Touché: Towards Ideal and Efficient Cache Compression By Mitigating Tag Area Overheads

Seokin Hong
Kyungpook National University
seokin@knu.ac.kr

Bulent Abali
IBM T. J. Watson Research Center
abali@us.ibm.com

Alper Buyuktosunoglu
IBM T. J. Watson Research Center
alperb@us.ibm.com

Michael B. Healy
IBM T. J. Watson Research Center
mbhealy@us.ibm.com

Prashant J. Nair
The University of British Columbia
prashantnair@ece.ubc.ca

ABSTRACT

Compression is seen as a simple technique to increase the effective cache capacity. Unfortunately, compression techniques either incur tag area overheads or restrict cache block placement to only include neighboring addresses. Ideally, we should be able to place compressed cache blocks without any restrictions or overheads.

This paper proposes Touché, a framework for storing multiple compressed blocks from arbitrary addresses within a cacheline without tag area overheads. The Touché framework consists of three components. The first component, called the "Signature" (SIGN) engine, creates shortened signatures from the tag addresses of compressed blocks. Due to this, the SIGN engine can store multiple signatures in each tag entry. On a cache access, the physical cacheline is accessed only if there is a signature match (which has a negligible probability of false positive). The second component, called the "Tag Appended Data" (TADA) mechanism, stores the full tag addresses with data. TADA enables Touché to detect false positive signature matches by providing the full tag address. The third component, called the "Superblock Marker" (SMARK) mechanism, uses a unique marker in the tag entry to indicate compressed cache blocks from neighboring physical addresses in the same cacheline. Touché is hardware-based and achieves an average speedup of 12% (ideal 13%) when compared to an uncompressed baseline.

CCS CONCEPTS

• **Hardware → On-chip resource management**; *Static memory*.

KEYWORDS

Compression, Caches, Tag Array, Data Array, Hashing

ACM Reference Format:
Seokin Hong, Bulent Abali, Alper Buyuktosunoglu, Michael B. Healy, and Prashant J. Nair. 2019. Touché: Towards Ideal and Efficient Cache Compression By Mitigating Tag Area Overheads. In *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52), October 12–16, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3352460.3358281

MICRO-52, October 12–16, 2019, Columbus, OH, USA

ACM ISBN 978-1-4503-6938-1/19/10…$15.00
https://doi.org/10.1145/3352460.3358281

1 INTRODUCTION

As Moore's Law slows down, the number of transistors-per-core for Last-Level caches (LLC) tends to be stagnating [6, 17, 18, 26, 28, 65, 66]. One can employ data compression at the hardware-level and increase the effective LLC capacity per core [4, 16, 35, 47]. Unfortunately, naively adopting data compression may incur significant tag area overheads [2, 5, 34]. This is because, in conventional caches each block needs a separate tag. As compression increases the number of blocks within the LLC, the LLC tag area overheads will also increase. We can reduce the tag area overheads by storing compressed blocks only from neighboring addresses [57–60]. This enables us to use a single overlapping tag for all compressed blocks. However, such an approach restricts the adoption of data compression to only regions that contain contiguous compressed blocks. Ideally, we would like to employ data compression within the LLC without any data placement restrictions and tag area overheads.

Tag overheads are a key roadblock for cache compression. For instance, if we store 4x more blocks, the effective LLC capacity can be increased by 4x. But we will also incur the area overheads for maintaining 4x unique tags. Furthermore, it is likely that these unique tags have no locality, cannot be combined together, and therefore incur significant area overheads [59, 60]. One can reduce the tag area overhead with placement restrictions. For instance, if we set a rule that only compressed blocks from neighboring memory addresses can reside in a physical cacheline, then we can overlap their tags. These contiguous compressed blocks are called "superblock" and their tag is called a "superblock-tag" [57, 58]. For a 4MB 8-way cache, superblock-tags can track 4 compressed blocks per cacheline with 1.35x tag area.

Restricting block placement by using superblocks reduces the benefits of compression. Figure 1 shows the effective LLC capacity for four designs executing 29 memory-intensive SPEC workloads in mixed and rate modes on a 4MB shared LLC [71]. The first design is a baseline LLC without data compression. The second design employs data compression in LLC while using superblocks. While such a design has a tag area of 1.35x as compared to the baseline, it also increases the effective LLC capacity only by 20%. This is because only blocks from neighboring addresses can be compressed and stored in the cacheline. The third design uses data compression to place compressed blocks from arbitrary addresses within the same cacheline. While this design increases the effective LLC capacity by 38%, it also requires 3.7x the tag area. The fourth design is an ideal design which places compressed blocks from arbitrary addresses in the same cacheline without any area overheads. This paper presents

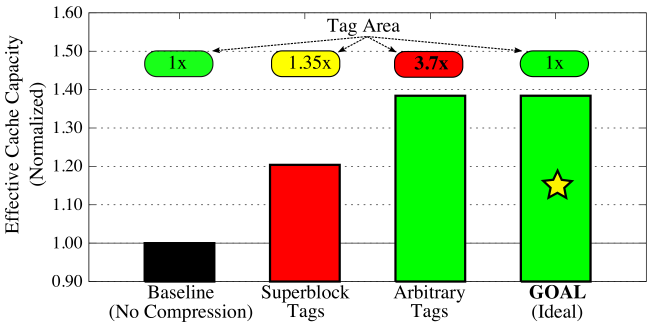

Figure 1: The effective capacity and tag area overheads for a 4MB last-level cache employing compression. Superblock-tags uses 1.35x tag area while providing 20% higher effective capacity. Arbitrary-tags uses 3.7x tag area while providing 38% higher effective capacity. The goal of this paper is to obtain 38% higher effective capacity with no tag overhead.

Touché, a framework that helps achieve the fourth design to enable near-ideal LLC compression.

Touché mitigates the tag area overheads by using a shortened signature of the full tag address for each compressed block. This has two key benefits. First, short signatures require fewer bits as compared to full tag addresses. Due to this, multiple signatures from different tags addresses can be placed in the space that was originally reserved for only a single tag address. Second, by enabling arbitrary signatures to reside next to each other, we can overcome restrictions of prior work that require compressed blocks to be from neighboring addresses. Furthermore, as compression creates unused space in the data array, tag addresses can be appended to compressed blocks and stored in this unused space.

The Touché framework consists of three components. The first component, called the "Signature" (SIGN) engine, creates shortened signatures from the tag addresses and places them in the tag array. The second component, called the "Tag Appended Data" (TADA) mechanism, appends full tag addresses to the compressed blocks and stores them in the data array. The third component, called the " Superblock Marker" (SMARK) mechanism, uses a unique marker in the tag-bits to enable Touché to identify superblocks that contain 4 contiguous compressed blocks from neighboring physical addresses. We describe each mechanism below:

(1) **Signature (SIGN) Engine**: The Touché framework is implemented within the LLC controller. The core provides the LLC controller with a 48-bit physical address for each request. The LLC controller uses this physical address to index into the appropriate set. At the same time, Touché invokes the SIGN engine to create a shortened 9-bit signature of the tag and looks up all the ways for a matching signature. On a signature match, the corresponding compressed block is accessed from the data array. As these signatures are only 9-bits long, several signatures, each belonging to a different tag address, can co-reside in a tag entry. For instance, a 4MB 8-way LLC with 64 Byte cachelines has tag entries that store 29-bit tag address. In this case, the SIGN engine can store up to three 9-bit signatures in the space that was designed for a single 29-bit tag address. This enables Touché to store up to three compressed blocks from arbitrary addresses within a cacheline without any tag area overheads.

Unfortunately, simply using shortened 9-bit signatures can lead to false positive tag matches. These cases are called as signature collisions. Signature collisions cause the LLC controller to incorrectly access blocks that do not have matching tag addresses for each access. For instance, in a workload that has a 0% cache hit-rate (worst case scenario), a 9-bit signature has an average signature collision rate of 0.19% (i.e., $\frac{1}{2^9}$). Furthermore, as each way in a set can have up to three 9-bit signatures, Touché potentially needs to check *twenty-four* 9-bit signatures in an 8-way LLC (worst case scenario) which results in a signature-collision rate of 4.58%. Therefore, it is essential to also check the full tag address on a signature collision.
(2) **Tag Appended Data (TADA) Mechanism**: The full tag addresses of the compressed blocks can be stored in the data array. Touché re-provisions a portion of the additional space that is obtained by compression to store the full tag addresses. To this end, Touché uses the "Tag Appended Data" (TADA) mechanism to append full tags beside the compressed blocks. The TADA mechanism appends metadata information on compressibility (3-bits), dirty and valid state (2-bits), and the full tag address (29-bits) to each compressed block. Overall, TADA increases the block size by only 34 bits (4.25 Bytes) and our experiments show that it only minimally reduces the effective LLC capacity. On an access, TADA interprets the last few bits in a compressed cacheline as metadata and tag addresses. As TADA checks the full tags on all signature matches and collisions, it guarantees the correctness of each LLC access.
(3) **Superblock Marker (SMARK) Mechanism**: Shortened 9-bit signatures enable storing up to three compressed blocks. However, there can be instances of cachelines that contain four compressed blocks from neighboring addresses (superblock). To address this scenario, Touché uses a "Superblock Marker" (SMARK) mechanism to generate a unique 16-bit marker. Touché stores this 16-bit marker in the tag bits, and uses this marker to indicate that there is a superblock present within a cacheline.

With a negligible probability (0.012%), the unique 16-bit marker can flag a match with the signatures that are stored by the SIGN engine. We call these scenarios as SMARK collisions. Fortunately, SMARK collisions cause no correctness problems. This is because even after a marker collision, the TADA mechanism will read the data array and check for full tag matches. During a collision, the tag addresses will not match and the compressed blocks are not processed by the LLC. The SMARK mechanism enables Touché to derive all the benefits of superblocks while also enabling the storage of up to three compressed blocks from arbitrary addresses.

Touché provides a speedup of 12% (ideal 13%) without any area overheads to the LLC. Touché requires comparators and lookup tables within the LLC controller. Therefore, Touché is a completely hardware-based framework that enables near-ideal compression.

2 BACKGROUND AND MOTIVATION

We provide a brief background on last-level cache organization and highlight the potential of data compression.

2.1 Last-Level Caches: Why Size Matters

Processors tend to have several levels of on-chip caches. Caches are designed to exploit spatial and temporal locality of accesses. Due to this, caches help improve the performance of processors as they reduce the number of off-chip accesses and reduce the latency of

memory requests. Caches are usually designed such that each level is progressively larger than its previous level. Consequently, the Last-Level Cache (LLC) tends to have the largest size, is typically shared, and occupies significant on-chip real-estate. Due to this, it is beneficial to increase the LLC capacity per core as this would enable the designers to fit a larger number of blocks on-chip and further reduce the number of off-chip accesses [70].

2.2 Last-Level Caches: Capacity Stagnation

Figure 2 shows the LLC capacity per core for commercial Intel and AMD processors from 2009 until 2018. On average, as the number of cores has increased, the LLC capacity per core has reduced. In current multi-core systems, the LLC capacity per core tends to be less than 1MB. Therefore, going into the future, it is beneficial to look at techniques to improve the effective capacity of the LLC [40].

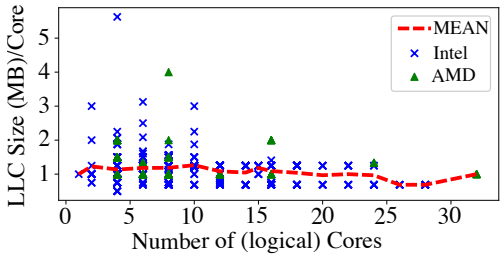

Figure 2: The Last-Level Cache (LLC) capacity per (logical) core for Intel and AMD processors from 2009 to 2018. On average, as the number of cores has increased, the LLC capacity per core has reduced.

2.3 Last-Level Caches: Organization

A Last-Level Cache (LLC) is organized into data arrays and tag arrays. Each cacheline in the data array has a corresponding tag entry in the tag array. Furthermore, groups of cachelines form "sets" and each cacheline in a set corresponds to a separate "way". As the size of the LLC is significantly smaller than the total physical address space, several blocks can map into the same set. Because of this, the LLC controller stores a tag address in the tag entry to uniquely identify the block in the cacheline. For instance, as shown in Figure 3, a 4MB 8-way LLC with 64-byte lines, uses 29 bits of tag address. On a cache access, all the tag entries for each of the ways in a set is searched in parallel by the LLC controller.

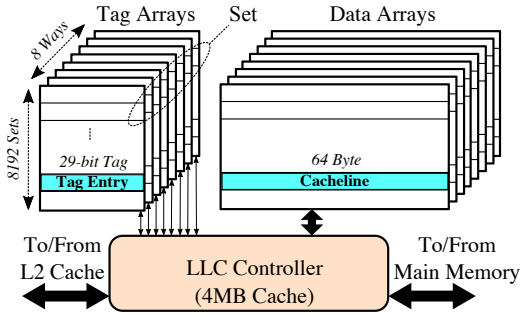

Figure 3: The organization of a 4MB Last-Level Cache (LLC). The LLC consists of data arrays, tag arrays, and an LLC controller. The tags are 29-bits long and all tag entries across the ways in a set are searched in parallel.

2.4 Compression: Higher Effective Capacity

Several prior works have proposed using efficient and low-latency algorithms to compress blocks, thereby storing more blocks and improving the effective LLC capacity. Typically, LLC compression techniques are implemented within the LLC controller.

2.4.1 Efficient Data Compression Algorithms. The Base Delta Immediate (BDI) and Frequent Pattern Compression (FPC) are two state-of-the-art low-latency compression algorithms [3, 47]. BDI uses the insight that data values tend to be similar within a block and therefore can be compressed by representing them using small offsets. FPC uses the insight that blocks contain frequent patterns like all-zeros, all-ones, etc. FPC represents frequent patterns with fewer bits. Prior work has shown that both BDI and FPC can be implemented to execute with a a single-cycle delay and can be implemented within the LLC controller [3, 47].

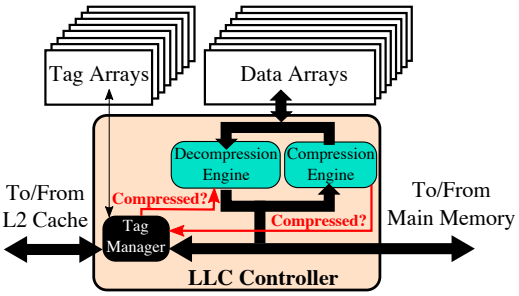

Figure 4: The LLC compression-decompression engine. The compression-decompression engine taps the data bus and stores compressibility information in the tag entries.

2.4.2 Compression-Decompression Engine. As shown in Figure 4, the LLC controller implements a compression-decompression engine that taps the bus going into the cache data array. The LLC controller contains a separate "tag manager" to manage tag entries. The compression-decompression engine implements both BDI and FPC and chooses the best algorithm. The tag manager maintains the compressibility information in the tag entries.

2.4.3 Distribution of Compressed Data Size. Figure 5 shows the distribution of the size of blocks after compression for 29 SPEC workloads containing rate and mixed workloads. On average, 55% of the blocks can be compressed to less than 48 bytes in size. Furthermore, 17% of the lines can be compressed to be less than 16 bytes in size. Therefore, several workloads tend to have blocks with low entropy and can benefit from compression.

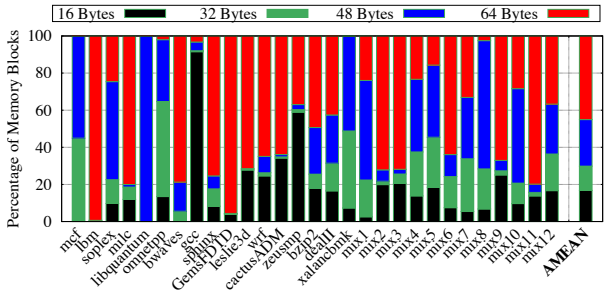

Figure 5: The distribution of the size of blocks for 29 SPEC workloads (rate and mix modes). On average, up to 55% of the blocks can be compressed to 48 Bytes.

2.5 LLC Compression: Tag Area Overheads

Modern computing systems tend to operate on 64-byte blocks. Figure 6 (a) shows the design of the tag entry and the cacheline in the data array for a 4MB 8-way LLC that does not employ compression. The tag entry for each block requires a valid bit and a dirty bit. Furthermore, we assume that replacement policy is maintained at the cacheline-level and the largest block in the selected cacheline is evicted. To reduce the number of encoding bits in the tag array, blocks are compressed into 16 byte, 32 byte, or 64 byte boundaries. To reduce the number of bits in the tag entry further, one can restrict cachelines to store blocks only from contiguous addresses. Such a contiguous set of blocks is called a superblock. Prior work has shown that superblocks with 4 compressed blocks can reduce the tag area overheads to 8 bits [58]. As shown in Figure 6 (b), a 4MB 8-way LLC that stores up to four blocks per cacheline will require 46 bits of tag entry. While superblocks help reduce tag area overheads, they limit the potential benefits of LLC compression as they restrict block placement to include only neighboring addresses. If one can store blocks from arbitrary addresses, we can unlock all the benefits of LLC compression. However, the disadvantage of this approach is that, as shown in Figure 6 (c), a 4MB 8-way LLC that stores up to four blocks per cacheline will require 127 bits of tag entry (3.7x higher than the baseline).

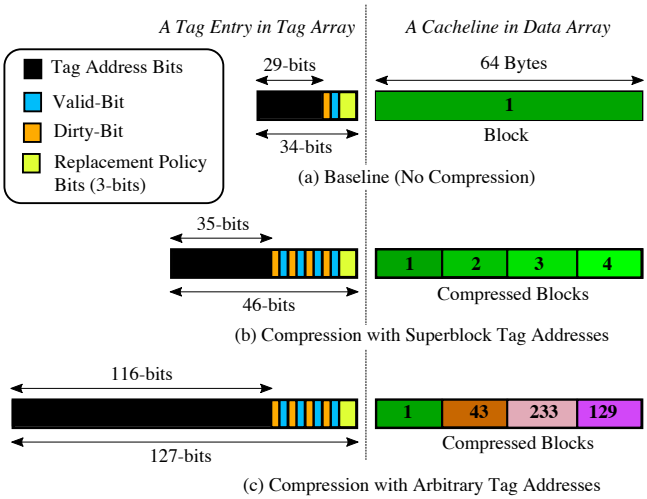

Figure 6: The tag area overheads for different techniques. (a) The baseline technique that does not employ any compression has no tag area overheads. (b) The superblock technique increases the tag area to 1.35x. (c) Storing arbitrary tags increases the tag area to 3.7x.

2.6 LLC Compression: Potential

Figure 7 shows the overheads and benefits of LLC compression for three techniques. The baseline technique does not employ compression, has no tag overheads and has an average hit-rate of 31.5%. The second technique employs superblocks for compression, has a tag area of 1.35x and increases the average hit-rate to 36%. The third technique highlights the potential hit-rate with compression when each cacheline can store up to 4 compressed blocks. Unfortunately, the third technique uses a tag area of 3.7x while also increasing the average LLC hit-rate to 38.5%.

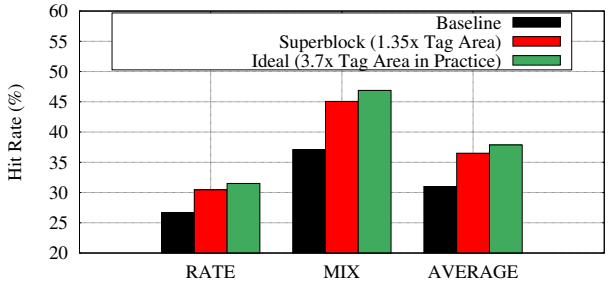

Figure 7: The potential of LLC compression. Enabling blocks from arbitrary addresses increases the average hit-rate of the LLC from 31.5% to 38.5%.

3 THE TOUCHÉ FRAMEWORK

3.1 An Overview

Figure 8 shows an overview of the Touché framework. Touché consists of three components. The first component, called the Signature (SIGN) Engine, generates shortened signatures of the tag addresses. The SIGN engine is designed within the tag manager. The second component, called the Tag Appended Data (TADA) mechanism, attaches full tag addresses to compressed memory blocks. The TADA mechanism taps the data bus after the compression-decompression engine and obtains the full tag address from the tag manager. The third component, called Superblock Marker (SMARK) mechanism, keeps track of superblocks by using a unique 16-bit marker in the tag entry. The SMARK mechanism is implemented in the tag manager. Touché requires changes only in the LLC controller.

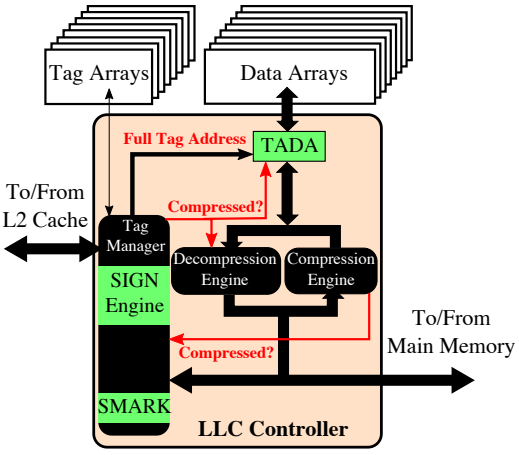

Figure 8: An overview of Touché. Touché consists of three components. The Signature (SIGN) Engine, the Tag Appended Data (TADA) mechanism, and the Superblock Marker (SMARK) mechanism. All components are implemented in the LLC controller with no changes to the LLC.

3.2 Signature (SIGN) Engine

The Signature (SIGN) Engine is implemented in the tag manager. The SIGN Engine generates shortened signatures from the full tag addresses supplied during the read and write accesses to the LLC.

3.2.1 Identifying Compressed blocks. On a LLC write, the compression-decompression engine informs the tag manager if the block is compressible; a block can be compressed to 16B, 32B or 48B. The tag

manager uses the original valid bit and the dirty bit in its tag entry to encode this information. We use the insight that, for valid blocks, the valid bit and the dirty bit can only exist in two states. For instance, a cacheline cannot be marked as both invalid and dirty at the same time. The tag manager uses this unused state to flag cachelines that contain compressed blocks. Thereafter, for a cacheline that stores compressed blocks, the 1^{st} and 2^{nd} bits of the tag address are used to indicate if any of these blocks are dirty.

As shown in Table 1, on a read, the tag manager checks the original valid and dirty bits in the tag entry to identify if the cacheline contains valid compressed blocks. If the cacheline is deemed to contain valid compressed blocks, then the tag manager reads the 1^{st} and 2^{nd} bits from the tag address to determine if any of the blocks are dirty.

Table 1: Identifying Compressed blocks

Cacheline Status	Valid Bit	Dirty Bit	Tag Address 1^{st} Bit	Tag Address 2^{nd} Bit
Invalid	0	0	N/A	N/A
Uncompressed: Valid and Clean	1	0	N/A	N/A
Uncompressed: Valid and Dirty	1	1	N/A	N/A
Compressed: Valid and Clean	**0**	**1**	**1**	**0**
Compressed: Valid and Dirty	**0**	**1**	**1**	**1**

3.2.2 Using Shortened Signatures. To store multiple signatures within a single tag entry, the SIGN engine shortens the full tag address into a 9-bit signature. For a 4MB 8-way LLC, the full tag address is 29-bit long. For a cacheline containing compressed blocks, as the top 2 bits of the tag address space in its tag entry are already used to indicate if any of blocks are dirty, we have 27 unused bits remaining in the tag address space. Therefore, we can store up to three 9-bit signatures corresponding to three compressed blocks.

Figure 9 shows the design of the signature generator in the SIGN engine. The signature generator uses the least 27-bits of the full tag address and divides it into three 9-bit segments. Each bit of these 9-bit segments is then XORed together to generate a 9-bit output. The 9-bit output then indexes into a 512 entry hash table. Each entry in the hash table is populated at boot-time with unique hashes. The hash table then generates a unique 9-bit signature for each 9-bit input. The overall latency of generating signatures is the delay of one 3-bit XOR gate and one parallel hash table lookup. For a high-performance processor executing at 3.2GHz, we estimate the signature generation to incur only 1 cycle. Furthermore, the latency of signature generation is masked by the latency of reading the tag entries for each LLC access (up to 5 cycles).

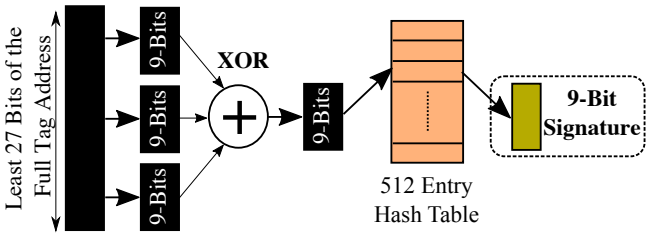

Figure 9: The signature generator within the SIGN Engine. The signature generator only requires one XOR operation and a hash table lookup for each LLC access.

3.2.3 Checking for Matching Signatures. On an LLC read, the tag manager reads the tag entries from all the ways of the indexed set. At the same time, the SIGN engine forwards its 9-bit signature to the tag manager. The tag manager identifies if the cacheline contains compressed blocks using the original valid and dirty bits. For an uncompressed cacheline, the tag manager ignores the signature and uses the full tag address to check for a match.

If the cacheline contains valid compressed blocks, the tag manager ignores the first two bits of the tag address. Thereafter, the remaining 27 bits in the tag address space of the tag entry are partitioned into three 9-bit entries. The tag manager then compares each of these three 9-bit entries with the 9-bit signature from the SIGN Engine. If the 9-bit entry does not match the 9-bit signature, then the block is guaranteed to be absent. On the other hand, if the 9-bit signature matches in any one of the ways, then the block is likely to be present. As a 9-bit signature is smaller than its full 29-bit tag address, there is a small chance of 0.19% ($\frac{1}{512}$) that each 9-bit entry comparison with the 9-bit signature can result in a false positive match. We call these false positive matches of signatures as "signature collisions".

3.2.4 Collision Rate of Signatures. As each tag entry can store up to three 9-bit signatures, an 8-way LLC would require up to twenty-four 9-bit signature comparisons for each access. As signatures are shorter than full tags, several tags may map into the same signature. As we perform twenty-four signature checks (in the worst-case), it is likely that some of LLC accesses will result in signature collisions. Figure 10 shows the probability of collisions as the number of signatures present in the 8-ways varies from zero (all ways are uncompressed) to twenty-four (all ways have three compressed blocks) for different LLC hit-rates.

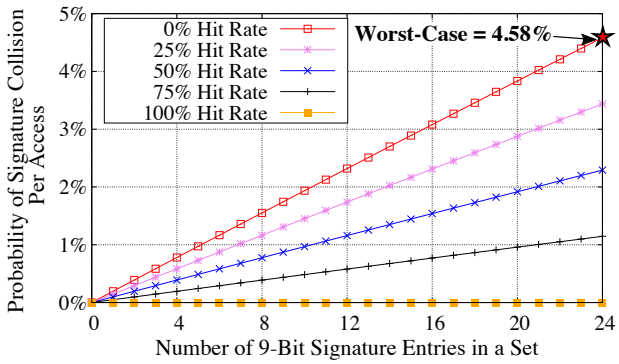

Figure 10: The probability of collision for a 9-bit signature as the number of signature entries vary in a set. In the worst-case, for an 8-way LLC, we expect a signature collision to occur 4.58% of the time for each access.

In the worst case, for a workload that has 0% hit-rate, we can expect a collision to occur 4.58% of the time. As signature collisions can cause the LLC to forward blocks with incorrect tag addresses to the cores, it is essential to check full tags.

3.3 Tag Appended Data (TADA) Mechanism

The Tag Appended Data (TADA) mechanism is implemented in the LLC controller and taps the data-bus between the compression-decompression engine and the data array.

3.3.1 Appending Full Tag Addresses to Data. During an LLC write, the TADA mechanism uses the full tags that are supplied by the tag manager. The TADA mechanism then appends the full tag addresses (29 bits), the valid-bit, the dirty-bit, and the compressibility information (3 bits) to the end of the cacheline (total 34 bits or 4.25 Bytes). Figure 11 shows a cacheline storing three compressed blocks and the TADA mechanism appending the information for each of these blocks at the end of the cacheline.

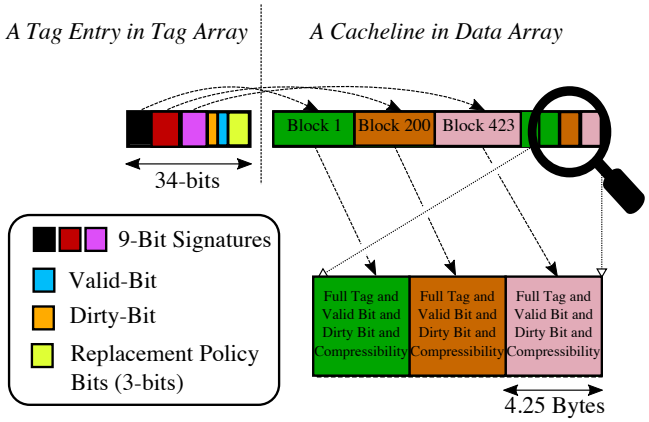

Figure 11: A cacheline storing compressed blocks with TADA mechanism. The TADA mechanism appends full tag addresses, valid bit, dirty bit, and compressibility information for each block at the end of the cacheline.

3.3.2 Appending Full Tag Addresses to Data. The TADA mechanism appends 34 bits (4.25 Bytes) of information to the end of the cacheline containing compressed blocks. As a result, TADA reduces the space available to store the compressed block. We can store three 16B compressed blocks or a pair of a 32B and a 16B compressed blocks in the data array; the block size is stored in the compressibility information field. Fortunately, this additional loss of space only causes a few lines to reduce their effective capacity. Figure 12 shows the reduction in effective LLC capacity due to TADA for an LLC that can store up to 3 arbitrary compressed blocks. TADA decreases the effective cache capacity by only 4.15% points as compared to an ideal scheme that can store three compressed blocks from arbitrary addresses without any storage overheads.

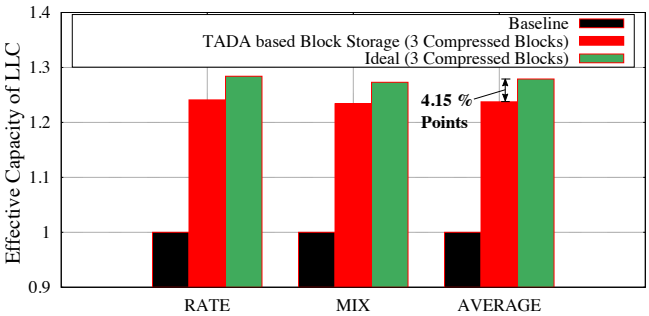

Figure 12: The reduction in the effective LLC capacity by the TADA storage overhead. While TADA uses 4.25 Bytes per compressed block, it decreases the effective LLC capacity only by 4.15% points as compared to an ideal scheme that does not require the metadata storage in the data array.

3.3.3 Detecting Collisions of Signatures. TADA helps detect signature collisions. This is because, on a signature collision, the cachelines from the selected way(s) in the data array are read by the tag manager. The TADA mechanism extracts the full tag address from the cachelines and checks if they match the full tag address of the LLC access. If there is no match, TADA flags a signature collision. Therefore, TADA guarantees the detection of signature collisions and thereby ensures correctness. Furthermore, TADA extracts the the compressibility information and supplies to the decompression engine. The valid and dirty bits of compressed blocks are also stored using TADA. Therefore, TADA helps to avoid using any additional bits in the tag entry to store additional information.

3.4 Latency Overheads

As the data array needs to be accessed during a signature collision, it can increase the LLC access latency.

3.4.1 Additional Accesses to Data Arrays. In the baseline system, an LLC access probes all the ways in the indexed set from the tag array. The cacheline from the data array is read-only in case of an LLC hit. As a tag access occurs for every access, irrespective of whether the access is a hit or a miss, the tag array is designed with lower access latency as compared to the data array. Typically, accessing the LLC tag array incurs a latency overhead of only 5 cycles. On the other hand, reading the LLC data array incurs an overhead of 30 cycles in modern processors [25].

In the Touché framework, the data arrays are likely to be accessed even in case of an LLC miss. This is because the SIGN engine may incur signature collisions and may invoke the TADA mechanism to access the data array to detect signature collisions. In the worst-case, for a workload with 0% hit-rate, this scenario may occur only 4.58% of the times. Therefore, signature collisions will increase the overall latency of LLC access. Table 2 shows the average latency of an LLC access during a collision.

Table 2: Additional Data Arrays Accessed on a Collision

Number of Data Arrays Accessed	Probability	Latency (cycles)
1	0.9768	35
2	0.0229	70
3+	0.0003	105+
Average: 1.0235	1	**35.82**

In the worst-case, all accesses can be a cache miss and a collision can occur 4.58% (probability of 0.0458) of the times. As shown in Table 2, collisions increase the access latency to 35.82 cycles. For a worst-case workload with a 0% hit-rate, the increase in the LLC tag access latency is denoted by Equation 1.

$$\text{New Tag Access Latency} = (1 - 0.0458) \times \text{Old Latency} + 0.0458 \times \text{Collision Latency} \quad (1)$$

As the old tag access latency is 5 cycles and the collision latency is 35.82 cycles, the new tag access latency of Touché is 6.4 cycles.

3.4.2 Mitigating Latency Overheads with Dynamic Touché: One can mitigate signature-collision latency overheads by compressing only when it is useful. To this end, Touché continuously monitors the average memory latency at the LLC controller. The average memory latency is defined as the total latency that is experienced by each request and this can emanate from the LLC and main memory.

Touché enables compression only when the average memory access latency is 100x greater than the latency overheads of signature collisions. As signature collisions increase the LLC tag access latency by 1.4 cycles, Touché enables compression only when the average memory latency is greater than 140 cycles. This has two key advantages. First, Touché is only enabled for workloads that showcase a large memory latency and benefit from LLC compression. Second, the latency overheads from Touché will be capped at 1%. As shown in Figure 13, for memory intensive benchmarks, the average memory latency for reads is 541 cycles (significantly higher than 140 cycles). Therefore, the observed latency overhead of Touché is only 0.26%.

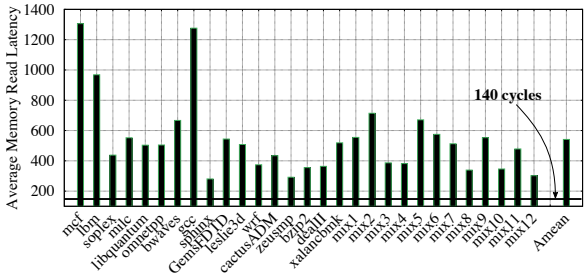

Figure 13: The average memory latency for reads. On average, the average memory access latency is 541 cycles. Therefore, Touché has a latency overhead of only 0.26%.

3.5 Superblock Marker (SMARK) Mechanism

The SIGN Engine enables storage of up to three blocks. However, some cachelines may contain superblocks.

3.5.1 Benefits of Including Superblocks. Figure 14 shows the hit-rate of Touché while maintaining up to three compressed blocks and compares it against a scheme that also stores superblocks (up to four neighboring blocks). For a superblock, Touché tries to compress each block to 15 Bytes. This enables Touché to get the benefits of storing both the superblock-tags and the arbitrary tags. If we can store superblocks and arbitrary blocks at the same time, we can increase the average hit-rate of Touché from 31.5% to 37.5%.

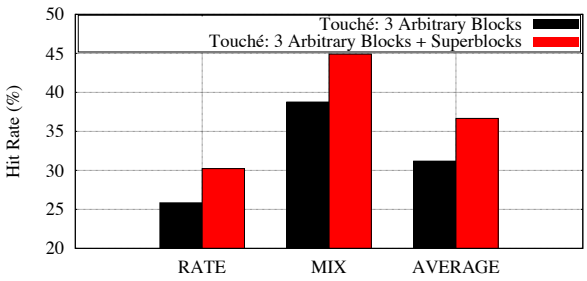

Figure 14: The average hit-rate of Touché with 3 blocks versus 3 blocks with superblocks. On average, the hit-rate increases to 37.5% by combining superblocks.

3.5.2 Identifying Potential Cachelines. During an LLC install, if the block is compressible and if the candidate cacheline already contains compressed blocks from its neighboring addresses, then this cacheline is also a superblock candidate. The TADA mechanism is used to identify superblock candidates by extracting the full tag addresses for all the blocks in a cacheline during an LLC install.

3.5.3 Generating Markers. Touché implements a "Superblock Marker" (SMARK) mechanism in the tag manager. SMARK mechanism generates a random 16-bit marker at boot-time and uses this marker throughout the operational time of the system.

Once the TADA mechanism identifies a superblock cacheline, it informs the tag manager. The tag manager then retrieves the 16-bit marker from the SMARK mechanism. It then informs the SIGN engine to ignore the last 2-bits (corresponding to four neighboring addresses) of the full tag address to generate a unique 9-bit signature. This ensures that neighboring addresses in the superblock generate the same 9-bit signature. Thereafter, the tag manager appends the 9-bit signature to the 16-bit marker and writes the 27-bit tag of the first block within the superblock at the end of the cacheline. The TADA mechanism also appends a dirty-bit per each compressed block to the data array. For a valid superblock, all compressed blocks within a superblock are always valid. Therefore, TADA does not need to append additional per-block valid bits. Figure 15 shows the superblock marker mechanism.

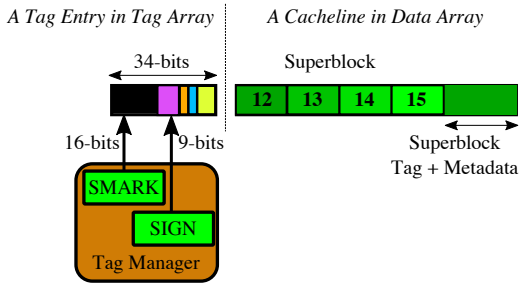

Figure 15: The Superblock Marker (SMARK) mechanism.

3.5.4 Checking for Matching Markers. On a read, the tag manager will check for matching 16-bit marker values in all the ways that store compressed blocks within a set. If there is a marker match, then the tag manager uses the 9-bit signature (generated from by ignoring the least two significant bits) and checks for a match.

If the signature matches, then the cacheline is read from the data array. The TADA mechanism extracts the 27-bit tag address and checks if the LLC access is for one of the blocks within the superblock. If there is a match, the block is processed by the LLC controller. It is likely, the cacheline may not contain the requested block and it may simply be a false positive match. Similar to signature collisions, we call this scenario as a marker collision.

3.5.5 Effect of Marker Collisions. Marker collisions are extremely rare. This is because, we use markers which are 16 bits long. For instance, in an 8-way cache, the probability of a marker collision for each access is only 0.012% and their impact on LLC latency is negligible. Furthermore, even in the case of marker collisions, the TADA mechanism ensures that the full tag address is checked before forwarding the compressed block. Therefore, SMARK works with TADA to *guarantee* correctness while storing superblocks.

3.6 Touché Operation: Reads and Writes

Figure 16 (a) and Figure 16 (b) show the flowchart for Touché for LLC access and install requests respectively. Touché invokes the SIGN, TADA, and SMARK mechanisms only for compressed data. For incompressible data, Touché works just like the baseline. Furthermore, TADA mechanism is always invoked for LLC hits of compressed blocks. This enables Touché to guarantee correctness.

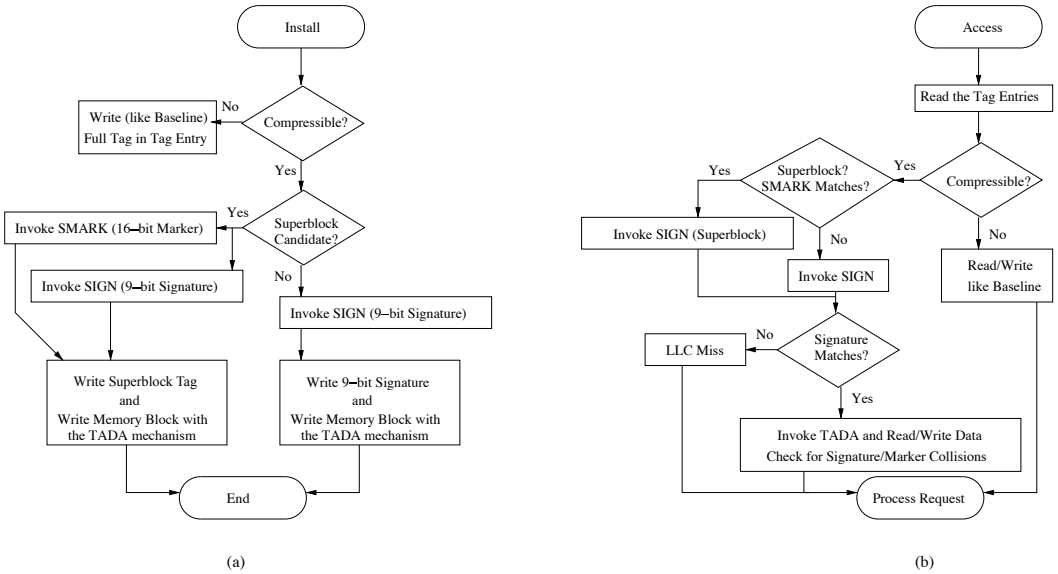

Figure 16: The flowchart detailing the high-level operations of Touché for install and access requests. (a) Shows the flowchart for install requests. (b) Shows the flowchart for access requests.

3.7 Discussions

In the baseline LLC, the tag entry contains metadata such as the replacement policy bits and coherence states (for private LLCs). We discuss how these affect the design of Touché.

3.7.1 Handling Cache Coherence: Touché assumes a shared LLC and therefore does not encounter coherence concerns. However, in case the LLC is private and a snooping-based coherence protocol is employed, Touché will need to maintain coherence states with minimal overheads. Touché will store the coherence state as well as the full tag for each compressed block in the data array. Thus, the LLC controller needs to access the data array for tag matching and checking the coherence states. This operation will increase the latency of tag matching for the coherence requests.

However, such an operation can be designed to incur low performance overheads. This is because, the additional access to the data array is required only if there is a signature match. Also, the coherence state can be updated *after* the critical requests are serviced. Additionally, if the coherence request is a "BusRd" (read request made by another core), the servicing core needs to send the entire block to the requesting core anyway. In this case, the additional data array access does not cause any performance overheads. Furthermore, as implemented in commercial processors, we can eliminate the performance impact of snooping-based coherence protocols by simply using a directory-based coherence protocol [27].

3.7.2 Handling Cache Replacement Policy: Each tag entry in the baseline system is already equipped with replacement information bits. As Touché stores multiple compressed blocks per cacheline, ideally, it would be preferable to equip each of these blocks with additional replacement bits in the tag entry. However, this would require us to add 3 ∼ 4 bits per compressed block in the tag entry.

To minimize the overheads for storing the replacement information, whenever a cacheline is accessed, Touché only updates the original replacement bits. Touché does not keep track of individual replacement bits for each block. During replacement, Touché selects the victim cacheline based on the original replacement bits and randomly evicts one block from within the victim cacheline.

4 EXPERIMENTAL METHODOLOGY

To evaluate the performance benefits of Touché, we develop a trace-based cache simulator based on the USIMM [9] (which is a detailed main memory system simulator). We extended the USIMM to model processor cores and a detailed cache hierarchy. Our processor model implements an out-of-order (OoO) execution of the benchmark trace. Our detailed cache model supports various replacement policies such as LRU, DRRIP [29], and DIP [52]. The baseline system configuration is described in Table 3. To enable efficient compression, the compression engine modeled in the cache model employs the BDI [47, 48] and FPC [3] compression algorithms and uses the algorithm that provides the best compression ratio for each cacheline. Based on prior work in BDI and FPC, we assume that compression and decompression of data incurs only a single-cycle latency. We compare Touché to a previous state-of-the-art scheme called YACC that uses only "superblocks" [58]. We also compare our scheme against an "Ideal" scheme that can store either three arbitrary blocks or a superblock (four neighboring blocks) without any area overheads. The Ideal scheme uses the same replacement policy as Touché (described in Section 3.7.2).

Table 3: Baseline System Configuration

Number of cores (OoO)	4
Processor clock speed	3.2 GHz
Issue width	8
L1 Cache (Private)	32KB, 8-Way, 64B lines, 4 cycles
L2 Cache (Private)	256KB, 8-Way, 64B lines, 12 cycles
Last Level Cache (Shared)	4MB, 8-Way, 64B lines
LLC Tag Access Latency	5 cycles
LLC Data Access latency	30 cycles
Memory bus frequency	1600MHz (DDR 3200MHz) [30]
Memory channels	2
Ranks per channel	1
Banks Groups	4
Banks per Bank Group	4
Rows per bank	64K
Columns (cache lines) per row	128
DRAM Access Timings: T_{RCD}-T_{RP}-T_{CAS}	22-22-22 [8]
DRAM Refresh Timings: T_{RFC}	420ns [39, 53]

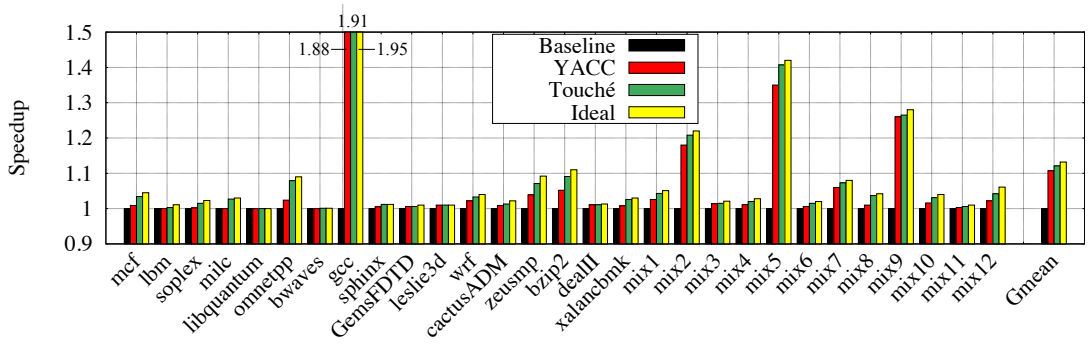

Figure 17: Speedup of Touché as compared to a baseline system that does not employ compression. On average, Touché achieves a speedup of 12% (Ideal – 13%, YACC – 10.3%) by enabling compressed blocks from arbitrary addresses to be placed next to each other while also allowing superblocks to be stored.

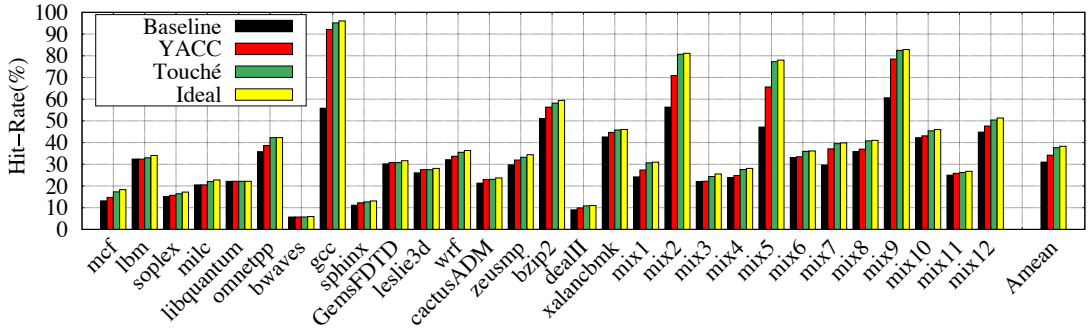

Figure 18: The Hit-Rate of Touché as compared to a baseline system that does not employ compression. On average, Touché increases the hit-rate by 6% points (Ideal – 7% points, YACC – 4% points) by storing a larger number of blocks within the LLC.

We chose memory intensive benchmarks, which have greater than 1 MPKI (LLC Misses Per 1000 Instructions), from the SPEC CPU2006 benchmarks. We warm up the caches for 2 Billion instructions and execute 4 Billion instructions. To ensure adequate representation of regions of compressibility [12] and performance [51], the 4 Billion instructions are collected by sampling 400 Million instructions per 1 Billion instructions over a 40 Billion instruction window. We execute all benchmarks in rate mode. We also create twelve 4-threaded mixed workloads by forming two categories of SPEC2006 Benchmarks; a low MPKI category and a high MPKI category. As described in Table 4, we randomly pick one benchmark from each category to form high MPKI mixed workloads and medium MPKI mixed workloads. We perform timing simulation until all the benchmarks in the workload finish execution.

Table 4: Workload Mixes

mix1	mcf, libquantum, GemsFDTD, wrf
mix2	lbm, gcc, bzip2, bwaves
mix3	milc, sphinx, leslie3d, zeusmp
mix4	soplex, omnetpp, cactusADM, dealII
mix5	xalancbmk, mcf, gcc, sphinx
mix6	omnetpp, lbm, milc, xalancbmk
mix7	astar, mcf, milc, calculix
mix8	omnetpp, gobmk, sjeng, libquantum
mix9	namd, gcc, lbm, dealII
mix10	soplex, tonto, hmmer, perlbench
mix11	GemsFDTD, bwaves, povray, zeusmp
mix12	wrf, xalancbmk, h264, gamess

5 RESULTS

This section discusses the performance, hit-rate, and sensitivity results of Touché.

5.1 Performance Impact

Figure 17 shows the speedup of Touché when compared to a baseline system that does not employ compression. On average, Touché has a speedup of 12%. Ideally, if we can place compressed memory blocks without any area overheads in tag and data arrays, we get a speedup of 13%. On the other hand, YACC achieves 10.3% speedup by capturing the superblocks. Our analysis shows that *gcc* benefits the most from LLC compression. *gcc* is extremely sensitive to the LLC capacity and as the miss rate of *gcc* drops from 45% to 5% (by 9x) due to Touché, *gcc* experiences a low memory access latency. This is because, at 5% miss-rate, almost all of its working set now fits in the LLC. Therefore, *gcc* shows a speedup of 91% due to Touché. For all other workloads, the drop in *miss-rate* is at most 2.4x (see Figure 18) and they show up to speedup of up to 50%.

5.2 Effect on Last-Level Cache Hit-Rate

Figure 18 shows the speedup of Touché when compared to a baseline system that does not employ compression. On average, Touché increases the hit rate by 6% points. In the ideal case, if we can place compressed memory blocks from arbitrary addresses without any area overheads in tag and data arrays, the hit-rate increases by 7% points. On the other hand, YACC increases the hit-rate by 4% points. Furthermore, some workloads like *gcc*, *mix2*, *mix5*, and *mix9* obtain significant increase in hit rate.

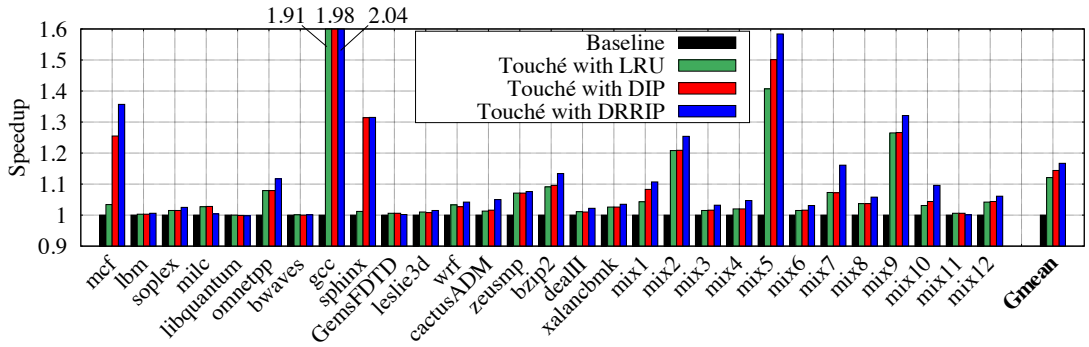

Figure 19: The sensitivity of Touché to different replacement policies. As Touché is only a LLC compression framework, it is orthogonal to the replacement policy. Touché shows an increasing speedup of 12%, 14.5%, and 16.7% for the LRU, DIP, and DRRIP replacement policies respectively.

We also observe that the hit-rates of Touché either increase or remain the same for benchmarks. Furthermore, Touché closely follows the hit-rate of an ideal LLC compression technique. The slight loss in hit-rate from the ideal LLC compression is because of the loss in capacity within the data array from the TADA mechanism.

5.3 Sensitivity to Replacement Policy

As Touché is an LLC compression technique, it is orthogonal to the replacement policy. Typically, the LLC controller will choose a victim cacheline based on a replacement policy. Touché then evicts a random block from within the victim cacheline. Therefore, different replacement policies can be combined alongside Touché.

Figure 19 shows the speedup of Touché for three different cache replacement policies. On average, Touché increases the speedup from 12% while using LRU, to 14.5% while using DIP. The speedup is further increased to 16.7% while using the DRRIP replacement policy. Therefore, irrespective of the replacement policy, Touché continues to provide high performance with efficient compression.

5.4 Impact on Memory Latency

Figure 20 shows the impact of Touché on the average memory latency for reads. As Touché provides a higher LLC hit rate, it also reduces the average memory read latency. On average, Touché reduces the memory read latency from 541 cycles to 489 cycles. In the ideal case, we can reduce the average memory read latency to 478 cycles as this scheme provides slightly higher hit-rate .

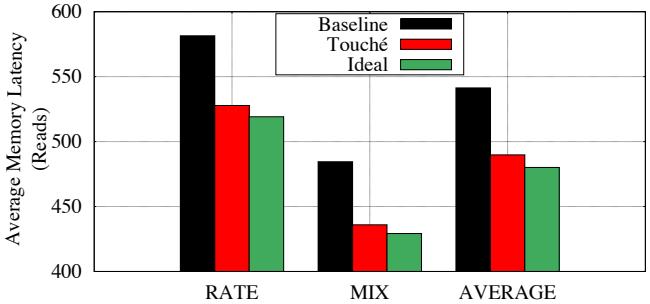

Figure 20: The average memory read latency for Touché. On average, Touché reduces the memory read latency from 541 cycles to 489 cycles.

5.5 Sensitivity to Last-Level Cache Size

Figure 21 shows the impact of LLC size on the effectiveness of Touché. Touché continues to be effective at different LLC sizes. For instance, even while using a 2MB cache, Touché provides an average speedup of 10%. Even after doubling the LLC size from 4MB to 8MB, Touché still provides an average speedup of 9%.

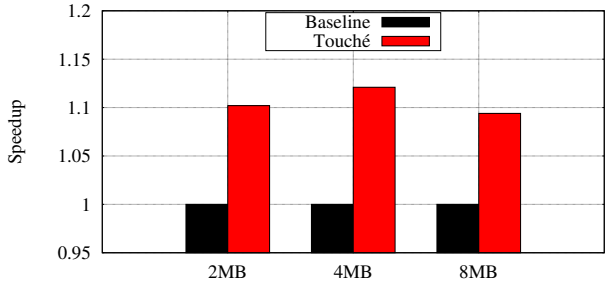

Figure 21: The sensitivity of Touché to the size of the LLC. Even after varying the LLC size, Touché continues to provide an average speedup of atleast 9%.

5.6 Impact on Low-MPKI Benchmarks

Until now, we have presented the results for high MPKI benchmarks. However, for implementation purposes, it is vital that Touché does not hurt the performance of low MPKI benchmarks. Figure 22 shows the impact of Touché on the performance of Low MPKI workloads from the SPEC2006 suite. Overall, Touché does not cause slowdown for any Low MPKI benchmark. On the contrary, Touché provides an average speedup of 1.9% for these workloads.

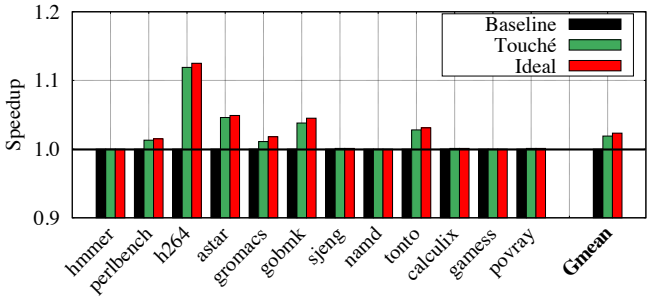

Figure 22: Impact of Touché on low MPKI workloads. Touché does not hurt the performance of any low MPKI benchmark. Touché provides an average speedup of 1.9% for these workloads.

Table 5: Comparison: Efficient Tag Management Techniques

Techniques	Last-Level Cache [57–60]	DRAM Cache		Variable Granularity Cache [37]	Touché
		Tags in Cache [38]	High-Bandwidth [74]		
Compression	✓	✗	✓	✗	✓
Stores Superblocks	✓	✗	✓	✗	✓
Stores Arbitary Blocks	✗	✓	✗	✓	✓
Maintains Entire Memory Block	✓	✓	✓	✗	✓
Tag Overhead	35%	0%	0%	0%	0%

6 RELATED WORK

In this section, we describe prior work that is closely related to the ideas discussed in this paper.

6.1 Efficient Compression Algorithms

Cache compression algorithms like Frequent Pattern Compression (FPC) [3], Base-Delta-Immediate (BDI) [47], and Cache Packer (C-PACK) [10] have low decompression latency and require low implementation cost (i.e., area overhead). The C-PACK algorithm can be improved further by detecting zero cache lines [57]. Recently, Kim et al. [33] introduce a bit-plane compression algorithm that uses a bit-plane transformation to achieve a high compression ratio. Touché is orthogonal to all of these compression algorithms. Touché can select any of these algorithms to meet the hardware budget, latency constraints, and application's requirements.

6.2 Cache Compression with Tag Management

Prior works have proposed compressed cache architectures to improve the effective cache capacity [2, 57–60]. For instance, a variable-size compressed cache architecture using FPC was proposed [2]. This architecture doubles the cache size when all cachelines are compressed while requiring twice as many tag entries.

Table 5 lists techniques that propose efficient tag management. To reduce tag overhead of the compressed cache, DCC [60] and SCC [57] use superblocks to track multiple neighbor blocks with a single tag entry. Recently, YACC [58] was proposed to reduce the complexity of SCC by exploiting the compression and spatial locality. YACC still restricts the mapping of compressed cachelines as it requires superblocks that contain cachelines only from neighboring addresses. Furthermore, YACC requires that those cachelines be of the same compressed size. Touché eliminates this fundamental limitation of the super block-based compressed cache. On average, YACC provides 10.3% speedup while requiring additional bits in the tag area resulting in 1.35x tag area. Touché provides 12% speedup without any area overheads. To increase LLC efficiency, Amoeba-Cache [37], proposes storing tag and data together while eliminating the tag area. However, to create space for tags, Amoeba-Cache stores only parts of the memory block within the cache. As DRAM caches do not encounter tag storage problems and tend to be bandwidth sensitive, Young et. al. [74] use compression in DRAM caches to improve both capacity and bandwidth dynamically.

6.3 Compression using Deduplication

Data deduplication exploits the observation that several memory blocks in the LLC contain the same identical value [15, 23, 67]. To improve efficiency, these techniques store only a single value of these memory blocks within the LLC and design techniques to maintain tags that point to such memory blocks.

Exploit the presence of identical memory blocks in the LLC, Dedup [67] changes the LLC to enable several tags to point to the same data. To this end, the tag array is decoupled from the data array. Each tag entry is then equipped with pointers to enable them to point to arbitrary memory blocks in the data array. Touché is orthogonal to Dedup, as Touché is compression technique that compresses arbitrary memory blocks independently and enables Dedup to be applied over it.

6.4 Main Memory Compression Techniques

Compression can also be used for main memory. Memzip compresses data for improving the bandwidth of the main memory [63]. Pekhimenko et. al. [48] and Abali et. al [1] have proposed efficient techniques to improve the effective capacity of main memory using compression. Compression can also be used in Non-Volatile Memories (NVM) to reduce energy and improve performance [46]. As compression increases the number of bit-toggles on the bus, Pekhimenko et. al. [50] minimizes bit-toggles and reduces the bus energy consumption. Recently, Compresso [13] memory system was proposed to reduce the additional data movement caused by metadata accesses for additional translation, changes in compressibility of the cacheline, and compression across cacheline boundaries. Similarly, DMC [36] was proposed to improve memory capacity.

Compression can use software support and increase the main memory capacity. Products like IBM MXT and VMWare ESX use "Balloon Drivers" to allocate and hold unused memory when data becomes incompressible or when Virtual Machines exceed capacity thresholds [7, 19, 64, 68, 69]. One can also use compression in the context of memory security. Morphable Counters [55] compress integrity tree and encryption counters to reduce the size and height of the integrity tree within the main memory.

6.5 Metadata Management for Main Memory

To reduce metadata bandwidth overheads from compression, Attaché [24] and PTMC [73] enables data and metadata to be accessed together. Deb et. al. [14] describes the challenges in maintaining metadata in main memory and recommend using ECC to store Metadata. While this technique is useful for memory modules that have ECC in them, LLC uses tag entries and does not have to rely on ECC to store metadata [11, 32, 41–45, 54, 56, 72]. However, Touché can be expanded to include the ECC within LLC to store metadata.

6.6 Other Relevant Work

Sardashti and Wood [61] observed that cachelines in the same page may not have similar compressibility. Hallnor et. al. [21] proposed using compressed data throughout the memory hierarchy. This approach reduces the overheads of compression and decompression at every level of memory hierarchy. Sathish et al. [62] try to save

memory bandwidth by using both lossy and lossless compression for GPUs. Recent work from Han et. al. [22] and Kadetotad et. al. [31] used compression with deep neural networks to significantly improve performance and reduce energy. These prior work are orthogonal to Touché.

Cache compression has also been used to reduce cache power consumption. Residue cache architecture [35] reduces the last-level cache area by half, resulting in power saving. Other prior works have been proposed to lower the negative impacts of compression on the replacement. ECM [5] reduces the cache misses using Size-Aware Insertion and Size-Aware Replacement. CAMP [49] 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. The power-performance efficiency of Touché can be improved using these prior work.

7 SUMMARY

The Last-Level Cache (LLC) capacity per core has stagnated over the past decade. One way to increase the effective capacity of LLC is by employing data compression. Data compression enables the LLC controller to pack more memory blocks within the LLC. Unfortunately, the additional compressed memory blocks require additional tag entries. The LLC designer needs to provision additional tag area to store the tag entries of compressed blocks. We can also restrict data placement within each cacheline to neighboring addresses (superblocks) and reduce the tag area overheads. Ideally, we would like to get the benefits of LLC compression without incurring any tag area overheads.

To this end, this paper proposes Touché, a framework that enables LLC compression without any area overheads in the tag or data arrays. Touché uses shortened signatures to represent full tag address and appends the full tags to the compressed memory blocks in the data array. This enables Touché to store arbitrary memory blocks as neighbors. Furthermore, Touché can be enhanced further to include superblocks. Touché is completely hardware based and achieves a near-ideal speedup of 12% (ideal 13%) without any changes or area overheads to the tag and data array.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd for their feedback. We also thank Amin Azar his feedback on compression. This work was partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) [funding reference number RGPIN-2019-05059] and by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) [funding reference number NRF-2019R1G1A1011403].

REFERENCES

[1] B. Abali et al. 2001. Performance of hardware compressed main memory. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*. 73–81. https://doi.org/10.1109/HPCA.2001.903253

[2] A. R. Alameldeen and D. A. Wood. 2004. Adaptive cache compression for high-performance processors. In *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.* 212–223. https://doi.org/10.1109/ISCA.2004.1310776

[3] Alaa R Alameldeen and David A Wood. 2004. Frequent pattern compression: A significance-based compression scheme for L2 caches. *Dept. Comp. Scie., Univ. Wisconsin-Madison, Tech. Rep* 1500 (2004).

[4] Angelos Arelakis and Per Stenstrom. 2014. SC2: A Statistical Compression Cache Scheme. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 145–156. http://dl.acm.org/citation.cfm?id=2665671.2665696

[5] S. Baek, H. G. Lee, C. Nicopoulos, J. Lee, and J. Kim. 2013. ECM: Effective Capacity Maximizer for high-performance compressed caching. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. 131–142. https://doi.org/10.1109/HPCA.2013.6522313

[6] BBC Science. 2019. The End of Moore's Law: What Happens Next. https://www.sciencefocus.com/future-technology/when-the-chips-are-down/. (2019). Accessed: 2019-03-08.

[7] C. D. Benveniste, P. A. Franaszek, and J. T. Robinson. 2001. Cache-memory interfaces in compressed memory systems. *IEEE Trans. Comput.* 50, 11 (Nov 2001), 1106–1116. https://doi.org/10.1109/12.966489

[8] K. K. Chang, P. J. Nair, D. Lee, S. Ghose, M. K. Qureshi, and O. Mutlu. 2016. Low-Cost Inter-Linked Subarrays (LISA): Enabling fast inter-subarray data movement in DRAM. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 568–580. https://doi.org/10.1109/HPCA.2016.7446095

[9] Niladrish Chatterjee et al. 2012. USIMM: the Utah SImulated Memory Module A Simulation Infrastructure for the JWAC Memory Scheduling Championship.

[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 Very Large Scale Integration (VLSI) Systems* 18, 8 (Aug 2010), 1196–1208. https://doi.org/10.1109/TVLSI.2009.2020989

[11] C. Chou, P. Nair, and M. K. Qureshi. 2015. Reducing Refresh Power in Mobile Devices with Morphable ECC. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 355–366. https://doi.org/10.1109/DSN.2015.33

[12] E. Choukse, M. Erez, and A. Alameldeen. 2018. CompressPoints: An Evaluation Methodology for Compressed Memory Systems. *IEEE Computer Architecture Letters* 17, 2 (July 2018), 126–129. https://doi.org/10.1109/LCA.2018.2821163

[13] E. Choukse, M. Erez, and A. R. Alameldeen. 2018. Compresso: Pragmatic Main Memory Compression. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 546–558. https://doi.org/10.1109/MICRO.2018.00051

[14] A. Deb et al. 2016. Enabling technologies for memory compression: Metadata, mapping, and prediction. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*. 17–24. https://doi.org/10.1109/ICCD.2016.7753256

[15] Timothy E. Denehy, Windsor W. Hsu, Timothy E. Denehy, and Windsor W. Hsu. 2003. Duplicate management for reference data. In *Research Report RJ10305, IBM.*

[16] Julien Dusser, Thomas Piquet, and André Seznec. 2009. Zero-content Augmented Caches. In *Proceedings of the 23rd International Conference on Supercomputing (ICS '09)*. ACM, New York, NY, USA, 46–55. https://doi.org/10.1145/1542275.1542288

[17] Hadi Esmaeilzadeh et al. 2011. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA '11)*. ACM, New York, NY, USA, 365–376. https://doi.org/10.1145/2000064.2000108

[18] H. Esmaeilzadeh et al. 2012. Dark Silicon and the End of Multicore Scaling. *IEEE Micro* 32, 3 (May 2012), 122–134.

[19] P. Franaszek and J. Robinson. [n. d.]. Design and analysis of internal organizations for compressed random access memories. In *IBM Report, RC 21146, year=1998.*

[20] J. Gaur, A. R. Alameldeen, and S. Subramoney. 2016. Base-Victim Compression: An Opportunistic Cache Compression Architecture. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 317–328. https://doi.org/10.1109/ISCA.2016.36

[21] E. G. Hallnor and S. K. Reinhardt. 2005. A unified compressed memory hierarchy. In *11th International Symposium on High-Performance Computer Architecture*. 201–212. https://doi.org/10.1109/HPCA.2005.4

[22] Song Han, Huizi Mao, and William J. Dally. 2015. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. *CoRR* abs/1510.00149 (2015). arXiv:1510.00149 http://arxiv.org/abs/1510.00149

[23] Bo Hong, Demyn Plantenberg, Darrell D. E. Long, and Miriam Sivan-Zimet. 2004. Duplicate Data Elimination in a SAN File System. In *MSST*.

[24] S. Hong, P. J. Nair, B. Abali, A. Buyuktosunoglu, K. Kim, and M. Healy. 2018. Attaché: Towards Ideal Memory Compression by Mitigating Metadata Bandwidth Overheads. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 326–338. https://doi.org/10.1109/MICRO.2018.00034

[25] Intel Inc. 2016. Intel 64 and IA-32 Architectures Optimization Reference Manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf. (2016). Accessed: 2019-04-08.

[26] Intel Inc. 2019. Broadwell: 5th Generation Intel Core i5 Processors. https://ark.intel.com/content/www/us/en/ark/products/88095/intel-core-i5-5675c-processor-4m-cache-up-to-3-60-ghz.html. (2019). Accessed: 2019-03-07.

[27] Intel Inc. 2019. Intel Ultra Path Interconnect: Directory-Based Protocol. https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview. (2019). Accessed: 2019-05-08.

[28] Intel Inc. 2019. Ivy Bridge: Intel Core X-series Processors. https://ark.intel.com/content/www/us/en/ark/products/77780/intel-core-i7-4930k-processor-12m-cache-up-to-3-90-ghz.html. (2019). Accessed: 2019-03-07.

[29] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. 2010. High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP).

In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. ACM, New York, NY, USA, 60–71. https://doi.org/10.1145/1815961.1815971

[30] JEDEC Standard. 2015. DDR4 Standard. In *JESD79-4*.

[31] Deepak Kadetotad et al. 2016. Efficient Memory Compression in Deep Neural Networks Using Coarse-grain Sparsification for Speech Applications. In *Proceedings of the 35th International Conference on Computer-Aided Design (ICCAD '16)*. ACM, New York, NY, USA, Article 78, 8 pages. https://doi.org/10.1145/2966986.2967028

[32] Samira Khan et al. 2014. The Efficacy of Error Mitigation Techniques for DRAM Retention Failures: A Comparative Experimental Study. In *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '14)*. ACM, New York, NY, USA, 519–532.

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

[34] Nam Sung Kim, Todd Austin, and Trevor Mudge. [n. d.]. Low-energy data cache using sign compression and cache line bisection. Citeseer.

[35] S. Kim, J. Kim, J. Lee, and S. Hong. 2011. Residue cache: A low-energy low-area L2 cache architecture via compression and partial hits. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 420–429.

[36] S. Kim, S. Lee, T. Kim, and J. Huh. 2017. Transparent Dual Memory Compression Architecture. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 206–218. https://doi.org/10.1109/PACT.2017.12

[37] S. Kumar, H. Zhao, A. Shriraman, E. Matthews, S. Dwarkadas, and L. Shannon. 2012. Amoeba-Cache: Adaptive Blocks for Eliminating Waste in the Memory Hierarchy. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. 376–388. https://doi.org/10.1109/MICRO.2012.42

[38] Gabriel H. Loh and Mark D. Hill. 2011. Efficiently Enabling Conventional Block Sizes for Very Large Die-stacked DRAM Caches. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. ACM, New York, NY, USA, 454–464. https://doi.org/10.1145/2155620.2155673

[39] P. Nair, Chia-Chen Chou, and M.K. Qureshi. 2013. A case for Refresh Pausing in DRAM memory systems. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*. 627–638. https://doi.org/10.1109/HPCA.2013.6522355

[40] P. J. Nair, B. Asgari, and M. K. Qureshi. 2019. SuDoku: Tolerating High-Rate of Transient Failures for Enabling Scalable STTRAM. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 388–400. https://doi.org/10.1109/DSN.2019.00048

[41] Prashant J. Nair, Dae-Hyun Kim, and Moinuddin K. Qureshi. 2013. ArchShield: Architectural Framework for Assisting DRAM Scaling by Tolerating High Error Rates. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 72–83. https://doi.org/10.1145/2485922.2485929

[42] P. J. Nair, D. A. Roberts, and M. K. Qureshi. 2014. Citadel: Efficiently Protecting Stacked Memory from Large Granularity Failures. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 51–62. https://doi.org/10.1109/MICRO.2014.57

[43] Prashant J. Nair, David A. Roberts, and Moinuddin K. Qureshi. 2015. FaultSim: A Fast, Configurable Memory-Reliability Simulator for Conventional and 3D-Stacked Systems. *ACM Trans. Archit. Code Optim.* 12, 4, Article 44 (Dec. 2015), 24 pages. https://doi.org/10.1145/2831234

[44] Prashant J. Nair, David A. Roberts, and Moinuddin K. Qureshi. 2016. Citadel: Efficiently Protecting Stacked Memory from TSV and Large Granularity Failures. *ACM Trans. Archit. Code Optim.* 12, 4, Article 49 (Jan. 2016), 24 pages. https://doi.org/10.1145/2840807

[45] P. J. Nair, V. Sridharan, and M. K. Qureshi. 2016. XED: Exposing On-Die Error Detection Information for Strong Memory Reliability. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 341–353. https://doi.org/10.1109/ISCA.2016.38

[46] P. M. Palangappa and K. Mohanram. 2016. CompEx: Compression-expansion coding for energy, latency, and lifetime improvements in MLC/TLC NVM. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 90–101. https://doi.org/10.1109/HPCA.2016.7446056

[47] G. Pekhimenko et al. 2012. Base-delta-immediate compression: Practical data compression for on-chip caches. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 377–388.

[48] Gennady Pekhimenko et al. 2013. Linearly Compressed Pages: A Low-complexity, Low-latency Main Memory Compression Framework. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 172–184. https://doi.org/10.1145/2540708.2540724

[49] G. Pekhimenko et al. 2015. Exploiting compressed block size as an indicator of future reuse. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 51–63. https://doi.org/10.1109/HPCA.2015.7056021

[50] G. Pekhimenko et al. 2016. A case for toggle-aware compression for GPU systems. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 188–200. https://doi.org/10.1109/HPCA.2016.7446064

[51] Erez Perelman et al. 2003. Using SimPoint for accurate and efficient simulation. *ACM SIGMETRICS Performance Evaluation Review* (2003).

[52] Moinuddin K. Qureshi et al. 2007. Adaptive Insertion Policies for High Performance Caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA '07)*. ACM, New York, NY, USA, 381–391. https://doi.org/10.1145/1250662.1250709

[53] M. K. Qureshi, D. Kim, S. Khan, P. J. Nair, and O. Mutlu. 2015. AVATAR: A Variable-Retention-Time (VRT) Aware Refresh for DRAM Systems. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 427–437. https://doi.org/10.1109/DSN.2015.58

[54] D Roberts and P Nair. 2014. FAULTSIM: A fast, configurable memory-resilience simulator. In *The Memory Forum: In conjunction with ISCA*, Vol. 41.

[55] G. Saileshwar et al. 2018. Morphable Counters: Enabling Compact Integrity Trees For Low-Overhead Secure Memories. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 416–427.

[56] G. Saileshwar et al. 2018. SYNERGY: Rethinking Secure-Memory Design for Error-Correcting Memories. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 454–465. https://doi.org/10.1109/HPCA.2018.00046

[57] S. Sardashti, A. Seznec, and D. A. Wood. 2014. Skewed Compressed Caches. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 331–342. https://doi.org/10.1109/MICRO.2014.41

[58] Somayeh Sardashti, Andre Seznec, and David A. Wood. 2016. Yet Another Compressed Cache: A Low-Cost Yet Effective Compressed Cache. *ACM Trans. Archit. Code Optim.* 13, 3, Article 27 (Sept. 2016), 25 pages. https://doi.org/10.1145/2976740

[59] S. Sardashti and D. A. Wood. 2013. Decoupled compressed cache: Exploiting spatial locality for energy-optimized compressed caching. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 62–73.

[60] S. Sardashti and D. A. Wood. 2014. Decoupled Compressed Cache: Exploiting Spatial Locality for Energy Optimization. *IEEE Micro* 34, 3 (May 2014), 91–99. https://doi.org/10.1109/MM.2014.42

[61] Somayeh Sardashti and David A. Wood. 2017. Could Compression Be of General Use? Evaluating Memory Compression Across Domains. *ACM Trans. Archit. Code Optim.* 14, 4, Article 44 (Dec. 2017), 24 pages. https://doi.org/10.1145/3138805

[62] Vijay Sathish, Michael J. Schulte, and Nam Sung Kim. 2012. Lossless and Lossy Memory I/O Link Compression for Improving Performance of GPGPU Workloads. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT '12)*. ACM, New York, NY, USA, 325–334. https://doi.org/10.1145/2370816.2370864

[63] A. Shafiee et al. 2014. MemZip: Exploring unconventional benefits from memory compression. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 638–649.

[64] T. B. Smith, B. Abali, D. E. Poff, and R. B. Tremaine. 2001. Memory Expansion Technology (MXT): Competitive impact. *IBM Journal of Research and Development* 45, 2 (March 2001), 303–309. https://doi.org/10.1147/rd.452.0303

[65] The Economist. 2019. Technology Quarterly: After Moore's Law. https://www.economist.com/technology-quarterly/2016-03-12/after-moores-law. (2019). Accessed: 2019-03-07.

[66] The Verge. 2019. Intel is forced to do less with Moore. https://www.theverge.com/2018/7/19/17590242/intel-50th-anniversary-moores-law-history-chips-processors-future. (2019). Accessed: 2019-03-08.

[67] Yingying Tian, Samira M. Khan, Daniel A. Jiménez, and Gabriel H. Loh. 2014. Last-level Cache Deduplication. In *Proceedings of the 28th ACM International Conference on Supercomputing (ICS '14)*. ACM, New York, NY, USA, 53–62. https://doi.org/10.1145/2597652.2597655

[68] R. B. Tremaine, T. B. Smith, M. Wazlowski, D. Har, Kwok-Ken Mak, and S. Arramreddy. 2001. Pinnacle: IBM MXT in a memory controller chip. *IEEE Micro* 21, 2 (Mar 2001), 56–68. https://doi.org/10.1109/40.918003

[69] ESX VMware. 2010. Understanding Memory Resource Management in VMware ESX 4.1. (2010).

[70] D. Weiss, M. Dreesen, M. Ciraula, C. Henrion, C. Helt, R. Freese, T. Miles, A. Karegar, R. Schreiber, B. Schneller, and J. Wuu. 2011. An 8MB level-3 cache in 32nm SOI with column-select aliasing. In *2011 IEEE International Solid-State Circuits Conference*. 258–260. https://doi.org/10.1109/ISSCC.2011.5746309

[71] www.spec.org. 2006. The SPEC2006 Benchmark Suite. (2006).

[72] Doe Hyun Yoon, Min Kyu Jeong, and Mattan Erez. 2011. Adaptive Granularity Memory Systems: A Tradeoff Between Storage Efficiency and Throughput. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA '11)*. ACM, New York, NY, USA, 295–306.

[73] V. Young, S. Kariyappa, and M. Qureshi. 2019. Enabling Transparent Memory-Compression for Commodity Memory Systems. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 570–581. https://doi.org/10.1109/HPCA.2019.00010

[74] Vinson Young, Prashant J. Nair, and Moinuddin K. Qureshi. 2017. DICE: Compressing DRAM Caches for Bandwidth and Capacity. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 627–638. https://doi.org/10.1145/3079856.3080243