Thesaurus: Efficient Cache Compression via Dynamic Clustering

Amin Ghasemazar
University of British Columbia
Vancouver, BC, Canada
aming@ece.ubc.ca

Prashant Nair
University of British Columbia
Vancouver, BC, Canada
prashantnair@ece.ubc.ca

Mieszko Lis
University of British Columbia
Vancouver, BC, Canada
mieszko@ece.ubc.ca

Abstract

In this paper, we identify a previously untapped source of compressibility in cache working sets: clusters of cachelines that are similar, but not identical, to one another. To compress the cache, we can then store the "clusteroid" of each cluster together with the (much smaller) "diffs" needed to reconstruct the rest of the cluster.

To exploit this opportunity, we propose a hardware-level on-line cacheline clustering mechanism based on locality-sensitive hashing. Our method dynamically forms clusters as they appear in the data access stream and retires them as they disappear from the cache. Our evaluations show that we achieve 2.25× compression on average (and up to 9.9×) on SPEC CPU 2017 suite and is significantly higher than prior proposals scaled to an iso-silicon budget.

ACM Reference Format:
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 (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3373376.3378518

1 Introduction

Over the past decade, on-chip last-level cache (LLC) capacity has grown substantially, from 24MB to 64MB for SRAM [29, 48] and 30MB to 120MB for eDRAM [52, 54]. Due to the slowdown of Moore's Law, however, further growth of LLCs is becoming increasingly costly.

As an alternative, researchers have explored programmer-transparent *data compression* techniques for the LLC. These techniques generally fall into two classes: (a) intra-cacheline compression, which places multiple memory blocks within each cacheline [36, 38, 45], and (b) inter-cacheline deduplication [55], which helps capture data redundancy across cacheline boundaries by detecting identical cachelines and storing only a single copy.

In this paper, we show that compression can be significantly improved — to 2.25× geomean — by clustering memory blocks that have *nearly identical*, rather than exact, data values. We propose a dynamic inter-cacheline compression technique which uses dynamic clustering to efficiently detect and compress groups of similar memory blocks.

Limitations of prior approaches. LLC-based compression can work with two workload properties: (a) limited entropy in the cached values and (b) regularity in the structure in the organization of these data. For instance, intra-cacheline compression schemes exploit the first property: they take advantage of low entropy of data within a small memory block (e.g., a cacheline) by compressing each block independently [2–5, 10, 14, 34, 36, 38, 43, 45, 46]. These can work well when the working set consists of arrays of primitive data types with a relatively low range of values. However, they do not capture the structural properties of more substantial, heterogeneous data structures, whose redundancy surfaces only when considering multiple cachelines.

Inter-cacheline compression, on the other hand, can exploit such scenarios by detecting and exploiting structure regularity across cacheline boundaries [55, 56]. Unfortunately, state-of-the-art inter-cacheline compression methods have significant drawbacks. Proposals like exact deduplication can only exploit data regularity if multiple LLC lines have *exact* data values [55], while techniques that directly leverage program-level data structure information require pervasive program changes and ISA extensions [56]. Cache compression methods that return approximate values [40, 41] work well for noise-resilient data (e.g., images), but are unsuitable for general-purpose workloads. The compression scheme we propose in this paper overcomes these limitations.

Untapped potential. For architectures like CPUs, programmability and ease of use are important considerations. Thus, LLC compression techniques that are both programmer-transparent and general-purpose are preferable. Dedup [55] is the only inter-cacheline proposal that meets both criteria. Unfortunately, Dedup misses out on many compression opportunities because it requires multiple memory blocks to have exact data values.

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland

ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378518

To see how much opportunity is lost, consider an ideal inter-cacheline compression scheme that inserts data by searching the entire LLC for similar cachelines and stores only the bytes that differ from the most similar existing cacheline whenever this representation is smaller than an uncompressed cacheline; we refer to this setup as *Ideal-Diff*. Figure 1 shows the effective LLC capacity for (a) a system without compression, (b) an idealized deduplication scheme that also instantly searches the LLC for exact matches (*Ideal-Dedup*), and (c) *Ideal-Diff*, on SPEC CPU 2017 suite [7]. The potential of detecting and compressing similar lines is significant: *Ideal-Diff* increases the LLC capacity by 2.5× over the baseline (geomean), compared to only 1.3× for *Ideal-Dedup*.

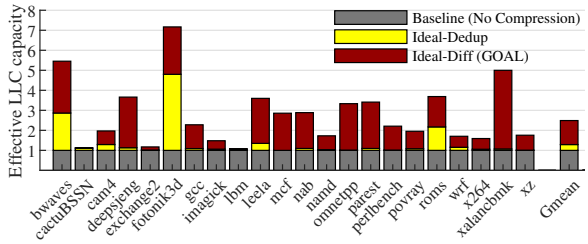
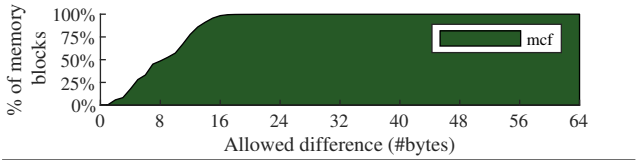


Figure 1. Effective LLC capacity from data compression by executing the SPEC-2017 suite [7]. On average, Ideal Deduplication (*Ideal-Dedup*) improves the effective LLC capacity by 1.3×. However, *Ideal-Diff*, that groups nearly identical memory blocks, can increase effective LLC capacity by 2.5×.

To achieve good compression with *Ideal-Diff*, the overheads of storing diffs must be low (i.e. the diffs must be relatively small). We observed that this tends to be true for a wide range of workloads. For example, Figure 2(top) illustrates this using an LLC snapshot of the *mcf* workload from SPEC CPU 2017 [7]. The working set contains very few duplicate memory blocks, making exact deduplication ineffective. Intra-cacheline techniques also have limited effectiveness, as the primary datatype contains a variety of fields with different types and ranges (see Listing 1).



l_1: 00002AAAC02419D8 00002AAAC0237610 0000000000000000 FFFFFFFFFECEF790 00006A04FFFFFE67 ···
l_2: 00002AAAC024B8B0 00002AAAC023EE60 0000000000000000 FFFFFFFFFECEF73C 00006A2CFFFFEEE7 ···
l_3: FFFFFFFFFFFFFFC4 00002AAAC11FE988 0000000000000000 0000000001312724 FFFFFFFFFECED8DC ···
l_4: FFFFFFFFFFFFFFC4 00002AAAC11FA920 0000000000000000 0000000013123A0 FFFFFFFFFECEDC60 ···

Figure 2. Top: Fraction of 64-byte cachelines in an LLC snapshot of *mcf* that can be deduplicated with at least one other cacheline *if* differences up to n bytes are permitted for $0 \leq n \leq 64$. Bottom: Two clusters of near-duplicate cachelines from *mcf*.

On the other hand, exploiting similarity across cacheline boundaries and relaxing the exact-duplicate requirement is very effective: almost *half* of the cached memory blocks differ from another block by a maximum of 8 bytes, and nearly *all* memory blocks differ only by a maximum of 16 bytes. Therefore, we can obtain significant LLC storage capacity by storing 16-byte diffs instead of full 64-byte blocks.

```
struct node {
    val (8 Bytes) potential;
    val (4 Bytes) orientation;
    ptr (8 Bytes) child, pred
    ptr (8 Bytes) sibling, sibling_prev;
    ptr (8 Bytes) basic_arc,firstout;
    ptr (8 Bytes) firstin, arc_tmp;
    val (8 Bytes) flow;
    val (8 Bytes) depth;
    val (4 Bytes) number;
    val (4 Bytes) time;
};
```

Listing 1. The node data structure in *mcf*

Challenges. Unfortunately, working sets usually do not contain a single reference memory block around which all other memory blocks could cluster; on the contrary, we may require *several* reference memory blocks with vastly different data values. This is the case in *mcf*: in Figure 2(bottom), lines $l_1 \ldots l_4$ all come from the same node data structure in *mcf*, but only $\{l_1, l_2\}$ and $\{l_3, l_4\}$ are near-duplicate pairs. This is because node takes up 68 bytes (see Listing 1) and is not aligned to the 64-byte cacheline size. The misalignment naturally creates several "clusters," each with its own reference memory block referred to as the "clusteroid." To achieve effective compression, therefore, multiple clusters must be identified; in addition, because the contents are input-dependent, this must happen dynamically at runtime.

Our proposal. This paper proposes Thesaurus, a cache compression method that relies on efficient dynamic "clustering" of memory blocks in the LLC. To form clusters dynamically, Thesaurus uses *locality-sensitive hashing* (LSH), which produces the same hash for similar blocks and different hashes for dissimilar blocks [25]. The LSH fingerprint of an incoming memory block becomes its "cluster ID": if other memory blocks with the same cluster ID are already cached, only the difference between the new memory block and an existing reference memory block for that cluster (the "clusteroid") is stored. Over time, as cluster members are evicted from the LLC, clusters that are not useful are naturally disbanded; this enables Thesaurus to adapt to changing workload phases by forming new clusters over time.

Broadly, this paper makes the following contributions:

1. We demonstrate significant similarity in the data values of memory blocks *across* different cachelines for a broad range of workloads.
2. We propose Thesaurus, an efficient LLC compression scheme based on clustering similar memory blocks using locality-sensitive hashing.

3. We develop practical dynamic cluster-detection hardware suitable for inclusion near the LLC, including a novel, hardware-friendly locality-sensitive hashing design.
4. We develop a replacement policy for the data array in the LLC that balances the development of new clusters with conserving existing clusters.

To the best of our knowledge, Thesaurus is the first LLC compression scheme based on dynamic clustering, and the first to leverage locality-sensitive hashing.

We evaluate Thesaurus on the SPEC CPU 2017 [7] suite. The cache footprints for these benchmarks are dominated by a range of different data structures. We show that Thesaurus compresses the cache footprint up to 9.9× (2.25× geomean) when compared to an LLC that does not employ compression — a substantial improvement over the 1.28× (geomean) achieved by the state-of-the-art inter-cacheline compression scheme Dedup [55] and the 1.48× (geomean) achieved with the state-of-the-art inter-cacheline scheme BΔI [38] given the same silicon area. The effective compression frees up space to cache additional data, allowing Thesaurus to achieve speedups up to 27% (7.9% on average) over an uncompressed baseline on the cache-sensitive benchmarks.

2 Background

In this section, we provide a brief background on last-level cache organization, and describe the key insights behind key prior LLC compression schemes.

2.1 Last-Level Cache (LLC) Organization

The Last-Level Cache (LLC) logically consists of several sets and each set contains multiple ways; modern LLCs have 4–8 ways per set. Physically, the LLC consists of tag and data arrays, usually with a dedicated tag and data array for each way in a set: e.g., an 8-way LLC has eight tag and eight data arrays. All ways in the set — and therefore all arrays — can be simultaneously queried during a cache access. Each cacheline in the data array is allocated one tag in its respective tag array: when an incoming memory block is placed in the LLC, it is assigned to the set that corresponds to its address, and replaces the tag and data entries for one of the ways. If this way previously contained valid data, this data is first evicted.

2.2 Intra-Cacheline Compression in LLC: BΔI

A simple technique to increase the capacity of the LLC is to employ intra-cacheline compression. In this scheme, each incoming memory block is examined in isolation, and, if possible, compressed independently of other blocks. For example, Base-Delta-Immediate (BΔI), a state-of-the-art intra-cacheline LLC compression technique [38], exploits the insight that, in many workloads, data values within a memory block are similar, and therefore can be compressed as a "base"

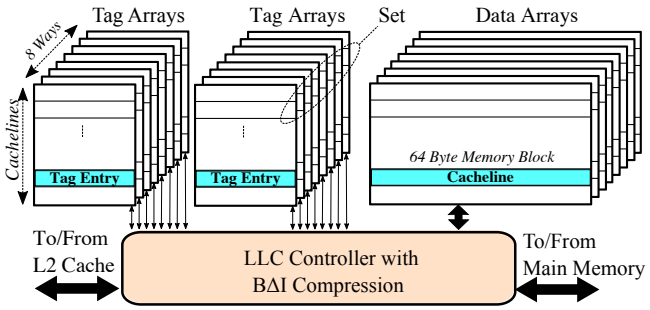


Figure 3. Last-Level Cache (LLC) implementing BΔI Compression. The LLC has multiple tags per cacheline to store additional tags for each of the compressed memory block within the physical cacheline.

value combined with small offsets. To store up to two memory blocks per cacheline, BΔI doubles the number of tag arrays for each way in the LLC, as shown in Figure 3.

2.3 Inter-Cacheline Compression in LLC: Deduplication

In contrast, the state-of-the-art inter-cacheline cache compression scheme Dedup exploits the existence of exactly identical memory blocks across the LLC in some workloads [55]. As shown in Figure 4, Dedup tries to store only a single copy of a block that would be represented by several copies in an uncompressed LLC. The key challenge here is that probing the entire cache to search for duplicates is impractical (unlike compressing a single cacheline, as BΔI does).

To overcome this limitation, Dedup uses a "hash table" that stores (say) 16-bit fingerprints of 64-byte memory blocks and their locations in the data array. While a "hit" must be verified against the actual 64-byte block to avoid false matches, in practice collisions are rare. Because cached values have some temporal locality, using a limited-size hash table with

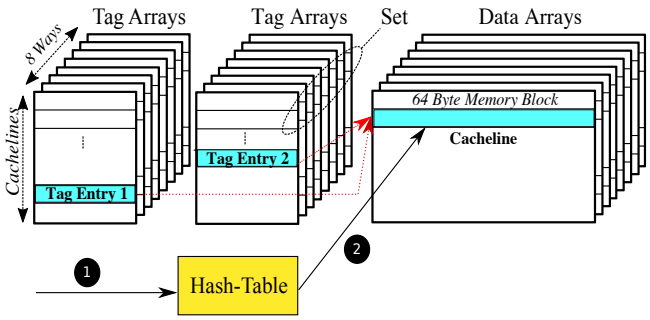


Figure 4. Last-Level Cache (LLC) implementing Deduplication (Dedup). Dedup enables multiple tags to point to the same cacheline containing a common memory block. During the LLC insertion of the memory block, Dedup uses a hash-table of the most recently used data values to identify exactly identical cachelines.

the most recently used fingerprints (say up to 1024 hashes) covers most of the duplication in typical working sets [55].

In addition to the limitations due to the exact-match requirement, Dedup has two performance challenges. One is that, as shown in Figure 4, data insertion involves a reference to the hash-table (action ❶) *followed by* a reference into the LLC data array (action ❷) to verify the exact contents of the memory block. Another limitation is that evicting a deduplicated memory block from the data array requires evicting *all* tags that point to it; in turn, this means that the tag array entries must contain two pointers to form a doubly-linked list for each deduplicated memory block value.

3 Motivation for In-Cache Clustering

To determine whether in-cache clustering is practical, and whether it should be dynamic, we asked three questions:

1. Do caches contain clusters with enough elements to provide substantial opportunities for compression?
2. Are there few clusters with clusteroids that could be hardcoded, or must clusteroids be computed at runtime?
3. Do cluster count and size vary among workloads enough to need a runtime-adaptive solution?

To answer these questions, we performed DBSCAN clustering [17] on LLC snapshots from the SPEC CPU 2017 suite, configuring similarity criteria for each workload to target 40% space savings. The experimental setup here is idealized in two ways: (a) the algorithm sees the entire LLC at once rather than each memory block separately at insertion time, and (b) DBSCAN uses far more computation and storage than is practical to implement within a cache controller.

Figure 5 shows that the LLC in most workloads has significant clusters of 10 or more members, with many exhibiting larger clusters of even 1,200 memory blocks (*povray*, *roms*). Because some workloads need many separate clusters to achieve substantial compression (e.g., *bwaves* and *cactuBSSN*), hard-coding cluster parameters in hardware is impractical. Finally, because cluster counts and sizes vary widely across the benchmark suite, clustering must be done dynamically at runtime, with performance considerations dictating a hardware-based or hardware-accelerated solution.

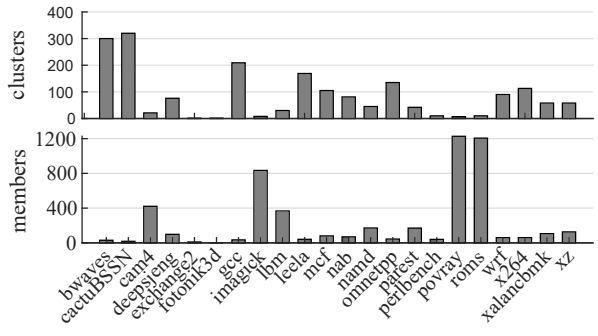


Figure 5. Cluster parameters after applying DBSCAN to LLC snaphots from different SPEC workloads. Workloads were run for 40B instructions after skipping the 100B instructions. Cluster distance was set to achieve an average 40% space savings in each snapshot.

4 Dynamic Clustering

Directly applying clustering techniques to cache compression is complicated by two challenges. Firstly, cache contents can change as often as every few cycles as lines are inserted and evicted, so there is never a stable image "snapshot" to be analyzed. Secondly, the need to incorporate clustering in a cache controller requires that is both relatively quick (on the order of a few cycles) and inexpensive to implement in hardware. These requirements exclude common clustering algorithms like k-means [32] or DBSCAN [17].

To overcome these challenges, we observe that an *approximate* clustering technique — one where a point is placed in the "correct" cluster with *high probability*, but can also end up in an entirely "wrong" cluster with *low probability* — is sufficient for cache compression. This is because the few lines that end up in the wrong cluster can simply be stored uncompressed; provided this happens rarely, compression ratio will not be significantly affected.

Thesaurus therefore uses a dynamic approximate clustering mechanism based on locality-sensitive hashing [25]. In this section, we will briefly discuss two key underlying concepts: (a) how locality-sensitive hashing can be used for approximate clustering, and (b) how locality-sensitive hashing can be efficiently implemented in hardware.

4.1 Locality-Sensitive Hashing (LSH)

LSH was initially developed as a data structure for the approximate-nearest-neighbour problem [25]. It has been especially popular in big-data environments, and used for streaming nearest-neighbour queries [31, 35], encrypted data search in cloud storage [58], detecting near-duplicate web pages [49], finding DNA patterns [8], unsupervised learning [22], computer vision [12], deep learning [16, 51], etc.

The idea is to create a family of hash functions that map points in some metric space to discrete buckets, so that the probability of hash collision is high for nearby points but low for points that are far away from each other. Specifically, given two points x and y in an d-dimensional real space $\mathbb{R}^d$ with a distance metric $\|x, y\|$, a family of hash functions $\mathcal{H}$ is called *locality-sensitive* if it satisfies two conditions:

1. if $\|x, y\| \leq r_1$, then $\Pr[h(x) = h(y)] \geq p_1$, and
2. if $\|x, y\| > r_2$, then $\Pr[h(x) = h(y)] \leq p_2$,

when h is chosen uniformly at random from $\mathcal{H}$, $r_1 < r_2$ are distances, and $p_1 > p_2$ are probabilities [25]. In the context of cache compression, we want a distance metric that (a) correlates with the number of bytes needed to encode

the difference between x and y, and (b) is easy to evaluate, the ℓ_1 metric being a natural candidate. Typically, we want $r_2 = (1+\varepsilon)r_1$ for some small $\varepsilon > 0$.

To see how such a hash family could be created, consider the space $\{0,1\}^d$ of d-bit strings under the Hamming distance metric, and, without loss of generality, choose two bit strings x and y in this space. Let $\mathcal{H} = \{h_1, \ldots, h_d\}$, where h_i simply selects the ith bit of its input. Intuitively, if $\|x, y\|$ is small (i.e., few bits differ), the probability that $h_i(x) \neq h_i(y)$ (i.e., selecting a different bit) with a randomly selected h_i will be small; in contrast, if $\|x, y\|$ is large (i.e., many bits differ), the probability that $h_i(x) \neq h_i(y)$ will be high. Observe that the difference between the two probabilities will be amplified if we again select an h_j at random and require x and y to match under both h_i and h_j to conclude that $h(x) = h(y)$.

The locality-sensitive hashing algorithm leverages this insight by mapping each point to an *LSH fingerprint* by concatenating the outputs of k randomly chosen functions in family $\mathcal{H}$. Typically, bit sampling is replaced with multiplication by a matrix randomly sampled from a suitably constructed normal distribution, as illustrated in Figure 6(left); such random projections preserve the distance between any two points to within a small error [19, 27]. By carefully selecting the LSH matrix, an arbitrarily high probability of finding a near neighbour within a chosen radius can be achieved (see [23] for details and a formal analysis).

4.2 Using LSH for Clustering and Compression

The fingerprint obtained by applying the chosen subset of $\mathcal{H}$ and concatenating the results naturally leads to a clustering scheme where all points with the same fingerprint are assigned to the same cluster. Crucially for cache compression, computing this fingerprint requires no preprocessing or queries of previously cached data.

At the same time, there are two challenges. One is that a cacheline may rarely be assigned to the "wrong" cluster, and be incompressible with respect to that cluster's clusteroid. Because this occurs very rarely, however, the effect on the overall compression ratio is negligible.

The other challenge is that cluster diameters vary across workloads (see Section 3), but combining LSH functions in a single fingerprint requires fixing the near-distance radius r_1 (see Section 4.1). Again, correctness is not compromised because the "misclassified" lines can be stored uncompressed; however, the near-distance radius must be carefully chosen to keep those events rare and provide good compression.

To effect compression, we must also select a base (clustroid) for each cluster: the base will be stored uncompressed, and lines in the same cluster will be encoded as differences with respect to this base (see Section 5.1). Because a clustering scheme based on LSH treats all points in a cluster equally and does not identify a true centroid, we simply choose the first cacheline to be inserted with a given LSH as the cluster base.

4.3 Hardware-Efficient LSH

A key disadvantage of the dimensionality reduction method for computing LSH fingerprints (see Figure 6(left)) is that applying the LSH matrix to the cacheline requires many expensive multiplication operations (e.g., 64 if we treat the cacheline as a byte vector); directly implementing this would incur unacceptable overheads in silicon area, latency, or both.

The hardware-efficient LSH implementation in Thesaurus combines two separate refinements of random projection. The first is that multiplication can be avoided by replacing the elements of the LSH matrix with +1, 0, or −1, chosen at random with probabilities $1/6$, $2/3$, and $1/6$ respectively, with negligible effects on accuracy [1]. Indeed, the sparsity can be further improved by reducing the probabilities of non-zero values to $d/\log(d)$ (where d is the number of dimensions in the original space), again at negligible accuracy loss [30]; this allows for very efficient hardware implementations [18]. (Refer to [1, 30] for a formal analysis of these optimizations.)

To reduce the resulting LSH fingerprints from many bytes to a small number of bits, we combine this with another refinement: the idea that each component of the LSH fingerprint vector can be replaced with 1 if it is positive or 0 if it is negative while retaining the chosen LSH probability bounds [9]. Besides resulting in small fingerprints, this allows us to select the fingerprint size at bit granularity by simply varying the number of hash functions used (i.e., number of LSH matrix rows).

Figure 6(centre) illustrates the LSH fingerprint computation in Thesaurus. The cacheline is first multiplied by a sparse matrix with entries from $\{-1, 0, 1\}$; then, only the sign of each scalar is retained, resulting in a fingerprint bit vector. Figure 6(right) illustrates the hardware implementation using only adders and comparators.

5 The Thesaurus Architecture

Briefly, Thesaurus operates by applying the LSH hash as each memory block is inserted. When another "base" memory block with the same LSH exists, the incoming memory block is stored as a byte-level difference with respect to the base (if the difference is small enough to result in compression); if no other memory blocks share the LSH, the incoming memory block becomes the new "base" for the cluster.

Below, we first outline the Thesaurus compression format and storage structures, then walk through an example, and finally detail how Thesaurus operates.

5.1 Compression Format

Thesaurus uses two primary data encodings: a compressed BASE+DIFF format, and an uncompressed RAW format used when compression is ineffective. We also use secondary data encodings to optimize for three common-case patterns: ALL-ZERO lines, BASE-ONLY for lines that do not need a diff, and 0+DIFF for lines that do not need a base.

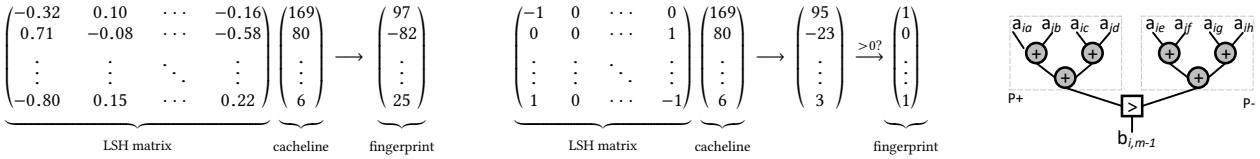


Figure 6. Left: computing the fingerprint of a cacheline using dimensionality reduction. Centre: a hardware-friendly variant developed for Thesaurus. Right: Hardware implementation using an adder tree and a comparator.

In the BASE+DIFF encoding, illustrated in Figure 7, memory blocks within a cluster are represented as a compacted byte difference with respect to a base memory block common to the entire cluster. The encoding consists of a 64-bit mask that identifies which bytes differ from the base ❷, followed by a sequence of the bytes that differ ❶. During decompression, the "base" and the compressed "diff" encoding are combined by replacing the relevant bytes ❸.

Lines encoded as ALL-ZERO are identified as such in the tag entry (see Section 5.2.1), and require no additional storage. Similarly, BASE-ONLY lines are equal to the cluster base, and so do not need a diff entry in the data array. Finally, 0+DIFF lines are encoded as a byte-difference from an all-zero cacheline.

5.2 Storage Structures

Figure 8 shows the storage structures used to implement Thesaurus and the connections among them. As is typical in prior compressed cache proposals [39, 55, etc.], the tag array and the data array are decoupled to enable the storage of a larger number of tags as compared to data entries.

Thesaurus stores the clusteroids for each possible LSH fingerprint in main memory, and caches the most recently used clusteroids within a small LLC-side structure similar to a TLB, which we refer to as the *base cache*.

5.2.1 Tag Array. The tag array is indexed by the physical address of the LLC request, and entries are formatted as shown in Figure 9. The *tag*, *coh*, and *rpl* fields respectively

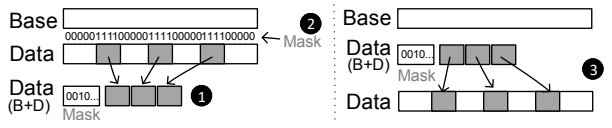


Figure 7. The BASE+DIFF compression encoding in Thesaurus. Left: compression; right: decompression.

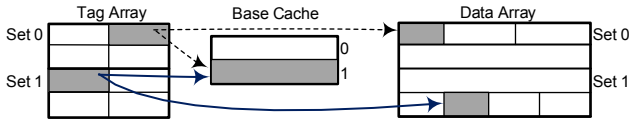


Figure 8. A two-set, two-way Thesaurus cache with two memory blocks cached in the BASE+DIFF format. The entries share the same base but have different diffs.

correspond to the tag, coherence state, and replacement policy state of a conventional cache. The new *lsh* field identifies the LSH fingerprint for the cached data, which points to the clusteroid for this LSH in the base table (see Section 5.2.3). The *setptr* field points to a set in the data array, whereas *segix* identifies the segment within the set (see Section 5.2.2 for details). Finally, the *fmt* field identifies the cacheline as an ALL-ZERO cacheline, a BASE+DIFF encoding, or an uncompressed RAW cacheline.

5.2.2 Data Array. As in a conventional cache, the data array is organized in individually indexed sets. To facilitate the storage of variable-length compressed diffs, however, each set is organized as a sequence of 8-byte segments: a single data array entry (i.e., cacheline in BASE+DIFF or RAW format) may take up anywhere from two to eight segments. To avoid intra-set fragmentation, segments in a set are compacted on eviction as in prior work [38, etc.].

As the data array is decoupled from the tag array, any set in the data array can store the incoming memory blocks. Therefore, each data array entry contains a *tagptr* that identifies the corresponding tag array entry. This entry is used to evict the tag if the data array entry is removed to free space for an incoming memory block.

Each set also contains a map called the *startmap*. The startmap helps identify which segments begin new entries; this enables intra-set compaction without the need to traverse the tag array and modify the (possibly many) tag entries to reflect new data locations. The startmap has as many entries as there are segments, where each entry is one of VALID-RAW, VALID-DIFF, or INVALID (128 bits total).

The startmap works in conjunction with the *segix* field in the tag array: segix identifies the *ordinal* index in the set (e.g., first, second, *n*th, etc.), while the startmap identifies which entries are valid. The location of the *n*th entry is obtained by adding the sizes of the first $n-1$ VALID-RAW or VALID-DIFF startmap entries. Because evicted entries can set their

Figure 9. Top: Data array entries for uncompressed data (left) and the BASE+DIFF/0+DIFF encodings (right). Bottom: Tag entry format (left) and the base table entry that contains base (right).

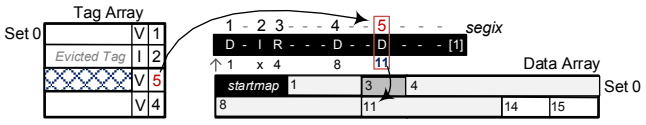


Figure 10. The segment index (here, 5) and the startmap combine to locate the compressed data block within a set.The second entry (shaded grey) is invalid, so it is skipped in the startmap. D=VALID-DIFF, R=VALID-RAW, I=INVALID.

startmap tags to INVALID without affecting the segix for the following entries, sets can be recompacted without updating the tag array.

Entries can come in two flavours: RAW and BASE+DIFF. Lines stored in the RAW format (Figure 9, top-left) contain a 15-bit tag pointer followed by 64 bytes of data across eight contiguous segments. Lines in the BASE+DIFF format (Figure 9, top-right) also begin with a 15-bit tag pointer; this is followed by a 64-bit map that identifies which bytes differ from the base, and then by a sequence of the differing bytes.

5.2.3 Base Table and Base Cache.

To store the clusteroid (base memory block) for each LSH fingerprint, Thesaurus uses a global in-memory array allocated by the OS, which we refer to as the *base table*. Each base table entry contains a counter of how many current cache entries are using this base. When the counter decreases to 0, the base is replaced with the next incoming cacheline for that LSH, which allows Thesaurus to adapt to changing working sets.

For performance, the base table is cached in a TLB-like table near the LLC, which we refer to as the *base cache*; for us, this is an pseudo-LRU-managed, 64-set, 8-way set-associative structure. The entries in this cache contain the base entry itself, an LSH tag, and replacement policy state.

5.3 Walk-through Examples

Figure 11(b) shows an example lookup the Thesaurus LLC. First, as in a conventional cache, the address is used to index the tag array ❶; in this example, the cacheline uses the BASE+DIFF encoding. The LSH stored in the tag entry is used to index into the base cache and retrieve the compression base ❷. Concurrently, the set index from the tag entry is used to retrieve the set. The segment index from the tag entry and the startmap from the data entry are combined to identify the beginning of the encoded cacheline in the set ❸. Finally, the bitmask and bytes stored in the diff entry are used to replace the corresponding bytes from the base entry, and the resulting line is returned ❹.

Figure 11(c) illustrates how the startmap is updated during an eviction, showing a set before and after evicting entry *d1*. Before the eviction, *d1* is the second VALID index in the startmap ❶, *d2* is the third, and so on. After the eviction, the tag array entry for *d*1 has been invalidated ❷, the other entries (*d0*, *d2*) have been compacted to form a contiguous set, and the startmap entry that previously identified *B* has

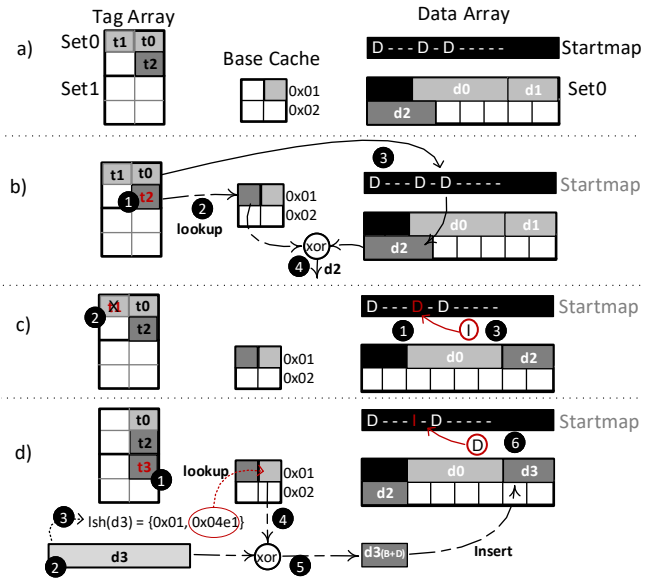


Figure 11. Thesaurus structures during cache operations: (a) initial state; (b) read request processing; (c) eviction; (d) new entry insertion.

become INVALID ❸. This means that *d2* is still the third overall entry, and the tag array entries for *d2* do not need to be updated to reflect the compaction.

Finally, Figure 11(d) shows an insertion of a new entry *d3*. First, the access misses in the tag array and a request is made to the backing memory ❶. When the cacheline data arrives, it is immediately returned to the requesting cache (e.g., L2) ❷. In parallel, the cacheline's LSH fingerprint is computed ❸ and used to index the base cache. In our example this access hits and returns the data for the base ❹. The incoming line is then XOR'd with the base, and the non-zero bytes of the resulting difference are encoded as a bitmask and a list of differing bytes ❺; in the example, this encoding takes up 16 bytes, or two segments. Next, this entry is inserted in the cacheline from the previous example containing *d0*, *d2*, and *d3*: the INVALID startmap entry is replaced by VALID-DIFF ❻, and the entry is inserted in the corresponding sequence in the set ❼.

5.4 Operational Details

5.4.1 Read Requests.

Figure 12 shows how Thesaurus services a read request. Shaded areas are on the critical path, while unshaded areas occur after the read has been serviced.

When the request is received, the address is looked up in the tag array as in a conventional cache. If the tag hits, the *setptr* is used to index the data array ❶ (for 0+DIFF and BASE+DIFF formats) and, in parallel, the *lsh* indexes the base cache ❷ (for BASE-ONLY and BASE+DIFF formats). If the LSH is not in the base cache, an access is made to the memory to retrieve the base entry for that LSH (not shown).

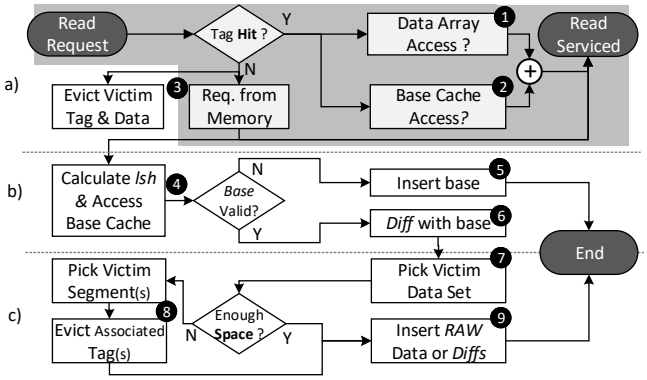


Figure 12. Processing a read request in Thesaurus: (a) critical-path lookup sequence; (b) cluster ID computation for new data brought in by a miss; (c) insertion of new data and possible data array evictions. Shaded steps are on the critical path, while unshaded steps are performed in parallel with servicing the read request.

If the tag misses, a request is made to the backing memory ❸; meanwhile, the victim tag and its corresponding data array entry are evicted and the new tag is inserted. The data is returned to the requesting cache as soon as it arrives; inserting the new line in the cache happens off the critical path as other read requests are handled.

If the newly arrived line consists of zeros, an ALL-ZERO tag is inserted and processing ends. Otherwise, the LSH for the newly arrived line is computed ❹, and used to index the base cache; if the LSH is not in the base cache, the data is inserted uncompressed (in RAW format) while the base is retrieved from the base table in memory (not shown). If there is currently no base for the LSH, the new line is installed as the base and processing ends ❺ by inserting a BASE-ONLY tag entry. Otherwise, the byte-difference with respect to the base is calculated ❻; if there are no differences, the a BASE-ONLY tag is inserted and no entries in the data array are made.

For non-base entries, the diff is packed together with a bitmask in the BASE+DIFF or 0+DIFF format; if compression is not possible, the entry will use the RAW format. In either case, the appropriate tag is inserted, and a block must be added to the data array. To make space, a data array victim set is selected ❼ as described in Section 5.4.3; if there is not enough space there, enough victim segments are selected to make space for then new block, and their tags evicted ❽. Finally, the block is inserted, possibly recompacting the set ❾.

5.4.2 Write and Atomic Requests. Accesses that modify the data can change the mode of compression or the size of the compressed block. If the diff is smaller, the data array entry is either removed (for ALL-ZERO or BASE-ONLY) or the block's bitmap is updated and the set is compacted. If the diff is larger, other entries are evicted from the set to make space ❽, and the set is updated and compacted.

For write operations in memory models without write acknowledgements (most extant ISAs), the entire write is performed off the critical path. For atomic read-modify-write operations (e.g., compare-and-swap), the read part is serviced as soon as possible, and the write part is completed off the critical path.

5.4.3 Replacement Policies. The tag array follows the corresponding conventional replacement policy (in this paper, we use pseudo-LRU); the base cache follows pseudo-LRU. Unlike in a conventional cache, however, the data array entry requires a separate replacement policy: in this case, a policy that favours evicting fewer data entries over recency makes sense, as not-recently-used data array entries will have been evicted anyway by the tag array replacement policy.

To choose a victim set, we use a best-of-n replacement policy [21, 55]. First, we randomly select four sets. If one of the sets has enough free segments to store the incoming (possibly compressed) block, it is chosen and no evictions are made; otherwise, we select the set with the fewest segments that would have to be evicted to make enough space.

Observe that the randomness ensures that frequently used blocks do not evict each other in a pathological pattern: if a block is evicted and soon thereafter reinserted, it will likely end up in a different set than the block that evicted it.

6 Evaluation

6.1 Methods

To evaluate effects on cache behaviour and performance, we implemented Thesaurus and the comparison baselines in the microarchitecture-level simulator ZSim [42]. We simulated an out-of-order x86 core similar to an i5-750, modelling on- and off-critical-path events as well as limited interconnect bandwidths; the simulated system is shown in Table 1. Compression was applied at the LLC level only.

To estimate silicon area and power impacts, we implemented all logic that is required in Thesaurus but not in a conventional cache in Verilog RTL, and synthesized these with Synopsys Design Compiler tool using FreePDK45 [53] standard cell library. We used CACTI 6.5 [33] to estimate the area, access time, and power of storage structures.

As the baseline, we modelled a conventional (uncompressed) LLC with 1MB capacity per core; we also modelled a hypothetical LLC with 2× the capacity (2MB), which has an

CPU	x86-64, 2.6GHz, 4-wide OoO, 80-entry ROB
L1I	32KB, 4-way, 3-cycle access lat., 64B lines, LRU
L1D	32KB, 8-way, 4-cycle access lat., 64B lines, LRU
L2	Private, 256KB, 8-way, 11-cycle lat., 64B lines, LRU
LLC	Shared 1MB, 8-way, 39-cycles lat., 64B lines, 8 banks
Memory	DDR3-1066, 1GB

Table 1. Configuration of the simulated system.

		Conv.	BΔI	Dedup	Thesaurus
Tag	#Entries	16384	32768	32768	32768
	Entry Size	37b	47b	81b	72b
	Total Size	74KB	188KB	324KB	288KB
Data	#Entries	16384	14336	11700	11700
	Entry Size	512b	512+0b	512+16b	512+32b
	Total Size	1024KB	896KB	754KB	777KB
Dict.	# Entries	-	-	8192	512
	Entry Size	-	-	24b	24+512b
	Total Size	-	-	24KB	33KB
	Total Size	**1.07MB**	**1.06MB**	**1.07MB**	**1.07MB**

Table 2. Storage allocation. All compressed caches were sized to fit in the same silicon size of a 1MB conventional cache with a 48-bit address space. The *Dict.* category accounts for the hash array required by Dedup and the base cache required by Thesaurus.

effective capacity similar to a 1MB Thesaurus cache. To study relative improvements over prior state-of-the art intra- and inter-cacheline compression, we also implemented BΔI [38] and Dedup [55]. All compressed configurations (Thesaurus, Dedup, BΔI) were sized to occupy the same silicon area as the baseline LLC: Table 2 compares the storage allocations.

To find a suitable LSH size, we swept sizes of 8–24 bits. We found that 12-bit LSHs result in good compression for most workloads while keeping the base table size low.

We evaluated all designs on the SPEC CPU 2017 suite [7]. For each benchmark, we skipped the first 100B instructions, and executed the last 20% of each 1B instructions. For miss rate and speedup measurements, we split the benchmarks into cache-sensitive (S) and cache-insensitive (NS). In our evaluation, a benchmark was considered cache-sensitive if doubling the cache size to 2MB improves the MPKI by more than 10%. (In a practical implementation, the LLC could dynamically detect cache-insensitive workloads by measuring average memory access times and disable LLC compression.)

6.2 Compression, Miss Rates, and Speedup

Figure 13 shows the improvements over the uncompressed baseline and state-of-the-art compressed caches. We also compare against the ideal clustering method (which searches the entire cache for the nearest match and diffs against it in one cycle) and a conventional cache with 2× the capacity.

Figure 13(a) shows the effective cache footprint reduction — i.e., the data array space taken up by the cached addresses normalized to the equivalent space that would have been required to hold the same addresses in a conventional cache. Overall, Thesaurus compresses the working sets by 2.25×, compared to 1.28× for exact deduplication and 1.48× for BΔI compression (all geomeans); this demonstrates that Thesaurus compresses more effectively than state-of-the-art cache compression techniques.

The LSH scheme in Thesaurus also captures nearly all data that can be effectively clustered: the cache footprint obtained using Thesaurus is within 5% of the ideal clustering scheme, which searches the entire cache for the nearest match. In a few cases, compression is, in fact, slightly *better* (e.g., *povray*, *perlbench*, *gcc*): this is because Thesaurus can diff against a clusteroid whose tag has since been evicted from the cache, while the ideal clustering model is restricted to currently cached entries.

Figure 13(b) shows that Thesaurus substantially reduces miss rates: for the cache-sensitive subset of the suite, MPKI drops to 0.78 of the conventional cache compared to 0.98 for exact deduplication and 0.89 for BΔI (all geomeans). Thesaurus is also within 1.5% of the ideal clustering model, and within 8% of the MPKI that can be attained with a conventional cache with 2× capacity. This is because, thanks to the effective compression, more data can be cached within the same silicon area, which benefits cache-sensitive workloads.

Finally, Figure 13(c) shows that the reduced MPKI rates result in execution time speedups over the baseline as well as Dedup and BΔI. Thesaurus is up to 27.2% faster than the conventional baseline (7.9% geomean), and up to 9.1% faster than BΔI (5.4% geomean). Indeed, performance is within 1.1% of the ideal clustering model, and within 2.2% of a conventional cache with 2× capacity.

6.3 Cost Analysis

Power. We used CACTI 6.5 [33] to estimate the read energy and leakage power of all cache structures; the results are shown in Table 3. While Thesaurus uses ∼12% more energy for each read and has a ∼14% leakage power overhead, these overheads are significantly lower than ∼57% increase in dynamic energy and ∼70% increase in leakage power for conventional cache with the same effective capacity.

Most importantly, the overhead of Thesaurus (∼0.06nJ per access at the 45nm node) is trivial compared to the energy of accessing external DRAM (32.61nJ using the same CACTI model). This means that Thesaurus can actually *save* energy when the entire memory hierarchy is considered. In order to measure this, we calculated the total added power of compressed cache (30.54mW + 6.4mW + 0.06nJ × access rate) and total power saved by accessing off-chip DRAM less frequently (32.61 nJ × access rate difference between Thesaurus and uncompressed).

While the power consumption of the Thesaurus LLC increases (from 36.87mW to 51.28mW) because of the added logic and more data array reads (due to the higher LLC hit rate), accounting for DRAM accesses results in power consumption savings of up to 101mW in the cache sensitive benchmarks. Cache-insensitive benchmarks do not see fewer off-chip DRAM accesses despite effective compression, and therefore power overheads are not outweighed by power savings; however, a practical implementation would detect

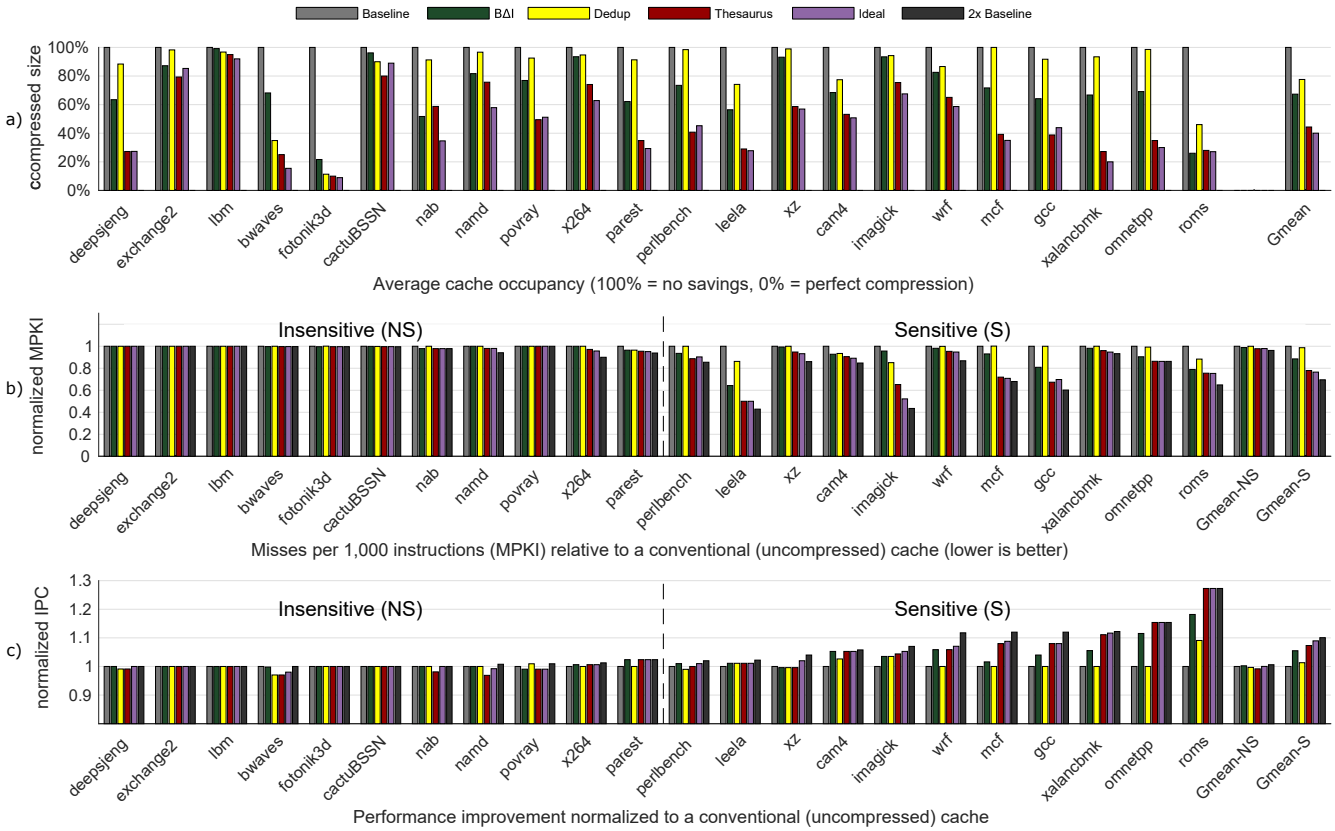


Figure 13. Compressed working set size, cache miss rates, and performance improvements of Thesaurus compared to baseline (uncompressed) cache, iso-silicon BΔI (intra-block) and Dedup (inter-block), Ideal-Diff, and an uncompressed cache with 2× capacity. All but ideal and 2× baseline are sized to the silicon area of the uncompressed cache.

cache-insensitive workloads and simply disable compression for cachelines they access.

Latency. We modelled access times to each LLC structure using CACTI; because compressed caches have smaller data array sizes, its overall access time is slightly reduced (~2% for Thesaurus). To measure compression and decompression latencies, we implemented the logic for compression, decompression, as well as locating and reading the compressed cachelines in 45nm ASIC; the results are shown in Table 4. At the relevant CPU frequency (2.66GHz), compression and decompression take one cycle each, while locating the compressed data block in the set (described in Section 5.2.2) takes four more cycles. This brings the total decompression latency to 5 cycles, which we used for the performance simulations.

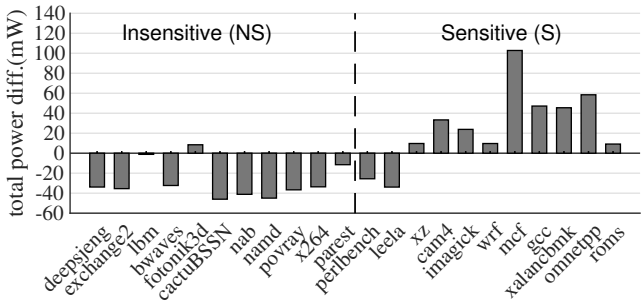


Figure 14. Difference in total power consumption using Thesaurus compared to the baseline. Positive values indicates less power consumption whereas negative values shows additional power consumption.

	45nm		32nm	
	dynamic energy	leakage power	dynamic energy	leakage power
Conv.	0.50 nJ	205.47 mW	0.28 nJ	109.96 mW
BΔI	0.55 nJ	196.47 mW	0.31 nJ	105.22 mW
Dedup	0.56 nJ	226.33 mW	0.32 nJ	121.06 mW
Thesaurus	**0.56 nJ**	**236.01 mW**	**0.31 nJ**	**125.85 mW**
Conv. 2×	0.78 nJ	349.21 mW	0.44 nJ	186.50 mW

Table 3. Dynamic read energy and leakage power per bank of compressed and conventional caches scaled to the same silicon area (1MB uncompressed = 5.56mm^2 in 45nm or 2.82mm^2 in 32nm).

Area. The logic required for Thesaurus incurs an area overhead of 0.06mm^2 in 45nm: this includes the compression (0.016mm^2) and decompression (0.013mm^2), the logic to locate the segments in the set using the indirect segix encoding

	latency	dynamic power	leakage power	area
comp.	1 cycle	0.116 mW	2.44 mW	0.016 mm^2
decomp.	1 cycle	0.084 mW	1.74 mW	0.013 mm^2
segix	4 cycles	0.035 mW	0.49 mW	0.007 mm^2
multi-bank	-	0.101 mW	1.42 mW	0.025 mm^2

Table 4. Synthesis results for the added logic area of Thesaurus: *segix* refers to locating the compressed block within a set (decoding the indirect segix format), while *multi-bank* refers to the muxing needed to access lines across multiple banks; 64-byte cachelines were used. Latency is in units of CPU cycles at the 2.66GHz frequency of the equivalent 45nm i5-750 core. All results obtained Synopsys DC and the 45nm FreePDK.

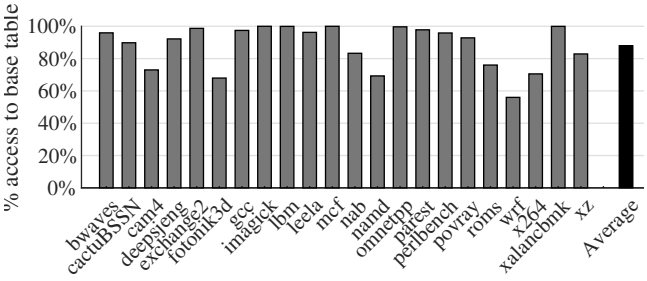


Figure 15. Fraction of cache insertions that are potentially compressible with respect to their clusteroid (avg. 87%).

(0.007mm^2), and the additional muxing needed to read a set across multiple banks (0.025mm^2). This is equivalent to 1% of the silicon area required for even a 1MB cache, and a tiny fraction of a 4-core i5-750 in the same 45nm node (296mm^2).

To compare with prior compressed caches, we also implemented and synthesized the BΔI scheme [38], which at 0.037mm^2 (20k NAND gates) is slightly smaller than Thesaurus (0.06mm^2 or 32k NAND gates); this is not surprising as BΔI offers much less compression. We also used NAND-gate estimates for prior work from [10, 56] to compare against other prior compression schemes, all of which incur more area overhead than Thesaurus: C-PACK [10] needs 0.075mm^2 in 45nm (40k NAND gates) and FPC [2] needs 0.544mm^2 (290k NAND gates) just for decompression [10], while BPC [28] needs 0.127mm^2 (68k NAND gates).

6.4 Clustering and Compression Details

To investigate the effectiveness of cacheline clustering based on locality-sensitive hashing, we first examined how many cache insertions are, in fact, compressed. Figure 15 shows that, on average, 87% (a significant majority) of cache insertions can potentially result in compression — i.e., their differences versus the relevant clusteroid are small enough that encoding them would take < 64 bytes even with the overhead of the difference bitmask. This observation validates both our choice of LSH for clustering and the effectiveness of our clusteroid selection strategy to increase compressibility.

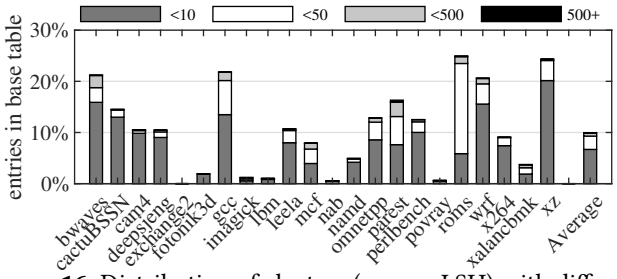


Figure 16. Distribution of clusters (= same LSH) with different sizes (average over the runtime of each benchmark).

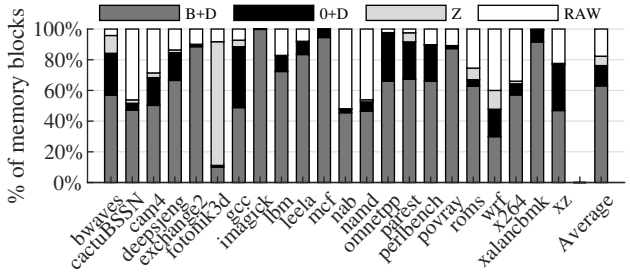


Figure 17. Frequency of different compression encodings in compressing benchmarks from SPEC. B+D=BASE+DIFF; 0+D=0+DIFF; RAW=uncompressed; Z=ALL-ZERO.

Figure 16 shows that most working sets have many small clusters rather than few large clusters. Nevertheless, because Thesaurus tracks clustroids for many LSH fingerprints, effective compression can still be obtained.

Next, we examined the compression encodings used by Thesaurus. Figure 17 shows that different workloads tend to benefit from different encodings. For most of the benchmarks, the byte-difference-based encodings are the most effective: the BASE+DIFF encoding covers an average of 76.2% of LLC insertions, while 0+DIFF covers a further 13.2%. Another 6.1% are covered by the ALL-ZERO encoding. Finally, 17.7% of LLC insertions are left uncompressed as the diff from the clusteroid (base) is too large.

Figure 18 shows the average size of the byte-difference block for the BASE+DIFF and 0+DIFF encodings. In most cases, the differences tend to be small, with a quarter of the benchmarks averaging about 8 bytes or less, and half of the benchmarks averaging about 16 bytes or less. This confirms that, for many benchmarks, caches have relatively tight data clusters with very small intra-cluster differences even at cacheline granularity.

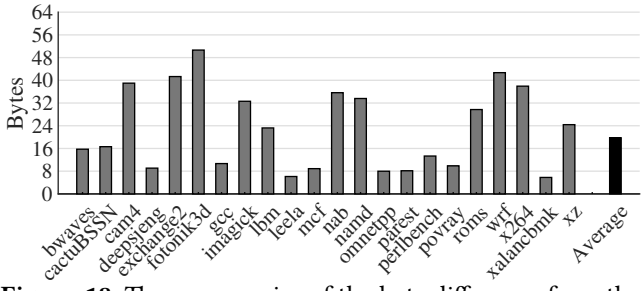


Figure 18. The average size of the byte difference from the relevant clusteroid for BASE+DIFF and 0+DIFF, in # bytes.

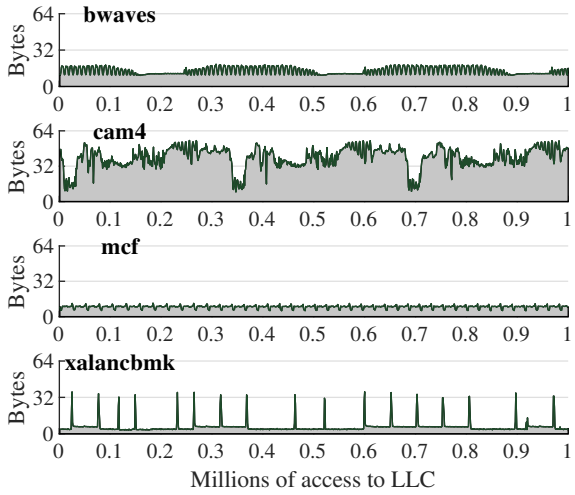


Figure 19. How the diff size varies over time: 1 million cache insertions after skipping the first 40B instructions.

In fact, diff sizes can change significantly over the run time of even a single workload. Figure 19 shows the diff sizes for 1 million cache insertions after the first 100 billion instructions for four workloads. In *mcf*, the data compression ratios are relatively stable; in *xalancbmk*, the tiny diffs of most accesses are punctuated by rare spikes of 32-byte diffs; *bwaves* has two distinct but small diff sizes; finally, *cam4* intersperses blocks that offer little compression with periodic short bursts of compressible data.

Together with the number of insertions that can be compressed (Figure 15), the diff sizes explain the compression ratios for each benchmark. For example, more than 90% of inserted blocks in *imagick* can be compressed, but the average diff size is 32.6 bytes, resulting in a compression ratio of 1.3× (see Figure 13). In contrast, *xalancbmk* and *mcf* combine a high (> 90%) proportion of compressible insertions with a small average diff size (6 and 9 bytes, respectively), for a total compression factors of 2.6× and 3.7×.

Finally, we established an efficient size for the base cache and examined its effectiveness. To establish size, we swept sizes ranging from 32 to 2048 entries; results are shown in Figure 20. Compared to a 94.8% hit rate for a 512-entry cache (33KB), a 2048-entry cache (+100KB storage) increases the hit rate by only 3.9%; we therefore used a 512-entry base cache for the remainder of our experiments.

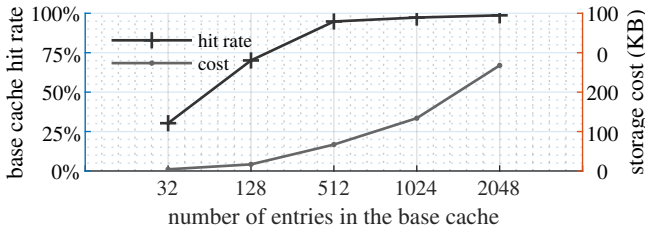


Figure 20. Base cache hit rate (left axis) and storage cost (right axis) for different base cache sizes.

On average, this cache has a 5.2% miss rate over all benchmarks; however, all but 8% of misses (i.e., all but 0.5% of accesses) miss when a line is being *inserted* in the cache, and are off the critical path. Because the data is inserted uncompressed while the clusteroid (base) is fetched into the cache, these misses represent a missed compression opportunity but do not affect insertion latency.

Nevertheless, the lost compression opportunities due to base cache misses can have a significant impact. The benchmarks with high off-critical-path base cache miss rates — *bwaves*, *nab*, *namd*, *x264* and, *wrf* with 8–13% — also lose the most compression opportunity compared to the idealized *Ideal-Diff* clustering (cf. Figure 13). For those workloads, a larger base cache would improve compression.

7 Related work

Prior work on cache compression can generally be categorized into three categories based on their compression granularities: (i) inter-block data compression, (ii) intra-block data compression, and (iii) techniques that do not operate at block granularity. Below, we briefly outline past proposals in all of these categories, and discuss prior work on orthogonal ideas on effective replacement policies with compression.

Inter-Block Data Compression. Inter-block data deduplication techniques leverage the observation that many cache blocks are either entirely zero [14, 15, 38] or are copies of other blocks that concurrently reside in the cache [11, 13, 24, 50, 55]. Instead of storing several identical copies, they aim to store only one copy of the block in the cache, and propose techniques to point the redundant data entries to this single copy.

To address the inter-block redundancy at the cache level, Dedup [55] modified a conventional cache to store one copy of the redundant data and allow multiple tags to pointing to the unique copy. Although the tag array is still arranged in sets and ways, the data array is decoupled from the tag array, and is designed to be explicitly accessed by pointers. In order to detect duplication, an augmented hashing technique is used for faster duplication detection. Thereafter, a quick look-up occurs in the hash table indexed by the hashed data.

Intra-Block Data Compression. For some applications, data values stored within a block have a low dynamic range resulting in redundancies [2, 38, 57]. Prior work categorized these into (a) repeated values (especially zeros) repeated in a data block, and (b) *near* values with the same upper data bits and different lower bits.

One way to reduce redundancy within the memory block is to capture the replicated data in dictionary entries and then point to that entry when new replicated data is presented. Frequent pattern compression [2] does this on a word-by-word basis by storing the last 16 observed values as a dictionary. Similarly, [57] uses an LZ77-like compression algorithm by reading through the input data word by word

and constructing a dictionary of observed sequences. The authors of [26] propose a small cache placed alongside the L1 data cache to store memory locations with narrow values; this compactly stores each 32-bit word at 1-, 2-, 4-, and 8-bit granularity.

Another method to reduce redundancy of nearly identical values is to try to separate repeated parts of values from distinct lower bits in a memory block. DISH [36] extracts distinct 4-byte chunks of a memory block and uses encoding schemes to compress them, with dictionaries potentially shared among a few contiguous blocks. It uses a fixed-width pointer that points to one of the n dictionary entries: i.e., a cache block is encoded as a dictionary, some fixed-width pointers, and some lower-bit deltas for each 4-byte chunk.

BΔI [38] uses one word-granularity "base" value for each compressed cache block, and replaces the other words in the block with their distances from the base value. BΔI is can compress zero lines, as well as various combinations of base value and offset sizes; the type of compression selected is encoded in the tag entry metadata. A data block is logically divided into eight fixed-size segments, and compressed blocks are stored as multiple segments allocated at segment granularity.

SC^2 [5] uses Huffman coding to compress memory blocks, and recomputes the dictionary infrequently, leveraging the observation that frequent values change rarely. HyComp [4] combines multiple compression algorithms and dynamically selects the best-performing scheme, based on heuristics that predict data types. Bit-Plane Compression [28] targets homogeneous arrays in GPGPUs to both improve the inherent data compressibility and to reduce the complexity of compression hardware over BΔI by compressing the deltas better. To reduce tag overhead of the compressed cache, DCC [47] and SCC [44] use "superblocks" formed by grouping adjacent memory blocks in the physical address space. More recently, YACC [45] was proposed to reduce the complexity of SCC by exploiting spatial locality for compression.

Broadly, intra-block methods are useful in compressing one block or possibly a "superblock" of contiguous memory blocks. However, they do not consider value redundancy among different non-contiguous memory blocks at far-away addresses, which still leads to repeated (albeit potentially compressed) data values in different parts of the cache.

Non-Block-Granularity Compression. Unlike scientific applications, whose working sets are often dominated by arrays of primitive-type values, many general-purpose applications traverse and operate on blocks. Based on this insight, Cross-Block-Compression algorithm (COCO) [56] uses data structure blocks (rather than cache blocks) as the unit of compression. The authors also present the first compressed memory hierarchy designed for block-based applications.

Replacement Policies With Compression Prior works have also looked at the impact of compression on cache replacement policy. ECM [6] reduces the cache misses using Size-Aware Insertion and Size-Aware Replacement. CAMP [37] exploits the compressed cache block size as a reuse indicator. Base-Victim [20] was also proposed to avoid performance degradation due to compression on the replacement. These proposals are effective for intra-cacheline compression, but do not consider the inter-cacheline interactions present in Thesaurus and Dedup [55].

8 Summary

In this paper, we have demonstrated that inter-cacheline compression can be very effective if small differences among the cached memory blocks are allowed. We proposed Thesaurus, an LLC compression based on dynamic cluster detection, and described an efficient implementation based on locality-sensitive hashing. Across SPEC CPU 2017, Thesaurus compresses LLC working sets 2.25× (geomean), compared with 1.28×–1.48× (geomean) achieved by state-of-the-art LLC compression schemes.

9 Acknowledgements

The authors gratefully acknowledge the support of the Natural Sciences and Engineering Research Council of Canada.

References

[1] D. Achlioptas. 2001. Database-friendly Random Projections. In *PODS 2001*.
[2] A. Alameldeen and D. Wood. 2004. *Frequent pattern compression: A significance-based compression scheme for L2 caches.* Technical Report. University of Wisconsin-Madison Department of Computer Sciences.
[3] A. R. Alameldeen and D. A. Wood. 2004. Adaptive cache compression for high-performance processors. In *ISCA 2004*.
[4] A. Arelakis, F. Dahlgren, and P. Stenström. 2015. HyComp: A hybrid cache compression method for selection of data-type-specific compression methods. In *MICRO 2015*. 38–49.
[5] Angelos Arelakis and Per Stenström. 2014. SC2: A Statistical Compression Cache Scheme. In *ISCA 2014*.
[6] S. Baek, H. G. Lee, C. Nicopoulos, J. Lee, and J. Kim. 2013. ECM: Effective Capacity Maximizer for high-performance compressed caching. In *HPCA 2013*.
[7] J. Bucek, K.-D. Lange, and J. von Kistowski. 2018. SPEC CPU2017 — Next-generation Compute Benchmark. In *ICPE 2018*.
[8] J. Buhler. 2001. Efficient large-scale sequence comparison by locality-sensitive hashing. *Bioinformatics* 17 (2001), 419–428.
[9] M. S. Charikar. 2002. Similarity Estimation Techniques from Rounding Algorithms. In *STOC 2002*.
[10] X. Chen, L. Yang, R. P. Dick, L. Shang, and H. Lekatsas. 2010. C-Pack: A High-Performance Microprocessor Cache Compression Algorithm. *IEEE Transactions on VLSI* 18 (2010), 1196–1208.
[11] D. Cheriton, A. Firoozshahian, A. Solomatnikov, J. P. Stevenson, and O. Azizi. 2012. HICAMP: Architectural Support for Efficient Concurrency-safe Shared Structured Data Access. In *ASPLOS 2012*.
[12] O. Chum, J. Philbin, and A. Zisserman. 2008. Near Duplicate Image Detection: min-Hash and tf-idf Weighting. In *BMVC 2008*.
[13] T. E. Denehy and W. W. Hsu. 2003. Duplicate management for reference data. In *Research Report RJ10305, IBM*.
[14] J. Dusser, T. Piquet, and A. Seznec. 2009. Zero-content Augmented Caches. In *ICS 2009*.
[15] M. Ekman and P. Stenström. 2005. A Robust Main-Memory Compression Scheme. In *ISCA 2005*.

[16] A. M. Elkahky, Y. Song, and I. He. 2015. A Multi-View Deep Learning Approach for Cross Domain User Modeling in Recommendation Systems. In *WWW 2015*.
[17] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD 1996*.
[18] S. Fox, S. Tridgell, C. Jin, and P. H. W. Leong. 2016. Random projections for scaling machine learning on FPGAs. In *FPT 2016*.
[19] P. Frankl and H. Maehara. 1988. The Johnson-Lindenstrauss lemma and the sphericity of some graphs. *Journal of Combinatorial Theory, Series B* 44 (1988), 355–362.
[20] J. Gaur, A. R. Alameldeen, and S. Subramoney. 2016. Base-Victim Compression: An Opportunistic Cache Compression Architecture. In *ISCA 2016*.
[21] A. Ghasemazar, M. Ewais, P. Nair, and M. Lis. 2020. 2DCC: Cache Compression in Two Dimensions. In *DATE 2020*.
[22] M. Ghayoumi, M. Gomez, K. E. Baumstein, N. Persaud, and A. J. Perlowin. 2018. Local Sensitive Hashing (LSH) and Convolutional Neural Networks (CNNs) for Object Recognition. In *ICMLA 2018*.
[23] A. Gionis, P. Indyk, and R. Motwani. 1999. Similarity Search in High Dimensions via Hashing. In *VLDB 1999*.
[24] B. Hong, D. Plantenberg, D. D. E. Long, and M. Sivan-Zimet. 2004. Duplicate Data Elimination in a SAN File System. In *MSST 2004*.
[25] P. Indyk and R. Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *STOC*.
[26] M. M. Islam and P. Stenström. 2010. Characterization and Exploitation of Narrow-width Loads: The Narrow-width Cache Approach. In *CASES 2010*.
[27] W. Johnson and J. Lindenstrauss. 1982. Extensions of Lipschitz mappings into a Hilbert space. *Conference in Modern Analysis and Probability* 26 (1982), 189–206.
[28] J. Kim, M. Sullivan, E. Choukse, and M. Erez. 2016. Bit-plane Compression: Transforming Data for Better Compression in Many-core Architectures. In *ISCA 2016*.
[29] S. Kottapalli and J. Baxter. 2009. Nehalem-EX CPU Architecture. In *HotChips 2009*.
[30] P. Li, T. J. Hastie, and K. W. Church. 2006. Very Sparse Random Projections. In *KDD 2006*.
[31] W. Liu, H. Wang, Y. Zhang, W. Wang, and L. Qin. 2019. I-LSH: I/O Efficient c-Approximate Nearest Neighbor Search in High-Dimensional Space. In *ICDE 2019*. 1670–1673.
[32] J. MacQueen. 1967. Some methods for classification and analysis of multivariate observations. In *Berkeley symposium on mathematical statistics and probability 1967*.
[33] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. 2007. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In *MICRO 2007*.
[34] T. M. Nguyen and D. Wentzlaff. 2015. MORC: A Manycore-oriented Compressed Cache. In *MICRO 2015* (Waikiki, Hawaii). ACM, New York, NY, USA, 76–88.
[35] J. Pan and D. Manocha. 2011. Fast GPU-based Locality Sensitive Hashing for K-nearest Neighbor Computation. In *GIS 2011*.
[36] B. Panda and A. Seznec. 2016. Dictionary Sharing: An Efficient Cache Compression Scheme for Compressed Caches. In *MICRO 2016*.
[37] G. Pekhimenko, T. Huberty, R. Cai, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. 2015. Exploiting compressed block size as an indicator of future reuse. In *HPCA 2015*.
[38] G. Pekhimenko, V. Seshadri, O. Mutlu, M. A. Kozuch, P. B. Gibbons, and T. C. Mowry. 2012. Base-delta-immediate compression: Practical data compression for on-chip caches. In *PACT 2012*.
[39] M. K. Qureshi, D. Thompson, and Y. N. Patt. 2005. The V-Way Cache: Demand Based Associativity via Global Replacement. In *ISCA 2005*.
[40] J. San Miguel, J. Albericio, N. Enright Jerger, and A. Jaleel. 2016. The bunker cache for spatio-value approximation. In *MICRO 2016*.
[41] J. San Miguel, J. Albericio, A. Moshovos, and N. Enright Jerger. 2015. Doppelgänger: a cache for approximate computing. In *MICRO 2015*.
[42] D. Sanchez and C. Kozyrakis. 2013. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems. In *ISCA 2013*.
[43] S. Sardashti, A. Seznec, and D. A. Wood. 2014. Skewed Compressed Caches. In *MICRO 2014*.
[44] S. Sardashti, A. Seznec, and D. A. Wood. 2014. Skewed Compressed Caches. In *MICRO 2014*.
[45] S. Sardashti, A. Seznec, and D. A. Wood. 2016. Yet Another Compressed Cache: A Low-Cost Yet Effective Compressed Cache. *ACM Trans. Archit. Code Optim.* 13 (2016), 27:1–27:25.
[46] S. Sardashti and D. A. Wood. 2013. Decoupled Compressed Cache: Exploiting Spatial Locality for Energy-optimized Compressed Caching. In *MICRO 2013*.
[47] S. Sardashti and D. A. Wood. 2014. Decoupled Compressed Cache: Exploiting Spatial Locality for Energy Optimization. *IEEE Micro* 34 (2014), 91–99.
[48] T. Singh, S. Rangarajan, D. John, C. Henrion, S. Southard, H. McIntyre, A. Novak, S. Kosonocky, R. Jotwani, A. Schaefer, E. Chang, J. Bell, and M. Co. 2017. Zen: A next-generation high-performance x86 core. In *ISSCC 2017*.
[49] G. Singh Manku, A. Jain, and A. Das Sarma. 2007. Detecting Near-duplicates for Web Crawling. In *WWW 2007*.
[50] D. Skarlatos, N. S. Kim, and J. Torrellas. 2017. PageForge: A Near-Memory Content-Aware Page-Merging Architecture. In *MICRO 2017*. 302–314.
[51] R. Spring and A. Shrivastava. 2017. Scalable and Sustainable Deep Learning via Randomized Hashing. In *KDD 2017*.
[52] W. J. Starke. 2009. POWER7: IBM's Next Generation, Balanced POWER Server Chip. In *HotChips 2009*.
[53] J. E. Stine, J. Chen, I. Castellanos, G. Sundararajan, M. Qayam, P. Kumar, J. Remington, and S. Sohoni. 2009. FreePDK v2.0: Transitioning VLSI education towards nanometer variation-aware designs. In *MSE 2009*.
[54] J. Stuecheli and W. J. Starke. 2018. The IBM POWER9 Scale Up Processor. In *HotChips 2018*.
[55] Y. Tian, S. M. Khan, D. A. Jiménez, and G. H. Loh. 2014. Last-level Cache Deduplication. In *ICS 2014* (Munich, Germany). 53–62.
[56] P.-A. Tsai and D. Sanchez. 2019. Compress Objects, Not Cache Lines: An Object-Based Compressed Memory Hierarchy. In *ASPLOS 2019*.
[57] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis. 1999. The Case for Compressed Caching in Virtual Memory Systems. In *ATEC 1999*.
[58] J. Zheng and J. Luo. 2013. A PG-LSH Similarity Search Method for Cloud Storage. In *CIS 2013*.