Cache Compression Using Base-Delta-Immediate

Shabbir. M. Aglodiya
Department of Electronics and Communications Engineering, Institute of Technology, Nirma University. Ahmedabad, Gujarat, India – 382481

21BEC006@nirmauni.ac.in

Sujal. R. Bhojani
Department of Electronics and Communications Engineering, Institute of Technology, Nirma University. Ahmedabad, Gujarat, India – 382481

21BEC016@nirmauni.ac.in

***Abstract* - This research aims to make computer memory work better by using a technique called cache compression. Cache is like a small, fast memory that stores frequently used data to speed up tasks. But storing all this data takes up space and can slow things down. So, we developed a new way to compress this data called BΔI compression. It's like squeezing a bunch of clothes into a smaller suitcase. Our method makes sure the data still fits in the cache but takes up less space and doesn't slow things down. This helps make computers run faster and more efficiently.**

Keywords – Cache compression, Caching, Memory, Base Delta Immediate.

I. INTRODUCTION

Multi-level on-chip cache systems are a feature of contemporary microprocessors that address the access constraints of main memory. Performance depends heavily on these caches, but choosing the right size necessitates making trade-offs. Larger caches increase power consumption, chip area, and access delay, but they can also minimise cache misses. Increasing the number of on-chip cores makes it harder to provide shared caches with adequate capacity. Merely increasing the size of each core's cache might squander chip resources, whilst decreasing it could result in more expensive off-chip cache misses.

One potential solution is to use data compression to increase effective cache capacity without suffering the drawbacks. Data compression has been successful in various computer systems to save storage capacity and data bandwidth. However, it hasn't been widely adopted in commodity microprocessors for cache capacity enhancement.

The ideal cache compression method should be quick, straightforward, and efficient for saving storage space. It's crucial that the compression ratio be substantial enough to offer significant benefits while keeping hardware complexity low to avoid offsetting these advantages. However, decompression time is the fundamental obstacle to the implementation of cache compression in commercial microprocessors. Unlike compression, which can happen in the background, decompression occurs during cache hits, where minimizing latency is vital for performance. Due to this, our focus is on compressing L2 caches and beyond.

Balancing the need for quick, effective and simple cache compression is challenging. Existing techniques often sacrifice one aspect for another, such as achieving a small compression ratio or having decompression latency or as well as high hardware complexity. To address these challenges, we offer Base-Delta-Immediate (BΔI) compression, a novel cache compression method.

II. APPROACH USING BΔI COMPRESSION

Basic Principle of Base-Delta-Immediate (BΔI) A feature of cache compression is that a large number of cache lines contain data with a narrow range of values—that is, relatively little variation between these values. In these situations, we may more effectively represent the cache lines by combining an array of very minor variations (called "deltas") with a common base value. This array ultimately occupies a lot less space than the original huge cache line. This is the origin of the terms "base" and "delta" in the name of our system.

First, we investigate the case of Base+Delta (B+Δ) compression, where there is just one arbitrary base. This case serves as the basis for all of our solutions. We investigate the potential of having numerous bases to further increase the compression efficiency. We find that the greatest outcomes for the workloads we analysed come from using two bases, one of which is always set to zero. With these two basis values, our method may effectively compress cache lines with two distinct dynamic ranges combined: one centred around zero (e.g., tiny integer values) and the other around an arbitrary value from the cache line's contents (e.g., pointer values). Mixed dynamic ranges are widely used in various applications, one of which being pointer-linked data structures.

As demonstrated later, BΔI compression offers several benefits: (i) it achieves a high cache compression ratio by leveraging common patterns in cache data, as evidenced by

real application examples and experimental validation; (ii) it incurs low decompression latency, as decompressing a cache line involves straightforward masked vector addition; and (iii) because the compression and decompression algorithms rely on straightforward comparison operations, subtraction, and vector addition, it has very little hardware overhead and implementation complexity.

This method makes the following contributions:

1. Base-Delta Immediate Compression (BΔI), a novel cache compression method, uses the limited dynamic range of cache lines to minimize their size. The low-latency compression and decompression algorithms of BΔI need just vector addition, subtraction, and comparison operations.

2. The proposed BΔI compression is used to get a compressed cache design. Compared to two contemporary cache compression strategies frequent pattern compression (FPC) and frequent value compression (FVC) this method offers a higher degree of compression at a lower decompression time. However, these techniques need complex and time-consuming decompression pipelines.

3. We compare the performance improvements of BΔI with three modern cache compression methods and an uncompressed minimum benchmark system. Our findings indicate that BΔI offers equivalent or superior compression for the majority of the applications we examined. Applications needing one core (8.1%) and several cores (9.5% / 11.2% for two or four cores) benefit from the improved performance. Performance can be improved by compression utilising BΔI by increasing the uncompressed cache space of the baseline system.

III. BACKGROUND

One useful technique for fitting big data sets into smaller storage places is data compression. Applying this to on-chip caches might enable the cache to store more compressed cache lines, so offering the advantages of a bigger cache with the reduced size and power consumption of a smaller one.
Numerous similar patterns have been identified as a result of prior research identifying a substantial level of repetition in the data accessible by practical applications:

1. Zeros: Zero is a commonly encountered value in application data, often used for initialization, representing NULL pointers or false Boolean values, and in sparse matrices. Many compression schemes focus on handling zero values efficiently.

2. Repeated Values: In large, interconnected memory regions, the same value may be repeated several times. This pattern is typical in applications that initialize big arrays with a single value or in multimedia data when consecutive pixels share the same color. The compression of repeated data can significantly reduce storage needs.

3. Narrow Values: Narrow values are small values stored using larger data types. They occur due to over-provisioning or data alignment in data structures. While most values could fit in smaller data types, programmers typically use larger types to accommodate worst-case scenarios, leading to inefficiencies. Compression techniques target these narrow values to optimize storage.

Other common data patterns include tables of pointers pointing to various locations in the same region of memory and images with small color gradients. These patterns can also be compressed using simple techniques, as demonstrated in prior proposals for main memory and image compression.

Table 1 Qualitative comparison of BΔI with prior work

	Characteristics			**Compressible data patterns**			
	Decomp. Lat.	Complex.	C. Ratio	Zeros	Rep. Val.	Narrow	LDR
ZCA [8]	**Low**	**Low**	Low	✔	×	×	×
FVC [33]	High	High	Modest	✔	Partly	×	×
FPC [2]	High	High	**High**	✔	✔	✔	×
BΔI	**Low**	Modest	**High**	✔	✔	✔	✔

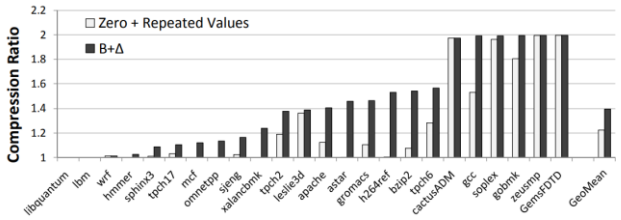
IV. BASE + DELTA ENCODING

Researchers have proposed a novel cache compression technique called Base+Delta (B+Δ) compression, which differs from previous approaches by focusing on compression opportunities at the cache line level. In this method, each cache line is either completely compressed or left uncompressed. This approach makes use of the fact that many cache lines include data with a constrained dynamic range, which makes it possible to express fluctuations between words within the line more effectively and with fewer bytes than the original values. By utilizing common base value and delta arrays (differences from the base), B+Δ compression achieves significant compression compared to storing the uncompressed cache line.

Although base+delta representation of values has been used in other fields, such texture compression on GPUs and bandwidth optimisation on CPU buses, this study is the first to investigate its usage for enhancing the use of on-chip cache in general-purpose CPUs.

To evaluate effectiveness of B+Δ compression across various applications, the researchers conducted a study comparing its compression ratio with a simple technique targeting two common data patterns: zeros and repeated values. Their experiments, performed on a 2MB L2 cache with various workloads, demonstrated that B+Δ compression achieves a substantially greater compression ratio (average of 1.4X) in comparison to the basic compression method. Nonetheless, a few assessments indicated that B+Δ reduction offered little to no improvement, leading to the development of a new technique called BΔI, which will be discussed in the subsequent sections.

Figure 1 Effective compression ratio with different value patterns.



V. COMPRESSION ALGORITHM

The B+ compression algorithm views a 64-byte cache line as a collection of fixed-size values, 16 4-byte values, or 32 2-byte values, for example. The next stage is to determine whether these values may be reduced in size by employing a list of deltas, or differences, produced from the base value and a base value itself. S = (v1, v2,..., vn) will represent the set of values to be compressed. The number of values on the cache line, n, is found by dividing the cache line size C (in bytes) by the value size k (in bytes). The algorithm seeks to find the ideal value size (k) and base value (B∗) in order to achieve maximum compression. After B* and k are established, the compression procedure yields {k, B∗, Δ = (Δ1, Δ2, ..., Δn)}, where Δi = vi - B∗ for each value vi.[1]

Observation 1: Cache is compressible only if, for every value of i, bytes needed to be stored Δi is less than k, where the size(Δi) signifies the smallest number of bytes for Δi. In other words, the differences between values must require few bytes than the values themselves for compression to be feasible.

Observation 2: To find the optimal value of B∗, the algorithm considers either the minimum or maximum value in S. Since the values in the cache are bounded by minimum and maximum values of S, the optimal base value should lie between these extremes.

To determine the size of each value (k), the algorithm considers multiple potential values simultaneously (e.g., 2, 4, and 8 bytes) to accommodate different data patterns. The cache is then compressed using the value that gives the highest compression rate.

Instead of computing the maximum or minimum of the set of values to determine B∗, which increases complexity and latency, the algorithm approximates B∗ by using the first value from the set. This simplification reduces compression latency and hardware complexity with only a minor impact on the compression ratio.

VI. DECOMPRESSION ALGORITHM

In order to apply the B+Δ decompression technique to decompress a compressed cache line, we require the base value B along with an array of differences Δ = (1Δ, 2 Δ,..., Δn) that indicate the differences between each value and the base. Then, vi = B∗ + Δi, or each difference, may be added to the base value to recover the original values. This method uses a SIMD kind of vector adder to compute the values in parallel. As a result, using simple adders, the full cache line can be decompressed in the same amount of time as performing integer vector addition. This effective method allows for speedy decompression of compressed cache lines with minimal overhead.

VII. BDI DESIGN AND OPERATION

Design:

Eight compressor units make up the compression logic: two for compression of zero and repeated values, and six for various base (8, 4, and 2 bytes) and delta (4, 2, and 1 bytes) sizes. A cache line is sent into each compressor unit, which outputs whether or not it can compress the cache. The unit creates the compressed cache if compression is possible.

The list of compressor units that can compress a specific cache line is determined by the compressor selection logic. The selection algorithm selects the compression technique with the shortest compressed size when there are several options (for example, 8-byte base with 1-byte delta and zero compression). A table lists the possible compressed sizes for each compression technique that is known statically. It is possible for all compressor units to run at once.

The decompression logic is simpler as it only needs to identify the compression method used and apply the corresponding decompression algorithm. The compressed cache line format includes information about the compression method, base value (if applicable), and the compressed data.

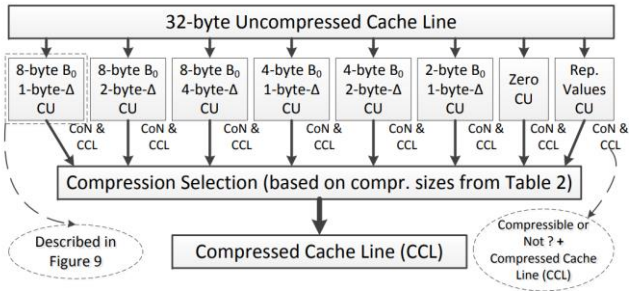


Figure 2 Compressor design. CU: Compressor unit.

Figure 2 shows how an 8-byte-base 1-byte-delta compressor unit is organised for a 32-byte cache line. The compressor first determines how much each element differs from the base element. This cache line is handled as though it were a group of four 8-byte values (V0, V1, V2, V3). The base (B0) is set to the initial value (V0), as was previously described. A 1-byte sign extension check is done to see if the differences (V1-B0, V2-B0, and V3-B0) can all be expressed

with a single byte. The cache line can be compressed if all the differences can be represented in a single byte, which means that all seven bits are either ones or zeros. This reduces the initial 32-byte cache line size to 12 bytes by storing the basis value B0 and the set of 1-byte differences. The compressor unit, however, is unable to compress the cache line if the 1-byte sign extension check is unsuccessful, meaning that at least one difference cannot be represented with a single byte. Other compressor units are arranged in a like manner.

This compression design might be improved, especially if latency is not as important as hardware complexity. For example, all compression units of 8-byte-base value can be combined to single unit to eliminate logic duplication.

Name	Base	Δ	Size	Enc.	Name	Base	Δ	Size	Enc.
Zeros	1	0	1/1	0000	Rep.Values	8	0	8/8	0001
Base8-Δ1	8	1	12/16	0010	Base8-Δ2	8	2	16/24	0011
Base8-Δ4	8	4	24/40	0100	Base4-Δ1	4	1	12/20	0101
Base4-Δ2	4	2	20/36	0110	Base2-Δ1	2	1	18/34	0111
NoCompr.	N/A	N/A	32/64	1111					

Table 2 BΔI encoding. All sizes are in bytes. Compressed sizes (in bytes) are provided for 32- and 64-byte cache lines.

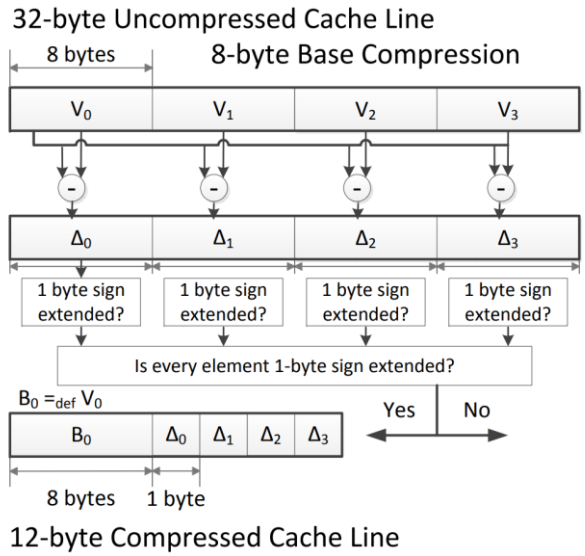
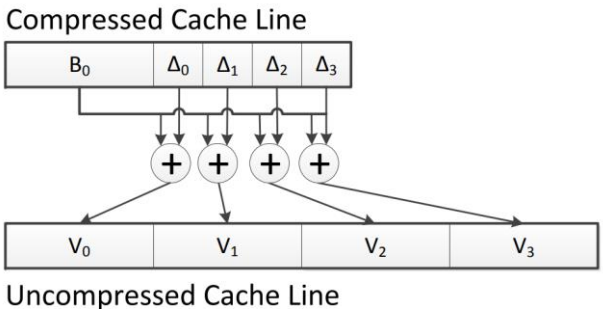


Figure 3 Compressor unit for 8-byte base, 1-byte Δ

In Figure 4, we see the layout of the decompression logic critical for minimizing latency. Its design is straightforward: when handling compressed line comprising a base B0 and differences Δ0, Δ1, Δ2, Δ3, the logic performs only the additions of the base to these to reconstruct the uncompressed cache. This approach ensures swift decompression, taking no longer than the latency of an adder, thereby enabling efficient decompression within the BΔI cache system.

Figure 4 Decompressor Design.



BΔI cache organization: introduces changes to the traditional cache architecture in order to take advantage of compression. In comparison to an uncompressed cache, more lines can store in the same amount of storage by compressing cache lines. A higher number of tags is needed to access these extra compressed lines. One strategy is to double the number of tags so that more data pieces in the storage may be accessed by them as pointers.

The required changes to the cache architecture are shown in Figure 3. Two tags per set in the tag store of a traditional 32-byte cache lines with 2-way cache correspond to individual data storage units. There are twice as many tags in the B-I design (bottom), and each tag additionally includes extra bits to indicate the kind and state of compression. The amount of data stored is divided into small fix segments (e.g., 8 bytes) yet stays the same overall. To make it easier to determine how many segments were utilized, each tag holds the beginning segment and the cache block's encoding.

Storage Cost Analysis: Because there are twice as many tags per set, this organisation allows for the storing almost double cache in the same storage. The increase in tag store size incurs a modest overhead, comparable to other designs. CACTI 5.3 is utilized to estimate additional latency and area costs, indicating a minor increase in cache access latency and a 2.3% increase in on-chip cache area for a 32nm technology node. These changes are relatively small compared to the significant area increase if both tag and data store sizes were doubled.

	Baseline	BΔI
Size of tag-store entry	21 bits	32 bits (+4–encoding, +7–segment pointer)
Size of data-store entry	512 bits	512 bits
Number of tag-store entries	32768	65536
Number of data-store entries	32768	32768
Tag-store size	84kB	256kB
Total (data-store+tag-store) size	2132kB	2294kB

Table 3 Storage cost analysis for 2MB 16-way L2 cache

Cache Eviction Policy: There are two situations in which deleting a single cache line (that is, the Least Recently Used (LRU) line in a cache with LRU) could not free up enough space for the incoming or updated cache line, necessitating the eviction of several cache lines from a compressed cache.

Initially, when adding a new line of cache, whether compressed or uncompressed, to the cache. Second, when a cache line that already exists gets longer than it used to be. The suggested method uses a slightly altered version of the LRU replacement policy in both scenarios, where the cache evicts several LRU cache lines to create space for the new or altered cache line. Since evictions are not on the crucial route of execution, this strategy has no effect on performance even if it could take longer. It is noteworthy that more efficient replacement procedures, such as one that holds off on evicting a zero-compressed cache line until the tag store needs more space, are conceivable and take the compressed cache line length into account. Such policies are not examined here; that is for future study.

The design of the BΔI compression technique is then discussed. While both B+Δ and BΔI share a common portion in their designs, there are a few notable differences. BΔI compression requires two steps (off the critical path), whereas B+Δ compression requires only one step. All items are compressed with a fixed delta size using an implicit base of zero in the first phase of BΔI compression. Any components that were not compressed in the first phase are tried to be compressed in the second step. The foundation for the second step is the first uncompressible element from the first stage. A bit mask, one bit for each element, indicating whether or not the associated base is zero, is stored throughout the compression step. For both bases, the delta's size—1, 2, or 4 bytes remains the same. The base (selected in the second phase) is masked and added to the array of deltas in B-I decompression. Which elements receive the base value is determined by the bit mask that was captured during the compression step.

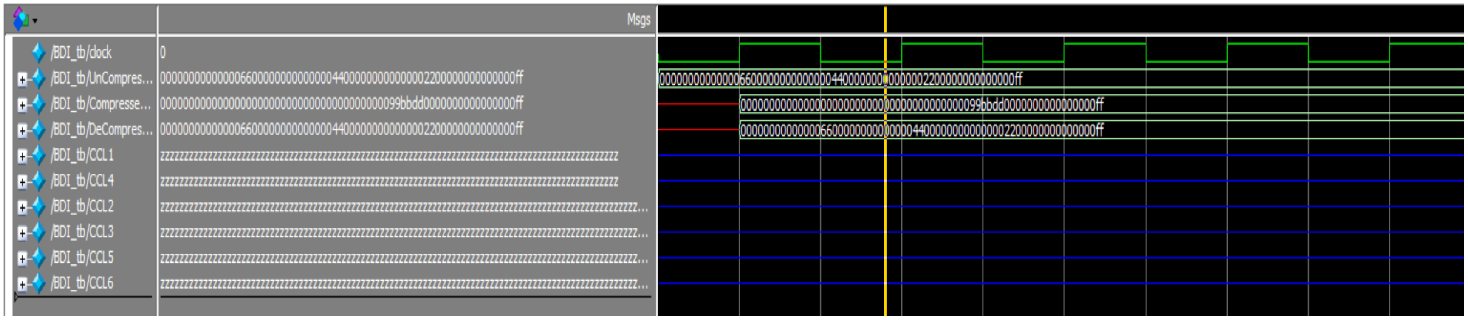
Operation:

We recommend implementing the BΔI design in higher-level caches such as L2 and L3, rather than the latency-sensitive L1 cache, where compression may negatively effect performance for applications that do not benefit from it. The integration of a B-I cache into a system consisting of main memory and a two-level cache hierarchy (L1, L2) is explained below. Compressed is only the L2 cache.

In this setup, four scenarios pertain to the operation of the compressed L2 cache:

1. On a hit on L2, the matching cache line is moved to the L1 cache. If compressed, data is decompressed before being transferred to L1.
2. When a miss occurs on L2, the corresponding cache line is fetched from memory and transferred to the L1 cache. The line is compressed and stored in the L2 cache.
3. A line is compressed when it is written back from L1 to L2. The new compressed line is appended and the previous copy of the line, if it exists in L2, is invalidated.
4. Prior to being delivered to the memory controller, the line is decompressed during a writeback from L2 cache to memory. Based on L2's eviction method, numerous cache lines may be eliminated in the second and third cases.

VIII. RESULTS




```
# input = 00000000000000660000000000000044000000000000002200000000000000ff
#
#  8_1: del1= 00000000000000dd, del2 =0000000000000bb, del3 = 0000000000000099
#
# CoN1 = 1, CCL1 = 99bbdd00000000000000000ff
# CoN2 = 1, CCL2 = 009900bb00dd00000000000000000ff
# CoN3 = 1, CCL3 = 00000099000000bb000000dd00000000000000000000000ff
#  BASE 4: del1= 000000ff, del2 =000000dd, del3 = 000000ff del4 = 000000bb del5 = 000000ff del6 = 00000099 del7 = 000000ff
#
# CoN4 = 1, CCL4 = ff99ffbbffddff000000000ff
# CoN5 = 1, CCL5 = 00ff009900ff00bb00ff00dd00ff0000000000ff
#  BASE 2: del1= 00ff, del2 =00ff, del3 = 00ff del4 = 00dd del5 = 00ff del6 = 00ff del7 = 00bb  del8= 00ff, del9 =00ff, del10 = 00ff del11 = 00ff del12 = 0099 del13 = 00ff del14 = 00ff del7 = 00ff
#
# CoN6 = 1, CCL6 = ffffff99ffffffffbbffffddffffff0000ff
# input = 00000000000000660000000000000044000000000000002200000000000000ff
#
#  8_1: del1= 00000000000000dd, del2 =0000000000000bb, del3 = 0000000000000099
#
# CoN1 = 1, CCL1 = 99bbdd00000000000000000ff
# CoN2 = 1, CCL2 = 009900bb00dd00000000000000000ff
# CoN3 = 1, CCL3 = 00000099000000bb000000dd00000000000000000000000ff
#  BASE 4: del1= 000000ff, del2 =000000dd, del3 = 000000ff del4 = 000000bb del5 = 000000ff del6 = 00000099 del7 = 000000ff
#
# CoN4 = 1, CCL4 = ff99ffbbffddff000000000ff
# CoN5 = 1, CCL5 = 00ff009900ff00bb00ff00dd00ff0000000000ff
#  BASE 2: del1= 00ff, del2 =00ff, del3 = 00ff del4 = 00dd del5 = 00ff del6 = 00ff del7 = 00bb  del8= 00ff, del9 =00ff, del10 = 00ff del11 = 00ff del12 = 0099 del13 = 00ff del14 = 00ff del7 = 00ff
#
# CoN6 = 1, CCL6 = ffffff99ffffffffbbffffddffffff0000ff
```

IX. CONCLUSION

In contrast to previous approaches, this work presents BΔI, an innovative hardware cache compression methodology that provides a simple yet effective way to boost effective cache capacity and enhance system performance. By expressing cache lines with two basis values (including an implicit base of zero) and an array of differences from these bases, BΔI takes use of the limited dynamic range of in-cache data. We illustrate the effectiveness of BΔI compression with real program data pattern samples.

By using parallel decompression to achieve a shorter decompression time while keeping a high average compression ratio, B-I performs better than earlier methods. We present the architecture and functionality of a cache that allows for low hardware overhead B-I compression.

Extensive experiments conducted on various workloads and system configurations suggest that BΔI compression within an L2 cache might significantly improve system performance for single-core and multi-core tasks, outperforming three cutting-edge cache compression algorithms. When B-I compression is used, performance advantages are sometimes doubled compared to the capacity of the L2/L3 cache. In summary, systems having one or more cores can address on-chip caches in an efficient and low-latency manner by utilising B-implemented data compression.

References

[1] G. V. S. O. M. P. B. G. M. A. K. T. C. M. Pekhimenko, "Base-delta-immediate compression: Practical data compression for on-chip caches.," *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pp. 377-388, 2012.

[2] X. L. Y. R. P. D. L. S. H. L. Chen, "C-pack: A high-performance microprocessor cache compression algorithm," *IEEE transactions on very large scale integration (VLSI) systems*, vol. 18, no. 8, pp. 1196-1208, 2009.

[3] E. G. S. K. R. Hallnor, "A unified compressed memory hierarchy.," *11th International Symposium on High-Performance Computer Architecture*, pp. 201-212, 2005.

[4] M. M. P. S. Islam, "Zero-value caches: Cancelling loads that return zero.," *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pp. 237-245, 2009.

[5] A. D. W. Alameldeen, "Adaptive cache compression for high-performance processors.," *ACM SIGARCH Computer Architecture News*, vol. 32, no. 2, p. 212, 2004.