can either store the MSB of all lines in the main cache—e.g., as in the **Frequent Value Cache (FVC)** [101]—in which case compression is used to avoid accessing the secondary storage; or part of the lines—e.g., as in the residue cache [46]—in which case hits to partial lines need to request the missing data from the lower cache levels.

4.6 Other Designs

The proposals presented so far consider the use of caches with a somewhat conventional tag and data array design. Compression is not limited to this layout, though—other cache designs that

significantly differ on the way data is found and accessed have been proposed over the years [36, 58, 71, 90], and can be adapted for compression support.

For instance, alloy caches [71] have been designed for DRAM caches. They embed the tag information along its data counterpart to make accesses concise (tag and data are accessed simultaneously, in a **Tag and Data (TAD)** entry). This idea is expanded upon to accommodate compression in **Dynamic-indexing Cache Compression (DICE)** [105], where each Tag and Data (TAD) entry supports a variable number of lines: as long as there is space available, blocks can co-allocate. The number of tags is informed by the addition of a bit to each tag informing whether there is a next tag, or if the data portion starts. A similar approach fixes the number of tags at two per compressed TAD to simplify indexing [4].

Most of the organization schemes described earlier focus on improving cache performance for single stream applications. Although they can be used in multi-core scenarios, it is not part of the method's design. The **Manycore-oriented Compressed Cache (MORC)** [58], however, focuses on multi-core architectures, by improving throughput. In Manycore-Oriented Compressed Cache (MORC) sets are substituted by logs, forming a log-based cache: data is appended to entries based on data commonality patterns, not data address. Writing data in MORC is as simple as adding to the end of the active log and invalidating the old version, if it exists. This means that locating data is not trivial, and requires the use of some level of indirection.

Some local memories do not need to store tags—the *scratchpads* [36]. Scratchpads are an alternative to conventional caches, mainly used in embedded systems. Contrary to transparent caches—whose address space is a subset of the main storage's—scratchpads use a different address space; thus, any requests to a given address belonging to the scratchpad space are guaranteed to hit. Data moved from the main storage to the scratchpads is stored as independent temporary copies, which can—but does not need to—be written back when evicted.

This tag-less kind of cache can be used to reduce the tag footprint of compressed caches. For example, Zippads [92] applies compression to a software-managed object-oriented cache hierarchy based on Hotpads [90]. In Zippads the data array is composed of objects, not lines; as a result, deduplication can be done between the same fields within objects, which yields better results than blindly comparing lines in object-oriented applications. New objects are always appended at the end of the data array; and, if a data expansion happens, the old entry becomes a forwarding pointer to the newly appended object. When a pad is filled, a process similar to garbage collection is triggered to free unused objects and recompact data.

4.7 Summary

Table 1 presents a summary of multiple state-of-the-art cache compaction methods with respect to the concepts presented in Sections 3 and 4. *Granularity* expresses whether deduplication occurs between values in a line, or between lines. The *Expansion* field informs the action adopted when a compressed block goes through a fat write. *Mapping* is the mapping between the tag and data storages. *Segment* contains the size of the smallest possible sub-division of a data entry, in bits. *Tag Optimization* informs if the tag representation is somehow optimized. Finally, the *Details* field adds some extra relevant information regarding the proposal.

5 COMPRESSION USEFULNESS

Compression has an inherent drawback: It adds a decompression step to cache hits that contain compressed data. Since one of the goals of increasing the effective cache capacity is to reduce the average access time, speedups are generally achieved when the total miss latency reduction overcompensates the hit latency increase. Consequently, compression can be a burden if an increase in cache capacity is not the workload's primary need. For instance, compression is a waste of energy

Table 1. Summary of Cache Compaction Layouts

Technique	Granularity	Expansion	Mapping	Segment	Tag Optimization	Details
Base-Victim [28]	Intra	Evict Victim	M:1	32	—	Logical division of lines. Victim line is stored clean.
Bunker Cache [73]	Inter	Not applicable	M:1	512	Address aliasing	Not precise. Multiple addresses are remapped to a single one.
CC [103]	Intra	Re-insert	M:1	256	—	Increases line length to store compression metadata.
DCC [77]	Intra	Make room	M:N	128	Superblock	Each segment has a pointer to its tag.
DICE [105]	Intra	Make room	M:M	32[a]	Shared Tag bit	Non-contiguous segments. Is an Alloy Cache [71].
Dedup [87]	Inter	Re-insertion	M:1	512	—	Doubly linked list.
Doppelgänger [54]	Inter	Re-insertion	M:1	512	—	Not precise. Doubly linked list. Extra cache.
DVSC [5]	Intra	Make room + Recompaction	M:N	8	—	Adaptive.
FCMS [104]	Intra	Not informed	M:1	64	Superblock	Has a small decompression buffer.
IIC-C [32]	Intra	Make room	M:N	256[b]	—	Non-contiguous segments. Adaptive.
MORC [58]	Intra,Inter	Re-insertion	M:N	N/A	Tag compression, Tags in data entry	Log-based storage.
O2W [4]	Intra	Make room	2:2	16	—	Two fixed line locations. Is an Alloy Cache [71].
Pair-matching [21]	Intra	Make room	M:1	256	—	—
SC^2 [12]	Intra	Make room	M:N	8	—	—
SCC [75]	Intra	Re-insertion	M:1	128	Superblock	Uses skewed indexing.
SCMS [51]	Intra	Make room	M:1	256[c]	—	Has a small decompression buffer.
Thesaurus [29]	Intra,Inter	Re-insertion + Recompaction	M:N	64[d]	—	Doubly linked list. Extra cache.
YACC [76]	Intra	Make room	M:1	128	Superblock	—

[a] Not informed. Inferred from the compression-metadata field size.
[b] In a 128-byte line configuration.
[c] The maximum compressed size is equal to the minimum compressed size.
[d] The minimum number of segments is two.

and a performance limiter whenever: a block is compressed, but does not manage to co-allocate with any other blocks; the extra access latency due to decompression surpasses the latency of a miss; a block co-allocates, but is never re-referenced before eviction.

Since the impact of the decompression latency is proportionally smaller on higher-latency caches, compression is typically proposed for the **Last-level Cache (LLC)**; nonetheless, some proposals tackle compression on caches closer to the core too [86, 103]. It is also worth noticing that compression can be co-applied with other miss ratio reduction techniques such as prefetching and optimized replacement policies [7, 66] to further enhance the system's performance (more on that in Section 7).

5.1 Adaptive Compression

A simple approach to reduce the likelihood of having compressed blocks stored without companions is to apply a threshold to the compressed size, such that a block is only stored in

compressed format if it is reduced to at most a certain number of bits [50]. This is implicitly used by superblock-based compaction schemes with fixed size sub-blocks—such as **Skewed Compressed Cache (SCC)** [75] and **Yet Another Compressed Cache (YACC)** [76]—since they enforce that compressed blocks must fit in either 16 or 32 bytes.

Preemptively deciding whether compression is useful based on a fixed—i.e., static—factor can be quite limiting considering that workloads tend to exhibit dynamic behavior. Besides, it limits compression: a block that has been compressed to a size $A > 50\%$ would still be able to co-allocate with a block whose compressed size is $B \leq 100 - A\%$; therefore, some techniques try to minimize this overhead by adaptively enabling and disabling compression, or switching between compressors based on the overall compressibility of the workload and the effects of compression on performance [10, 13, 63]. Others focus on determining how much of the storage should be dedicated to compressed data [88, 93].

Enabling and disabling compression can be done indirectly: the **Indirect Index Cache with Compression (IIC-C)** [32], for example, modifies its replacement policy based on the observation that the less misses there are, the lower is the usefulness of compression. In Indirect Index Cache with Compression (IIC-C) the need to move blocks between the different priority queues of its specialized replacement policy is proportional to the number of misses; thus, blocks are only compressed when they move. In any case, the majority of the adaptive compressed systems take an active role on compression control. For instance, one can use a counter to track whether compression is being helpful by training it positively on avoided or avoidable misses—accesses to blocks that would only be present in the cache due to the existence of compressible entries—and training it negatively on penalized hits—accesses that would be hits regardless of the use of compression [5]. Since the usefulness of compression is workload-dependent, a fine-grained approach (e.g., a counter per core) can improve enablement-prediction accuracy [99].

Notwithstanding the potential benefits, switching between enabling or disabling compression may introduce overhead. For example, when turning a compressed cache into an uncompressed one, there may be not enough space for all currently allocated blocks; in this case, some of them will need to be evicted. For this reason, Xie et al. [99] propose using a Decision Switching Filter, which applies a low and a high threshold to the measured access time to determine when to switch compression on or off.

5.2 Avoiding Frequent Decompressions

When a compressed block is read too frequently, the benefits of avoiding accesses to the lower-level memory may be subdued by the accumulated decompression latency. This drawback can be lessened by preemptively decompressing such blocks, so that future reads skip the decompression step. By definition, this is done to a certain extent by the caches in between the core and the compressed memory [32]. Anyhow, one can add a decompression buffer to the compressed cache itself, which must be kept coherent with the cache [51, 88]. The coherence handling can be avoided by expanding the block in the cache itself [65]; however, this has the downside of reducing the effective cache capacity due to extra evictions. Deciding which blocks to decompress follows guidelines similar to conventional replacement policies—e.g., access frequency or recency [51]—possibly incorporating the compressed size into this selection process [65].

6 COMPRESSION ALGORITHMS

So far, we have discussed the organization of compressed caches; however, a compression algorithm is needed to effectively compress data. Cache compression schemes are typically simplified versions of conventional data compression algorithms. This is due to hardware complexity constraints, and the need for speed: Since decompression is done on the critical execution path, it must

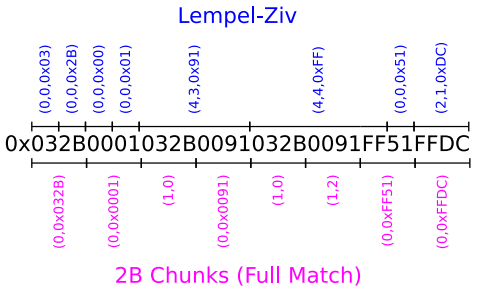

Fig. 6. Comparison of the tuples generated when using a Lempel-Ziv compressor (tuples above the data input, in blue) and a compressor with a fixed granularity of 16 bits (tuples below, in pink). Data is parsed from left to right.

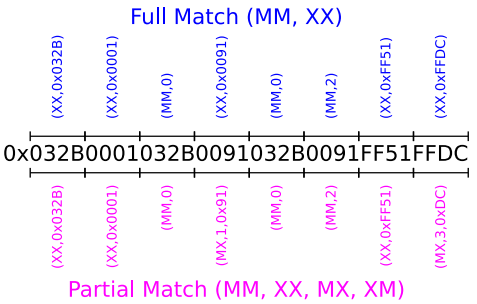

Fig. 7. Comparison of the generated tuples when using a compressor supporting only full 2-byte matches (patterns MM and XX)—above, in blue—and a compressor with partial 2-byte matching (patterns MM, XX, MX, and XM)—below, in pink. Dictionary entries are not allocated for MM instances. Data is parsed from left to right.

be fast enough not to severely degrade the average hit latency [27, 69, 74]; therefore, cache compressors usually try to find a suitable trade-off between compression efficiency and decompression latency.

Another key difference between hardware and data compressors is the input size: hardware compression algorithms are typically applied at a cache line granularity—e.g., all compression information is acquired from a chunk of 64 bytes [6, 21, 69]—while conventional algorithms do not abide to this restriction; thus, the latter are able to achieve much better compression ratios due to higher ratio between deduplication opportunities and input size.

In this section, we describe the evolution of hardware data compression, as well as what is the goal of each kind of solution. The algorithms depicted here generate better results for integer datasets, as they are usually designed for that reason, but compression can also be applied to code [16, 25, 79] and floating point data [10, 17, 52]. We have opted not to enter into detail on these non-integer-based compression methods to focus on general data compression proposals.

6.1 Dictionary-based Compression

In 1977, Lempel and Ziv proposed *LZ77* [108], a dictionary-based coder that replaces repeated occurrences of data by length-distance tuples. It works by parsing the data in a sliding window, storing, for each entry, the relative distance from the current input to its longest last seen copy, its size, and the next unmatched value—which we will represent by the tuple (*distance*,*length*,*value*). It is clear that the compression efficiency depends on the width of the distance and length fields, as well as the input size.

For example, the process of compressing the cache line from Figure 6, when parsing from left to right using a Lempel-Ziv compressor with a 1-byte minimum granularity is hereby described (corresponds to the tuples on top, in blue). Bytes *0x03*, *0x2B*, *0x00*, and *0x01* had never been seen beforehand, so they have a 0-byte distance from their respective previous tuple, and 0 known bytes (length). The fifth byte—*0x03*—however, was already seen four bytes before (dictionary match), which allows increasing the parsing window's granularity by one byte. The value *0x032B* has also been seen previously, so the window is extended again. This is repeated until the unforeseen byte *0x91* is included in the window. The distance and length of the window, as well as the divergent

byte are then recorded, generating the tuple (*4,3,0x91*). The window is then reset to its minimum granularity and the process is repeated until the whole cache line is parsed. Due to the algorithm's asymptotic efficiency, better compression ratios are noticeable as the input size increases.

Based on the original algorithm, many variations have been proposed, such as adding a flag bit to indicate if a compressed tuples contains a pointer (i.e., $length \neq 0$) or unmatched values (*LZSS*) [84]; using backward dictionary pointers (*LZ78*) [109]; pre-initializing the dictionary (*LZW*) [97]; and performing parallel comparisons [27, 89], which can then be implemented in hardware to be used in systems with compressed memories [19, 32, 85]. In particular, when porting this compression algorithm to be used in caches, its compressibility must be sacrificed to attain viable decompression latency and area overheads.

As mentioned previously, a hardware Lempel-Ziv compressor starts parsing input at a tiny granularity—e.g., byte [89]—but, as long as there are dictionary matches, the granularity grows. Although using this variable-sized input allows reaching better compression ratios, the execution speed is quite low, since the algorithm's parallel-processing capabilities are limited by the design's budget. This, combined with the need to keep decompression latency low, has led proposals to concede compressibility to attain better decompression latencies. A straightforward way to simplify compression is to limit the granularity of the input: cache lines are then parsed in fixed-size **chunks**, and each chunk is individually compared against the dictionary entries to generate its respective compressed representation. In general, the dictionary entry's size matches the chunk size, but that is not always the case [92].

With the use of fixed-size chunks the tuple representation needs to change. First, the length field is fixed at the chunk size; thus, it is no longer required. Second, the value field is only needed when there is no dictionary match, and the distance field is not used unless there is a match; thus, both fields become conditionally optional, relying on the value of a new field—the *pattern* field—which indicates if there is a match or not. Finally, the distance field can be renamed as the *pointer* field, since it contains the index of the dictionary entry to which the chunk refers. To summarize, the tuple becomes (*pattern,pointer*) for matches and (*pattern,value*) for non-matches. Figure 6 depicts how the tuple generation would differ when changing from a Lempel-Ziv compressor to a fixed-chunk-based compressor.

6.1.1 Adding Patterns. This change in granularity introduces an issue: in Lempel-Ziv the dictionary entries are compared seeking perfect matches, so even a single differing bit would make the matching fail. At a byte granularity this is a fair compromise; but, when the chunk size is larger, duplicating a whole chunk due to a couple of non-matching bits can severely restrain compressibility. This issue can be diminished by loosing constraints to perform **partial matching**. Partial matching adds new patterns, which accept a few differing bits at predefined positions, duplicating only these said bits—i.e., the tuple for matches becomes (*pattern,pointer,differingBits*). The number of bits is determined by the pattern granularity, and will be assumed to be 8 throughout this chapter, although other values can be used [15, 98].

For instance, assume that the symbol *X* means that the byte at that position of the chunk does not need to match the dictionary entry, and that *M* represents a match at that position. Then, a compressor that uses a 32-bit dictionary, and only allows perfect matches supports just the patterns MMMM and XXXX. One could, however, have different combinations of M and X bytes to perform partial matching and increase the likelihood of compression. In the previous example, the chunk *0x0091* would not be compressible, since it had not been previously seen; and thus it generates the tuple (XX, *0x0091*) under perfect-matching-only compression. When partial matching is supported—say, adding MX to the pattern list—that chunk would be compressible to (MX, 1,

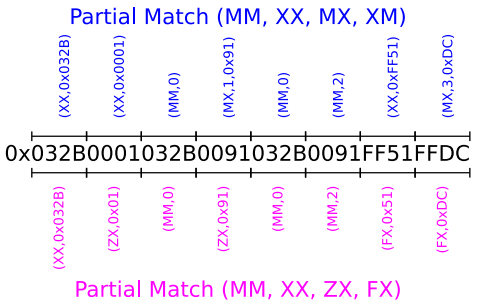

Fig. 8. Comparison of the tuples generated when using partial matching with symbols M and X (above, in blue), and partial matching with symbols M, X, Z, and F (below, in pink). Each compressor has four patterns. Dictionary entries are not allocated for MM instances. Data is parsed from left to right.

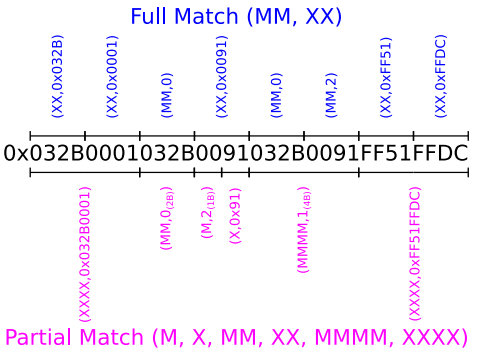

Fig. 9. Comparison of the tuples generated when using a single 2-byte dictionary (above, in blue), and three dictionaries (1, 2, and 4-byte) (below, in pink). The subscript of the index indicates which dictionary it refers to. Dictionaries using larger chunks are prioritized to reduce metadata overhead. Data is parsed from left to right.

0x91)—i.e., its last byte is different from the dictionary's entry at index 1, but the first byte matches (Figure 7).

With more patterns, the pattern field expands, incurring a larger per-chunk overhead. An alternative to cope with this extra cost is to use variable-length encoding to prioritize generating smaller pattern codes [3, 21, 47, 58–60]. An analogous idea can be applied to reduce the pointer field size [47]. These solutions, however, make the compressor more complex, resulting in higher latencies. It is worth noticing that not all possible patterns are useful; for example, a partial match on the most-significant bytes of a chunk (e.g., patterns MMMX, MMXX, and MXXX) is significantly more likely to happen than other partial-matching patterns [69]. Besides, when parsing a chunk, each dictionary entry is checked against the chunk using each of these patterns; thus, the hardware cost and complexity is higher, the bigger is the number of patterns. Consequently, it is desirable to limit the number of patterns to generate a cost-effective hardware implementation. This set of patterns is usually defined statically at design time, but compressors can also have a profiling stage to detect and select good workload-specific patterns [30, 47].

6.1.2 Adding Symbols. So far, we have discussed comparing chunks strictly against dictionary entries; yet, there are patterns that are expected to be seen frequently and do not rely on the dictionary. For example, bytes containing only-zeros or only-ones are common, mainly among the most-significant bits [23, 24, 48, 85]. Therefore, other symbols—such as a zeros byte (*Z*); a ones byte (*F*); a byte that is the sign extension of the bit that precedes it (*S*); or a byte that is repeated throughout the whole chunk (*R*)—could be added to generate patterns with lower metadata footprint. This happens because a value that is implicitly defined by the use of a pattern code, does not need to use a dictionary-entry pointer. Furthermore, the first occurrence of a value does not necessarily imply a full chunk duplication anymore (Figure 8).

6.1.3 Adjusting the Dictionary Size. As stated previously, a compressor's complexity is proportional to the number of patterns and dictionary entries; therefore, one could reduce the number of entries a chunk must be compared against to reduce its intricacy. For instance, Diff1 [15] selects the first word of a line as the sole dictionary entry, and calculates a single minimum delta *d* such

that the $32 - d$ MSB of the dictionary entry matches all words' respective bits. Then, all other values are stored as d-bit differences relative to that dictionary entry. Since not all words will differ by the same number of bits, many difference bits will be duplicated; therefore, variants of this technique—Diff2 and Diff3—assign a delta d_w to each word w to exchange metadata overhead for difference deduplication.

If both techniques are combined (limiting both the number of patterns and dictionary entries to a minimum), then the compressor's complexity can be reduced to the point of attaining negligible decompression latencies. This approach is taken by the **Base-Delta-Immediate (BDI)** compressor [69], which enforces that the dictionary contains exactly one entry—the first chunk in the cache line—while only allowing two patterns—a match except last d bits, and an all-zeros value except last d bits. Since d is defined at design, the position of every chunk's data is fixed; therefore, the compressor becomes as simple as an adder tree, and a single-cycle decompression latency is achievable.

Although defining statically which chunks will be part of the dictionary significantly reduces hardware complexity, it also severely restrains compressibility. A solution that can achieve a good trade-off between reducing complexity and keeping a similar compression ratio is to reduce the dictionary size, yet make it dynamic. This is the case of Diff2 [15] and **Frequent Pattern Compression with limited Dictionary support (FPC-D)** [4], which limit comparison of a chunk, respectively, to its former or two previous chunks. This restrictive design works well due to the spatial value locality of data: nearby chunks are likely to contain similar values [11]. Frequent Pattern Compression with limited Dictionary support (FPC-D) also increases the number of symbols and embeds the match location in the pattern code to increase efficacy and reduce the increase in complexity, respectively.

6.1.4 Multiple Dictionaries. Sometimes data can repeat in patterns larger than the dictionary entry's size. This means that the dictionary is not able to capture the granularity of the data, and will likely not be able to match (i.e., generate more no-match patterns). Furthermore, smaller dictionaries parse the input in smaller chunks, which generate proportionally more metadata overhead than larger chunks. One could reduce this metadata overhead by simultaneously using multiple dictionaries, each compressing at a different chunk size (Figure 9). Under such scheme, the dictionary using the largest chunk size is typically prioritized, as it has less entries—thus, generating the lowest pointer-related overhead. Overall, the compression efficiency is improved under this configuration, at the cost of a higher algorithmic complexity.

A variant of this idea is explored by **Large-block Encoding (LBE)** [58], which has four dictionary entry sizes: 32, 64, 128, and 256 bits. To reduce the dictionary-related area, the entries of the larger dictionaries are represented as pointers to the smallest dictionary, adding an extra level of indirection to the decompression step. In the previous example, the 16-bit dictionary entry *0x0001* would be physically stored as two 4-bit pointers (to 8-bit dictionary's *0x00*, and *0x01*, respectively). Furthermore, to decrease (de)compression complexity and reduce the number of entries per dictionary, entries for the Y-bit dictionary are only added at Y-bit boundaries. That is, the entries *0x0102*, *0x0300*, and *0x0400* would not be generated, and the space required to store the maximum number of 16-bit dictionary entries needed would be halved, removing one bit from their pointer representation.

6.1.5 Sharing Dictionaries. So far the dictionary-based compressors presented apply compression to lines individually; however, just as values can be deduplicated within a line, different lines can contain similar values, so there is a potential advantage in sharing their dictionaries [57, 58, 63, 106]. Whenever a dictionary is shared among multiple lines, the compressor's efficiency can be increased (more dictionary entries can be deduplicated). In addition, since the dictionaries are

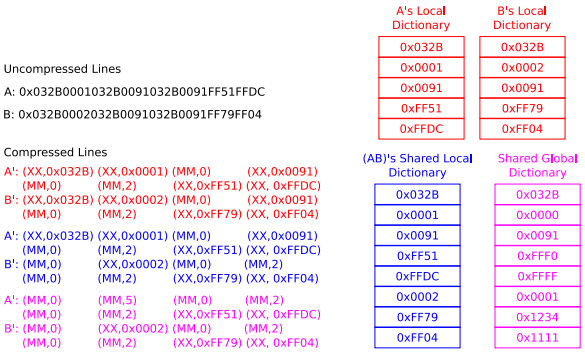

Fig. 10. Example of full match (MM and XX) compression using: (1) local dynamic dictionaries (top A'B' pair, in red)—lines are compressed isolatedly; (2) shared local dynamic dictionaries (middle, blue)—both lines construct a single dictionary (e.g., A's entries can be used when deduplicating B); and (3) shared global static dictionaries (bottom, pink). The global dictionary was populated arbitrarily, simulating a previous sampling stage. Its entries can be referenced to by every compression.

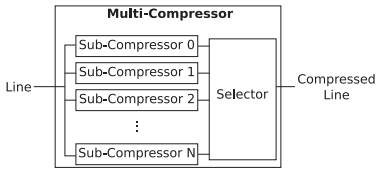

Fig. 11. Example of a multi-compressor consisting of N+1 sub-compressors. The selector can work based on the fastest decompression latency or the smallest compressed size, for example.

spanning multiple lines, they can be enlarged to contain more entries. This increases the probability of successfully compressing individual lines, at the cost of slightly larger dictionary pointers.

Compressors that share dictionary entries, however, increase the compressed cache's management complexity. In general, shared dictionaries are mostly static: updates should happen infrequently—e.g., when the current dictionary does not abide to the workload's characteristics [12]—since any modification to a dictionary entry may cause the update or eviction of multiple cache lines. This drawback can also be lessened by controlling the amount of lines that can share a dictionary—e.g., a shared dictionary per data entry [63]. In this case, writes and overwrites, besides loading and comparing dictionary entries, also require the identification of the prospective dictionary; furthermore, accesses to a line are tightly coupled with other lines, which may increase the average hit latency, raise energy consumption, and overload read and write buffers.

Shared dictionaries are generally used by profile-driven compressors [12, 15, 103]. These compressors fully separate the dictionary from the cache; the dictionary is stored in a global static table, and encompasses the majority of values expected to be seen on cache lines. To do so the compressor passes through a brief sampling stage, where the workloads' data is parsed to create a probabilistic model based on the frequency of values. This model then populates the table—which is frozen after sampling—and blocks are compressed by comparing the chunk's values to the table's entries. Since the pointers become larger due to the bigger dictionary, one can apply variable-length prefix coding methods—such as Huffman coding [35]—to reduce the metadata overhead of the most frequent values. Figure 10 exemplifies the difference between local dictionaries, 2-line dictionary sharing, and global static dictionaries.

To find the n most frequent values, Yang et al. [103], for example, propose having a **Content-addressable Memory (CAM)** [78] table with $2 \cdot n$ entries, each consisting of a value and a counter. During the sampling phase the counters are updated for each repeated value, and if the counter saturates, its respective value is swapped with the next entry. If the value was not present in the table, then one of the entries with the lowest counter is replaced by it. At the end of this stage,

the n first entries are used in the encoding. This allows a hardware-efficient way to achieve approximate sorting. In case the compression ratio worsens, another sampling stage can be started concurrently using a second decompressor; therefore, data inside the cache may have two encodings. Nonetheless, re-sampling does not need to be frequent, due to the little variation in value locality over time—even in case multiple applications are being simultaneously executed; thus, it could be software-managed to reduce compression latency [12].

6.2 Other Techniques

Dictionary compression methods achieve a certain degree of compression, and are usually effective given their processing speeds; yet, they are not the only compression methods that exist—data compression has been a field of research for long [94]. Compression can be as simple as bitwise operations [9, 43, 102], or as complex as using mathematical transformations [94]. Furthermore, a single technique does not necessarily achieve the minimum entropy on all workloads; thus, multiple layers of compression can be applied simultaneously to increase coverage. Needless to say, re-compressing the compressed data comes at a very high latency cost.

An example of multi-layer compression is **Bit-plane Compression (BPC)** [43], which combines delta and pattern compression through sequential bitwise transformations to further improve the compression ratio. In Bit-Plane Compression (BPC) compression passes through 4 stages to achieve higher compression: **Delta-Bit Plane (DBP)** transformation, which stores the first chunk as a base, and then calculates deltas between consecutive chunks (unlike Base-Delta-Immediate (BDI), which uses the difference between word and bases); bit plane rotation, which reassembles (isolates) the Delta-BitPlane (DBP) bits based on their position within the delta; then a XOR is applied between consecutive bit-planes; and lastly, the bit-planes are scanned for pattern matching.

6.3 Multi-Compressors

Each compressor has its own merits and capabilities; thus, they can be combined to improve their individual shortcomings. These are **multi-compressors**—compressors composed of multiple sub-compressors—as depicted in Figure 11. Examples of multi-compressors in the literature include BDI [69], Hybrid Compression [10], and **Dictionary Sharing (DISH)** [63]. Multi-compressors need extra bits to identify which of their sub-compressors is associated with some compressed data. These bits can be stored in either the tag or data entry. The former configuration requires a slight expansion of the tag array, but can be beneficial when performing sequential accesses, since decoding which compressor will decompress the data can be done in parallel with the tag and coherence checks. However, if the encoding metadata is stored in the data itself, or if the data access happens in parallel with the tag access, then this decoding adds 1-2 cycles to the decompression step, which correspond to the time to decode and select the sub-compressor.

6.4 Latency Trade-offs

Caches typically display low hit latencies, and decompression adds to it; thus, the higher the decompression latency, the harder it is for the system to cope with it. The speed of a compression algorithm is highly correlated with its parallelizability; therefore, the higher the amount of chunks on which a given chunk's value can depend on, the higher is the ripple effect, and the costlier it is to parallelize its operations. For example, in **Cache Packer (C-Pack)** [21] a chunk can be represented as a reference to any of the previously seen chunks (dictionary entries); in FPC-D [4] a chunk can reference one of the two most-recently seen chunks; and in base-delta compressors [69] any chunk can only refer to a single, statically determined (i.e., the first), chunk of the line. The complexities of these algorithms are reflected in their speeds: by having the highest number

Table 2. Summary of Data Cache Compressors

Technique	Dictionary Size	Entry Size (bits)	Num. Patterns	Compression	Decompression
BDI [69]	0, 1, 2, 2, 2, 2, 2, 2	64, 64, 64, 64, 64, 32, 32, 16	1, 1, 2, 2, 2, 2, 2, 2	8, 8, 8, 8, 8, 8, 4, 4, 2 bytes/cycle	8 · 8, 8 · 8, 8 · 8, 8 · 8, 8 · 8, 16 · 4, 16 · 4, 32 · 2 bytes/cycle
BPC [43]	1	32	5	7 cycles	7 cycles
COCO [92]	At least 128[a]	Variable[b]	Up to 2^{64}	8 bytes/cycle	8 bytes/cycle
C-Pack [21]	16	32	6	2 · 4 bytes/cycle	2 · 4 bytes/cycle
DFPC [30]	0	32	4 (+ 4 dynamic)	16 · 4 bytes/cycle	16 · 4 bytes/cycle
Diff1/2/3 [15]	1/1/8	32	32	16 · 4 bytes/cycle	16 · 4 bytes/cycle
DISH [63]	4, 8 (shared)	32	1	4, 4 bytes/cycle	16 · 4, 16 · 4 bytes/cycle
FPC [6]	0	32	8	3 cycles	5 cycles
FPC-D [4]	2	32	16	4 · 4 bytes/cycle	8 · 4 bytes/cycle
Huffman [12]	512/1024 (single/multi-core) (shared)	32	2	6 cycles	14 cycles
LBE [58]	128 (shared)	32	5, 3, 3, 3	4, 8, 16, 32 bytes/cycle	4, 8, 16, 32 bytes/cycle
Lempel-Ziv [27, 89]	4 · 16	32	2	4 · 1 byte/cycle	4 · 2 bytes/cycle
X-Match,X-RL [47]	16	32	10–16[c]	4 bytes/cycle	4 bytes/cycle

[a] Assuming the use of a 8 KB dictionary, and a maximum object size of 64 B.
[b] COCO works on objects, not basic data types.
[c] Depends on the probabilities of the patterns in the set that generates the Huffman codes.

of deduplication opportunities, Cache Packer (C-Pack) achieves high compressibility, yet its hardware is harder to parallelize; FPC-D exchanges some opportunities to simplify its implementation and increase its parallelizability; finally, a base-delta compressor's circuit can be as simple and fast as an adder tree.

6.5 Summary

Table 2 presents a summary of the state-of-the-art compression algorithms. All values assume a 64-byte cache line configuration, for uniformity. The latencies shown here are merely suggestive under the assumption of *a worst case scenario*, that the input data is available in its entirety when compression starts (e.g., no delays due to bus width not matching line size), and the use of single data rate. Also, compressors are typically parallelizable, so the level of parallelization claimed in their original proposal is shown as a multiplier on the latency. In real implementations the levels of parallelization are possibly different, since they depend on the area budget, clock frequency of the (de)compression units, among other design decisions. Moreover, the latencies stated in bytes per cycle do not take into account the cycles needed to perform other pipeline operations on the compressed data, such as packing and shifting.

For multi-compressors, the values shown are for their respective sub-compressors. In general, it should be assumed that the average compression latency of a multi-compressor matches the latency of its slowest sub-compressor. In practice, some optimizations can be applied to stop execution early—e.g., if a sub-compressor that achieves the smallest possible size among the remaining sub-compressors is successful, then the others can be halted. Finally, the decompression latency of multi-compressors only takes into account the time to decompress the data of the sub-compressors. In reality, 1 or 2 extra cycles may be required to decode which sub-compressor to use.

7 INTERACTIONS

There are multiple components and policies in a cache, each of which is designed under specific assumptions. Frequently, one of these assumptions is that each single line occupies a whole data entry; however, with cache compression, this is not the case any longer. In this section, we list some of the side effects and interactions that have been observed in compressed caches, and how some proposals modify such policies and components to attain a cohesive design.

7.1 Interaction with Indexing Policies

Changes to the cache design to accommodate compression do not need to be a complete makeover; some techniques simply change the way blocks are indexed to take advantage of the characteristics of compression. For example, under traditional (uncompressed) indexing adjacent lines map to adjacent sets. This means that compressed caches co-allocate lines that are spatially distant from each other when using this scheme. As mentioned previously, spatially close lines present similar compressibility, and are more likely to be used within the same timeframe; thus, compressed systems that modify indexing to maximize co-allocation can improve performance and bandwidth consumption [39, 105].

For example, Young et al. [105] suggest using traditional indexing when data is not compressible, and a special spatial indexing otherwise **(Bandwidth-aware Indexing (BAI))**. Cache lines under Bandwidth-Aware Indexing (BAI) are either on the same set or the neighboring set as they would be under traditional indexing, so that consecutive cache lines map to the same set, instead of several blocks apart. This allows successive data to be compressed together in the same set; therefore, accesses to a compressed line can provide multiple spatially neighboring lines, while still keeping some similarity with the traditional approach. Since this scheme relies on the compressibility of the data set, special care must be taken to determine when to prioritize each indexing policy.

The indexing policy's hashing functions can also be modified to better support compression. For instance, cache skewing [80] applies a different hash per way. This may hinder co-allocation, since lines that co-allocate well may be placed apart from each other. A compressed cache that uses skewing—e.g., Skewed Compressed Cache (SCC) [75]—can cope with that issue by mapping blocks based on a function of the way, address, and their compression factors. This reduces the number of conflict misses without changing the cache's associativity.

7.2 Interaction with Replacement Policies

Although conventional replacement policies can be used with most of the compression techniques previously described, compression and replacement policies can interact negatively [14, 28, 64, 65, 67]. For example, assuming the use of a LRU replacement policy, when a MRU cache line is co-allocated with the LRU block, either the MRU line will be co-evicted, or the policy must be given flexibility to search for another less controversial entry. In addition, inserting a block may require the eviction of multiple lines due to the removal of a shared tag, or evictions not freeing enough data-entry space for the new block. This means that a compression-aware policy might result in a more efficient cache space utilization. For example, a compressed data's size is important to determine how many other blocks can be co-allocated with it; therefore, a replacement policy that takes the co-allocatability of a block into account could improve the compaction ratio. Nonetheless, co-allocatability should not be considered the paramount factor; otherwise, a small cache line with low temporal locality would occupy space indefinitely [14].

Multiple compression-aware replacement policies (e.g., ECM [14], CAMP [67], and HoPE [65]) derive from **Re-reference Interval Prediction (RRIP)** [37], a simple—yet efficient—replacement policy. Re-Reference Interval Prediction (RRIP) assigns a value to each entry, indicating its distance

from being re-referenced. If this value is 0, then it is likely to be re-referenced in the near future, otherwise its next reference is distant. On hits the value is reset to 0, and on misses all values are incremented until a victim (block with the highest possible value) is found. RRIP can become size aware by, for instance, assigning different initial values for newly inserted entries based on their compressed sizes. For example, **Effective Cache Maximizer (ECM)** [14] was proposed for segment-based compressed layouts. It decides the size thresholds to assign different values dynamically, based on the physical memory usage ratio and the average segment usage per set. When choosing from the pool of eviction candidates, Effective Cache Maximizer (ECM) uses the size information to select the entry that frees the most data space.

Panda et al. [64] notice that over 55% of the blocks in a compressed cache are used only once before being evicted; therefore, it would not be beneficial to waste tag storage with these blocks. To deal with this situation, they propose the **Synergistic cache layour for Reuse and Compression (SRC)**, which applies the principle of the reuse cache [8] to caches using superblock compression, such as Yet Another Compressed Cache (YACC) [76] and SCC [75]. Synergistic cache layout for Reuse and Compression (SRC) expands superblocks with a "first use" field to defer data allocation. Whenever an access to an invalid block whose superblock was already present happens, instead of allocating the data, the bit is set. This way, an effective write of the data contents only happens when blocks are reused.

Alternatively, one can take an opportunistic approach and not include replacement data for the co-allocated lines. For instance, the Base-Victim cache [28] *logically* divides cache entries into a *Baseline* and a *Victim* Cache, associating two tags per physical data entry. Whenever a Baseline entry is evicted, a special step is added to the eviction: if the evicted entry can co-allocate with another Baseline block, then it is saved as a clean entry in that Baseline block's respective Victim area; therefore, future read references can still be served by this cache level. As the Victim Cache consists of clean lines, evicting them is free of data traffic. The replacement policy only keeps track of the main line (Baseline Cache) in every entry; therefore, by design, the Base-Victim cache cannot have a miss rate higher than an uncompressed cache. Read and write hits in the Baseline cache are trivial, and write hits to the Victim Cache cannot happen due to the enforced inclusive property. When a read hit happens in the Victim Cache, the block is promoted to the Baseline in a fashion similar to a regular miss, with the advantage of not needing to access the lower level memory.

7.3 Interactions with Prefetching

Other works address the interactions with prefetching [1, 7, 12, 68]. Prefetching and compression can have an interesting synergy: bad prefetches can bring blocks that end up polluting the cache; however, in a compressed cache these blocks may co-allocate with useful blocks, reducing the drawbacks of those bad prefetches. In addition, good prefetches may take further advantage of the increase in the cache's effective capacity.

For instance, **Prefetched Blocks Compaction (PBC)** [39] explores the similarity of prefetched lines to co-allocate them through inter-line compression and maximize use of the effective cache capacity. It splits the cache into two different-sized caches: a compacted and a conventional part. To avoid recompactions and data expansions, the compacted cache only stores prefetched (non-dirty) entries; therefore, the block must be moved to the conventional cache when a writeback happens from a higher cache. Blocks are also moved when there are no available entries in the compactable cache. To control underutilization when not many prefetches are generated, or the dataset is not sufficiently compactable, Prefetched Blocks Compaction (PBC) can change the size of the compactable cache dynamically.

7.4 Interactions with Security

Besides the direct drawback on the hit latency, compressed systems can be hazardous at a security level: the compressibility of data leaks information that can be explored to retrieve the data itself [40]. The key idea is that an S-bits data block whose compressed size is S' bits reduces the exploration space of attacks, since only a subset of the possible 2^S values can compress to S' through a given compression algorithm.

In the context of hardware data compression, Tsai et al. [91] have applied this exploit to BDI [69]. Assuming that an attacker controls the contents of a line, except for the x-bit secret it wants to unveil, their proposal combines brute force with exploration space reduction. An attacking line is generated such that it is only compressible if the secret's $b - d$ most significant bits—where b and d are the base and delta sizes, respectively—match the respective bits in the base being forced. All possible variations of the $b - d$ bits are tested until compression is successful—i.e., such $b - d$ bits of the secret become known. Then, the exploration space can be halved, and d is reduced to d' to discover the next $\frac{b-d}{2} - d'$ bits. This process is repeated until all bits are found. Although conceived for BDI, this concept can be applied to any dictionary-based compressor by selectively filling the dictionary and identifying which pattern matches the secret. While this article did not show the possibility of a realistic attack on a compressed cache, it pointed out that compressed caches can leak information to an attacker.

8 CONCLUSION

The cache hierarchy provides a great speedup when compared to directly accessing the memory; yet, it increases the system's cost, energy consumption, and bandwidth requirements. In addition, due to the capacity limitations of each level, they add extra—although significantly smaller than the hierarchy-less access latency—delays to misses. Several works have addressed these pitfalls to reduce their impact, many of which under the domain of cache compression. While compressed caches manage to reduce some of the drawbacks of their uncompressed counterparts, they come at a different cost, and with their own sets of challenges. Regardless, their benefits and the increasingly high cost of larger memories have resulted in the slow emergence of a few commercial applications over the past decades.

In this manuscript, we have thoroughly explored the problems faced when designing a compressed cache, presenting how the literature typically tackles them. The primary decision is the goal of the compressed system: although results may overlap, choosing whether to focus on increasing effective capacity, decreasing energy consumption, improving bandwidth, or even reducing the overhead of orthogonal techniques impacts the system differently. Then, the compression algorithm, compaction layout and how they interact with each other and the rest of the cache must be decided. The cache itself must change: there must be a way to map more blocks into physical data entries, as well as determining their compressed sizes. Adding more tag entries solves the issue, but significantly adds overhead. Alternative representations have smaller costs, but typically imply sacrificing mapping freedom or compressibility.

The compressor cannot significantly affect the area budget or the access latency; therefore, proposals tend to trade efficiency for speed, which is typically achieved by keeping circuit complexity low and increasing the degree of parallelization. Yet, even at small latencies, compression can still be a burden for some workloads; thus, it is essential to provide means to reduce its negative impact. Solutions usually include caching decompression results to avoid the drawbacks of the decompression step on the critical path, and dynamically deciding based on the execution characteristics whether compression is useful, or should be temporarily disabled.

Despite all the advancements in compressed systems, the general average effective capacity improvement they provide is still quite low compared to their theoretical upper limit. Most of the hardware compression algorithms are simplifications of dictionary-based data-compression algorithms; yet, other approaches exist, and future hardware compression proposals can further learn from decades of software data-compression literature. In addition, compression has the ability to enhance a level in the cache hierarchy, and its potential is even higher when applied to multiple levels—a completely compressed memory hierarchy is then the holy grail of hardware compression. Although a few previous works partially address such systems, it is an unsolved challenge. Different levels have different restraints, both in latency and area budget, so it is important to be flexible enough to attain a good per-level trade-off between compressibility and speed, while still maintaining compressor compatibility between different memory levels.

REFERENCES

[1] Ali-Reza Adl-Tabatabai, Anwar M. Ghuloum, and Shobhit O. Kanaujia. 2007. Compression in cache design. In *Proceedings of the 21st Annual International Conference on Supercomputing (ICS'07)*. ACM, 190–201. DOI: https://doi.org/10.1145/1274971.1274999

[2] Anant Agarwal and Stephen D. Pudar. 1993. *Column-Associative Caches: A Technique for Reducing the Miss Rate of Direct-Mapped Caches*. Association for Computing Machinery. 179–190 pages. DOI: https://doi.org/10.1145/165123.165153

[3] Edward Ahn, Seung-Moon Yoo, and Sung-Mo Steve Kang. 2001. Effective algorithms for cache-level compression. In *Proceedings of the 11th Great Lakes Symposium on VLSI (GLSVLSI'01)*. ACM, 89–92. DOI: https://doi.org/10.1145/368122.368872

[4] Alaa R. Alameldeen and Rajat Agarwal. 2018. Opportunistic compression for direct-mapped DRAM caches. In *Proceedings of the International Symposium on Memory Systems (MEMSYS'18)*. Association for Computing Machinery, 129–136. DOI: https://doi.org/10.1145/3240302.3240429

[5] Alaa R. Alameldeen and David A. Wood. 2004. Adaptive cache compression for high-performance processors. *SIGARCH Comput. Archit. News* 32, 2 (Mar. 2004), 212. DOI: https://doi.org/10.1145/1028176.1006719

[6] Alaa R. Alameldeen and David A. Wood. 2004. Frequent pattern compression: A significance-based compression scheme for L2 caches. *Department of Computer Science, University of Wisconsin-Madison, Technical Report No.* 1500.

[7] Alaa R. Alameldeen and David A. Wood. 2007. Interactions between compression and prefetching in chip multiprocessors. In *Proceedings of the IEEE 13th International Symposium on High Performance Computer Architecture (HPCA'07)*. IEEE, 228–239. DOI: https://doi.org/10.1109/HPCA.2007.346200

[8] 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'13)*. Association for Computing Machinery, 310–321. DOI: https://doi.org/10.1145/2540708.2540735

[9] Chloe Alverti, Georgios Goumas, Konstantinos Nikas, Angelos Arelakis, Nectarios Koziris, and Per Stenström. 2015. Memory link compression to speedup scientific workloads. In *Proceedings of the 8th Workshop on Programmability Issues for Heterogeneous Multicores*. Amsterdam, Netherlands.

[10] Angelos Arelakis, Fredrik Dahlgren, and Per Stenstrom. 2015. HyComp: A hybrid cache compression method for selection of data-type-specific compression methods. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO'15)*. Association for Computing Machinery, 38–49. DOI: https://doi.org/10.1145/2830772.2830823

[11] Angelos Arelakis and Per Stenstrom. 2014. A case for a value-aware cache. *Comput. Archit. Lett.* 13, 1 (2014), 1–4.

[12] 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, 145–156. DOI: https://doi.org/10.1109/ISCA.2014.6853231

[13] Akhil Arunkumar, Shin-Ying Lee, Vignesh Soundararajan, and Carole-Jean Wu. 2018. Latte-cc: Latency tolerance aware adaptive cache compression management for energy efficient gpus. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'18)*. IEEE Computer Society, 221–234. DOI: https://doi.org/10.1109/HPCA.2018.00028

[14] Seungcheol Baek, Hyung Gyu Lee, Chrysostomos Nicopoulos, Junghee Lee, and Jongman Kim. 2013. ECM: Effective capacity maximizer for high-performance compressed caching. In *Proceedings of the IEEE 19th International Symposium on High-Performance Computer Architecture (HPCA'13)*. IEEE Computer Society, 131–142. DOI: https://doi.org/10.1109/HPCA.2013.6522313

[15] L. Benini, D. Bruni, A. Macii, and E. Macii. 2002. Hardware-assisted data compression for energy minimization in systems with embedded processors. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'02)*. IEEE Computer Society, 449.

[16] Árpád Beszédes, Rudolf Ferenc, Tibor Gyimóthy, André Dolenc, and Konsta Karsisto. 2003. Survey of code-size reduction methods. *ACM Comput. Surv.* 35, 3 (Sept. 2003), 223–267. DOI: https://doi.org/10.1145/937503.937504

[17] Martin Burtscher and Paruj Ratanaworabhan. 2010. gFPC: A self-tuning compression algorithm. In *Proceedings of the Data Compression Conference (DCC'10)*. IEEE Computer Society, 396–405. DOI: https://doi.org/10.1109/DCC.2010.42

[18] Chin-Long Chen and M. Y. Hsiao. 1984. Error-correcting codes for semiconductor memory applications: A state-of-the-art review. *IBM J. Res. Dev.* 28, 2 (1984), 124–134.

[19] David Chen, Enoch Peserico, and Larry Rudolph. 2003. A dynamically partitionable compressed cache. In *Proceedings of the Singapore-MIT Alliance Symposium*.

[20] Long Chen, Yanan Cao, and Zhao Zhang. 2013. Free ECC: An efficient error protection for compressed last-level caches. In *Proceedings of the IEEE 31st International Conference on Computer Design (ICCD'13)*. IEEE Computer Society, 278–285. DOI: https://doi.org/10.1109/ICCD.2013.6657054

[21] 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.* 18, 8 (2010), 1196–1208. DOI: https://doi.org/10.1109/TVLSI.2009.2020989

[22] Pat Conway, Nathan Kalyanasundharam, Gregg Donley, Kevin Lepak, and Bill Hughes. 2010. Cache hierarchy and memory subsystem of the AMD Opteron processor. *IEEE Micro* 30, 2 (2010), 16–29.

[23] Julien Dusser, Thomas Piquet, and André Seznec. 2009. Zero-content augmented caches. In *Proceedings of the 23rd International Conference on Supercomputing (ICS'09)*. Association for Computing Machinery, 46–55. DOI: https://doi.org/10.1145/1542275.1542288

[24] Magnus Ekman and Per Stenstrom. 2005. A robust main-memory compression scheme. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA'05)*. IEEE Computer Society, 74–85. DOI: https://doi.org/10.1109/ISCA.2005.6

[25] Jens Ernst, William Evans, Christopher W. Fraser, Todd A. Proebsting, and Steven Lucco. 1997. Code compression. *SIGPLAN Not.* 32, 5 (May 1997), 358–365. DOI: https://doi.org/10.1145/258916.258947

[26] Alexandra Ferreron, Dario Suarez-Gracia, Jesus Alastruey-Benede, Teresa Monreal-Arnal, and Pablo Ibanez. 2016. Concertina: Squeezing in cache content to operate at near-threshold voltage. *IEEE Trans. Comput.* 65, 3 (Mar. 2016), 755–769. DOI: https://doi.org/10.1109/TC.2015.2479585

[27] P. Franaszek, J. Robinson, and J. Thomas. 1996. Parallel compression with cooperative dictionary construction. In *Proceedings of the Conference on Data Compression (DCC'96)*. IEEE Computer Society, 200. Retrieved from http://dl.acm.org/citation.cfm?id=789084.789497.

[28] Jayesh Gaur, Alaa R. Alameldeen, and Sreenivas Subramoney. 2016. Base-victim compression: An opportunistic cache compression architecture. In *Proceedings of the ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA'16)*. IEEE Press, 317–328.

[29] Amin Ghasemazar, Prashant Nair, and Mieszko Lis. 2020. Thesaurus: Efficient cache compression via dynamic clustering. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*. Association for Computing Machinery, 527–540. DOI: https://doi.org/10.1145/3373376.3378518

[30] Yuncheng Guo, Yu Hua, and Pengfei Zuo. 2018. DFPC: A dynamic frequent pattern compression scheme in NVM-based main memory. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'18)*. IEEE, 1622–1627. DOI: https://doi.org/10.23919/DATE.2018.8342274

[31] Erik G. Hallnor and Steven K. Reinhardt. 2000. *A Fully Associative Software-Managed Cache Design*. Association for Computing Machinery. 107–116. DOI: https://doi.org/10.1145/339647.339660

[32] Erik G. Hallnor and Steven K. Reinhardt. 2005. A unified compressed memory hierarchy. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA'05)*. IEEE Computer Society, 201–212. DOI: https://doi.org/10.1109/HPCA.2005.4

[33] John L. Hennessy and David A. Patterson. 2012. *Computer Architecture: A Quantitative Approach*. Elsevier.

[34] Seokin Hong, Bulent Abali, Alper Buyuktosunoglu, Michael B. Healy, and Prashant J. Nair. 2019. Touché: Towards ideal and efficient cache compression by mitigating tag area overheads. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'19)*. Association for Computing Machinery, 453–465. DOI: https://doi.org/10.1145/3352460.3358281

[35] David A. Huffman et al. 1952. A method for the construction of minimum-redundancy codes. *Proc. IRE* 40, 9 (1952), 1098–1101.

[36] Bruce Jacob, David Wang, and Spencer Ng. 2010. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann.

[37] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, 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)*. Association for Computing Machinery, 60–71. DOI: https://doi.org/10.1145/1815961.1815971

[38] Lei Jiang, Bo Zhao, Youtao Zhang, Jun Yang, and Bruce R. Childers. 2012. Improving write operations in MLC phase change memory. In *Proceedings of the IEEE International Symposium on High-Performance Comp Architecture*. IEEE Computer Society, 201–210. DOI: https://doi.org/10.1109/HPCA.2012.6169027

[39] Raghavendra K., Biswabandan Panda, and Madhu Mutyam. 2015. PBC: Prefetched blocks compaction. *IEEE Trans. Comput.* 65 (01 2015), 1–1. DOI: https://doi.org/10.1109/TC.2015.2493533

[40] John Kelsey. 2002. Compression and information leakage of plaintext. In *Proceedings of the International Workshop on Fast Software Encryption (Lecture Notes in Computer Science)*, Vol. 2365. Springer, 263–276. DOI: https://doi.org/10.1007/3-540-45661-9_21

[41] Georgios Keramidas, Konstantinos Aisopos, and Stefanos Kaxiras. 2006. Dynamic dictionary-based data compression for level-1 caches. *Archit. Comput. Syst.* 3894 (2006), 114–129. DOI: https://doi.org/10.1007/11682127_9

[42] Mushfique Junayed Khurshid and Mikko Lipasti. 2013. Data compression for thermal mitigation in the hybrid memory cube. In *Proceedings of the IEEE 31st International Conference on Computer Design (ICCD'13)*. IEEE Computer Society, 185–192. DOI: https://doi.org/10.1109/ICCD.2013.6657041

[43] Jungrae Kim, Michael Sullivan, Esha Choukse, and Mattan Erez. 2016. Bit-plane compression: Transforming data for better compression in many-core architectures. *SIGARCH Comput. Archit. News* 44, 3 (June 2016), 329–340. DOI: https://doi.org/10.1145/3007787.3001172

[44] Jungrae Kim, Michael Sullivan, Seong-Lyong Gong, and Mattan Erez. 2015. Frugal ECC: Efficient and versatile memory error protection through fine-grained compression. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15)*. Association for Computing Machinery, Article 12, 12 pages. DOI: https://doi.org/10.1145/2807591.2807659

[45] N. Kim, Todd Austin, and Trevor Mudge. 2002. Low-energy data cache using sign compression and cache line bisection. In *Proceedings of the 2nd Annual Workshop on Memory Performance Issues (WMPI'02)*.

[46] Soontae Kim, Jongmin Lee, Jesung Kim, and Seokin Hong. 2011. Residue cache: A low-energy low-area L2 cache architecture via compression and partial hits. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'11)*. ACM, 420–429. DOI: https://doi.org/10.1145/2155620.2155670

[47] Morten Kjelso, Mark Gooch, and Simon Jones. 1996. Design and performance of a main memory hardware data compressor. In *Proceedings of the 22nd EUROMICRO Conference: Beyond 2000: Hardware and Software Design Strategies*. IEEE Computer Society, 423–430. DOI: https://doi.org/10.1109/EURMIC.1996.546466

[48] M. Kjelso, M. Gooch, and S. Jones. 1998. Empirical study of memory-data: Characteristics and compressibility. *IEE Proc. Comput. Dig. Techn.* 145, 1 (1998), 63–67. DOI: https://doi.org/10.1049/ip-cdt:19981797

[49] Sumeet Kumar, Prateek Pujara, and Aneesh Aggarwal. 2004. Bit-sliced datapath for energy-efficient high performance microprocessors. In *Proceedings of the International Workshop on Power-Aware Computer Systems (Lecture Notes in Computer Science)*, Vol. 3471. Springer, 30–45. DOI: https://doi.org/10.1007/11574859_3

[50] Jang-Soo Lee, Won-Kee Hong, and Shin-Dug Kim. 1999. Design and evaluation of a selective compressed memory system. In *Proceedings of the International Conference on Computer Design (ICCD'99)*. IEEE Computer Society, 184–191. DOI: https://doi.org/10.1109/ICCD.1999.808424

[51] Jang-Soo Lee, Won-Kee Hong, and Shin-Dug Kim. 2000. An on-chip cache compression technique to reduce decompression overhead and design complexity. *J. Syst. Architect.* 46, 15 (2000), 1365–1382.

[52] Peter Lindstrom and Martin Isenburg. 2006. Fast and efficient compression of floating-point data. *IEEE Trans. Visual. Comput. Graph.* 12, 5 (Sept. 2006), 1245–1250. DOI: https://doi.org/10.1109/TVCG.2006.143

[53] Gabriel H. Loh and Mark D. Hill. 2011. Efficiently enabling conventional block sizes for very large die-stacked DRAM caches. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'11)*. Association for Computing Machinery, 454–464. DOI: https://doi.org/10.1145/2155620.2155673

[54] Joshua San Miguel, Jorge Albericio, Andreas Moshovos, and Natalie Enright Jerger. 2015. DoppelgäNger: A cache for approximate computing. In *Proceedings of the 48th International Symposium on Microarchitecture* (*MICRO'15*). Association for Computing Machinery, 50–61. DOI: https://doi.org/10.1145/2830772.2830790

[55] Sparsh Mittal. 2016. A survey of techniques for approximate computing. *ACM Comput. Surv.* 48, 4, Article 62 (Mar. 2016), 33 pages. DOI: https://doi.org/10.1145/2893356

[56] Sparsh Mittal and Jeffrey S. Vetter. 2015. A survey of architectural approaches for data compression in cache and main memory systems. *IEEE Trans. Parallel Distrib. Syst.* 27, 5 (2015), 1524–1536.

[57] Tri M. Nguyen, Adi Fuchs, and David Wentzlaff. 2018. CABLE: A cache-based link encoder for bandwidth-starved manycores. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'18)*. IEEE Press, 312–325. DOI: https://doi.org/10.1109/MICRO.2018.00033

[58] Tri M. Nguyen and David Wentzlaff. 2015. MORC: A manycore-oriented compressed cache. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO'15)*. Association for Computing Machinery, 76–88. DOI: https://doi.org/10.1145/2830772.2830828

[59] Jose Luis Nunez, Claudia Feregrino, Stephen Bateman, and Simon Jones. 1999. The X-MatchLITE FPGA-based data compressor. In *Proceedings of the 25th EUROMICRO Conference*, Vol. 1. IEEE Computer Society, 1126–1132. DOI: https://doi.org/10.1109/EURMIC.1999.794458

[60] Jose Luis Nunez, Claudia Feregrino, Simon Jones, and Stephen Bateman. 2001. X-MatchPRO: A ProASIC-based 200 Mbytes/s full-duplex lossless data compressor. In *Proceedings of the International Conference on Field Programmable Logic and Applications (Lecture Notes in Computer Science)*, Vol. 2147. Springer, 613–617. DOI: https://doi.org/10.1007/3-540-44687-7_65

[61] Howard T. Olnowich. 1985. Set associative sector cache. U.S. Patent 4,493,026.

[62] David J. Palframan, Nam Sung Kim, and Mikko H. Lipasti. 2015. COP: To compress and protect main memory. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. Association for Computing Machinery, 682–693. DOI: https://doi.org/10.1145/2749469.2750377

[63] Biswabandan Panda and André Seznec. 2016. Dictionary sharing: An efficient cache compression scheme for compressed caches. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. IEEE Press, Article 1, 12 pages. DOI: https://doi.org/10.1109/MICRO.2016.7783704

[64] Biswabandan Panda and André Seznec. 2018. Synergistic cache layout for reuse and compression. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT'18)*. Association for Computing Machinery, Article 4, 13 pages. DOI: https://doi.org/10.1145/3243176.3243178

[65] Jaehyun Park, Seungcheol Baek, Hyung Gyu Lee, Chrysostomos Nicopoulos, Vinson Young, Junghee Lee, and Jongman Kim. 2017. HoPE: Hot-cacheline prediction for dynamic early decompression in compressed LLCs. *ACM Trans. Des. Autom. Electron. Syst.* 22, 3, Article 40 (Apr. 2017), 25 pages. DOI: https://doi.org/10.1145/2999538

[66] Bhargavraj Patel, Nikos Hardavellas, and Gokhan Memik. 2015. SCP: Synergistic cache compression and prefetching. In *Proceedings of the 33rd IEEE International Conference on Computer Design (ICCD'15)*. IEEE Computer Society, 164–171. DOI: https://doi.org/10.1109/ICCD.2015.7357098

[67] Gennady Pekhimenko, Tyler Huberty, Rui Cai, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2015. Exploiting compressed block size as an indicator of future reuse. In *Proceedings of the IEEE 21st International Symposium on High Performance Computer Architecture (HPCA'15)*. IEEE Computer Society, 51–63. DOI: https://doi.org/10.1109/HPCA.2015.7056021

[68] Gennady Pekhimenko, Vivek Seshadri, Yoongu Kim, Hongyi Xin, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2013. Linearly compressed pages: A low-complexity, low-latency main memory compression framework. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'13)*. Association for Computing Machinery, 172–184. DOI: https://doi.org/10.1145/2540708.2540724

[69] 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)*. Association for Computing Machinery, 377–388. DOI: https://doi.org/10.1145/2370816.2370870

[70] Prateek Pujara and Aneesh Aggarwal. 2005. Restrictive compression techniques to increase level 1 cache capacity. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'05)*. IEEE, IEEE Computer Society, 327–333. DOI: https://doi.org/10.1109/ICCD.2005.94

[71] Moinuddin K. Qureshi and Gabe H. Loh. 2012. Fundamental latency trade-off in architecting DRAM caches: Outperforming impractical SRAM-tags with a simple and practical design. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'12)*. IEEE Computer Society, 235–246. DOI: https://doi.org/10.1109/MICRO.2012.30

[72] Moinuddin K. Qureshi, M. Aater Suleman, and Yale N. Patt. 2007. Line distillation: Increasing cache capacity by filtering unused words in cache lines. In *Proceedings of the IEEE 13th International Symposium on High Performance Computer Architecture (HPCA'07)*. IEEE Computer Society, 250–259. DOI: https://doi.org/10.1109/HPCA.2007.346202

[73] Joshua San Miguel, Jorge Albericio, Natalie Enright Jerger, and Aamer Jaleel. 2016. The Bunker cache for spatio-value approximation. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. IEEE Computer Society, 43:1–43:12. DOI: https://doi.org/10.1109/MICRO.2016.7783746

[74] Somayeh Sardashti, Angelos Arelakis, Per Stenström, and David A. Wood. 2015. A primer on compression in the memory hierarchy. *Synth. Lect. Comput. Architect.* 10, 5 (2015), 1–86. DOI: https://doi.org/10.2200/S00683ED1V01Y201511CAC036

[75] 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'14)*. IEEE Computer Society, 331–342. DOI: https://doi.org/10.1109/MICRO.2014.41

[76] 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. DOI: https://doi.org/10.1145/2976740

[77] 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'13)*. Association for Computing Machinery, 62–73. DOI: https://doi.org/10.1145/2540708.2540715

[78] Kenneth James Schultz, Garnet Frederick Randall Gibson, Farhad Shafai, and Armin George Bluschke. 1999. Content addressable memory. U.S. Patent 5,859,791.

[79] Seok-Won Seong and Prabhat Mishra. 2008. Bitmask-based code compression for embedded systems. *IEEE Trans. Comput.-aided Design Integr. Circ. Syst.* 27, 4 (2008), 673–685.

[80] André Seznec. 1993. A case for two-way skewed-associative caches. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA'93)*. Association for Computing Machinery, 169–178. DOI: https://doi.org/10.1145/165123.165152

[81] A. Seznec. 1994. Decoupled sectored caches: Conciliating low tag implementation cost. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA'94)*. IEEE Computer Society Press, 384–393. DOI: https://doi.org/10.1145/191995.192072

[82] Ali Shafiee, Meysam Taassori, Rajeev Balasubramonian, and Al Davis. 2014. MemZip: Exploring unconventional benefits from memory compression. In *Proceedings of the IEEE 20th International Symposium on High Performance Computer Architecture (HPCA'14)*. IEEE, IEEE Computer Society, 638–649. DOI: https://doi.org/10.1109/HPCA.2014.6835972

[83] Alan Jay Smith. 1982. Cache memories. *ACM Comput. Surv.* 14, 3 (Sept. 1982), 473–530. DOI: https://doi.org/10.1145/356887.356892

[84] James A. Storer and Thomas G. Szymanski. 1982. Data compression via textual substitution. *J. ACM* 29, 4 (Oct. 1982), 928–951. DOI: https://doi.org/10.1145/322344.322346

[85] Martin Thuresson, Lawrence Spracklen, and Per Stenstrom. 2008. Memory-link compression schemes: A value locality perspective. *IEEE Trans. Comput.* 57, 7 (2008), 916–927.

[86] Xinhua Tian and Minxuan Zhang. 2007. A unified compressed cache hierarchy using simple frequent pattern compression and partial cache line prefetching. In *Proceedings of the International Conference on Embedded Software and Systems (Lecture Notes in Computer Science)*, Vol. 4523. Springer, 142–153. DOI: https://doi.org/10.1007/978-3-540-72685-2_14

[87] Yingying Tian, Samira M. Khan, Daniel A. Jiménez, and Gabriel H. Loh. 2014. Last-level cache deduplication. In *Proceedings of the 28th ACM International Conference on Supercomputing (ICS'14)*. Association for Computing Machinery, Munich, Germany, 53–62. DOI: https://doi.org/10.1145/2597652.2597655

[88] R. Brett Tremaine, Peter A. Franaszek, John T. Robinson, Charles O. Schulz, T. Basil Smith, Michael E. Wazlowski, and P. Maurice Bland. 2001. IBM memory expansion technology (MXT). *IBM J. Res. Dev.* 45, 2 (2001), 271–285.

[89] R. Brett Tremaine, T. Basil Smith, Mike Wazlowski, David Har, Kwok-Ken Mak, and Sujith Arramreddy. 2001. Pinnacle: IBM MXT in a memory controller chip. *IEEE Micro* 21, 2 (2001), 56–68.

[90] Po-An Tsai, Yee Ling Gan, and Daniel Sanchez. 2018. Rethinking the memory hierarchy for modern languages. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture* (*MICRO'18*). IEEE Press, 203–216. DOI: https://doi.org/10.1109/MICRO.2018.00025

[91] Po-An Tsai, Andres Sanchez, Christopher W. Fletcher, and Daniel Sanchez. 2020. Safecracker: Leaking secrets through compressed caches. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*. Association for Computing Machinery, 1125–1140. DOI: https://doi.org/10.1145/3373376.3378453

[92] Po-An Tsai and Daniel Sanchez. 2019. Compress objects, not cache lines: An object-based compressed memory hierarchy. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*. ACM, 229–242. DOI: https://doi.org/10.1145/3297858.3304006

[93] Irina Chihaia Tuduce and Thomas R. Gross. 2005. Adaptive main memory compression. In *Proceedings of the USENIX Annual Technical Conference*. USENIX, 237–250. Retrieved from http://www.usenix.org/events/usenix05/tech/general/tuduce.html.

[94] J. Uthayakumar, T. Vengattaraman, and P. Dhavachelvan. 2018. A survey on data compression techniques: From the perspective of data quality, coding schemes, data type and applications. *J. King Saud Univ.-Comput. Info. Sci.* 33, 2 (2021), 1319–1578. DOI: https://doi.org/10.1016/j.jksuci.2018.05.006

[95] Luis Villa, Michael Zhang, and Krste Asanović. 2000. Dynamic zero compression for cache energy reduction. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO'00)*. Association for Computing Machinery, 214–220. DOI: https://doi.org/10.1145/360128.360150

[96] Hong Wang, Tong Sun, and Qing Yang. 1995. CAT—Caching address tags: A technique for reducing area cost of on-chip caches. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA'95)*. Association for Computing Machinery, 381–390. DOI: https://doi.org/10.1145/223982.224448

[97] Terry A. Welch. 1984. A technique for high-performance data compression. *Computer* 6, 17 (1984), 8–19.

[98] Paul R. Wilson, Scott F. Kaplan, and Yannis Smaragdakis. 1999. The case for compressed caching in virtual memory systems. In *Proceedings of the USENIX Annual Technical Conference*. USENIX, 101–116. Retrieved from http://www.usenix.org/events/usenix99/full_papers/wilson/wilson.pdf.

[99] Yuejian Xie and Gabriel H. Loh. 2011. Thread-aware dynamic shared cache compression in multi-core processors. In *Proceedings of the IEEE 29th International Conference on Computer Design (ICCD'11)*. IEEE Computer Society, 135–141. DOI: https://doi.org/10.1109/ICCD.2011.6081388

[100] Chao Yan and Russ Joseph. 2018. Cocoa: Synergistic cache CoMpression and error CoRrection in CaPacity sensitive last level caches. In *Proceedings of the International Symposium on Memory Systems (MEMSYS'18)*. Association for Computing Machinery, 117–128. DOI: https://doi.org/10.1145/3240302.3240304

[101] Jun Yang and Rajiv Gupta. 2002. Energy efficient frequent value data cache design. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO'02)*. IEEE Computer Society Press, 197–207. Retrieved from http://dl.acm.org/citation.cfm?id=774861.774883.

[102] Jun Yang, Rajiv Gupta, and Chuanjun Zhang. 2004. Frequent value encoding for low power data buses. *ACM Trans. Des. Autom. Electron. Syst.* 9, 3 (July 2004), 354–384. DOI: https://doi.org/10.1145/1013948.1013953

[103] Jun Yang, Youtao Zhang, and Rajiv Gupta. 2000. Frequent value compression in data caches. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 33)*. Association for Computing Machinery, Monterey, 258–265. DOI: https://doi.org/10.1145/360128.360154

[104] Keun Soo Yim, Jang-Soo Lee, Jihong Kim, Shin-Dug Kim, and Kern Koh. 2004. A space-efficient on-chip compressed cache organization for high performance computing. In *Proceedings of the 2nd International Conference on Parallel and Distributed Processing and Applications (ISPA'04)*. Springer-Verlag, 952–964. DOI: https://doi.org/10.1007/978-3-540-30566-8_109

[105] Vinson Young, Prashant J. Nair, and Moinuddin K. Qureshi. 2017. DICE: Compressing DRAM caches for bandwidth and capacity. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*. Association for Computing Machinery, 627–638. DOI: https://doi.org/10.1145/3079856.3080243

[106] Qi Zeng, Rakesh Jha, Shigang Chen, and Jih-Kwon Peir. 2018. Data locality exploitation in cache compression. In *Proceedings of the IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS'18)*. IEEE, 347–354. DOI: https://doi.org/10.1109/PADSW.2018.8644558

[107] Youtao Zhang, Jun Yang, and Rajiv Gupta. 2000. Frequent value locality and value-centric data cache design. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'00)*. Association for Computing Machinery, 150–159. DOI: https://doi.org/10.1145/378993.379235

[108] Jacob Ziv and Abraham Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Trans. Info. Theory* 23, 3 (1977), 337–343.

[109] Jacob Ziv and Abraham Lempel. 1978. Compression of individual sequences via variable-rate coding. *IEEE Trans. Info. Theory* 24, 5 (1978), 530–536.

Received September 2020; revised February 2021; accepted March 2021