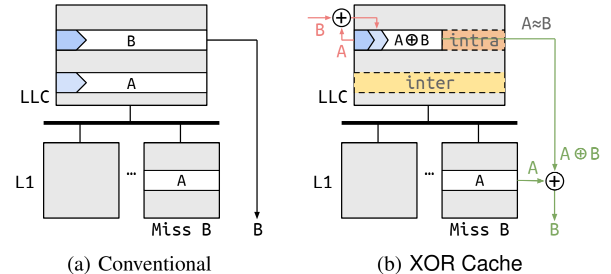
The XOR Cache: A Catalyst for Compression

Abstract—**Modern computing systems allocate significant amounts of resources for caching, especially for the last level cache (LLC). We observe that there is untapped potential for compression by leveraging redundancy due to private caching and inclusion that are common in today's systems. We introduce the XOR Cache to exploit this redundancy via XOR compression. Unlike conventional cache architectures, XOR Cache stores bitwise XOR values of line pairs, halving the number of stored lines via a form of inter-line compression. When combined with other compression schemes, XOR Cache can further boost intra-line compression ratios by XORing lines of similar value, reducing the entropy of the data prior to compression. Evaluation results show that the XOR Cache can save LLC area by $1.32\times$ and power by $1.67\times$ at a cost of 3.58% performance overhead compared to a $2\times$ larger uncompressed cache.**

I. Problem and Motivation

Today's computing systems dedicate tens to hundreds of megabytes of SRAM to caching, which contributes to a significant portion of die area, e.g, AMD's Zen3's 32 MB L3 cache occupies around 40% of die area [8]. Additionally, the power consumption of these systems also surges, further straining the overall energy efficiency. The demand for resources in the cache hierarchy will continue to increase due to the growth in dataset size and memory wall problem. However, large caches do not necessarily translate into better performance despite having more capacity; additionally, they come at the cost of high access latency, usually in tens of cycles. Given their resource-demanding nature, these factors combined make traditional large caches inefficient for future systems. To bridge this efficiency gap, computer architects continuously seek better ways to optimize the cache hierarchy by reducing the cache footprint yet maintaining performance. Cache compression [6], [10] is one of the promising lines of research. Caches can store compressed data in cache lines upon insertion and decompress them upon access. This approach reduces the cache footprint with the overhead of only a few extra cycles added to access latency. Effective cache compression exploits data redundancy to maintain the benefits of a larger cache but with significantly reduced hardware costs.

We introduce the XOR Cache, a new compressed LLC architecture that leverages redundancy that spans multiple caches, inherent to the memory hierarchies of today's systems. More specifically, XOR Cache harnesses **redundancy due to inclusion and private caching**. First, while prior cache compression paradigms only exploit value redundancy within a single cache level for compression opportunity, XOR Cache focuses on taming redundancy due to inclusion between the upper and lower level cache, crossing the boundary of cache level. It has been shown that inclusive cache hierarchies can introduce a significant amount of data redundancy, thereby decreasing the effective capacity of the cache at the higher level[1] [5]. Unfortunately, this data duplication has been overlooked, beyond the trivial solution of relaxing the strictly-inclusive property. To bridge this gap, XOR Cache transforms what was once considered a drawback—redundancy due to inclusion—into untapped compressibility. Second, to implement its compression and decompression scheme, XOR Cache leverages redundancy due to private caching to achieve data forwarding between the private caches.



(a) Conventional (b) XOR Cache

Fig. 1: High-level overview. Unlike a conventional cache, XOR Cache stores the bitwise XOR of line pairs.

Fig. 1 shows a high-level overview of the XOR Cache. Unlike a conventional cache that stores lines as-is, in XOR Cache, an inclusive cache line, e.g., line A, since it already exists in the L1s, can XOR with another, e.g., line B, in the higher level as a single line A⊕B, denoted by the pink arrows. Upon access, we can simply forward the XORed line A⊕B to the lower level and perform another XOR operation to reverse the compression, denoted by the green arrows. This provides two novel benefits: **1) XORing two lines allows us to save one line of storage in the LLC; and 2) when the two lines have similar data values, XORing them reduces entropy, making them more compressible and catalyzing the effectiveness of other compression schemes.**

A. Redundancy in Cache Hierarchy (Inter-Line Compression)

Unless made strictly exclusive, the LLC typically contains either all, i.e., an inclusive LLC, or some of the cache lines, i.e., a non-inclusive, non-exclusive (NINE) LLC, that exist in the lower-level caches. This duplication is a missing piece in conventional cache compression works, as they typically only exploit redundancy within a single cache level. However, it can create extra opportunities for compression across the cache level boundary. To exploit such opportunities, upon insertion, XOR Cache performs a bitwise XOR between the inserted line and another selected line and stores the result in the LLC data array. By doing so, it effectively co-locates two cache

[1] We refer to the cache further from the processors as the higher level cache and vice versa.

lines in one physical slot, resulting in a 2:1 compression ratio. A compressed line pair can stay compressed as long as one of the original lines is shared at the lower level to maintain the ability to recover, which we call the minimum sharer invariant. Leveraging XOR compression alone, we can achieve a best-case compression ratio of 2, allowing us to downsize the LLC data array by half. We denote this as a form of **inter-line compression**, labeled as *inter* in Fig. 1b.

B. Synergy of XOR Compression (Intra-Line Compression)

XOR compression can work independently from prior cache compression schemes; however, synergy exists between it and other schemes when we carefully select the XOR candidate lines. When combined, the XOR Cache can boost (*catalyze*) the compression ratios of other schemes. For example, in Fig. 1b, when the selected line A is similar to B, i.e., A≈B, the XORed line A⊕B can exhibit lower entropy and be further compressed. This compression ratio boost is achieved by exploiting the similarity within the XORed lines, denoted as a form of **intra-line compression**, labeled as *intra* in the figure. For each compression scheme combined with XOR, an ideal XOR policy exists that maximizes the compression ratio. The XOR pair selection policy, which we refer to as XOR policy for short, determines which two cache lines should be XORed with each other, which is the crux of XOR Cache's design space. We quantify this synergy by profiling the LLC values, as shown in Fig. 2. As examples, we show the potential of XOR Cache's synergy with BΔI [10] and Thesaurus [6], which are state-of-the-art cache compression schemes. The *idealBank* XOR Cache, which searches the best candidates across the entire LLC bank, can boost the baselines by $2.08\times$ and $2.02\times$ on average and up to $4.7\times$ and $4.6\times$, demonstrating XOR Cache's potential for catalyzing compression.

II. Background and Related Works

Abundant works in cache compression [6], [10] have demonstrated success in increasing the effective storage capacity. A compressed cache exploits data redundancy to maintain the benefits of a larger cache but with significantly reduced hardware costs. To implement cache compression, multiple tag entries can map to the same data entry. The common practice is to adopt a decoupled tag data array structure, which can either be a fixed many-to-one mapped or a flexible doubly linked list. Compression algorithms can be classified into intra- and inter-line based on the compression granularity. Intra-line compression captures value similarity within a single memory or cache line. [10]. These works are typically dictionary-based, matching against predefined common values or patterns at sub-line granularity. Inter-line works [6] compress multiple similar lines together and store only one copy of the line, along with additional metadata if needed.

III. Uniqueness of the Approach

A. XOR Compression

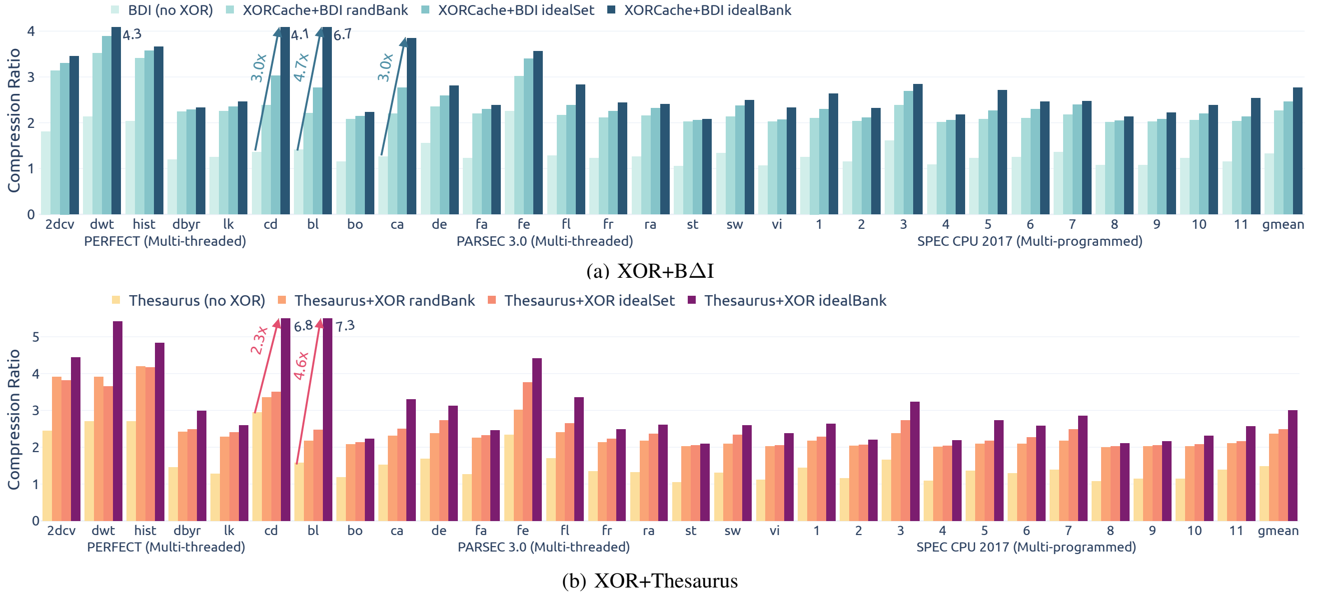
1) Compression and Decompression Algorithm: XOR compression employs simple and symmetric compression and decompression algorithms. XOR Cache stores the bitwise XOR results of line pairs. Upon access, the XORed line performs another bitwise XOR operation with one of the two original lines. Therefore, compression and decompression are perfectly symmetrical since XOR with a given input is a self-inverse function. Additionally, the compressor and decompressor hardware is extremely simple. They are simply an array of XOR gates of length 512, assuming 64B cache lines. Given that they are simple bitwise operations, they can be embedded in the cache controller or even closer to the SRAM cells.

2) XOR Policy: We can adopt an **opportunistic** XOR policy by allowing XOR compression whenever any candidate, i.e., a standalone line, is available. This maximizes the inter-line compression ratio from XOR compression. Alternatively, we can be more selective about performing XOR compression by adopting a **synergistic** XOR policy. As mentioned earlier in Section I-B, if similar lines are XORed together, the resultant $A \oplus B$ line is likely to exhibit lower entropy and contain many zeros, thereby enabling further intra-line compression. We can see this by examining a concrete example of two lines from the bodytrack benchmark in Fig. 3. They are very similar, with only a few bit differences, i.e., low in hamming distance. Individually, they have limited intra-line compressibility, but when XORed as a pair, the entropy of the XORed line can be significantly reduced. However, the key challenge is identifying line pairs that are similar to each other. As a practical implementation, we consider a map table-based synergistic XOR policy for finding similar lines within a bank using manageable hardware complexity. A map function is applied on the cache line to generate a map value, which serves as a signature of line bits and is used as an index to the map table. More details will be discussed in detail in Section III-C1.

B. XOR Cache Coherence

We present XOR Cache's protocol based on MSI. Fig. 4 shows the transition between stable states. In addition to stable states, we denote another state Shared0, which is specific scenario of Shared when the line has no sharers in the private cache, for clarity. Our implementation assumes a general NINE cache hierarchy, where inclusion is maintained for clean lines, and exclusion is enforced for dirty lines since their owner is in the lower level, and the upper level does not directly service requests on dirty lines. Note that this does not impose considerably more transient states, as Shared0 is a state that already exists in MSI; we simply highlight it for clarity. The directory tracks accurate sharer information requiring explicit eviction and upgrade notification. In our work, we do not assume support for silent upgrades and thus do not consider the Exclusive state.

1) Expansion: XOR Cache needs to perform expansion, i.e., unXOR a pair, in the following three cases before any of the lines in the pair goes into an unrecoverable state. First, when a line in an XORed pair is **upgrading to Modified**, denoted by solid blue arrows on *getM* requests, the XORed pair needs to expand. Second, expansion is also needed on transitions from Shared to Shared0 state on **the last *putS* request**, i.e., dotted blue arrow, required by the minimum



(a) XOR+BΔI

(b) XOR+Thesaurus

Fig. 2: Compression ratio from LLC profiling. *randBank* randomly XORs two lines from the same bank. *idealSet* and *idealBank* search the entire set and bank to find the best candidate that minimizes data storage.

```
Line A     0020 003C 6D7F 0000 7C20 003C 6D7F 0000 7C20 003C …
Line B     0020 004C 6D7F 0000 7C20 004C 6D7F 0000 7C20 004C …
Line A⊕B   0000 0070 0000 0000 0000 0070 0000 0000 0000 0070 …
```

Fig. 3: Two similar lines A and B from bodytrack benchmark in PAESEC3.0 suite. The XORed line A⊕B has low entropy.

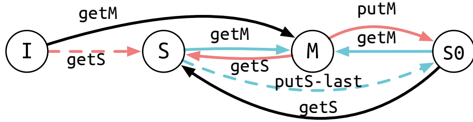


Fig. 4: Transitions between stable states. I for Invalid; S for Shared; M for Modified; S0 is a special Shared state when the number of sharers is zero. Pink and blue arrows denote transitions that can possibly result in XOR compression and require XOR expansion, respectively.

sharer invariant. Third, **dirty replacement in the upper level**, not shown in Fig. 4 for simplicity, also needs expansion to recover the original data and then write back to memory.

2) Decompression: When a data request arrives at the lower level for an XORed line, XOR Cache needs to decompress the line to service the request. To decompress, XOR Cache may need to forward requests to the lower level. There are three data forwarding cases, and the sequence diagrams for handling each case are shown in Fig. 5. We continue with the same scenario as in Fig. 1 where we have a miss on line B while the LLC holds A⊕B. In the first case where A has at least one sharer, and B's requestor happens to be one of them, the LLC forwards the data A⊕B to B's requestor. The requestor then performs another bitwise XOR operation on A⊕B and its local copy of A to retrieve the demand data B. We name this case **local recovery**. In the second case, where the requestor of B does not share A, but line B has at least one sharer in the lower level, LLC forwards the request for B to one of B's sharers and it then supplies the data to B's requestor. We call this case **direct forwarding**. In the third case, when the requestor of B similarly does not share A, and B no longer has any sharer, A must have at least one sharer in the lower level by design, so the LLC forwards the request for B along with the data A⊕B to A's sharer. A's sharer reads out its local copy of A and performs another bitwise XOR operation between A⊕B and A to retrieve B remotely, then send it back to B's requestor. We name this case **remote recovery.**

C. XOR Cache Architecture

1) Organization: To allow two arbitrary lines in a bank to be XORed, XOR Cache adopts a flexible decoupled tag-data organization and uses a map table to identify XOR candidates, as shown in Fig. 6a. **The tag array** is managed as a linked list with each tag entry shown in Fig. 6b. *XORed* is a 1-bit indicator of whether the line is XORed with a partner. *XORptr* points to the tag entry of its partner. *DataPtr* points to the corresponding entry in the data array. **The data array** can have an arbitrary number of sets. As shown in Fig. 6c, the data array entry stores a reverse pointer to the tag array entry, i.e., *tagptr*. We use a random replacement policy for the data array. **The map table**, at a high level, is a small storage structure for identifying similar XOR candidates. As shown in Fig. 6a, it contains tag pointers of all the standalone lines. It is indexed using a map value generated by applying a map function on the data. On cache line insertion, it first computes the map value and index the map table. If it hits, meaning a valid XOR candidate exists, then XOR compression is triggered, and the entry is cleared. Otherwise, the line is

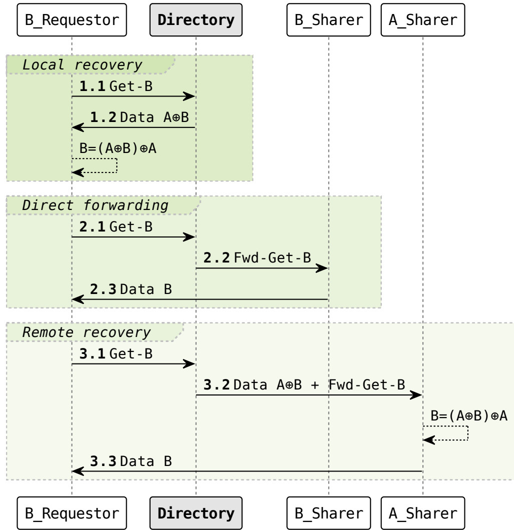


Fig. 5: Three forwarding cases when A and B are XORed. From top to bottom are local recovery, direct forwarding, and remote recovery.

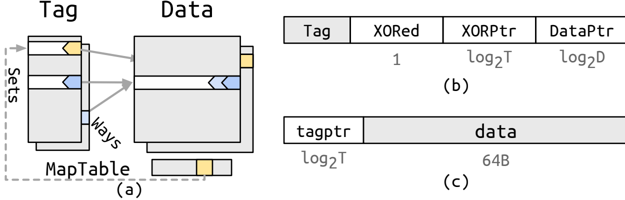


Fig. 6: XOR Cache organization. Grey blocks are identical to the uncompressed baseline; T is the number of tag entries; D is the number of data entries.

inserted as uncompressed, and the map table allocates an entry. We will discuss XOR Cache's insertion flow in detail in Section III-C2. We consider four hash functions as our **map function** candidates. Two are locality-sensitive hash functions based on random projection (LSH-RP) similar to [6] and bit sampling (LSH-BS). The other two are based on byte labeling. For every byte in the cache line, a boolean label 0 is generated if the byte is 0x0, or 1 otherwise, effectively capturing byte-level sparsity. For baseline byte labeling (BL), the generated byte labels are first permuted and then XOR folded into the map value. Sparse byte labeling (SBL) only considers a subset of bytes in the line, i.e., the most significant 6 bytes per every 8-byte word. We compare them in Section IV-B.

2) Operations: We only highlight XOR Cache's insertion operation flow for brevity. Upon data insertion, XOR Cache accesses the map table to find an available candidate to co-locate with. This insertion flow is off the critical path. The map value is calculated by applying the map function to the returned data and it is used to index into the map table. On a map table hit, it reads the tag pointer and then the data, performs bitwise XOR, and inserts the XORed data into the data array to replace the original data. Conversely, on map table miss, the tag pointer is inserted into the map table, data is inserted into the data array, and the tag is updated.

IV. Results

A. Methodology

We simulate XOR Cache with its coherence protocol using the Ruby model in the gem5 simulator [7] and uses full-system simulation. We use CACTI [2] for memory evaluation and Synopsys design compiler for hardware synthesis using 32nm technology. Table I lists the simulated hardware system configuration for uncompressed baseline. Our upper-lower cache size ratio reflects Skylake [9]. We shrink the data array size for all compressed caches by a factor based on our profiling results in Fig. 2 and Section IV-B. We assume 1 cycle latency for BΔI on the critical path. We assume XOR decompression is within the same cycle, as it only incurs 0.12 ns delay. We include the PERFECT [3] openmp version and PARSEC 3.0 [4] for **multi-threaded** workloads and SPEC CPU 2017 [1] reference workloads for **multi-programmed** evaluation.

TABLE I: Hardware system configuration.

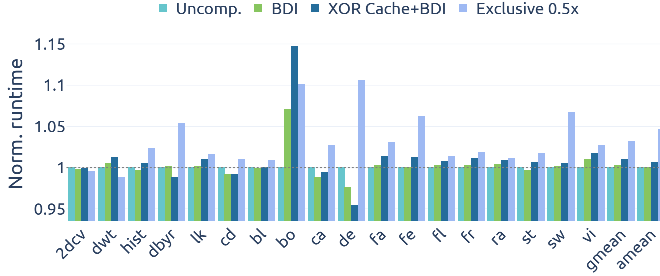
Name	Baseline Configuration
CPU	4 core, 3GHz x86-64
L1I	16KiB, 4 way, 2 cycle, 64B line, LRU
L1D	32KiB, 4 way, 2 cycle, 64B line, LRU
Dir	256 set, 8 way, 2 cycle, LRU
L2 LLC	64KiB per bank, 8 way, 4 cycle, 64B line, LRU, 4 banks, non-inclusive
Memory	DualChannelDDR4-2400, 3GiB

B. XOR Compression Synergy

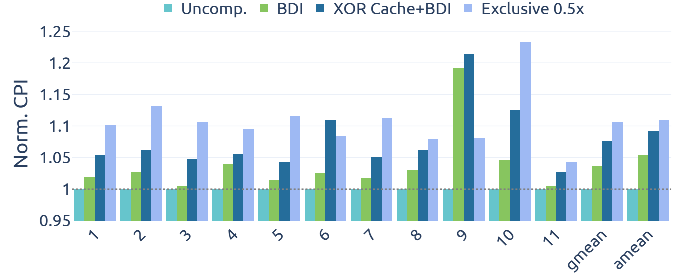
In our map table-based implementation with a synergistic XOR policy, lines are only XORed upon map table hits. We profile the inter-line and intra-line compression ratios of the four map functions as shown in Fig. 8. The X-axis shows the number of map value bits, and the Y-axis shows compression ratio. As shown in Fig. 8a, XOR compression coverage as the number of map value bits increases as the number of unique map values increases exponentially and the probability of finding a candidate in the matching bin decreases. However, as shown in Fig. 8b, the accuracy of identifying similar candidate lines improves as intra-line compression ratio increases due to the increased value similarity between the XORed line pairs. LSH-BS takes more than 30 bits to achieve intra-line compression ratio synergy, whereas LSH-RP needs 12 bits. BL and SBL both achieve similar intra-line compression ratios; however, SBL maintains a higher inter-line compression ratio since it effectively removes the noise from high entropy bits in words. We use 7-bit SBL, which balances inter- and intra-line compression and results in an average compression ratio of ~ 2.5, justifying our choice of using a $2.5\times$ smaller data array for XOR Cache with BΔI.

C. Compression Ratio Analysis

XOR Cache's compression ratio is less than the upper bound in the practical setting for the existence of Modified lines and extensive sharing between cores. Regardless, opportunistic XOR Cache achieves an inter-line compression ratio of 1.58. Combined with its intra-line component, synergistic XOR Cache achieves a compression ratio of 1.84, $1.35\times$ greater than BΔI.



(a) Performance overhead. (PERFECT, PARSEC)



(b) Performance overhead. (SPEC)

Fig. 7: Performance analysis. (a) shows normalized runtime of multi-threaded benchmarks; (b) shows geometric mean of CPI of multi-programmed benchmarks.

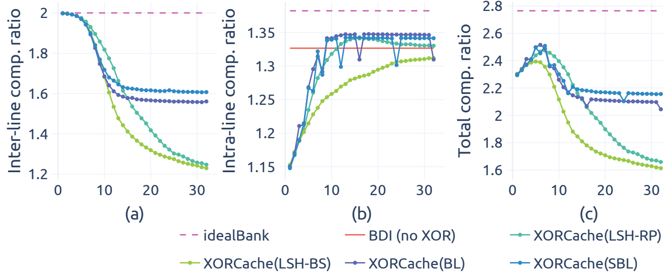


Fig. 8: Compression ratio with for four map functions.

D. Area and Power Analysis

XOR Cache achieves $1.32\times$ and $1.31\times$ smaller area than the uncompressed and BΔI counterparts, respectively. In spire of the additional activity, XOR Cache still achieves a significant $1.67\times$ and $1.47\times$ LLC total power reduction and $1.23\times$ and $1.15\times$ cache hierarchy power reduction compared to the uncompressed cache and BΔI baselines, respectively.

E. Performance Analysis

We compare XOR Cache's performance overhead against three baselines: the uncompressed NINE LLC, the BΔI LLC, shown in Fig. 7. In summary, across all benchmarks, with increased LLC misses and hit latency, XOR Cache adds a performance overhead of 3.58% and 1.91% compared to the uncompressed baseline and BΔI, respectively.

V. Contributions

In this work, we introduce the XOR Cache, which exploits data redundancy due to inclusion and private caching to perform effective inter-line and intra-line compression simultaneously. Unlike conventional caches, XOR Cache stores bitwise XOR results of line pairs, resulting in a 2:1 compression ratio. When combined with other compression schemes, XOR Cache can further boost compression ratio by reducing entropy of the data values. By exploring the design space of XOR Cache, our work makes the following contributions:

1) We present a novel cache compression scheme that harnesses redundancy due to inclusion and private caching, which are understudied in prior works.
2) We show that XOR compression can synergistically boost the compression ratio when combined with other cache compression schemes.
3) We implement XOR Cache using the Ruby memory model in the gem5 simulator [7] and evaluate it in full-system simulation on multiple benchmark suites.

Evaluation results show that it reduces LLC area by $1.32\times$ and power by $1.67\times$ with comparable performance (average overhead of 3.58%) to a $2\times$ larger uncompressed cache.

References

[1] "SPEC CPU® 2017," https://www.spec.org/cpu2017/.
[2] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 2, jun 2017.
[3] K. Barker, T. Benson, D. Campbell, D. Ediger, R. Gioiosa, A. Hoisie, D. Kerbyson, J. Manzano, A. Marquez, L. Song, N. Tallent, and A. Tumeo, *PERFECT (Power Efficiency Revolution For Embedded Computing Technologies) Benchmark Suite Manual*, Pacific Northwest National Laboratory and Georgia Tech Research Institute, December 2013, http://hpc.pnnl.gov/projects/PERFECT/.
[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *International Conference on Parallel Architectures and Compilation Techniques*, 2008.
[5] M. Chaudhuri, "Zero inclusion victim: Isolating core caches from inclusive last-level cache evictions," in *International Symposium on Computer Architecture*, 2021.
[6] A. Ghasemazar, P. Nair, and M. Lis, "Thesaurus: Efficient cache compression via dynamic clustering," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 527–540. [Online]. Available: https://doi.org/10.1145/3373376.3378518
[7] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj *et al.*, "The gem5 simulator: Version 20.0+," *arXiv preprint arXiv:2007.03152*, 2020.
[8] H. Mujtaba. (2020) Amd ryzen 5000 zen 3 'vermeer' undressed, first ever high-res die shots close ups pictured & detailed. [Online]. Available: https://wccftech.com/amd-ryzen-5000-zen-3-vermeer-undressed-high-res-die-shots-close-ups-pictured-detailed/
[9] D. L. Mulnix. (2022) Intel xeon processor scalable family technical overview. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html
[10] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-delta-immediate compression: practical data compression for on-chip caches," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 377–388. [Online]. Available: https://doi.org/10.1145/2370816.2370870