Last-level Cache Deduplication

Yingying Tian‡ Samira M. Khan†‡ Daniel A. Jiménez‡ Gabriel H. Loh§
‡Texas A& M University †Carnegie Mellon University ‡Intel Labs §AMD Research
‡{tian,djimenez}@cse.tamu.edu †samirakhan@cmu.edu §gabriel.loh@amd.com

ABSTRACT

Caches are essential to the performance of modern microprocessors. Much recent work on last-level caches has focused on exploiting reference locality to improve efficiency. However, value redundancy is another source of potential improvement. We find that many blocks in the working set of typical benchmark programs have the same values. We propose cache deduplication that effectively increases last-level cache capacity. Rather than exploit specific value redundancy with compression, as in previous work, our scheme detects duplicate data blocks and stores only one copy of the data in a way that can be accessed through multiple physical addresses. We find that typical benchmarks exhibit significant value redundancy, far beyond the zero-content blocks one would expect in any program. Our deduplicated cache effectively increases capacity by an average of 112% compared to an 8MB last-level cache while reducing the physical area by 12.2%, yielding an average performance improvement of 15.2%.

1. INTRODUCTION

Caches play an essential role in modern microprocessors to bridge the gap between fast processor speed and high access latency of main memory. A simple solution to improve cache performance is to increase the cache size. However, increased cache size leads to increased power and area consumption. In a chip-multi processor (CMP), often more than half of the chip area is occupied by caches that contribute to significant power consumption [37, 18, 31]. In a conventional cache, each block is associated with a requested memory block address and a copy of the data. Different cache blocks with different addresses can contain copies of identical data. These duplicated blocks waste cache capacity because they store redundant information. As an example, Figure 1 shows the average percentage of duplicated blocks stored in a 2MB last-level cache (LLC) in 18 randomly selected SPEC CPU2006 benchmarks [11]. Thirteen of 18 benchmarks have more than 20% duplicated cache blocks. For benchmarks *zeusmp* and *GemsFDTD*, more than 90% of cache blocks are duplicated. On average, 35% of cache blocks store duplicated data in the LLC.

Cache compression has been proposed to improve effective cache capacity [1, 2, 4, 19, 20, 40, 38, 10, 28]. Storing compressed cache blocks potentially reduces cache misses by increasing effective capacity. However, the processes of compression and decompression significantly increase cache access latency, thus degrading performance. The zero-content augmented cache [8] was proposed to reduce the storage of cache blocks that contain null data. Storing only physical addresses and valid bits of null blocks in an augmented cache saves cache area and improves overall performance. However, the percentage of zero-content blocks is relatively small on average, the performance improvement is also small.

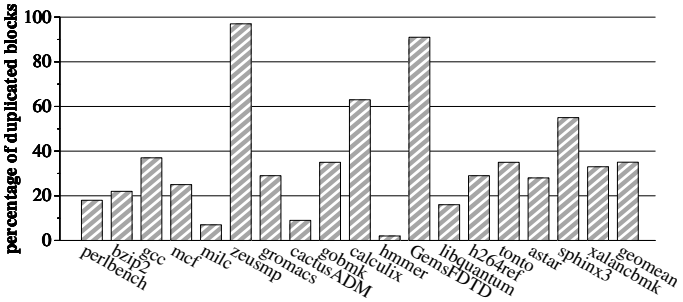

Figure 1: Average percentage of duplicated blocks in LLC.

We propose cache deduplication to improve performance significantly by exploiting value redundancy to increase effective cache capacity. Cache deduplication eliminates duplicated data stored in the cache. Instead of storing different copies of identical data, duplicated blocks are stored as references to distinct data. The saved capacity can be used to store more blocks to increase the overall effectiveness of the cache, reducing the frequency of costly off-chip memory accesses.

This paper makes the following contributions:

- We find that widespread duplication exists in caches and quantify the cache duplication effect in 18 SPEC CPU2006 benchmarks.
- We propose a unified cache-deduplication technique to improve cache performance with increased effective cache capacity. By exploiting block-level value redundancy, cache deduplication significantly increases cache effectiveness with limited area and power consumption.
- We propose a novel LLC design with cache deduplication. Compared to a conventional LLC, the deduplicated LLC uses similar chip area and power consumption while performing comparably to a double-sized conventional LLC.

Based on our experiments, cache deduplication improves performance by 15.2% compared to a baseline 8MB LLC,

ICS'14, June 10–13 2014, Munich, Germany.

http://dx.doi.org/10.1145/2597652.2597655 .

which is comparable to a conventional 16MB LLC and superior to 12MB and 14MB caches, while using 12.2% less physical area than a conventional 8MB LLC.

In this paper, Section 2 motivates the proposed technique. Section 2.2 describes cache deduplication in detail, followed by a novel design of deduplicated LLCs in Section 3. Section 4 discusses the experimental methodology we use, followed by the experimental results in Section 5. A detailed analysis is presented in Section 6. Related work is discussed in Section 7, and we conclude in Section 8.

2. MOTIVATION AND CHALLENGES

2.1 Motivation

Conventional cache design wastes capacity because it stores duplicated data. When a memory request is issued, the data fetched from the main memory also is brought into caches for future requests. This data is associated with a tag derived from its physical memory address. However, cache blocks with different block addresses may contain identical data. The same chunk of data is duplicated in the cache because the tags differ.

We measure the percentage of distinct blocks stored in a 2MB 16-way LLC during the execution of 18 SPEC CPU2006 benchmarks, each running for an interval of one billion instructions. We count the number of distinct blocks every 10 million instructions. The ratio of distinct blocks varies with the workload, but there always are duplicated blocks stored in the cache for all the benchmarks. Among the benchmarks, *hmmer* has the smallest percentage of duplicated blocks (2.7% on average) and *zeusmp* has the largest percentage of duplicated blocks (97.8%). On average, 35.1% of cache blocks are duplicated for all the benchmarks.

```
if (serEng.needToStoreObject(objToStore)) {
   int vectorLength = objToStore->size();
   serEng<<vectorLength;
   for ( int i = 0; i < vectorLength; i++) {
       XercesStep* data = objToStore->
           elementAt(i);
       serEng<<data;
    }
}
```

Listing 1: storeObject() in XTemplateSerializer.cpp

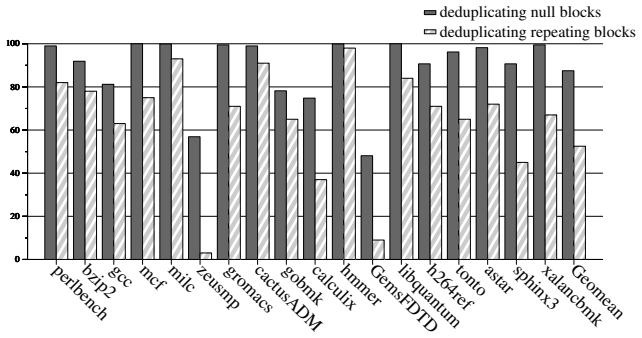

Figure 2: Percentage of distinct blocks for null-block deduplication and all repeating-block deduplication.

This phenomenon happens mainly because of program behaviors and input characteristics. Program behavior such as copying and assignment generate duplicate data stored at different memory locations. Listing 1 shows a code snippet of assignment in the SPEC CPU2006 benchmark *xalancbmk*. Elements in the vector *objToStore* are stored in the buffer *serEng*. After running this code, there are two copies of the same data stored in the cache. A similar phenomenon happens with memory operations like *memcpy()* [1].

Another source of duplication is program input. For example, the input of the SPEC CPU2006 benchmark *zeusmp* is "a spherical blastwave with radius r=0.2 and located at the origin" [11], which contains perfect symmetry, leading to a significant amount of data similarity (97.8% of cache blocks are duplicated in our experiment). Similar input characteristics exist in benchmark *GemsFDTD*, in which more than 90% of cache blocks are duplicated. Symmetric data is common especially in scientific workloads, causing copious duplication of non-zero values.

Previous cache-compression techniques proposed to compress specific values that cause data duplication [1, 8, 28] such as zero. Based on our experiments, eliminating zero-content (null) blocks can save only 13% of the cache capacity, while eliminating all possible duplication leads to 47.5% of cache blocks removed/invalidated, as shown in Figure 2. In other words, almost half of the cache capacity can be saved with data deduplication.

The majority of duplication contains non-zero data values resulting from input and/or computation with a random distribution of the number of copies depending on program behavior. As an example, we take a random execution point of *xalancbmk* to show the nature of duplication degree and duplicated data. At a random execution point, in a 2MB cache, there are 14,931 distinct blocks out of 29,278 of cache blocks (i.e., 51% of blocks are distinct). There are 2,414 chunks of data associated with two tags each, so 16% of blocks are duplicated once. There are 1,157 zero-content blocks. If only zero-content blocks are compressed, only 4% of total capacity is saved. If all the duplication can be eliminated from the cache, more than 38% of the capacity of the 2MB cache can be saved, which is about three times larger than a modern processor's typical 256KB L2 cache.

2.2 Challenges of Cache Deduplication

Data deduplication is a specific compression technique to eliminate duplicated copies of repeating data. It has been used widely in disk-based storage systems [6, 39, 12]. With data deduplication, only a single instance of identical data is stored physically. The redundant data is stored as references to the corresponding data in a deduplicated data storage to improve storage utilization. Although commonly used in disk storage and main-memory compression, data deduplication is a challenge in caches with limited overhead due to the following concerns:

How to detect duplication: The first challenge is the way to compare data to detect possible duplication. Duplication can be detected by comparing the analyzed data either with all the stored data or to a specific part of a tree-based data array. Because caches contain a large number of blocks, direct comparison with all blocks is prohibitively expensive. A tree-based structure requires more metadata to maintain the tree while the time complexity is still too high for a large number of nodes. Indexing using a hash function is a fast solution to find the data with which to compare. However, simply using a hash function to index the data array is inefficient because of underutilization of

[1]Some compilers and ISAs generate specialized code so that certain copies bypass the cache. For instance, Intel's C compiler and libraries will use a non-temporal store for *memcpy()* if the size of the data moved is larger than 256KB [9]. However, shorter instances of copying continue to lead to significant cache data duplication.

the data array. A practical duplication-detection technique must be fast as well as storage-efficient.

When to detect duplication: The second challenge is the point at which to process duplication detection. Caches play an important role in bridging the performance gap between processors and the main memory, in which access latency is critical to the overall system performance. The process of duplication detection should not affect the cache latency.

Deduplication granularity: Previous work [40, 38, 2, 1, 8, 28] used sub-block level granularity to compress all possible compressible data. Granularity at the sub-block-level may lead to a higher rate of deduplication, but it also causes increased access latency, additional power overhead, and more complex hardware design. Although the effective capacity can be increased more with sub-block-level deduplication, the system performance may be degraded because of the increased access latency. The trade-offs among compression degree and increased cache latency and overhead makes compression granularity another challenge for cache deduplication.

Write hit and replacement of duplicated blocks: The last challenge in cache deduplication design is dealing with write hits and replacement of duplicated blocks. When a store instruction writes duplicate data, the updated block must be allocated a new entry to differentiate from the previous value. When duplicate data is invalidated or evicted from a deduplicated cache, all tags that are associated with this data also should be invalidated. Previous work proposed storing all possible tags in each data entry [23], which is impractical in a cache design due to the limited capacity. An intelligent and low-overhead data management is required in a practical cache deduplication design.

3. DEDUPLICATED LAST-LEVEL CACHE

We propose a practical LLC design eliminating duplicated cache blocks that we call a deduplicated LLC. To address the challenges cited in the previous section, deduplicated LLC uses augmented hashing to detect duplication, which is fast and makes the most of the utilization of the cache capacity. It uses post-process detection [17] to hide possibly increased cache latency. It uses block-level-deduplication granularity to compare the analyzed block with the data already stored in the cache, regardless of its content, to exploit data duplication fully with limited overhead. For the replacement policy of the duplicated blocks, we propose the distinct-first random replacement (DFRR) policy for efficiency.

3.1 Structure

Figure 3 shows the structure of a deduplicated LLC. It consists of three decoupled structures: a tag array, a data array, and a hash table. With cache deduplication, the mapping from the data store to the tag store is no longer one-to-one. The structure of the data store is decoupled from that of the tag store. The data array is used only to place distinct data, while the tag array keeps the semantics of cache blocks by storing blocks with tags, pointers to the data array, and other metadata. More than one tag can share a data block. Cache-management techniques (e.g., intelligent replacement policy, increased number of blocks, and so on) are related only to the tag array. With the decoupled structures, changes in the tag array need not affect the design of the data array.

3.1.1 Tag Array

The tag array is a set-associative structure that keeps the semantics of cache blocks. Each entry in the tag array contains the following fields: required metadata of a cache block as in a conventional cache (e.g., tag bits, LRU bits, valid bit, and dirty bit), a reference that indexes the data array, and two references that point to other tag entries that maintain a doubly-linked list of tags all pointing to the same data block. The reference to a data entry, referred to as a *tag-to-data pointer (Tptr)*, identifies a distinct entry in the data array. When there is a tag match, *Tptr* directly indexes the data associated with this cache block. When a tag is inserted in the tag array, it also is inserted into the doubly-linked list of tags of duplicated blocks (if there are any) associated with the corresponding data.

When a tag is replaced from the tag array, it also is deleted from the linked list. With these pointers, all tags stored in the tag array that share identical data are linked. The linked list of tags of duplicated blocks is referred to as the tag-list and the two pointers in each tag entry are referred to as tag-list pointers. When there is a replacement in the data array, all associated tags can be tracked along with the tag-list of the data block and invalidated. The replacement of the data array will be discussed in Section 3.2.3; in practice, this process has very low latency. The tag array can be treated as a conventional cache storing only metadata. It uses requested memory addresses to search specific sets for matching tags. When cache misses occur, the tag array uses the regular cache replacement policy (i.e., least-recently used (LRU) to choose replacement candidates rather than replacement in the data array, which uses the DFRR policy).

In our experiments, we use the traditional least-recently-used (LRU) replacement policy in the tag array for fair evaluation. The left-most structure shown in Figure 3 gives an example of the tag array in a deduplicated LLC. This tag array is a 4-way set-associative structure, with three sets. As shown at the bottom of the structure, the second (from left to right) tag entry in *set[2]* contains the tag *t9*, the *Tptr* that indexes the corresponding data *d1 - 0x1*. One tag-list pointer to the previous block in the tag-list - *t6* and the other tag-list pointer is set as NULL because there is no next block of *t9*. As drawn in bold in Figure 3, Blocks t3, t2, t1, t5, t4, and t8 are in the tag-list of duplicated data *d0*, and t6 and t9 are in their own list. Blocks t7 and t10 are distinct blocks, because there is only one tag in the tag-list of each data block.

3.1.2 Data Array

Each entry in the data array contains a data frame, a counter, a pointer, and a one-bit deduplication flag. The counter (referred to as *Ctr*) indicates the number of tags stored in the tag array that share this data. When a tag is inserted into the tag array, the corresponding *Ctr* in the data array is incremented by 1. When a tag is replaced or invalidated from the tag array, the corresponding *Ctr* is decremented by 1. When a *Ctr* becomes zero, the data block can be reused. The pointer (referred to as a data-to-tag pointer (Dptr)) identifies the head of the tag-list. *Dptr*s of invalid entries are used to keep a free list of available data entries. The one-bit deduplication flag indicates whether the current data block has been analyzed for deduplication (discussed in Section 3.3). The data array can be treated as a direct-mapped cache, accessed only by *Tptr*s from the corresponding tag entries. The structure shown in the middle of Figure 3 gives an example of a data array. There are six entries in the data array; four of them are valid. Data *d0*, located in *0x0*, is shared by six blocks (*Ctr* equals 6), heading with tag *t3* in the tag-list. Data blocks *d2* and *d3* are distinct blocks, linking to only one tag each, *t7* and

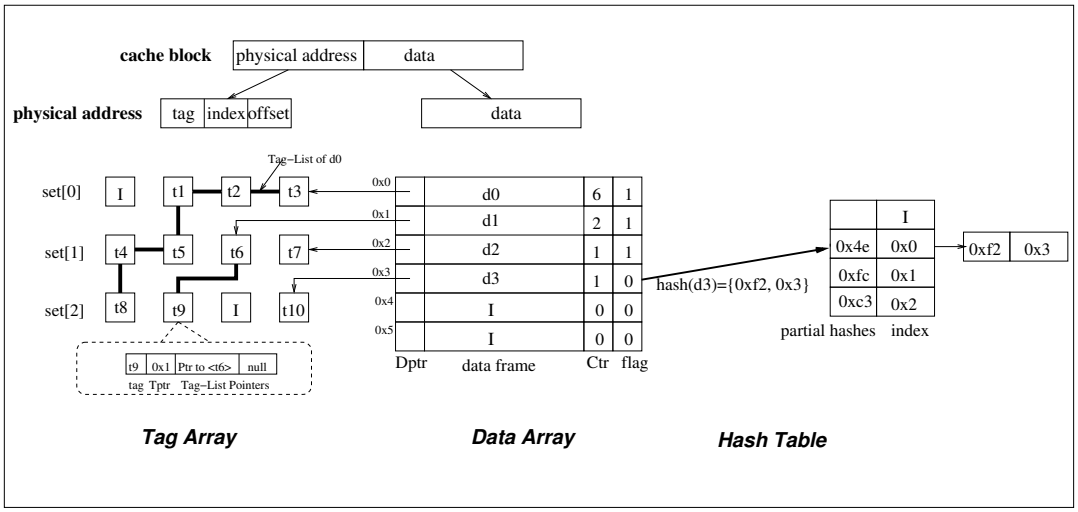

Figure 3: Structure of a deduplicated LLC. Blocks t1, t2, t3, t4, t5 and t8 are duplicated blocks, sharing identical data d0; t6 and t9 share data d1; t7 is a distinct block with data d2; and, t10 is inserted as a distinct block and has not been analyzed for deduplication yet.

t10, respectively. However, *d3* has not been analyzed for duplication detection yet (i.e., the flag is unset).

3.1.3 Hash Table

The third structure in deduplicated LLC is an augmented hash table. We use an augmented hash table to implement a two-level look-up to make the most of the cache capacity. The first level of look-up occurs in the hash table indexed by the hashed data, and the second level occurs in the data array redirected by the indices stored in the hash node. To reduce the number of hash collisions, the hash table is implemented as a sequence of small associative arrays representing buckets. Each node in a bucket contains a 16-bit pointer indexing the data array, a 1-bit valid bit, and a 15-bit partial-hash value.

On each duplication detection, the new data are hashed to a hash table entry containing a bucket of nodes as shown in the right-most structure in Figure 3. To reduce access to the data array, each node stores a partial hash value as well as the index into the data array. The new data is compared with indexed data only if the partial hash values match. For the hash function, we use five-level exclusive-OR gates using the same technology used for hashing long branch history for high-performance branch predictors [35]. Each level of the exclusive-OR gate halves the number of bits by taking the exclusive-OR of the upper half of the input bits with the lower half of the input bits. Hashing is completed within one cycle assuming a clock period of at least 10 FO4 delays.

Based on our experiments, a small hash table is sufficient to keep the percentage of hash collisions extremely low (less than 1%). However, hash collisions are practically unavoidable when hashing a large set of possible keys (cache data). We describe hash collision resolution in Section 3.4.

3.2 Operations

A deduplicated cache has different operations on cache hits and cache misses. On a cache access, the tag of the requested block is compared in parallel with all tags in a specific set of the tag array. If the look-up fails, a cache miss has occurred; otherwise, a cache hit has occurred.

3.2.1 Cache Miss

On a cache miss, the requested block is brought from the main memory as in a conventional cache. The placement of the cache block then is separated into two parts: placement in the tag array and placement in the data array. The data of the block is placed in an invalid data entry randomly chosen from the free list maintained using the *Dptr*s. The tag of the block is placed in the corresponding set of the tag array indexed using the memory address. The *Tptr* in the tag entry and the *Dptr* in the data entry then are updated to point to each other, and *Ctr* is increased by 1. If there is no invalid entry in the set of the tag array, the regular replacement policy (LRU in our experiments) is used to choose a replacement victim. If there is no invalid entry in the data array, we use DFRR to choose a data replacement victim (detailed in Section 3.2.3).

At this time, the requested cache block is not analyzed for duplication (with the deduplication flag unset). Instead, it is placed in the cache directly with an unset deduplication flag, indicating it has not been processed for deduplication, and without incurring any deduplication latency. The duplication detection to this block will not be launched until next cache miss occurs, as described in Section 3.3. The corresponding hash node of the data replacement victim then is invalidated.

3.2.2 Cache Hit

A cache hit can be either a read hit or a write hit. In a deduplicated cache, write hits modify the data of blocks, incurring re-hash of the updated data for another duplication detection, while read hits are unrelated to deduplication. Thus, the operations on read hits and write hits are different:

- When there is a read hit in the tag array, the *Tptr* in the matching entry directly indexes the data array to retrieve the requested data. Replacement information then is updated in the tag array. The data array is unchanged.
- When there is a write hit in the tag array, the requested data is indexed by the *Tptr*. If it is a distinct block (*Ctr* equals 1), the data can be modified immediately and the deduplication flag is unset to indicate an

unanalyzed block. If it is a duplicated block, instead of modifying the data array directly, an invalid data entry is allocated to place the updated data. In this case, the write hit to a duplicated data is processed similar to a cache miss. Then the dirty bit in the tag entry is updated as well as the replacement information.

3.2.3 Distinct-first Random Replacement

We use a DFRR policy in data array replacement. To find a replacement candidate, the DFRR policy goes to a random position of the data array and checks if the data is distinct. If it is distinct, the entry is chosen as replacement victim; if not, another random entry is checked. To limit the amount of checking, up to four locations can be checked on each replacement. If there is no distinct block among the checked blocks, the block with the fewest duplicates out of the four entries is replaced. Corresponding tag entries are back-invalidated in the tag array to maintain integrity.

Based on our experiments, on each data replacement, on average 1.004 blocks are checked randomly to find the replacement victim. The intuition behind DFRR is that no invalid data entry means there are too many distinct blocks, so one or two random checks will be enough to find a distinct block to replace. The latency of finding a new data entry can be hidden completely.

3.3 An Example of Hash-based Post-Process Deduplication

We propose to use hash-based post-process duplication detection to process deduplication fast with limited overhead. Hash-based post-process duplication detection is launched on LLC misses to avoid possible increased latency. The cache block that is under deduplication detection is blocked. Delaying the detection process until the cache is less busy and the processed block has less chance to be accessed (due to locality) helps avoid dynamically increased cache latency. Figure 4 gives an example of how it works. In this example, the tag array is a 4-way associative structure with two sets, the data array has three entries, and the hash table has four buckets. Each bucket contains a chain of two nodes. Each valid tag entry contains a *Tptr* pointing to the corresponding data entry. For simple illustration, we do not show the replacement states in the tag array, nor do we show *Dptr*s, *Ctr*s, and deduplication flags in the data array.

On a cache miss to Block A, the requested block is fetched from the main memory. The tag is inserted in the tag array and the data *d1* is inserted in an invalid data entry, as in Step 1. On the next cache miss to Block B, during the memory access time, the previously placed data *d1* of Block A is detected for duplication. The hash value of *d1* indexes a bucket in the hash table (Step 2). Because the bucket is empty, the location of *d1* and its hash value *hd1* are placed in this bucket. After Block B is fetched from the memory, it is filled in the cache (Step 3).

On a cache miss to Block C, the previously placed data *d2* of Block B needs duplication detection. The bucket of *d2* is also empty, so the position of *d2*, *0x1*, and its hash value *hd2* are inserted in the bucket (Step 4). Block C later is filled in the cache by placing the tag in the tag array and inserting the data in an empty data entry at *0x2* (Step 5).

On a cache miss to Block D, the data of Block C (located at *0x2*) hashes to a bucket containing a hash value *hd1* and index *0x0*. Because the hash of the data of Block C equals *hd1*, the data is compared with the data located at *0x0*, resulting in a match (Step 6). Thus, the *Tptr* of Block C is updated to *0x0*, and the data entry in *0x2* is invalidated (Step 7). The *Dptr* of *d1* is updated to point to Block C. After requested Block D is fetched, it is filled in the cache by placing its data in the empty entry at *0x2* (Step 8).

On a cache miss to Block E, the previously placed data *d4* of Block D is analyzed for deduplication. The hash value of *d4* does not equal the one stored in the hash node, so there is no further data comparison. A hash collision incurs. The location of *d4* and its hash value are inserted in the chain of the hashed bucket (Step 9).

3.4 Hash Collision Resolution

Hash collisions are unavoidable with a practical hash function. In a deduplicated cache, a hash collision occurs when the hash bucket is full. Thus, a strategy is required for hash collision resolution:

- If there is a distinct block indexed in the current bucket, this block is back-invalidated from the data array and the tag array, respectively. The bucket node then is updated to the location of the colliding data. This procedure can be treated as a replacement in a hash bucket.
- Because of the extremely low probability (lower than 0.1% in our experiments), if data indexed in the current bucket are all duplicated, no replacement occurs in this bucket. The current deduplication procedure just exits and a new detection is launched if there is any unanalyzed data. In this case, we may lose a chance to eliminate a possibly duplicated block. However, it will not cause any extra cache misses to degrade the cache performance because the mapping from the tag to the data is kept one to one.

Based on our experiments, a hash bucket with 16 nodes is sufficient to keep the rate of hash collision as low as 1%. We give detailed analysis concerning hashing in Section 6.4.

4. METHODOLOGY

This section outlines the experimental methodology used in this work.

4.1 Simulation Environment

We use the MARSSx86 cycle-accurate simulator [26], a full- system simulation of the x86-64 architecture that runs both single-core and multi-core workloads to evaluate the proposed deduplicated LLC. It models an out-of-order 4-wide x86 processor with a 128-entry re-order buffer and coherent caches with MESI protocol as well as on-chip interconnections.

The micro-architectural parameters are consistent with Intel Core i7 processors [21], including a three-level cache hierarchy: L1 I-caches and L1 D-caches, L2 caches, and a shared LLC. The L1 and L2 caches are private to each core. The L1 I-cache and D-cache are 4-way 32KB each and the L2 cache is unified 8-way 256KB. The shared LLC is a unified 16-way 2MB-per-core cache. The default replacement policy for each cache is LRU. Access latencies to the L1 cache, L2 cache, LLC, and main memory are 4, 10, 40, and 250 cycles respectively, in keeping with the methodology of recent cache research work [13, 15, 14, 7]; we show in Section 6.6 that our results are not changed significantly with alternate latencies. For the deduplicated LLC, both the number of sets and the associativity of the tag array can be increased to accommodate more blocks. We evaluate both ideas by doubling the number of sets and associativity of the tag array, respectively. The reason to double the size of the tag array is to compare the duplicated LLC with a double-sized conventional LLC. The actual size of the tag array can

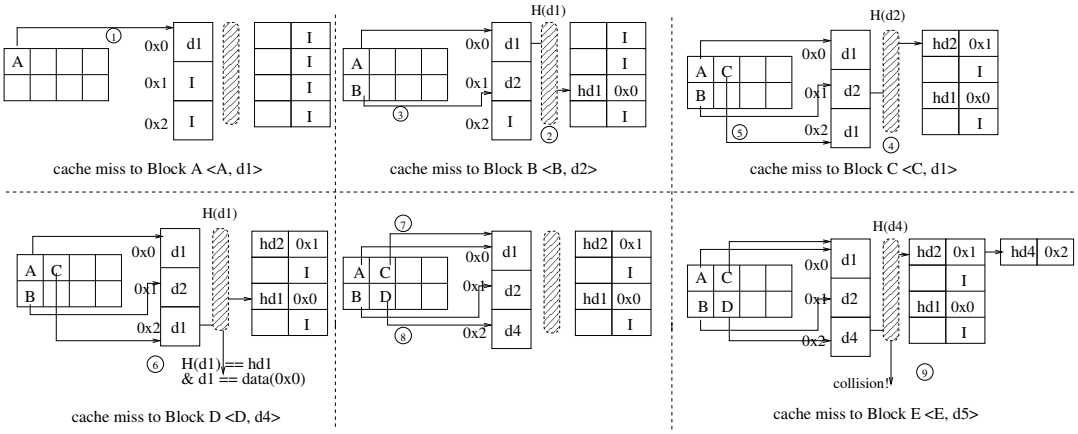

Figure 4: An example of hash-based post-process last-level cache deduplication.

be increased arbitrarily to achieve better performance with commensurate power and area consumption. Based on the experiments, the evaluated deduplicated LLC with a double-sized tag array fits in the area of the LLC of the baseline. We show a detailed cost analysis in Section 6.

The replacement policy in the tag array is LRU, while the replacement policy in the data array is the proposed DFRR.

We also compare our work with two cache-compression techniques: adaptive cache compression [1] and ZCA cache [8]. With adaptive cache compression, the L1 and L2 caches have the same configuration as in a conventional cache hierarchy. Data stored in L1 and L2 caches are uncompressed and only the LLC supports compression. The compressed LLC is a unified 16-way (up to 32-way dynamically) 2MB-per-core set-associative cache with decoupled tag and data stores. Instead of storing a 64-byte data block, the data store is broken into 8-byte segments. An uncompressed 64-byte block is stored as eight 8-byte segments, while a compressed block is compressed into one to seven segments. Data segments are stored continuously in each set with tag order. We conservatively ignore the very high cost of replacement in the contiguous storage variant of the compressed cache.

In our experiments, the access latency of a compressed LLC is constant at 24 cycles. We ignore the decompression latency of 5 cycles to evaluate the cache-deduplication technique better. We also assume that the compression process, occurring on each LLC replacement, can be hidden by the memory-access latency. Thus, the extra compression latency is ignored in our experiments.

With the ZCA cache technique, the L1 and L2 caches have the same configuration as the baseline. The L3 cache is a 2MB-per-core set-associative main cache along with an 8,192-entry, 8-way ZCA cache consuming 156KB of storage overhead. Because accesses to the ZCA cache are in parallel with accesses to the main cache, the access latency is unchanged.

4.2 Benchmarks

The benchmarks used in the experiments are selected randomly from the SPEC CPU2006 benchmark suite. We use SimPoint [29] to identify a single one-billion-instruction characteristic interval (i.e., SimPoint) of each benchmark. Each benchmark is compiled for the x86-64 instruction set and run with the first *ref* input provided by the *runspec* command. Benchmarks are categorized into three groups based on the average percentage of duplicated blocks:

- *Deduplication-sensitive benchmarks*: average percentage of duplicated blocks is greater than 50%;
- *Deduplication-friendly benchmarks*: average percentage of duplicated blocks is between 20% and 50%; and,
- *Deduplication-insensitive benchmarks*: average percentage of duplicated blocks is lower than 20%.

Table 1 shows the group and the percentage of duplicated blocks of each benchmark as well as the LLC misses per 1,000 instructions (MPKI), instructions per cycle (IPC), and the number of instructions fast-forwarded (FFWD) to reach the interval given by SimPoint in a baseline system. Memory-intensive benchmarks are shown in boldface.

Group	Benchmark	% Duplicated Blocks	MPKI (LRU)	IPC (LRU)	FFWD
Dedup-sensitive (S)	**zeusmp**	97.1%	9.05	0.580	405B
	GemsFDTD	90.6%	16.46	0.466	1060B
	calculix	63%	0.04	1.130	4433B
	sphinx3	54.6%	9.00	0.530	3195B
Dedup-friendly (F)	gcc	37.3%	1.38	1.292	64B
	gobmk	34.9%	0.35	1.072	133B
	tonto	34.9%	0.04	1.259	44B
	xalancbmk	33.4%	35.95	0.144	178B
	h264ref	30%	0.09	1.700	8B
	gromacs	28.8%	0.59	1.244	1B
	astar	27.9%	9.7	0.366	185B
	mcf	24.7%	83.54	0.126	370B
	bzip2	22.1%	0.886	1.127	368B
Dedup-insensitive (I)	**perlbench**	18.2%	1.67	0.882	541B
	libquantum	16.1%	24.82	0.162	2666B
	cactusADM	9%	24.7	0.22	81B
	milc	7%	1.01	1.299	272B
	hmmer	2.7%	2.75	0.844	942B

Table 1: The 18 SPEC CPU2006 benchmarks with LLC cache misses per 1,000 instructions for LRU, instructions per cycle for LRU in a 2MB cache, and number of instructions fast-forwarded to reach the simpoint (B = billions). Memory-intensive benchmarks in **boldface**.

For multi-core workloads, we randomly generate 12 mixes of quad-core workloads from the 18 benchmarks, listed in Table 2 with their characteristics of duplication. Each benchmark in a workload runs simultaneously with the others, restarting after one billion instructions, until all of the benchmarks have executed at least two billion instructions.

5. EXPERIMENTAL RESULTS

In this section we analyze the performance and overhead of cache deduplication.

Mixes	Benchmarks
mix1 (FFSF)	gcc, gobmk, zeusmp, xalancbmk
mix2 (ISSF)	milc, sphinx3, zeusmp, gobmk
mix3 (SSSF)	GemsFDTD, zeusmp, calculix, xalancbmk
mix4 (FFSS)	astar, gobmk, calculix, GemsFDTD
mix5 (FISF)	sphinx3, milc, zeusmp, xalancbmk
mix6 (IFSS)	hmmer, gcc, sphinx3, calculix
mix7 (IFFF)	hmmer, gcc, xalancbmk, gromacs
mix8 (FSSF)	gcc, calculix, GemsFDTD, h264ref
mix9 (FFII)	gobmk, gromacs, hmmer, perlbench
mix10 (FIIF)	h264ref, hmmer, libquantum, xalancbmk
mix11 (IISF)	libquantum, hmmer, GemsFDTD, tonto
mix12 (ISFF)	perlbench, zeusmp, mcf, gcc

Table 2: 12 mixes of quad-core workload ('F' stands for deduplication-friendly, 'S' for deduplication-sensitive and 'I' for deduplication-insensitive).

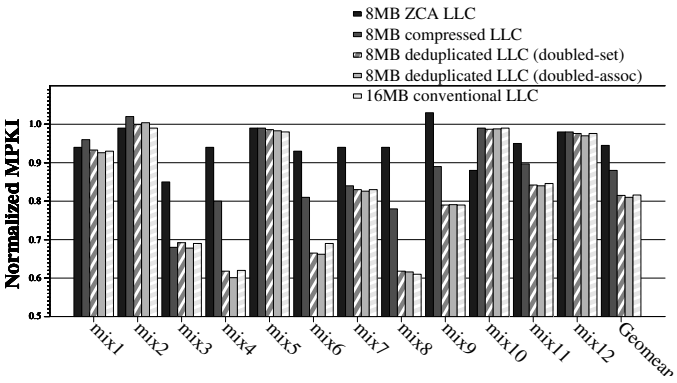

Figure 5: Reduction in LLC misses normalized to 8MB conventional LLC.

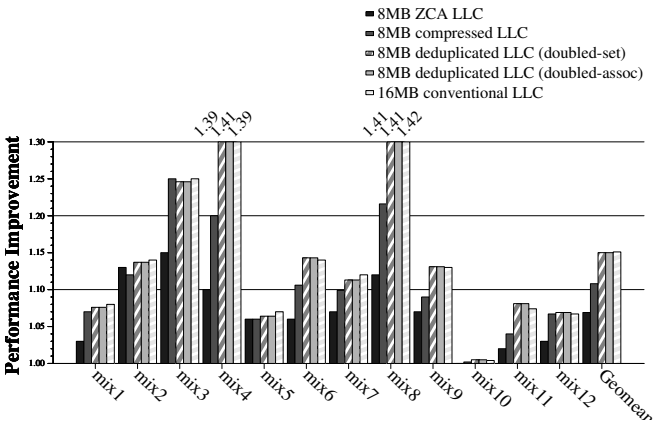

Figure 6: Performance Improvement normalized to 8MB conventional LLC.

5.1 Performance Improvement

In a deduplicated cache, both the number of sets and the associativity of the tag array can be increased to place more cache blocks. In a compressed cache, the number of sets cannot be increased and the associativity is increased dynamically up to twice as large as an uncompressed cache. In a ZCA cache, up to 64MB null blocks can be mapped.

We compare the performance of each technique with a double-sized conventional cache as an upper bound (doubled-sets). In our experiments, we show the performance improvement (normalized to an 8MB conventional LLC) of an 8MB compressed LLC, an 8MB deduplicated LLC with doubled number of sets (16,384 sets, 16-way), an 8MB deduplicated LLC with doubled associativity (8,192 sets, 32-way), an 8MB conventional LLC with a 8,192-entry ZCA cache, and a 16MB conventional LLC (16,384 sets, 16-way).

Figure 5 shows the LLC cache misses normalized to an 8MB conventional LLC of each technique for quad-core workloads. On average, ZCA cache reduces the LLC misses by 5.5%. Cache compression reduces the LLC misses by 12%. Cache deduplication in a doubled-set LLC reduces average misses by 18.5%. Cache deduplication in a doubled-associativity LLC reduces average misses by 19%. The doubled-size conventional LLC reduces the cache misses by 18.4%.

Reducing cache misses translates into improved performance. Figure 6 shows the performance improvement of each technique normalized to an 8MB conventional LLC. The ZCA cache improves performance by 6.9%. The compressed cache yields an average speed-up of 10.8% compared to the baseline. Cache deduplication in a doubled-set LLC gives an improvement of 15%, and cache deduplication in a doubled-associativity LLC yields a speed-up of 15.2%. The upper-bound 16MB conventional cache delivers an average speed-up of 15.1% compared to the 8MB baseline. A 12MB conventional LLC delivers an 8.7% speed-up, and a 14MB LLC delivers an 8.9% speed-up.

Overall, the deduplicated LLC performs comparably to a double-sized conventional LLC.

6. DETAILED ANALYSIS

In this section, we give detailed analysis of cache deduplication with respect to capacity, storage, and power overhead, hashing effectiveness, and the cache sensitivity to different sizes of hash table.

6.1 Effective Cache Capacity

Figure 7 shows the average amount of duplication in each quad-core workload. On average, each block of data stored in the data array is shared by 2.23 tags. In other words, effective cache capacity is increased by 112% with cache deduplication. For workloads mix6, mix7, mix9, mix10, and mix11, which all contain the most deduplication-insensitive benchmark *hmmer*, cache deduplication still works by eliminating duplication by about 38%.

6.2 Storage

Although the effective capacity is increased, the physical area is reduced. Table 3 shows the detailed storage requirements of both the baseline and the deduplicated LLC in a quad-core CMP. The 8MB deduplicated LLC occupies only 87.8% of the physical area of a conventional 8MB LLC (i.e., it reduces physical area by 12.2% compared to the conventional LLC). The area savings lead to reduced leakage power cost, as shown in Section 6.3.

6.3 Power and Energy

Table 4 shows the results of CACTI 6.5 simulations [24] to determine the leakage and dynamic power of the deduplicated LLC compared to the conventional LLC. The tag array is modeled as the tag store of a conventional 16MB set-associative cache. The data array is modeled as a 4MB direct-mapped cache with 37 bits of tags. The hash table is modeled as the data store of a 512KB direct-mapped cache with block size of 4 bytes.

Due to the nature of deduplicated caches, accesses to the LLC are increased while accesses to the main memory are decreased. Based on the experiments, compared to an 8MB

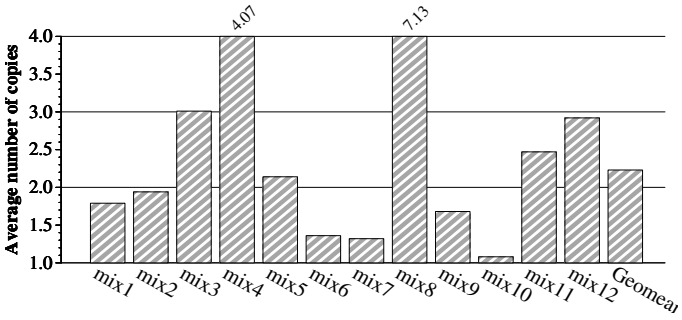

Figure 7: Average amount of duplication.

	Conventional LLC	Deduplicated LLC
Each tag store entry contains:		
Tag	29 bits	28 bits
Status (valid+dirty+LRU)	6 bits	6 bits
Tptr	-	17 bits
*Rptr*s	-	36 bits
Number of tag entries	131,072	262,144
Total size of tag store	560KB	2784KB
Each data store entry contains:		
Data	512 bits	512 bits
Dptr	-	18 bits
Ctr	-	18 bits
Dedup flag	-	1 bit
Number of data entries	131,072	65,536
Total size of data store	8192KB	4392KB
Additional structure(s):		
Size of hash table	-	8,192
Length of chain	-	16
Size of node	-	32 bits
Total size of hash table	-	512KB
TOTAL SIZE	8,752KB	7,688KB

Table 3: Storage cost analysis.

conventional cache, the number of accesses to the tag array of the 8MB deduplicated cache is increased by 38% and the number of accesses to the data array is increased by 33%. The number of accesses to the off-chip main memory is decreased by 26% with the deduplicated LLC.

Compared to the energy cost of accessing caches, the energy cost of accessing the off-chip memory is significantly higher. According to the results of previous work [33], the energy consumed to activate and precharge a page and to read a block is *5nJ* with a row buffer size of 8KB. Thus, as shown in Table 5, the average dynamic energy consumption of the deduplicated LLC accesses is 3.3% higher than that of the conventional LLC, while the dynamic energy cost of the memory accesses is reduced by 34.5% with the deduplicated LLC.

	Structures	Dynamic Energy per Read Port (nJ)	Dynamic Power per Read Port at max freq (W)	Leakage Power per Bank (W)
Conventional	Tag store	0.0389	0.0605	0.5205
	Data store	1.3148	2.0482	3.0297
	Total	1.3537	2.1087	3.5502
Deduplicated	Tag array	0.1225	0.2564	0.9207
	Data array	0.8793	2.3149	1.8441
	Hash table	0.0234	0.0746	0.0445
	Total	1.0543	2.6534	2.9278

Table 4: Dynamic and leakage power of each LLC design.

6.4 Hashing

Number of Look-ups.

Figure 8 shows the average number of look-ups in each deduplication process. On each duplication detection, the analyzed data is compared with all the data indexed in the hash bucket until a match occurs or it mismatches with all the data. On average, there are 4.9 look-ups in each

	Structures	Dynamic Energy (J)
Conventional	Tag store	0.0005
	Data store	0.0175
	Memory	0.0222
Deduplicated	Tag array	0.0021
	Data array	0.0156
	Hash table	0.0009
	Memory	0.0165

Table 5: Dynamic energy cost of each LLC and main memory.

duplication detection. The number of look-ups is related to the deduplication latency, described in Section 6.5. For workloads such as mix6, mix10, and mix11, the number of look-ups is higher because of the nature of deduplication-insensitive benchmarks: most analyzed data is distinct, causing more look-ups in each duplication detection.

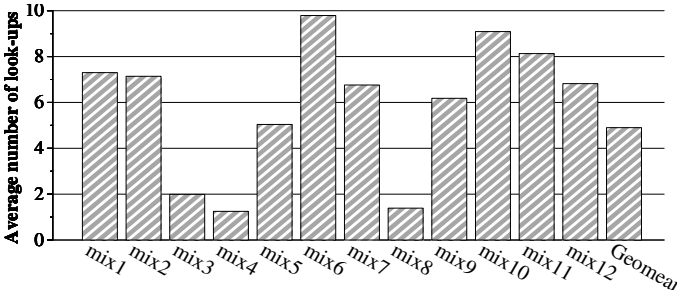

Figure 8: Average number of look-ups for data comparison.

Hash Collisions.

With a practical hash algorithm, hash collisions are unavoidable. Figure 9 shows the average percentage of hash collisions for each quad-core workload. On average, the percentage of hash collisions is as low as 1%.

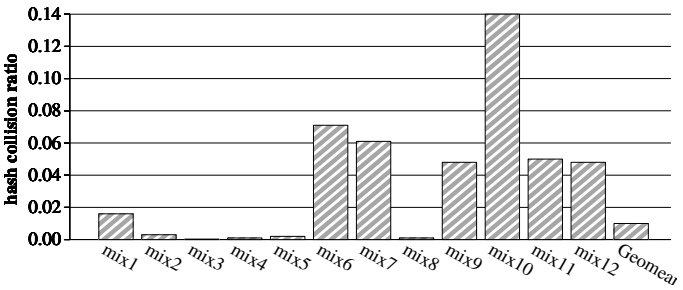

Figure 9: Hash collision.

6.5 Process Latency

The deduplication latency is hidden by the memory access. On each LLC miss, the duplication detection is launched to analyze a previously stored cache block. The analyzed data is hashed to a bucket and compared with all the data indexed in that bucket until a match occurs or mismatches with all the indexed data. Data comparison is completed well within one cycle using a simple circuit, assuming 12 FO4 delays [2]. Thus, the duplication detection takes *(number of look-ups × (1 + data comparison))* cycles on average, which is less than 10 cycles and thus totally hidden by the memory-access latency of 250 cycles.

In adaptive cache compression, as claimed in [1, 2], compression latency is 3 cycles and decompression latency is 5

cycles. The extra access latency is on the critical path to degrade performance. Even if the compression latency of 3 cycles can be hidden by the memory-access latency, the decompression latency is unavoidable.

6.6 Hash Table Sensitivity

The size of the hash table in our experiments is 8,192 buckets with 16 nodes per bucket, leading to a 512KB storage overhead. Reducing the size of the hash table to 4,096 buckets leads to an increased number of look-ups of 5.7 on average, and the percentage of hash collision is increased to 1.3%. The performance improvement is barely changed; the difference is 0.1%. We performed experiments to measure the behavior of our technique in the presence of context switching. Space constraints prevent a full discussion, but our results indicate that our technique yields at least the same improvement compared to the baseline configuration in the presence of OS context-switching among multiple applications.

7. RELATED WORK

Data deduplication is used in disk-based storage systems to reduce storage consumption [6, 39, 12]. Address correlation [32] analyzed the phenomenon of data duplication in the L1 cache without giving a feasible implementation. Non-redundant data cache [22] proposed a sub-block-level cache-deduplication technique, requiring value-based data storage and an extra value search on the critical path. Content-based block caching [23] was an inline deduplication technique designed to improve disk-based storage systems requiring significant storage overhead, which is impractical in caches. The mergeable cache architecture [3] proposed to merge blocks from different processes with the same address and similar data. Villa *et al.* [36] proposed compressing zero-content data in the cache for energy reduction. Dusser *et al.* [8] proposed an augmented cache to store null blocks to increase effective cache capacity. The HICAMP architecture [5] applies deduplication to main memories. It uses an associative hash table, suffers from underutilization, practical design issues, and lack of consideration for collisions in the hash table. CATCH [16] proposed using cache-content-deduplication only in instruction caches without data modification.

Data compression is another technology to eliminate redundant data. Yang *et al.* [38] proposed frequent value compression in first-level caches. Zhang *et al.* [40] proposed the frequent value cache (FVC) based on the observation of frequent value locality to hold only frequently accessed values in a compressed form. Alameldeen *et al.* [2] proposed frequent pattern compression (FPC), a pattern-based compression scheme for L2 caches. By storing common word patterns in a compressed form with certain prefixes, FPC provides a compression ratio comparable to more complex schemes. To reduce useless decompression overhead, Alameldeen *et al.* [1] proposed an adaptive policy to trade dynamically between the benefit of compression with the cost overhead. Hallnor *et al.* [10] proposed a unified compression scheme to compress and decompress data in the LLC, main memory, and memory channels. Although the unified compression scheme eliminates the additional compression and decompression expense required in transferring data between the LLC and the main memory, it cannot avoid compression/decompression overhead incurred with data transferring between different cache levels. Base-delta-immediate compression [28] is another data-compression algorithm representing data using a base value and an array of differences. For value- or pattern-based compression, besides the complex compression and decompression logic and unavoidable decompress latency, another drawback is that most cache-management policies cannot be used efficiently in a compressed cache because of the variation of block sizes. Linearly compressed pages [27] is another recently proposed technique for main memory compression.

The V-way cache [30] was proposed to vary the associativity of a cache on a per-set basis to increase the effective cache capacity. Line distillation was proposed to retain only used words and evict unused words in a cache block to increase effective cache capacity. Motivated by skew-associative caches [34] and cuckoo hashing [25], Zcache [31] was proposed to provide higher associativity than the number of physical ways by increasing the number of replacement candidates.

8. CONCLUSION

We propose a practical deduplicated last-level cache with limited overhead to improve performance by increasing effective cache capacity. By exploiting block-level data duplication, cache deduplication significantly increases the effectiveness of the cache with limited area and power consumption. Compared to a conventional LLC, a deduplicated LLC uses similar chip area and power consumption while performing comparably to a double-sized conventional LLC.

This paper evaluates cache deduplication in LLCs. In future work, we will extend cache deduplication to core caches and exploit redundant information-elimination techniques in other storage units. The sequentially allocated nature of the data array in the deduplicated cache offers opportunities for power gating.

9. REFERENCES

[1] A.R. Alameldeen and D.A. Wood. Adaptive cache compression for high-performance processors. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 212–223. IEEE, 2004.

[2] A.R. Alameldeen and D.A. Wood. Frequent pattern compression: A significance-based compression scheme for l2 caches. *Dept. of Computer Sciences, University of Wisconsin-Madison, Tech. Rep*, 2004.

[3] S. Biswas, D. Franklin, A. Savage, R. Dixon, T. Sherwood, and F.T. Chong. Multi-execution: multicore caching for data-similar executions. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 164–173. ACM, 2009.

[4] D. Chen, E. Peserico, and L. Rudolph. A dynamically partitionable compressed cache. In *Proceedings of the Singapore-MIT Alliance Symposium*, January 2003.

[5] D. Cheriton, A. Firoozshahian, A. Solomatnikov, J.P. Stevenson, and O. Azizi. Hicamp: architectural support for efficient concurrency-safe shared structured data access. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, pages 287–300. ACM, 2012.

[6] T.E. Denehy and W.W. Hsu. Duplicate management for reference data. *Research Report RJ10305, IBM*, 2003.

[7] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):35, 2012.

[8] J. Dusser, T. Piquet, and A. Seznec. Zero-content augmented caches. In *Proceedings of the 23rd international conference on Supercomputing*, pages 46–55. ACM, 2009.

[9] R. W. Green. Memory movement and initialization: Optimization and control. `http://software.intel.com/`, April 4th, 2013.

[10] E.G. Hallnor and S.K. Reinhardt. A unified compressed memory hierarchy. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 201–212. IEEE, 2005.

[11] J.L. Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[12] B. Hong, D. Plantenberg, D.D.E. Long, and M. Sivan-Zimet. Duplicate data elimination in a san file system. In *Proceedings of the 21st Symposium on Mass Storage Systems (MSSâĂŹ04), Goddard, MD*, 2004.

[13] A. Jaleel, E. Borch, M. Bhandaru, SC Steely, and J. Emer. Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (tla) cache management policies. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 151–162. IEEE, 2010.

[14] A. Jaleel, H.H. Najaf-Abadi, S. Subramaniam, S.C. Steely, and J. Emer. Cruise: cache replacement and utility-aware scheduling. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 249–260. ACM, 2012.

[15] S.M. Khan, Y. Tian, and D.A. Jimenez. Sampling dead block prediction for last-level caches. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 175–186. IEEE, 2010.

[16] M. Kleanthous and Y. Sazeides. Catch: A mechanism for dynamically detecting cache-content-duplication and its application to instruction caches. In *Proceedings of the conference on Design, automation and test in Europe*, pages 1426–1431. ACM, 2008.

[17] P. Koutoupis. Data deduplication with linux. *Linux Journal*, 2011(207):7, 2011.

[18] N.A. Kurd, S. Bhamidipati, C. Mozak, J.L. Miller, T.M. Wilson, M. Nemani, and M. Chowdhury. Westmere: A family of 32nm ia processors. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 96–97. IEEE, 2010.

[19] J.S. Lee, W.K. Hong, and S.D. Kim. Design and evaluation of a selective compressed memory system. In *Computer Design, 1999.(ICCD'99) International Conference on*, pages 184–191. IEEE, 1999.

[20] J.S. Lee, W.K. Hong, and S.D. Kim. Adaptive methods to minimize decompression overhead for compressed on-chip caches. *International journal of computers & applications*, 25(2):98–105, 2003.

[21] D. Levinthal. Performance analysis guide for intel core i7 processor and intel xeon 5500 processors. *Intel Performance Analysis Guide*, 2009.

[22] C. Molina, C. Aliagas, M. García, A. Gonzàlez, and J. Tubella. Non redundant data cache. In *Proceedings of the 2003 international symposium on Low power electronics and design*, ISLPED '03, pages 274–277, New York, N.Y., USA, 2003. ACM.

[23] C.B. Morrey III and D. Grunwald. Content-based block caching. In *Proceedings of 23rd IEEE Conference on Mass Storage Systems and Technologies*, College Park, Maryland, May 2006.

[24] N. Muralimanohar, R. Balasubramonian, and N.P. Jouppi. Cacti 6.0: A tool to model large caches. *Research report hpl-2009-85, HP Laboratories*, 2009.

[25] R. Pagh and F.F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[26] A. Patel, F. Afram, and K. Ghose. Marss-x86: A qemu-based micro-architectural and systems simulator for x86 multicore processors. In *1st International Qemu Users' Forum*, pages 29–30, 2011.

[27] G. Pekhimenko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. Linearly compressed pages: a low-complexity, low-latency main memory compression framework. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 172–184. ACM, 2013.

[28] G. Pekhimenko, V. Seshadri, O. Mutlu, T. C. Mowry, P. B. Gibbons, and M. A. Kozuch. Base-delta-immediate compression: A practical data compression mechanism for on-chip caches. In *Proceedings of the 21st ACM International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[29] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder. Using simpoint for accurate and efficient simulation. In *Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '03, pages 318–319, New York, N.Y., USA, 2003. ACM.

[30] M.K. Qureshi, D. Thompson, and Y.N. Patt. The v-way cache: Demand-based associativity via global replacement. In *Computer Architecture, 2005. ISCA'05. Proceedings. 32nd International Symposium on*, pages 544–555. IEEE, 2005.

[31] D. Sanchez and C. Kozyrakis. The zcache: Decoupling ways and associativity. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 187–198. IEEE, 2010.

[32] R. Sendag and P.F. Chuang. Address correlation: Exceeding the limits of locality. *IEEE Comput. Architecture Letters*, 1(1):13–16, January 2002.

[33] O. Seongil, S. Choo, and J.H. Ahn. Exploring energy-efficient dram array organizations. In *Circuits and Systems (MWSCAS), 2011 IEEE 54th International Midwest Symposium on*, pages 1–4. IEEE, 2011.

[34] A. Seznec. A case for two-way skewed-associative caches. In *ACM SIGARCH Computer Architecture News*, volume 21, pages 169–178. ACM, 1993.

[35] A. Seznec. Analysis of the o-geometric history length branch predictor. In *Computer Architecture, 2005. ISCA'05. Proceedings. 32nd International Symposium on*, pages 394–405. IEEE, 2005.

[36] L. Villa, M. Zhang, and K. Asanovic. Dynamic zero compression for cache energy reduction. In *Microarchitecture, 2000. MICRO-33. Proceedings. 33rd Annual IEEE/ACM International Symposium on*, pages 214–220, 2000.

[37] D.F. Wendel, R. Kalla, J. Warnock, R. Cargnoni, S.G. Chu, J.G. Clabes, D. Dreps, D. Hrusecky, J. Friedrich, S. Islam, et al. Power7, a highly parallel, scalable multi-core high end server processor. *Solid-State Circuits, IEEE Journal of*, 46(1):145–161, 2011.

[38] J. Yang, Y. Zhang, and R. Gupta. Frequent value compression in data caches. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 258–265. ACM, 2000.

[39] T. Yang, H. Jiang, D. Feng, Z. Niu, K. Zhou, and Y. Wan. Debar: A scalable high-performance de-duplication storage system for backup and archiving. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[40] Y. Zhang, J. Yang, and R. Gupta. Frequent value locality and value-centric data cache design. In *ACM SIGOPS Operating Systems Review*, volume 34, pages 150–159. ACM, 2000.