The XOR Cache: A Catalyst for Compression

Zhewen Pan
zhewen.pan@wisc.edu
University of Wisconsin–Madison
Madison, WI, USA

Joshua San Miguel
jsanmiguel@wisc.edu
University of Wisconsin–Madison
Madison, WI, USA

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 1.93× and power by 1.92× at a cost of 2.06% performance overhead compared to a larger uncompressed cache, reducing energy-delay product by 26.3%.

CCS Concepts

• **Computer systems organization** → **Multicore architectures**.

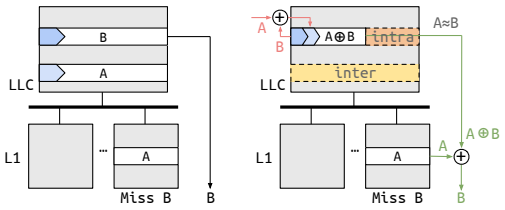
Keywords

Cache hierarchy, Cache compression, Low-power architecture

ACM Reference Format:
Zhewen Pan and Joshua San Miguel. 2025. The XOR Cache: A Catalyst for Compression. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3695053.3730995

1 Introduction

Today's computing systems dedicate tens to hundreds [39] 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 [38]. 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 the 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 can make traditional large caches potentially inefficient for future systems. To bridge this efficiency gap, we seek better ways to optimize the cache hierarchy by reducing the cache footprint and power while maintaining performance.



(a) Conventional (b) XOR Cache

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

Cache compression [3, 8–10, 16, 24, 41, 42, 45–47, 49, 51, 54] is one of the promising lines of research. Caches can compress 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 compressibility to achieve 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 higher and lower level cache. It has been shown that inclusive cache hierarchies can introduce a significant amount of data redundancy, thereby decreasing the effective capacity of the lower-level cache[1] [15, 26, 37, 40, 48, 55]. 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, XOR Cache leverages redundancy due to private caching to decompress data via forwarding between the private caches, supported by its coherence protocol. XOR Cache achieves efficient compression by exploiting both forms of redundancy.

Figure 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 lower level as a single line A⊕B, denoted by the pink arrows. On an LLC access, we can simply forward the XORed line A⊕B to the higher level and perform another XOR operation to reverse the compression, denoted by the green arrows. Note that lines in L1s are never XORed, so L1 hits do not impose extra latency. Doing so provides two novel benefits:

[1]We refer to the cache further from the processors as the lower-level cache.

ISCA '25, Tokyo, Japan

ACM ISBN 979-8-4007-1261-6/2025/06
https://doi.org/10.1145/3695053.3730995

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. **The goal of this work is to leverage the enhanced compressibility enabled by XOR Cache to reduce area and power of the LLC.**

1.1 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 higher-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 lines in one physical slot, resulting in a 2:1 compression ratio. A compressed line pair can stay compressed when at least one of the original lines is shared at the higher 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** (*inter* in Figure 1b).

1.2 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 Figure 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. With XOR compression, each pair of cache lines is first XORed, and then we can apply the baseline existing compression schemes on the XORed data.

We quantify this synergy by profiling the LLC snapshots when running a wide range of benchmarks, as shown in Figure 2. For this analysis, we assume any two lines in the same bank or in the same set can potentially be XORed without imposing the minimum sharer invariant (Section 1.1). As examples, we show the potential of XOR Cache's synergy with BΔI [45] and Bit-plane (BPC) [30] compression, which are intra-line cache compression schemes, and Thesaurus [24] as an inter-line compression scheme. BΔI [45] exploits the low dynamic range of values within a cache line and encodes a line using a single base value and an array of deltas with smaller sizes. BPC [30] applies three transformations to boost data compressibility and uses run-length encoding and frequent pattern compression to compress the transformed data. Thesaurus [24] dynamically clusters cache lines using locality-sensitive hashing and compress lines against the centroids.

The XOR pair selection policy, or 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. Let's consider three hypothetical XOR policies, namely *randBank*, *idealSet*, and *idealBank*. For *randBank*, a line randomly selects and XORs with another line in the same bank. For *idealSet*, a line considers all the lines in the same **set** as XOR candidates and selects the most ideal XOR candidate that results in the most compressible XORed line. For example, in Figure 3, the line at set 0 and way 0 can potentially XOR with any of the lines in the same set, i.e., in the blue box. Here, the ideal XOR candidate in set 0 is the line at way 2, since they only have one bit difference. Similarly, for *idealBank*, a line considers all the lines in the same **bank** and XORs with the one that results in the smallest size. In Figure 3, the line at set 0 and way 0 can XOR with anyone in the orange box. The line at set 1 and way 3 is the most ideal as it has exactly the same value. Shown in Figure 2, the leftmost sets of bars are the baseline compression ratio without XOR; the other three sets of bars show the compression ratio of XOR Cache using the three aforementioned XOR policies. The lines are first XORed with each other, and then the selected baseline compression scheme further compresses the XORed line.

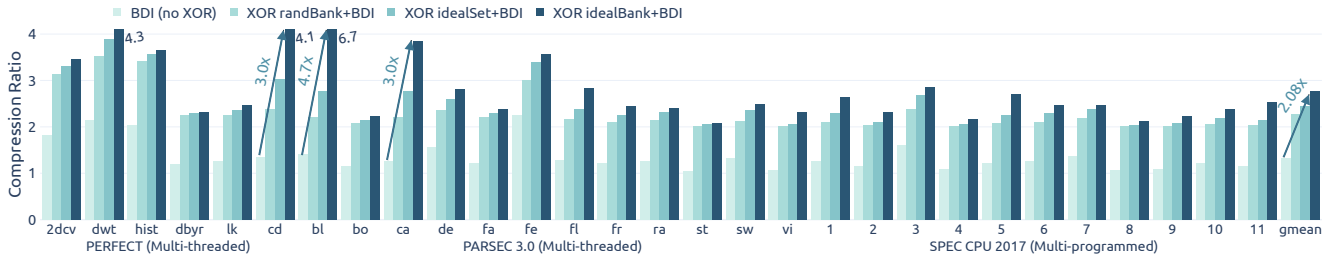
We make the following observations. First, the two search-based XOR policies, i.e., *idealBank* and *idealSet*, achieve higher compression ratios than the random XOR policy, which is value-agnostic. This demonstrates the importance of considering the value similarity when choosing the XOR candidates. Second, *idealBank* outperforms *idealSet* because it has a significantly larger search scope, and is therefore more likely to find ideal XOR candidates. However, the implementation cost of *idealBank* can also significantly outweigh that of *idealSet*. Though the search-based policies by no means represent practical implementations (we discuss practical implementation later in Section 3.2), it showcases the upper bound compression ratio of XOR Cache. More specifically, **the *idealBank* XOR compression, which searches for the best candidates across the entire LLC bank, can boost compression ratio of BΔI, BPC and Thesaurus by** 2.08×**,** 2.09× **and** 2.02× **on average and up to** 4.7×**,** 3.0× **and** 4.6×**, respectively, demonstrating XOR Cache's potential for catalyzing compression.**

1.3 Challenges

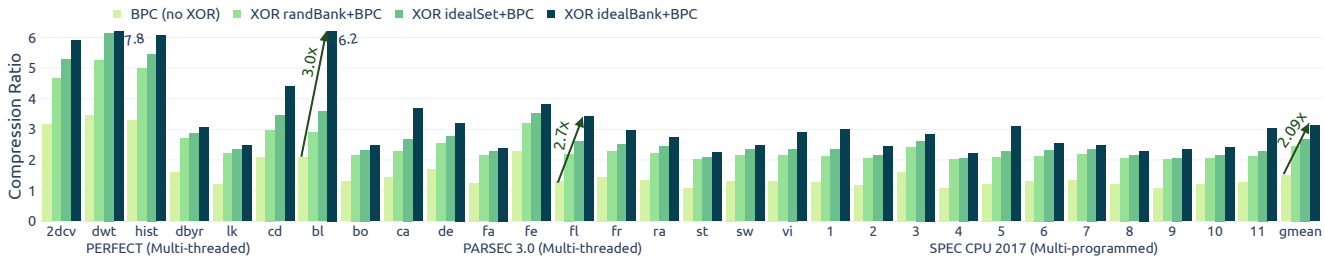
There are two main challenges that must be addressed to unlock the full potential of XOR Cache. The first challenge lies in **designing the XOR policy that achieves synergy**. There is ample design space, ranging from random to value-aware, each with its own trade-offs between complexity, effectiveness, and performance. The second involves the **coherence protocol design**. Unlike prior works, XOR compression crosses the boundary of a single cache level; therefore, the coherence protocol needs a significant redesign. Besides maintaining coherence, the protocol needs to ensure that uncompressed data values are always recoverable.

This work proposes XOR Cache, a compressed LLC architecture that allows aggressive data array downsizing, while maintaining performance. By exploring the design space of XOR Cache, we show how the synergy between XOR and other schemes can boost the compression ratio. This work makes the following contributions:

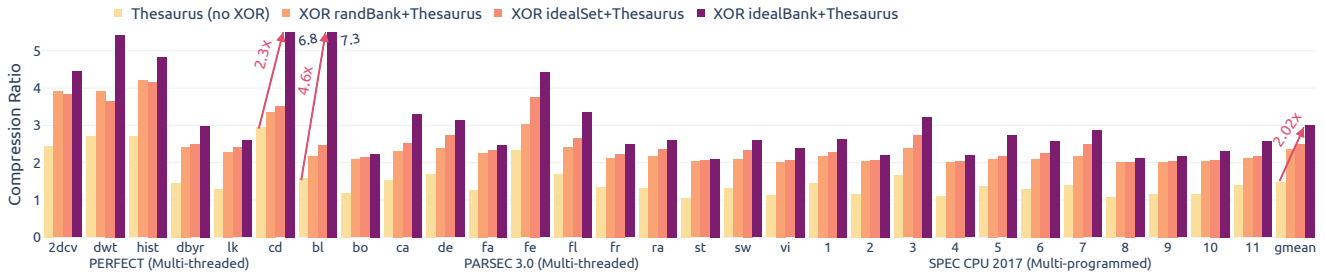
(1) We present a novel cache compression scheme that harnesses redundancy due to inclusion and private caching, which are understudied in prior works.



(a) XOR+BΔI



(b) XOR+BPC



(c) XOR+Thesaurus

Figure 2: Compression ratio from LLC profiling. (a) shows compression ratio of XOR with BΔI; (b) shows compression ratio of XOR with BPC; (c) shows compression ratio of XOR with Thesaurus. A cache line can randomly XOR with another from the same bank (*randBank*), or search the entire set/bank to find the best candidate that minimizes data storage (*idealSet*/*idealBank*).

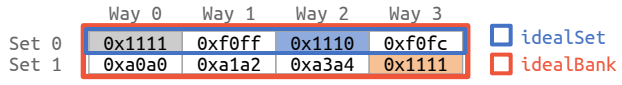


Figure 3: *IdealSet* and *idealBank* XOR policy. The example LLC bank is 4-way set associative with 2 sets. The boxes denote search scopes; the shaded lines represent the ideal XOR candidates for the line at set 0 and way 0.

(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 [36] and evaluate it in full-system simulation on multiple benchmark suites.

The remainder of the paper is organized as follows. Section 2 provides background on cache concepts and summarizes cache compression research. Section 3 describes XOR compression and XOR policy. Sections 4 and 5 discuss XOR Cache's coherence protocol support for data forwarding and architecture implementation. Section 6 presents our evaluation methodology and experimental results. Section 7 discusses related work, and Section 8 concludes.

2 Background

We provide background on inclusion policy in modern cache hierarchies and cache coherence protocol design choices, followed by a summary of low-power and compressed cache works.

2.1 Modern Cache Hierarchy

Cache hierarchies are typically classified into inclusive, NINE, and exclusive, each with its own pros and cons [6, 15, 26, 37, 40, 48, 55]. In an inclusive cache hierarchy, all the lines in the higher level caches must remain a subset of lines in the lower level cache. In contrast, the lower level cache in the exclusive cache hierarchy cannot contain any line at the higher level. The NINE hierarchy sits in between, where a line in the higher level may or may not appear in the lower level. Conventionally, the inclusive hierarchy is widely adopted due to its low required snooping bandwidth and low complexity in the coherence protocol naturally provided by inclusion. However, maintaining inclusion poses two main challenges: back invalidation and low effective capacity. While several works include proposed solutions to combat the former [15, 26], the

latter has received less attention. A strictly inclusive cache usually adopts the trivial solution of keeping the higher level cache small and the lower level large to limit the amount of data duplication. More recently, the trend has been moving towards NINE cache hierarchies [4, 6, 25, 31, 53, 56], which is free of back invalidation problems and has more flexibility in sizing. Regardless, both inclusive and NINE cache hierarchies inevitably contain redundant cache lines due to inclusion, causing the effective capacity to shrink.

2.2 Cache Coherence

This section summarizes the design choices in conventional cache coherence relevant to the XOR Cache's functionality.

2.2.1 Eviction Notification. An LLC shared by all cores is usually the point of coherence in the system. Therefore, it needs a directory structure for storing coherence states, e.g., sharer list and owner of lines cached below. The most critical information a directory tracks is the sharer list. It is a per-line list of higher level cache IDs where the line exists. A higher level cache is added to the sharer list on read requests and removed from the list on eviction notifications. This sharer list can be imprecise for the following reasons, depending on the coherence protocol and directory organization implementation. First, some implementations adopt an incomplete directory, e.g., limited pointer or coarse bit vector. These directories only track a subset of sharers for scalability reasons [40]. Second, imprecision can also stem from the coherence protocol. On clean evictions, the higher level cache may support silent eviction, i.e., opt out of eviction notification and silently drop the clean line to avoid excess communication on eviction. Note this imprecise sharer information does not compromise correctness as the list may contain false positive sharers but never false negatives. However, it does have performance and cost implications as unnecessary invalidation requests and acknowledgments are sent out to and from the false positive sharers [22]. In XOR Cache, to ensure recoverability, a cache line pair can remain compressed if at least one of the two lines is shared. We call this *the minimum sharer invariant*. Therefore, we adopt a full bit vector directory implementation, and our cache coherence protocol sends explicit notifications on clean evictions.

2.2.2 Upgrade Notification. Some protocols also opt out of upgrade notifications with the `Exclusive` state. A uniquely shared line can be directly promoted to `Exclusive` upon reading. When the owner later modifies the line, it can silently upgrade it to `Modified` state, as it is guaranteed that no other sharer exists. XOR Cache benefits from explicit upgrade notification when a line transitions from `Shared` to `Modified` state, as the LLC is informed that its copy is potentially stale and should not remain XORed.

2.3 Low-Power Cache Architecture

Prior research has explored various approaches to reduce cache power consumption. Techniques to lower leakage power include dynamically turning off dead blocks [1, 27] or cache ways [5], and placing cold blocks into a low-power drowsy state [23]. Other strategies focus on reducing the overall power by operating caches near the threshold voltage [20] or employing mixed-cell design [28]. To address dynamic power, prior work proposes using a small filter cache [33] to handle frequent accesses more efficiently. Additionally, reducing cache size has been achieved by only storing reused blocks [4] or through cache compression [32].

Among these, **compression** emerges as a particularly promising technique, exploiting data redundancy for compressibility [3, 7–10, 16, 18, 21, 24, 29, 30, 35, 41–47, 49–51, 54]. A compressed cache maintains the benefits of a larger cache with significantly reduced hardware costs. To implement cache compression, designs adopt a decoupled tag data array structure. Some also require additional hardware structures for metadata storage [24, 54]. 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. [3, 8, 30, 45, 54]. These works are typically dictionary-based, matching against predefined common values or patterns at sub-line granularity. **Inter-line** works [24, 29, 41, 44, 46, 47, 49] compress multiple similar lines together and store only one copy of the line, along with additional metadata. Typically, a hash function selects the lines compressed together based on address or data value.

3 XOR Compression

We explain how our proposed XOR compression works and address the first challenge mentioned in Section 1.3.

3.1 Compression and Decompression Algorithm

Our implementation of XOR compression employs simple and symmetric compression and decompression algorithms. As discussed in Section 1.1, 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 merely 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, 51]. In fact, any form of reversible computing [2] is compatible with this type of compression. In this work, we choose XOR due to its simplicity and symmetry as mentioned earlier. We only consider 2-way XORing, i.e., XORing exactly two lines, and leave exploration of other reversible functions for future work.

3.2 XOR Policy: Finding XOR Candidates

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 in Section 1.2, if similar lines are allowed to XOR 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 Figure 4. 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 similar to each other. As in Figure 2, the exhaustive search across the entire bank, i.e., *idealBank*, achieves a remarkable compression ratio boost

[2] Given an operator $\circ$ and its reverse operator $\circ^{-1}$, and $C = A \circ B$. Given A and C, operator $\circ$ is reversible if we can uniquely determine B by $C \circ^{-1} A$.

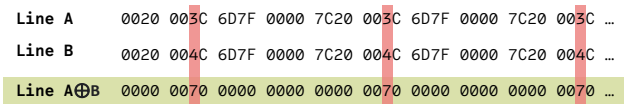


Figure 4: Two similar lines A and B from bodytrack benchmark in PAESEC3.0 suite. The XORed line A⊕B has low entropy.

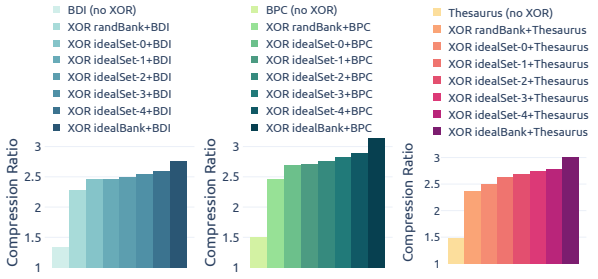


Figure 5: Sensitivity study of *idealSet* compression ratio on the effect of spatio-value locality. X in *idealSet*-X denotes the number of index bits shifted towards the MSBs.

from the baseline. However, it is idealistic and prohibitively expensive in hardware. *idealSet* shows less synergy due to a smaller scope of suitable candidates, as a set contains significantly fewer lines than the entire bank. In the sensitivity study in Figure 5, we also consider variants of *idealSet* where we leverage spatio-value locality and intentionally create more similar candidates by shifting the index bits in the cache line address towards the MSBs by 1-4 bits. We find that shifting the index bits can improve exhaustive search within a set by 5.47% and 11.66% on average and up to 28.32% and 4.04×, respectively. The results, i.e., *idealSet-1,2,3,4*, show more prominent synergy, closing the gap between the ideal and more realistic compression ratios.

As a practical implementation, we consider a map table-based synergistic XOR policy for finding similar lines within a bank using manageable hardware complexity. The map table, similar to those in prior works on deduplication [24, 46, 49], is an additional level of indirection that effectively implements a hash table. A map function is applied on the cache line to generate a map value, which serves as a signature of the cache line value and is used as an index to the map table. The map table implementation and map function selection will be discussed in Section 5.1.3.

4 XOR Cache Coherence

This section describes the coherence protocol, addressing the second challenge discussed in Section 1.3. Though it can be more natural to reason about coherence for XORed lines as pairs of states, we describe the protocol in terms of decoupled state transitions here for brevity. After performing **XOR compression** (Section 4.2), XOR Cache relies on its coherence protocol to perform **decompression** to service data requests (Section 4.3) and **unXORing** to maintain data recoverability (Section 4.4). We discuss the inter-line dependency and its implication on deadlock-freedom in Section 4.5.

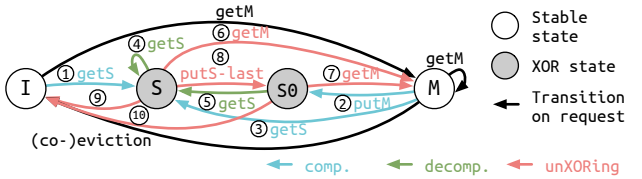


Figure 6: LLC transitions between stable states. `I` for `Invalid`; `S` for `Shared`; `M` for `Modified`; `S0` is a special `S` state when the number of sharers is zero; compression, decompression, and unXORing edges are in blue, green, and red, respectively.

4.1 Assumption

In this work, we present XOR Cache's protocol based on MSI, though the concepts generalize to other protocols. Figure 6 shows the LLC transition between stable states. In addition to the three stable states, namely `Modified`, `Shared`, and `Invalid`, we denote another state `Shared0`, for clarity. It is the specific scenario of `Shared` when the line has no sharers in the private cache. Note that `Shared0` state is fundamentally different from `Exclusive` state in MESI protocols, as `Shared0` lines do not have write permission without sending explicit write requests. Our implementation assumes mixed inclusive cache hierarchy, where inclusion is maintained for clean lines, and exclusion is enforced for dirty lines since their owner is in the higher level, and the lower level does not directly service requests on dirty lines. These assumptions are clarified in Table 1, where `S` lines allocate both directory and LLC entries, while `M` lines only allocate a directory entry. When promoted to `M`, a line deallocates its entry in the LLC. `S0` lines are those with no sharer at the higher level; therefore, they only allocate LLC entries. Note that this does not impose considerably more transient states, as `S0` is a state that already exists in MSI; we simply highlight it for clarity in the description. 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. In the following sections, we dissect the protocol by discussing three types of transitions special to XOR Cache: compression, decompression, and unXORing, as highlighted in blue, green, and red in Figure 6.

Table 1: Coherence stable states and mapping in storage.

State	`Invalid`	`Shared`	`Modified`	`Shared0`
Directory	✗	✓	✓	✗
LLC	✗	✓	✗	✓

4.2 Compression

Upon line insertion, XOR Cache attempts to XOR it with an existing line in the data array. The inserted line can be 1) a line from memory on a demand miss, denoted by ① in Figure 6, or 2) a write-back line from the higher level due to dirty eviction, i.e., ②, or 3) writer data update on a downgrade, i.e., ③. No coherence change is required by compression; we only highlight the transitions at the end of which it can happen.

4.3 Decompression

When a `getS` request (④ and ⑤ in Figure 6) arrives at the LLC for an XORed line, XOR Cache needs to decompress it to service the

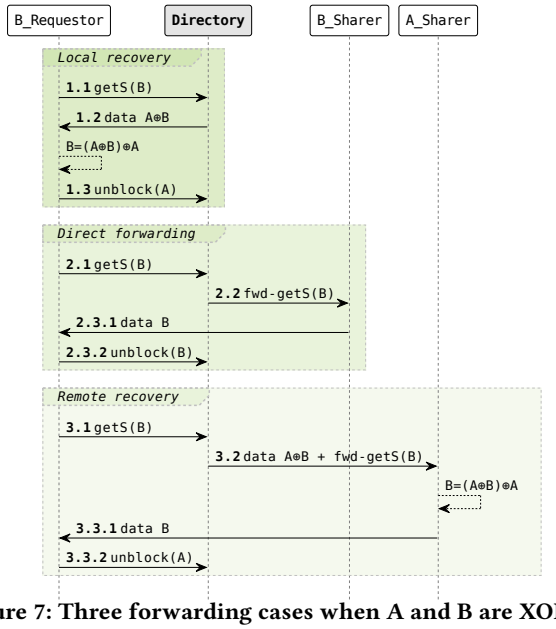


Figure 7: Three forwarding cases when A and B are XORed. From top to bottom are local recovery, direct forwarding, and remote recovery.

request. To decompress, XOR Cache forwards the request to the higher level. As shown in Table 2, there are three data forwarding cases, and the sequence diagrams for handling them are shown in Figure 7. We continue with the same scenario as in Figure 1, where we have a miss on line B, i.e., `getS(B)`, while the LLC holds A⊕B.

Table 2: Forwarding choices when requestor issues `getS(B)`, assuming A and B are XORed in LLC.

Case	A State (`Shared`/ `Shared0`)	B State (`Shared`/ `Shared0`)	B Requestor shares line A
Local recovery	`Shared`	-	✓
Direct forwarding	-	`Shared`	✗
Remote recovery	`Shared`	`Shared0`	✗

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 B has at least one sharer at the higher level, LLC forwards the request for B to one of B's sharers. B's sharer then supplies the data. Note that this case does not need any XOR operation and is handled as conventional cache-to-cache forwarding. We call this case **direct forwarding**. In the third case, the requestor of B similarly does not share A, and B no longer has any sharer. In this case, A must have at least one sharer 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, performs another bitwise XOR between A⊕B and A to retrieve B remotely, and then sends it back to B's requestor. We name this case **remote recovery.**

In these cases, unblock control messages (1.3, 2.3.2, and 3.3.2) are sent from the private cache that supplies the data to the directory, informing its data service completion. These are necessary for a completely unblocking cache controller implementation without assuming any network ordering requirement. Unblock messages are similarly needed in the baseline when downgrading from and upgrading to `M` state. For direct forwarding and remote recovery, when multiple sharers exist, we randomly select the forwarder. There are two directory-to-cache messages (1.2 and 3.2 in Figure 7) including both addresses A and B. We assume an extra 8 bytes for these packets. Note the case where both A and B are in `S0` state is impossible due to the minimum sharer invariant. This invariant is guaranteed by design by enforcing the necessary unXORing, discussed next in Section 4.4.

4.4 UnXORing

4.4.1 When unXORing happens. XOR Cache occasionally needs to unXOR a pair before any of them goes into an unrecoverable state in the following three cases. First, when a line B, XORed with A, is **upgrading to `Modified`**, denoted by ⑥ and ⑦ in Figure 6, B's writer is expected to update its value, rendering the LLC copy potentially stale and thus unrecoverable. Therefore, on `getM` requests, the XORed pair, A⊕B, needs to unXOR. Second, XOR Cache requires that an XORed line can stay compressed if at least one line in the XOR pair has at least one sharer (the minimum sharer invariant). Therefore, unXORing is needed on **the last `putS` request** transitioning from `S` to `S0`, i.e., ⑧. Third, when handling **`eviction`** from `S` and `S0`, i.e., ⑨ and ⑩, unXORing is needed to recover the original data and optionally write back to memory if 1) only one of the two paired lines is performing `eviction` due to insufficient tag space or 2) both lines are performing `co-eviction` due to insufficient data space and at least one of them is dirty.

4.4.2 unXORing implementation. Note the line triggering unXORing, say B, may be in `S` or `S0` state; ⑥, ⑧, and ⑨ are `S` state triggered, while ⑦ and ⑩ are `S0` state triggered. These two cases require different implementations of unXORing. To unXOR, the LLC has to obtain one of the original lines in the higher level cache by issuing a special write-back request. If the triggering line B is in `S` state (⑥, ⑧, and ⑨), the write-back request is sent to the triggering line's own sharer. It can be implemented in the protocol directly as it only involves one address. Otherwise, when the triggering line, B, is in `S0` (⑦ and ⑩), both A and B are involved. The paired line, A, transitions into a transient state and serves as the proxy for retrieving data to perform unXORing. This effectively creates a dependence between A and B, as a request on B may trigger state transition on A. The implication of this inter-line dependence on deadlock freedom is discussed in Section 4.5.

4.4.3 Free of uncontrolled expansion. In other compressed cache schemes, an update to a line may cause data size change and thus may induce a variable number of LLC evictions due to insufficient space in the data array, which may lead to performance problems. In XOR Cache, due to unXORing, except in the `co-eviction` case, expansion also happens. Especially when XOR compression is combined with other intra-line compression schemes, the likelihood of data expansion increases. If the victim happens to be an XORed data block, both lines should perform eviction, i.e., `co-eviction`. Say that line A, XORed with line B, triggers unXORing through the

abovementioned operations. Then the recovered line B and optionally A[3] after intra-line compression individually try re-insertion. This data expansion may cause another XORed pair, C and D, to co-evict. If any of C and D is dirty, unXORing is performed. However, this is guaranteed not to cause further expansion, as the recovered lines C and D only occupy the transaction buffer space as opposed to the actual cache space. Therefore, XORed data block eviction never causes expansion, so it can not cause further data eviction and is guaranteed to be sunk.

4.5 Deadlock Freedom

To show that XOR Cache's coherence protocol is correct and practical, this section proves the following two properties: 1) there is no cyclic dependence between requests (free of protocol deadlock) in Section 4.5.1, and 2) its implementation does not require any extra virtual network (free of virtual network deadlock) in Section 4.5.2.

4.5.1 Free of cyclic dependence between requests. As in Section 4.4, XOR Cache introduces inter-line dependence through unXORing. To verify deadlock freedom, we combine model checking with analysis. We leverage the model checking tool Murphi [19] to verify deadlock freedom with a single address. To avoid the combinatorial explosion problem, we analytically evaluate deadlock freedom with multiple addresses similar to [34]. We adopt an unblocking private cache controller and blocking LLC controller implementation. With our controller implementation, only LLC-bound requests may ever be blocked. Recall that the S state line A in the XORed pair serves as the proxy for its paired line B to send out a write-back request to A's sharer in the case of `getM(B)`. In reply to the write-back request, A's sharer should send an LLC-bound write-back data response. Note that this response cannot be blocked by messages other than LLC-bound requests, as line A cannot be in other blocking transient state[4], therefore the dependence established by XOR Cache does not result in cyclic dependence between requests.

4.5.2 No extra virtual network (VN) required. With a single address, two virtual networks, one dedicated to LLC-bound requests and the other to the rest of messages, are sufficient to guarantee virtual network deadlock freedom. With the new dependence introduced by XOR Cache, it is sufficient to ensure VN deadlock freedom by enforcing that LLC-bound requests such as `getM` cannot be in the same virtual network with this LLC-bound write-back response. This is already satisfied by the virtual network assignment for a single address. As a result, we argue that XOR Cache's coherence protocol is deadlock-free, and more importantly, its implementation does not require any extra virtual network on top of the baseline.

In summary, XOR Cache's coherence protocol supports decompression on LLC access to an XORed line (Section 4.3) and unXORing for maintaining data recoverability (Section 4.4). The protocol requires 18.8% more transient states for implementing decompression and unXORing. Additionally, an additional pair of messages (write-back request and response in Section 4.5) and a forwarded `getS` request (`fwd-getS` in Figure 7) are needed, which account for 18.2% overhead in message support.

[3]If unXORing is triggered by `getM`, the triggering line is not re-inserted as exclusion is enforced on dirty lines. As an optimization to avoid excessive expansion, on the last `putS(A)` request, we may choose to insert B only and drop A with no back-invalidation required.

[4]Otherwise, it would have caused unXORing in the first place, or A would not have been XORed with B

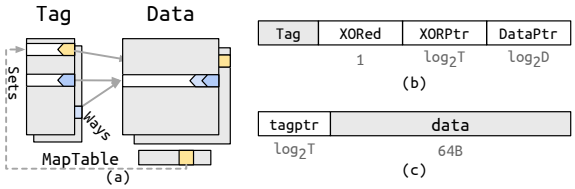


Figure 8: XOR Cache organization. a) Decoupled tag-data store and map table; b) Tag entry; c) Data entry; Grey blocks are identical to the uncompressed baseline; T is the number of tag entries; D is the number of data entries.

5 XOR Cache Architecture

This section outlines the XOR Cache's storage organization and how it handles cache operations.

5.1 Organization

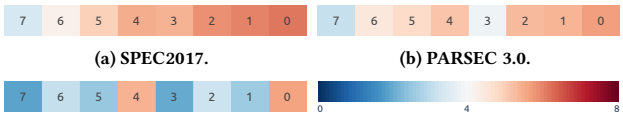
To allow two arbitrary lines in a bank to XOR, XOR Cache adopts a decoupled tag-data organization and uses a map table to identify XOR candidates, as shown in Figure 8a.

5.1.1 Tag Array. The tag array is managed as a linked list with each tag entry shown in Figure 8b. *XORed* is a 1-bit indicator of whether the line is XORed with a partner. *XORptr* points to the tag entry of its partner. Note that in our implementation, a line can only have one partner, so XOR Cache only needs one tag pointer instead of two in the conventional case. We leave the exploration of XORing beyond pairs of lines to future work. *DataPtr* points to the corresponding entry in the data array.

5.1.2 Data Array. As shown in Figure 8c, 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. Note that when combined with other compression schemes, XOR Cache needs to support variable data entry size. We use a similar 8B-segmented data array and *startmap* per set as in Thesaurus to convert the ordinal index stored in the tag entry to the actual segment ID. Details are omitted here for brevity and can be found in [24]. We assume that data compaction happens after eviction, expansion, and contraction, similar to prior works.

5.1.3 Map Table and Map Function. At a high level, the map table is a small storage structure for identifying similar XOR candidates. As shown in Figure 8a, the map table contains tag pointers of standalone lines. It is indexed using a map value generated by applying a map function to the data. On cache line insertion, it first computes the map value and accesses the map table. If it hits, meaning a valid XOR candidate exists, then XOR compression is triggered, and the map table entry is cleared. Otherwise, the line is inserted as an uncompressed line, and the map table allocates an entry for the standalone line. We will discuss XOR Cache's insertion flow in detail in Section 5.2.5. The map table is introduced as an additional level of indirection, similar to [24, 47, 49].

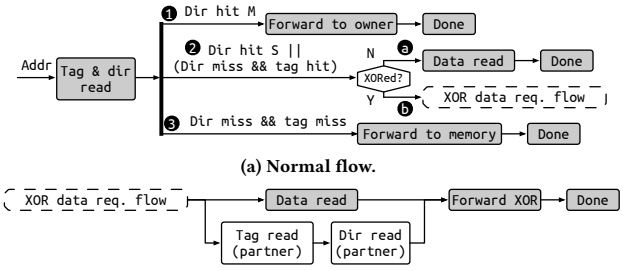
We consider four hash functions as our map function candidates. Two are locality-sensitive hash functions based on random projection (LSH-RP) similar to [24] and bit sampling (LSH-BS). The other two are based on byte labeling. For every byte in the cache line, a boolean label 0 is generated if the byte is 0x0, or 1 is generated otherwise. Byte labeling effectively captures the sparsity of a cache line at a byte level. For baseline byte labeling (BL), the generated



(a) SPEC2017. (b) PARSEC 3.0.

(c) PERFECT. (d) Byte-level entropy scale.

Figure 9: Average byte-level entropy per 8-byte word.



(a) Normal flow.

(b) XOR Cache flow. *Forward XOR* refers to the forwarding cases in Table 2.

Figure 10: Data request flow. The critical path is in grey.

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. This sparse version of byte labeling exploits the fact that the low-ordered bits in words exhibit the highest entropy [29, 44] as shown in Figure 9. Therefore, excluding low-order bytes ideally reduces noise in the map value. For all four map functions, the number of map value bits can vary as a parameter. We compare them in a sensitivity study in Section 6.2 and also discuss the coverage and accuracy trade-off with a varying number of map value bits. Since the map table only needs one entry per unique map value and we keep the map value in a small number of bits, it can be implemented as a direct-mapped structure to minimize the overhead.

5.2 Operations

Like other compressed caches, XOR Cache performs decompression when servicing data requests, which is discussed in Section 5.2.1. Sections 5.2.2, 5.2.3 revisit the necessary unXORing cases. Finally, Section 5.2.5 describes XOR Cache's insertion flow.

5.2.1 Reads. As shown in Figure 10a, a read request that arrives at the LLC first performs a parallel lookup in the directory and tag array. If the address hits in the directory, the line is either in `M` or `S` state. In `M` state ❶, the read request performs the normal flow by forwarding to the owner. In `S` state ❷, i.e., when we have a directory hit on S state, the line could either be standalone, i.e., ⓐ, or XORed with another line, i.e., ⓑ, indicated by the 1-bit *XORed* field in the tag entry. For case ⓐ, we perform the normal flow by retrieving data from the data array. In the latter case ⓑ, the tag entry has a valid partner pointed by *XORPtr*. This triggers the XOR Cache's data request flow as shown in Figure 10b. Reading the XORed data can start immediately by following the *DataPtr* in the tag entry of the requested line. In parallel, the request also needs to access the XORed partner's coherence metadata to decide which of the three forwarding cases discussed in Section 4.3 to follow. First, we follow *XORPtr* to access the tag entry of the XOR partner. Then, a second lookup in the directory is performed to retrieve

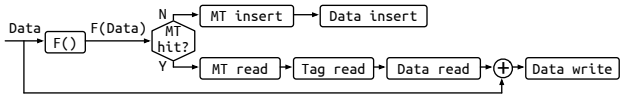


Figure 11: Insertion flow (off critical path). F() denotes the map function.

the coherence metadata of the partner. Recall from Table 2 that depending on the state of the requested line and the partner, this read access is handled in three different cases. It can be serviced by sending the XORed line back, piggybacking on the partner that already exists in the requestor, i.e., local recovery. Alternatively, it can be forwarded to a sharer of the requested line corresponding to direct forwarding. Note that in this case, since the request hits in the directory, the line is guaranteed not to be in `S0` state, so remote recovery is not needed.

If the address misses in the directory but hits in the tag array, i.e., also case ❷, the line is in `S0` state. Similarly, it can be standalone or XORed. For XORed `S0` line ⓑ, the second lookup is also triggered, and the request is either piggybacked on the existing partner, i.e., local recovery, or forwarded to the partner's sharer, i.e., remote recovery. Lastly, on an LLC read miss ❸, the request is forwarded to memory as in normal operation flow. After the data is returned from memory, the insertion flow in Section 5.2.5 is triggered.

5.2.2 Writes and Upgrades. On write request to an `I` or `M` line, it is forwarded to either memory or the owner as usual. In other cases, when a write request arrives for an XORed line, XOR Cache must unXOR the XORed pair. This involves an extra writeback hop from the higher level cache to the LLC. As XOR Cache forces exclusion on dirty lines, it will be evicted from the tag and data array and inserted into the directory.

5.2.3 Writebacks. Clean writeback requests, i.e., `putS` are short data-less messages that allow the directory to track a precise sharer list. On an XORed line's last writeback request, it looks up its partner's coherence metadata to determine if unXORing is triggered. A negative acknowledgment is sent out as a reply, and the private cache should retry the clean writeback along with clean data. Dirty writebacks, i.e. `putM`, are handled similarly to insertion, which will be discussed in Section 5.2.5. The line performs XOR compression if a candidate exists.

5.2.4 Evictions. When co-evicting an XORed line, unXORing is needed to perform dirty writeback to memory. Both tags are evicted through the reverse and forward pointers.

5.2.5 Insertions. Upon data insertion, XOR Cache tries to find an available candidate in the map table to co-locate with. The insertion could be from memory due to demand fetch or from private caches due to writeback. This insertion flow, shown in Figure 11, is off the critical path. In a map table-based implementation, the map value is calculated by applying the map function to the returned data. The map value is then 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, if no candidate exists, i.e., on map table miss, the tag pointer is inserted into the map table, data is inserted, and tag is updated. For both lines, *DataPtr*s in the tag array point to the XORed data entry, and the *XORPtr*s point to each other.

Table 3: Hardware system configuration.

CPU	4 core, 3GHz x86-64
L1I	32KiB, 4 way, 4 cycle, 64B line, LRU, Private
L1D	32KiB, 4 way, 4 cycle, 64B line, LRU, Private
L2	256KiB, 8 way, 9 cycles, 64B line, LRU, Private
L3	1MiB per bank, 16 way, 40 cycle, 64B line, LRU, Shared, 4 banks
Memory	DualChannelDDR4-2400

Table 4: Per bank LLC storage breakdown. *Other* refers to the base cache in Thesaurus and map table in XOR Cache.

Clusivity		Mixed inclusive					Exclusive	
Compression		Un-comp.	BΔI	The-saurus	BPC	XOR +BΔI	Un-comp.	BΔI
Tag	#Entries	16384					13312	
	Entry size	32b	49b	49b	49b	63b	32b	49b
	Size (KiB)	64	98	98	98	126	52	79.63
Data	#Entries	16384	12288	10240	10240	6144	13312	10240
	Entry size	512b				512+40b	512b	
	Size (KiB)	1024	768	640	640	414	832	640
Other	#Entries	-		512	-	128	-	
	Entry size	-		512b	-	14b	-	
	Size (KiB)	-		32	-	0.22	-	
Total size (KiB)		1088	866	770	738	540.22	884	719.63

Table 5: SPEC CPU 2017 benchmark random mixes.

Run	Benchmarks
1	505.mcf_r, 541.leela_r, 510.parest_r, 503.bwaves_r
2	520.omnetpp_r, 548.exchange2_r, 500.perlbench_r, 557.xz_r
3	511.povray_r, 521.wrf_r, 538.imagick_r, 549.fotonik3d_r
4	502.gcc_r, 544.nab_r, 519.lbm_r, 527.cam4_r
5	523.xalancbmk_r, 525.x264_r, 508.namd_r, 554.roms_r
6	531.deepsjeng_r, 507.cactuBSSN_r, 521.wrf_r, 538.imagick_r
7	525.x264_r, 500.perlbench_r, 549.fotonik3d_r, 519.lbm_r
8	503.bwaves_r, 541.leela_r, 557.xz_r, 508.namd_r
9	527.cam4_r, 511.povray_r, 507.cactuBSSN_r, 505.mcf_r
10	520.omnetpp_r, 554.roms_r, 502.gcc_r, 531.deepsjeng_r
11	523.xalancbmk_r, 544.nab_r, 510.parest_r, 548.exchange2_r

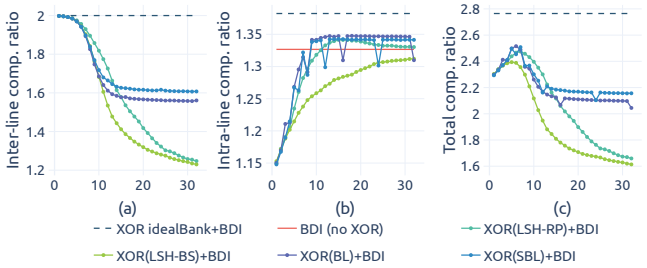


Figure 12: Comp. ratio with four map functions (Section 5.1.3). (a) inter-line comp. ratio; (b) intra-line comp. ratio; (c) total comp. ratio. The *x*-axis is the number of map value bits.

6 Evaluation

XOR Cache enables effective compression through XOR compression. With it, XOR Cache reduces area and power with comparable performance to the uncompressed cache. The goal of this section is to evaluate XOR Cache's compressibility (Sections 6.2 and 6.3) and its effectiveness in reducing area and power (Section 6.4). We finally show its performance overhead (Section 6.5) and energy-delay product improvement (Section 6.8).

6.1 Methodology

6.1.1 Simulator and Simulated System. We simulate XOR Cache with its coherence protocol using the Ruby model in the gem5 simulator [36]. We use CACTI 7.0 [11] to evaluate area, power, and latency of memory structures and use Synopsys design compiler for compressor hardware synthesis using 32nm technology. Table 3 lists the configuration of the simulated hardware system with a 3-level cache hierarchy similar to [24]. It represents a system with a high LLC-to-MLC size ratio, i.e., 4:1, which is a pessimistic configuration for XOR Cache due to limited XOR compression opportunity.

6.1.2 LLC Configuration. Table 4 lists the configurations of the LLC used in a 3-level cache hierarchy for XOR Cache and other baselines, unless otherwise specified. The uncompressed baseline is 1 MiB per bank LLC (Table 3); the compressed baselines include **intra-line** compressed BΔI and BPC LLCs, and an **inter-line** compressed Thesaurus LLC. Thesaurus [24] dynamically clusters cache lines using locality-sensitive hashing and compresses lines against the centroids. Our XOR Cache is based on xor compression synergistically with intra-line BΔI compression. We shrink the data array size for all compressed caches by a factor based on our profiled geometric mean compression ratio in Figure 2. Specifically, we use a 1.3× smaller data array for BΔI, and a 1.5× smaller data array for Thesaurus and BPC. XOR Cache with BΔI adopts a 2.5× smaller data array. The map table is direct-mapped with 128 entries (Section 5.1.3). Besides these mixed inclusive LLCs, we also include an exclusive LLC with the same effective capacity.[5] We also include the exclusive LLC with BΔI applied atop.

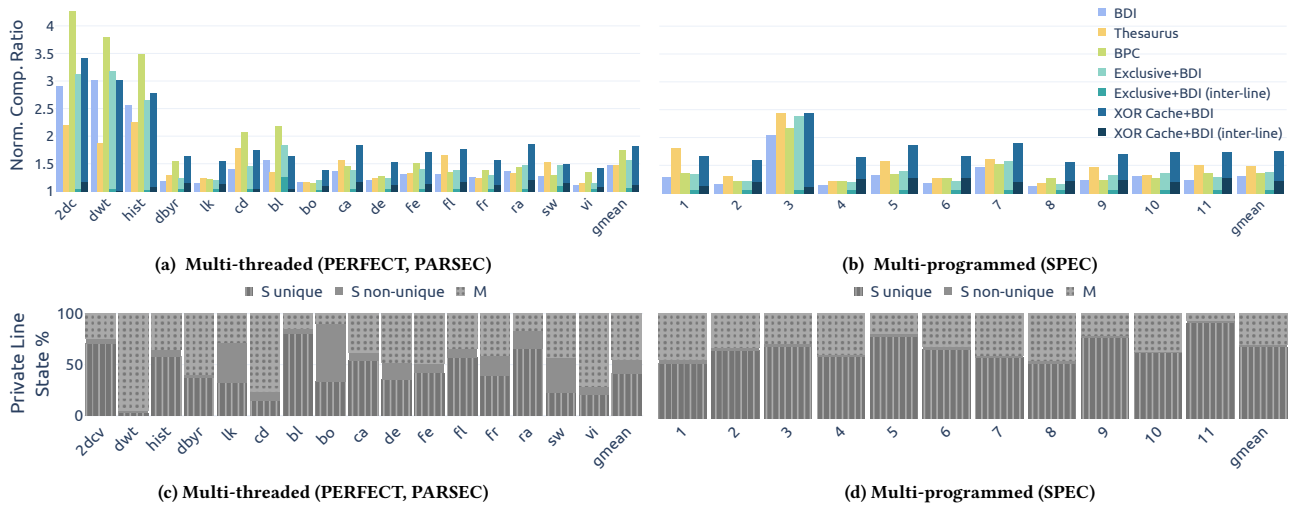
We pessimistically assume a uniform LLC latency of 40 cycles, despite the potential for lower latency given the smaller data array. We assume 1 cycle latency for BΔI, 5 cycles for Thesaurus, and 7 cycles for Bit-plane decompression. For XOR Cache's (de)compressor, the synthesized XOR gate array only incurs 0.12 ns delay, so we assume performing bit-wise XOR is within the same cycle as the read. Note that we also model forwarding latency as part of XOR decompression, in addition to the above decompressor latencies.

6.1.3 Benchmarks. We evaluate three benchmark suites in gem5 full-system simulation. For **multi-threaded** workloads, we include the PERFECT [12] openmp version, which targets image processing workloads, and PARSEC 3.0 [13] with simlarge datasets. For PERFECT and PARSEC 3.0, we simulate the entire region of interest. We include SPEC CPU 2017 [14] reference workloads for **multi-programmed** evaluation. In each run, we launch a random mix of 4 SPEC rate benchmarks, each with one copy, shown in Table 5. We fast-forward 100B instructions and collect statistics using the following 1B detailed instructions on every core.

6.2 XOR Compression Synergy

In our map table-based implementation of XOR Cache, we use a small map table with a map function to choose the XOR candidates based on value similarity. We profile the LLC snapshots to compare the four map functions (LSH-RP, LSH-BS, BL, and SBL)

[5] We size the exclusive LLC according to the proportion of `S0` lines as the baseline.



(a) Multi-threaded (PERFECT, PARSEC)

(b) Multi-programmed (SPEC)

(c) Multi-threaded (PERFECT, PARSEC)

(d) Multi-programmed (SPEC)

Figure 13: Compression ratio analysis.

introduced in Section 5.1.3. Figure 12 shows the inter-line and intra-line geometric mean compression ratios across all three benchmark suites. The X-axis shows the number of map value bits, and the Y-axis shows the compression ratio. Additionally, compression ratios of XOR *idealBank* with BΔI and baseline BΔI (without XOR) are included for reference. For all four map functions, we see a coverage-accuracy tradeoff.

Coverage. As shown in Figure 12a, all four map functions show decreased XOR compression coverage as the number of map value bits increases. The reason is that the number of unique map values increases exponentially and the probability of finding a candidate in the matching bin decreases. Hence, the inter-line XOR compression opportunity is reduced.

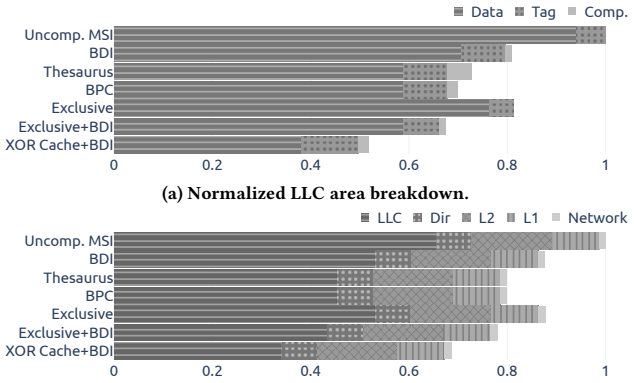
Accuracy. At the same time, however, as shown in Figure 12b, the accuracy of identifying similar candidate lines improves as the intra-line compression ratio increases due to the increased value similarity between the XORed line pairs. The reason is that the more map function value bits are used, the fewer false similar candidates appear in the same bin. 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 as BΔI as early as 7 bits; however, SBL maintains a higher inter-line compression ratio since it effectively removes the noise from high entropy bits in words.

Tradeoff. To balance the XOR compression coverage and accuracy of similar candidate selection, we need to select the proper number of map value bits. Figure 12c shows that the sweet spot balancing inter- and intra-line compression ratios occurs at around 7 bits for BL and SBL. Considering all the benchmarks for our profiling results, for the rest of the evaluation, we use 7-bit SBL, which results in an average compression ratio of ~ 2.5, justifying our choice of using a 2.5× smaller data array for XOR Cache with BΔI.

6.3 Compression Ratio Improvement

Figure 13 shows a compression ratio analysis of XOR Cache and baselines. Figure 13a and 13b show the per benchmark compression ratio normalized relative to the uncompressed LLC. We make the following observations. First, XOR Cache with BΔI uniformly boosts baseline BΔI compression ratio across all benchmarks, demonstrating XOR compression's effectiveness. Second, it achieves a higher compression ratio than the exclusive LLC with BΔI. Note that the Exclusive LLC (with no BΔI) also has a greater than 1 compression ratio. This comes from the fact that it does not store any S state line, unlike the baselines storing inclusive lines. However, this ratio is low (1.06×) due to the already limited redundancy between LLC and private caches (4:1 ratio) in the baseline hierarchy (Table 3). For exclusive LLC with BΔI, there exists no redundancy due to inclusion. Conversely, XOR Cache leverages this redundancy for effective compression. Third, XOR Cache also achieves a higher average compression ratio than Thesaurus and BPC. Thesaurus achieves a lower compression ratio on PERFECT, whereas BPC works well exploiting value similarity for homogeneous data. For the multi-programmed SPEC workloads, both baselines achieve a lower compression ratio due to limited value similarity at word and cache line level.

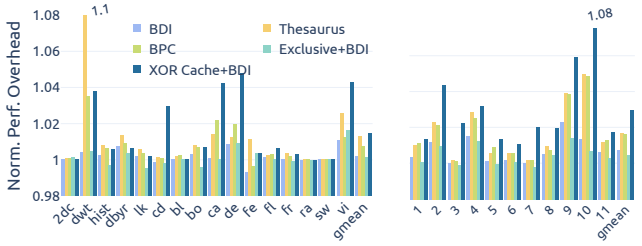
XOR Cache's inter-line compression ratio by XOR compression alone is marked by the darker blue shade. In the practical setting, it is less than the upper bound ratio of 2. The reasons here are threefold. First, there is **limited redundancy between the LLC and the private caches**. In our evaluated configuration (Table 3), the private cache lines only make up at most a quarter of lines in the LLC, leaving the vast majority of LLC lines in `S0` state. They have limited opportunity to be XORed, therefore limiting the inter-line compression ratio. Second, the inter-line compressibility is limited due to the **existence of `Modified` lines**. Note that even though our setup enforces exclusion for `M` lines in the LLC, they contend with `S` lines for private cache space, i.e., the more `M` lines, the fewer `S` lines in the private cache. Recall that XOR Cache leverages shared data in the private caches, i.e., redundancy due to inclusion for compressibility; the existence of `M` lines limits our achieved compression ratio. Third, workloads may have **extensive sharing between cores** in data and instructions; These private cache lines map to the same set



(a) Normalized LLC area breakdown.

(b) Normalized cache hierarchy power breakdown.

Figure 14: Normalized area and power breakdown.



(a) Multi-threaded (PERFECT, PARSEC) (b) Multi-programmed (SPEC)

Figure 15: Normalized performance overhead. (a) shows norm. runtime of multi-threaded runs; (b) shows the norm. geometric mean of CPI of multi-programmed runs.

of S lines in the LLC. In the extreme case, all the private caches may share exactly the same set of N lines; the LLC has only N S lines, and the remaining majority of lines end up as S0. Again, S0 lines have limited XOR opportunity as they can only be XORed with S lines (the minimum sharer invariant). This imbalanced number of S0 and S lines can limit the XOR compressibility. We validated the last two hypotheses above by examining the private cache line distribution shown in Figure 13c and 13d. All the valid private cache lines are classified into M, S non-unique, and S unique; S non-unique lines are those with more than one sharers, whereas S unique lines are exclusive to one private cache. Based on the last two reasons, XOR compression opportunity (dark blue area in Figure 13a and 13b) is proportional to the weight of S unique lines (bottom dark gray area with vertical lines), with the exception of *blacksholes (bl)* as the private cache is not fully utilized to due to its small footprint. For example, *dwt*'s low compression ratio is because more than 90% private cache lines are in M state (top light gray area with dots). PERFECT and PARSEC 3.0 generally have more sharing (the middle S non-unique region), due to their multi-threaded nature. Therefore, they achieve lower inter-line compression ratios compared to the multi-programmed case.

Takeaway. Unlike the exclusive LLC that *eliminates* redundancy due to private caching, XOR Cache embraces it and tames it for more compressibility by synergistic XOR compression. When both combined with the BΔI intra-line scheme, XOR Cache achieves 16.2% and 27.8% higher compression ratio for multi-threaded and multi-programmed workloads than the Exclusive baseline, which lacks such synergy. Moreover, XOR Cache with BΔI also achieves a higher compression ratio than intra- and inter-line schemes by 23.1% (BΔI), 4.5% (BPC), and 23.4% (Thesaurus) on multi-threaded workloads, and 34.9% (BΔI), 28.5% (BPC), and 18.4% (Thesaurus) on multi-programmed workloads.

6.4 Area and Power Improvement

We obtain the normalized LLC area and cache hierarchy power breakdown in Figure 14a and 14b assuming 32nm technology.

6.4.1 Area. To support XOR compression, XOR Cache only needs 0.01 mm^2 extra area. Though decoupling the tag and data arrays adds marginal metadata overhead, the data array size reduction as a result of compression dominates. *Comp.* includes compressors, map table, and map function logic for XOR Cache and includes compressors and storage (e.g., base cache) for other baselines.

Takeaway. XOR Cache achieves 1.93× smaller LLC area than the uncompressed, 1.56×, 1.41×, and 1.35× smaller than BΔI, Thesaurus and BPC counterparts, and 1.30× than the Exclusive LLC with BΔI.

6.4.2 Power. Figure 14b shows the normalized power breakdown of the cache hierarchy. In addition to LLC and private caches, we also include the network dynamic power using the model in [52]. With XOR compression, XOR Cache's additional private cache accesses due to local and remote recovery contribute to a mere 1.99% of total private cache accesses. The increased activity adds overhead to dynamic power. However, the leakage power still dominates the total LLC power contribution due to the filtering effect of private caches. XOR Cache generates 23.4% more network traffic due to additional forwarding messages, which translates to increased dynamic network power. Though this may seem a lot, especially in a scaled-out system, with the network bandwidth scaling trend in emerging chiplet-based systems [17], we do not expect the additional traffic to translate to significant bandwidth overhead. Additionally, this overhead is still less than the Exclusive LLC, which adds 24.6% more traffic for maintaining strict exclusivity.

Takeaway. In spite of its additional activity, XOR Cache still achieves a significant 1.92× LLC power reduction and 1.46× cache hierarchy power reduction compared to the uncompressed cache.

6.5 Performance Overhead

To quantify XOR Cache's performance overhead, we use normalized runtime for multi-threaded benchmarks; we take the geometric mean of CPI from every core on the 1B detailed simulated instructions, for multi-programmed benchmarks. Figure 15 shows performance overhead ofXOR Cache and four baselines (Table 3), normalized to the uncompressed MSI baseline. BΔI adds a fixed 1-cycle decompression latency on the critical path; BPC and Thesaurus have higher decompression latency due to complex decompressors, and Thesaurus also has base cache miss. XOR Cache shows a slowdown of 1.45% on multi-threaded and 2.95% on multi-programmed workloads. This difference between these two sets of workloads is due to two reasons: 1) multi-programmed workloads generally observe less compressibility; and 2) more LLC hits (~ 15%) follow the remote recovery decompression path (Section 4.3), which is the

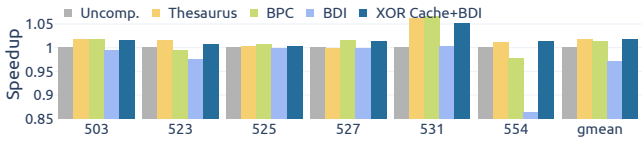


Figure 16: Iso-storage performance.

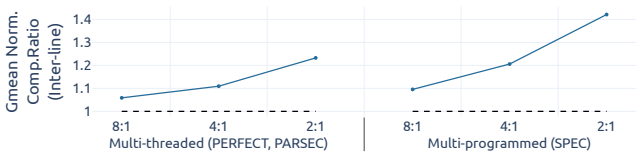


Figure 17: Geometric mean of normalized inter-line compression ratio. X-axis denotes LLC-to-private cache size ratio.

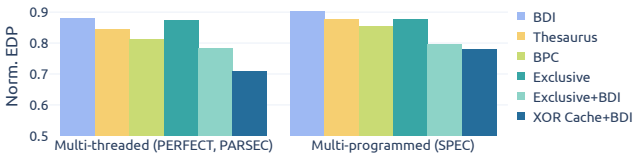


Figure 18: Normalized energy-delay product.

slowest among all three, in multi-programmed workloads. Overall, XOR Cache incurs a marginal geomean norm. performance overhead of 2.06%.

6.6 Case Study: Iso-Storage Performance

Though our primary goal is to reduce LLC area and power, we include a case study on iso-storage performance of XOR Cache against the three compressed baselines. We run 4-core multi-programmed workloads with every core running a copy of a SPEC benchmark. Figure 16 highlights the subset of workloads that are most sensitive to LLC size, where the performance difference is more than 3% using a 2× LLC. XOR Cache achieves an average of 1.78% and up to 5.22% speedup over the uncompressed iso-storage LLC, which is higher than that of BΔI (−2.89%), Thesaurus (1.75%) and BPC (1.28%), thanks to its higher compression ratio. Across all workloads, XOR Cache yields a modest speedup of 0.21%, still highest among the compression schemes.

6.7 Sensitivity Study

6.7.1 Core Count. In addition to the 4-core results, we provide an additional set of 8-core multi-threaded[6] results. Scaling up XOR Cache to 8-core only sees 18.7% network traffic overhead and 1.55% performance overhead, comparable to 18.3% traffic and 1.45% performance overhead in 4-core multi-threaded case.

6.7.2 LLC Size. Figure 17 shows the inter-line compression ratio sensitivity to LLC size with two additional sets of experiments with doubled and halved LLC size, corresponding to an 8:1 and 2:1 LLC-to-private cache size ratio. As discussed in Section 6.1.1 and 6.3, since XOR Cache exploits the redundancy between LLC and private caches to perform XOR compression, as the LLC-to-private cache ratio lowers, there is more compression opportunity for XOR Cache in both multi-threaded and multi-programmed cases.

6.8 Summary: Energy-Delay Product

Figure 18 summarizes our evaluation by measuring the energy-delay product (EDP), normalized to the uncompressed MSI baseline. As a result of effective compression, XOR Cache saves significant power and area with a marginal performance loss. Therefore, the EDP of XOR Cache is the lowest among all, being 26.3% lower than the uncompressed baseline.

7 Related Work

Abundant works leverage inter-line compressibility for heterogeneous data. Deduplication [49] uses a heuristic XOR folding-based hash to identify identical lines and stores only unique ones, eliminating redundancy at the cache line level. MORC [41] proposes compressing lines leveraging temporal value locality for log-based caches. Thesaurus [24] dynamically clusters cache lines using locality-sensitive hashing and compress lines against the centroids. Several inter-line compression works identify that high-order bits in words exhibit low entropy while low-order bits exhibit high entropy. BCD [44] proposes to deduplicate low entropy bits within memory lines. EPC [29] builds on a similar observation and stores a set of frequent patterns of low entropy bits. XOR Cache exploits this similar observation with its map function. Wang et al. [51] also propose a cache compression scheme by XORing multiple lines using in-SRAM bit-line computing. However, it does not target redundancy due to inclusion as XOR Cache does. Bunker [46] and Doppelgänger [47] cache propose to exploit similarity based on address and value, respectively. They work well for approximate computing on image processing workloads. Going beyond the granularity of cache lines, Zippads [50] compresses objects of the same type. SC2 [9, 10] builds Huffman coding based on value statistics and exploits word-level redundancy. XOR Cache fits into the cache compression landscape as a new inter-line compression work. Similar to prior works [24, 46, 49] on inter-line compression, XOR Cache also adopts a hash table-based approach to capture similarity across lines. However, the goal is to pair every two lines to create structured sparsity to catalyze intra-line compression.

8 Conclusion

In this paper, 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, effectively collocating a pair of lines in a single slot. When combined with other compression schemes, XOR Cache can further boost compression ratio by exploiting the compressibility of the XORed data values. We tackle two key challenges in XOR Cache: designing the XOR policy based on value similarity and implementing the cache coherence protocol. Evaluation results show that as a result of XOR Cache's effective compression, it reduces LLC area by 1.93× and power by 1.92× with a comparable performance (average overhead of 2.06%) to a larger uncompressed cache, reducing energy-delay product by 26.3%.

Acknowledgments

We thank all reviewers for their valuable feedback. This work is supported by the Wisconsin Alumni Research Foundation and NSF under award No. CNS-2045985.

[6] Most 8-core multi-programmed SPEC runs fail to complete due to limited memory.

References

[1] Jaume Abella, Antonio González, Xavier Vera, and Michael F. P. O'Boyle. 2005. IATAC: a smart predictor to turn-off L2 cache lines. *ACM Trans. Archit. Code Optim.* 2, 1 (March 2005), 55–77. doi:10.1145/1061267.1061271

[2] Shaizeen Aga, Supreet Jeloka, Arun Subramaniyan, Satish Narayanasamy, David Blaauw, and Reetuparna Das. 2017. Compute Caches. In *International Symposium on High Performance Computer Architecture*.

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

[4] Jorge Albericio, Pablo Ibáñez, Victor Viñals, and José M. Llabería. 2013. The reuse cache: Downsizing the shared last-level cache. In *International Symposium on Microarchitecture*.

[5] D.H. Albonesi. 1999. Selective cache ways: on-demand cache resource allocation. In *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*. 248–259. doi:10.1109/MICRO.1999.809463

[6] Mohammad Alian, Siddharth Agarwal, Jongmin Shin, Neel Patel, Yifan Yuan Yuan, Daehoon Kim, and Ren Wang. 2022. IDIO: Network-Driven, Inbound Network Data Orchestration on Server Processors. In *International Symposium on Microarchitecture*.

[7] Alexandra Angerd, Angelos Arelakis, Vasilis Spiliopoulos, Erik Sintorn, and Per Stenström. 2022. GBDI: Going Beyond Base-Delta-Immediate Compression with Global Bases. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 1115–1127. doi:10.1109/HPCA53966.2022.00085

[8] 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* (Waikiki, Hawaii) *(MICRO-48)*. Association for Computing Machinery, New York, NY, USA, 38–49. doi:10.1145/2830772.2830823

[9] Angelos Arelakis and Per Stenstrom. 2014. SC2: a statistical compression cache scheme. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)* (Minneapolis, Minnesota, USA) *(ISCA '14)*. IEEE Press, 145–156.

[10] Angelos Arelakis and Per Stenstrom. 2014. SC2: a statistical compression cache scheme. *SIGARCH Comput. Archit. News* 42, 3 (jun 2014), 145–156. doi:10.1145/2678373.2665696

[11] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Trans. Archit. Code Optim.* 14, 2, Article 14 (jun 2017), 25 pages. doi:10.1145/3085572

[12] Kevin Barker, Thomas Benson, Dan Campbell, David Ediger, Roberto Gioiosa, Adolfy Hoisie, Darren Kerbyson, Joseph Manzano, Andres Marquez, Leon Song, Nathan Tallent, and Antonino Tumeo. 2013. *PERFECT (Power Efficiency Revolution For Embedded Computing Technologies) Benchmark Suite Manual.* Pacific Northwest National Laboratory and Georgia Tech Research Institute. http://hpc.pnnl.gov/projects/PERFECT/.

[13] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *International Conference on Parallel Architectures and Compilation Techniques*.

[14] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-Generation Compute Benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering* (Berlin, Germany) *(ICPE '18)*. Association for Computing Machinery, New York, NY, USA, 41–42. doi:10.1145/3185768.3185771

[15] Mainak Chaudhuri. 2021. Zero Inclusion Victim: Isolating Core Caches from Inclusive Last-level Cache Evictions. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*.

[16] Xi Chen, Lei Yang, Robert P. Dick, Li Shang, and Haris Lekatsas. 2010. C-pack: a high-performance microprocessor cache compression algorithm. *IEEE Trans. Very Large Scale Integr. Syst.* 18, 8 (aug 2010), 1196–1208. doi:10.1109/TVLSI.2009.2020989

[17] Grigory Chirkov and David Wentzlaff. 2023. Seizing the Bandwidth Scaling of On-Package Interconnect in a Post-Moore's Law World. In *Proceedings of the 37th International Conference on Supercomputing* (Orlando, FL, USA) *(ICS '23)*. Association for Computing Machinery, New York, NY, USA, 410–422.

[18] Esha Choukse, Mattan Erez, and Alaa R. Alameldeen. 2018. Compresso: Pragmatic Main Memory Compression. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 546–558. doi:10.1109/MICRO.2018.00051

[19] David L. Dill. 1996. The Mur ϕ verification system. In *Computer Aided Verification*, Rajeev Alur and Thomas A. Henzinger (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 390–393.

[20] Ronald G. Dreslinski, Gregory K. Chen, Trevor Mudge, David Blaauw, Dennis Sylvester, and Krisztian Flautner. 2008. Reconfigurable energy efficient near threshold cache architectures. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*. 459–470. doi:10.1109/MICRO.2008.4771813

[21] Albin Eldstål-Ahrens, Angelos Arelakis, and Ioannis Sourdis. 2023. FlatPack: Flexible Compaction of Compressed Memory. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (Chicago, Illinois) *(PACT '22)*. Association for Computing Machinery, New York, NY, USA, 96–108. doi:10.1145/3559009.3569653

[22] Ricardo Fernández-Pascual, Alberto Ros, and Manuel E. Acacio. 2017. To Be Silent or Not: On the Impact of Evictions of Clean Data in Cache-Coherent Multicores. *J. Supercomput.* 73, 10 (oct 2017), 4428–4443.

[23] Krisztián Flautner, Nam Sung Kim, Steve Martin, David Blaauw, and Trevor Mudge. 2002. Drowsy caches: simple techniques for reducing leakage power. In *Proceedings of the 29th Annual International Symposium on Computer Architecture* (Anchorage, Alaska) *(ISCA '02)*. IEEE Computer Society, USA, 148–157.

[24] Amin Ghasemazar, Prashant Nair, and Mieszko Lis. 2020. Thesaurus: Efficient Cache Compression via Dynamic Clustering. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 527–540. doi:10.1145/3373376.3378518

[25] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2016. Cross Processor Cache Attacks. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security* (Xi'an, China) *(ASIA CCS '16)*. Association for Computing Machinery, New York, NY, USA, 353–364. doi:10.1145/2897845.2897867

[26] Aamer Jaleel, Eric Borch, Malini Bhandaru, Simon C. Steely Jr., and Joel Emer. 2010. Achieving Non-Inclusive Cache Performance with Inclusive Caches: Temporal Locality Aware (TLA) Cache Management Policies. In *International Symposium on Microarchitecture*.

[27] S. Kaxiras, Zhigang Hu, and M. Martonosi. 2001. Cache decay: exploiting generational behavior to reduce cache leakage power. In *Proceedings 28th Annual International Symposium on Computer Architecture*. 240–251. doi:10.1109/ISCA.2001.937453

[28] Samira M. Khan, Alaa R. Alameldeen, Chris Wilkerson, Jaydeep Kulkarni, and Daniel A. Jiménez. 2013. Improving multi-core performance using mixed-cell cache architecture. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. 119–130. doi:10.1109/HPCA.2013.6522312

[29] Jinkwon Kim, Mincheol Kang, Jeongkyu Hong, and Soontae Kim. 2022. Exploiting Inter-block Entropy to Enhance the Compressibility of Blocks with Diverse Data. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 1100–1114. doi:10.1109/HPCA53966.2022.00084

[30] Jungrae Kim, Michael Sullivan, Esha Choukse, and Mattan Erez. 2016. Bit-Plane Compression: Transforming Data for Better Compression in Many-Core Architectures. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 329–340. doi:10.1109/ISCA.2016.37

[31] Sowoong Kim, Myeonggyun Han, and Woongki Baek. 2022. DPrime+DAbort: A High-Precision and Timer-Free Directory-Based Side-Channel Attack in Non-Inclusive Cache Hierarchies using Intel TSX. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 67–81. doi:10.1109/HPCA53966.2022.00014

[32] 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* (Porto Alegre, Brazil) *(MICRO-44)*. Association for Computing Machinery, New York, NY, USA, 420–429. doi:10.1145/2155620.2155670

[33] J. Kin, Munish Gupta, and W.H. Mangione-Smith. 1997. The filter cache: an energy efficient memory structure. In *Proceedings of 30th Annual International Symposium on Microarchitecture*. 184–193. doi:10.1109/MICRO.1997.645809

[34] Weihang Li, Andrés Goens, Nicolai Oswald, Vijay Nagarajan, and Daniel J. Sorin. 2024. Determining the Minimum Number of Virtual Networks for Different Coherence Protocols. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*.

[35] Y. Li and M. Gao. 2023. Baryon: Efficient Hybrid Memory Management with Compression and Sub-Blocking. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 137–151. doi:10.1109/HPCA56546.2023.10071115

[36] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, et al. 2020. The gem5 simulator: Version 20.0+. *arXiv preprint arXiv:2007.03152* (2020).

[37] Milo M. K. Martin, Mark D. Hill, and Daniel J. Sorin. 2012. Why On-Chip Cache Coherence is Here to Stay. *Commun. ACM* 55, 7 (jul 2012), 78–89. doi:10.1145/2209249.2209269

[38] Hassan Mujtaba. 2020. *AMD Ryzen 5000 Zen 3 'Vermeer' Undressed, First Ever High-Res Die Shots Close Ups Pictured & Detailed.* https://wccftech.com/amd-ryzen-5000-zen-3-vermeer-undressed-high-res-die-shots-close-ups-pictured-detailed/

[39] Ashley O. Munch, Nevine Nassif, Carleton L. Molnar, Jason Crop, Rich Gammack, Chinmay P. Joshi, Goran Zelic, Kambiz Munshi, Min Huang, Charles R. Morganti, Sireesha Kandula, and Arijit Biswas. 2024. 2.3 Emerald Rapids: 5th-Generation Intel® Xeon® Scalable Processors. In *2024 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 67. 40–42. doi:10.1109/ISSCC49657.2024.10454434

[40] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2020. *A Primer on Memory Consistency and Cache Coherence, Second Edition.* Springer Cham.

[41] Tri M. Nguyen and David Wentzlaff. 2015. MORC: a manycore-oriented compressed cache. In *Proceedings of the 48th International Symposium on Microarchitecture* (Waikiki, Hawaii) *(MICRO-48)*. Association for Computing Machinery, New York, NY, USA, 76–88. doi:10.1145/2830772.2830828

[42] Biswabandan Panda and André Seznec. 2016. Dictionary sharing: An efficient cache compression scheme for compressed caches. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. doi:10.1109/MICRO.2016.7783704

[43] Gagandeep Panwar, Muhammad Laghari, Esha Choukse, and Xun Jian. 2024. DyLeCT: Achieving Huge-page-like Translation Performance for Hardware-compressed Memory. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*.

[44] Sungbo Park, Ingab Kang, Yaebin Moon, Jung Ho Ahn, and G. Edward Suh. 2021. BCD deduplication: effective memory compression using partial cache-line deduplication. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 52–64. doi:10.1145/3445814.3446722

[45] 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* (Minneapolis, Minnesota, USA) *(PACT '12)*. Association for Computing Machinery, New York, NY, USA, 377–388. doi:10.1145/2370816.2370870

[46] Joshua San Miguel, Jorge Albericio, Natalie Enright Jerger, and Aamer Jaleel. 2016. The Bunker Cache for spatio-value approximation. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. doi:10.1109/MICRO.2016.7783746

[47] Joshua San Miguel, Jorge Albericio, Andreas Moshovos, and Natalie Enright Jerger. 2015. Doppelgänger: A cache for approximate computing. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 50–61. doi:10.1145/2830772.2830790

[48] Yingying Tian, Samira M. Khan, and Daniel A. Jiménez. 2013. Temporal-Based Multilevel Correlating Inclusive Cache Replacement. *ACM Trans. Archit. Code Optim.* 10, 4, Article 33 (dec 2013), 24 pages. doi:10.1145/2541228.2555290

[49] 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* (Munich, Germany) *(ICS '14)*. Association for Computing Machinery, New York, NY, USA, 53–62. doi:10.1145/2597652.2597655

[50] Po-An Tsai and Daniel Sanchez. 2019. Compress Objects, Not Cache Lines: An Object-Based Compressed Memory Hierarchy. In *International Conference on Architectural Support for Programming Languages and Operating Systems*.

[51] Xiaowei Wang, Charles Augustine, Eriko Nurvitadhi, Ravi Iyer, Li Zhao, and Reetuparna Das. 2021. Cache Compression with Efficient in-SRAM Data Comparison. In *2021 IEEE International Conference on Networking, Architecture and Storage (NAS)*. 1–8. doi:10.1109/NAS51552.2021.9605440

[52] P.T. Wolkotte, G.J.M. Smit, N. Kavaldjiev, J.E. Becker, and J. Becker. 2005. Energy Model of Networks-on-Chip and a Bus. In *2005 International Symposium on System-on-Chip*. 82–85. doi:10.1109/ISSOC.2005.1595650

[53] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. 2019. Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World. In *2019 IEEE Symposium on Security and Privacy (SP)*. 888–904. doi:10.1109/SP.2019.00004

[54] 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* (Monterey, California, USA) *(MICRO 33)*. Association for Computing Machinery, New York, NY, USA, 258–265. doi:10.1145/360128.360154

[55] Li Zhao, Ravi Iyer, Srihari Makineni, Don Newell, and Liqun Cheng. 2010. NCID: A Non-Inclusive Cache, Inclusive Directory Architecture for Flexible and Efficient Cache Hierarchies. In *Proceedings of the 7th ACM International Conference on Computing Frontiers*.

[56] Zirui Neil Zhao, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. 2024. Last-Level Cache Side-Channel Attacks Are Feasible in the Modern Public Cloud. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 582–600. doi:10.1145/3620665.3640403