Combining Inter and Intra-Line Cache Compression

by

Mohammad Ewais

BSc, Alexandria University, 2014

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Applied Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL
STUDIES
(Electrical and Computer Engineering)

The University of British Columbia
(Vancouver)

June 2018

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

Combining Inter and Intra-Line Cache Compression

submitted by **Mohammad Ewais** in partial fulfillment of the requirements for the degree of **Master of Applied Science** in **Electrical and Computer Engineering**.

Examining Committee:

Mieszko Lis, Electrical and Computer Engineering
Supervisor

Tor Aamodt, Electrical and Computer Engineering
Supervisory Committee Member

Steven Wilton, Electrical and Computer Engineering
Supervisory Committee Member

Abstract

Caches are essential to today's microprocessors. They close the huge speed gap between processors and memories. However, cache design presents an important tradeoff. A bigger cache size should increase performance and allow processors to perform faster, but it is also limited by its silicon, area, and power consumption costs. Today's caches often use half of the silicon area in processor chips and consume a lot of power. Instead of physically increasing the cache size, effective cache capacity can be substantially increased if the data inside the cache is compressed.

Current cache compression techniques focus only on one granularity, either compressing inside one cache line, or compressing similar cache lines together. In this work, we combine both compression techniques to leverage both inter-line and intra-line compression. We find that combining both techniques results in better compression than previously described methods, and also maintains the same performance as a normal uncompressed cache when running incompressible applications. We study and address the design considerations and tradeoffs that arise from such design. We address issues related to the design like cache structure and replacement policies. Then we present an implementation that achieves the best possible compression and performance while maintaining overheads as low as possible.

Lay Summary

Today's computer systems have two main elements: processors to do the computations, and memories to hold the data. However, memories that are large enough for today's applications can be a lot slower than processors, limiting their speed. A work around is to use smaller and faster memories placed in between the processor and the memory, called caches, that can only hold the critical data needed by the processor. A bigger cache means more critical data can be accessed at high speed, but it also makes the cache slower. This work combines and builds upon previous data compression techniques to allow the caches to store more data without increasing the physical cache size, giving the benefits of a bigger cache while maintaining the same speed.

Preface

This dissertation is based on joint work done by PhD student Amin Ghasemazar and myself in the same research group. None of the text of the dissertation is taken directly from previously published or collaborative articles.

Amin and myself worked on the idea and design together. We both independently implemented the BDI, Dedup, and straightforward DedupBDI caches in Chapters 2 and 3. We individually worked on verifying our own implementations then together worked on cross verification of both implementations. The offline analysis done in Chapter 2, the experiments done in Sections 3.2 and 3.3, the ideal and final cache implementations in 3.2 and 3.3, the area and power estimations done in Section 4.7, and gathering the final results reported in this work were done by me.

Table of Contents

List of Tables

List of Figures

Acknowledgments

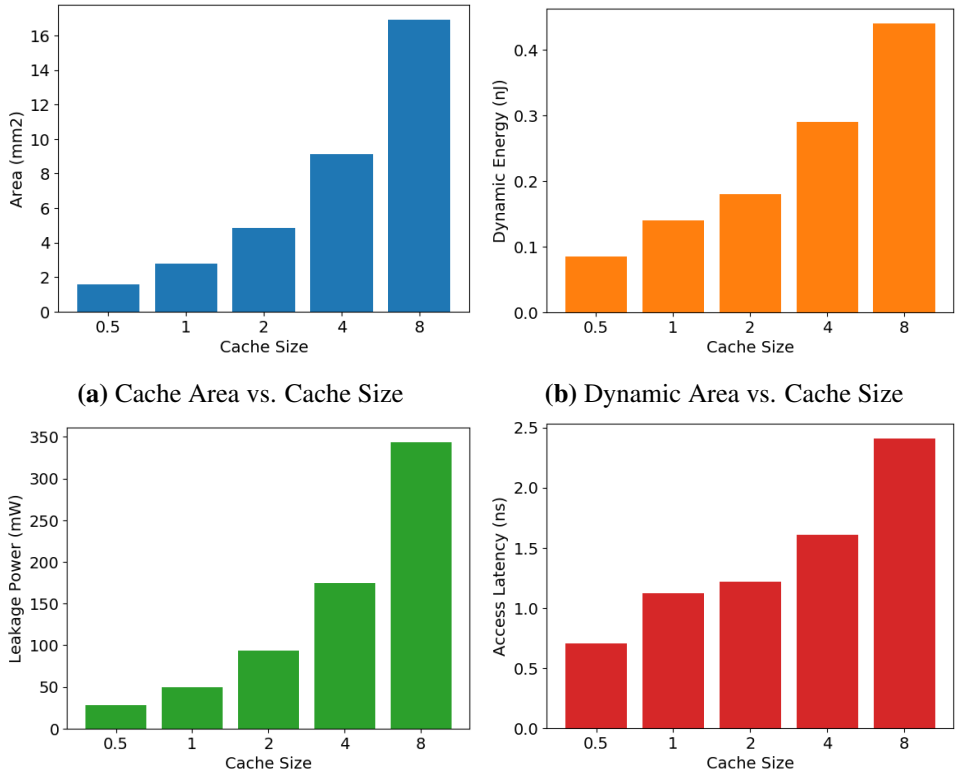
Many thanks to my mother and brother Hanaa and Ahmed for their continuous support, I owe you for life. I'm eternally grateful to my wife Toqa who has provided me with moral and emotional support. Special thanks to my friend and coworker Amin Ghasemazar. It was a fantastic opportunity to work with you. I'm very grateful to my advisor and mentor Mieszko Lis. Thank you for all your help.

Chapter 1

Introduction

Modern computer systems suffer from a huge speed gap between microprocessors and DRAM main memories. The traditional approach to tolerate the huge DRAM latency is to use large multi-megabyte caches. However, caches always present a design tradeoff. While increasing cache size can lead to lower miss rates and thus better performance, it also requires higher silicon area, increases higher access latencies and consumes more power [21]. In modern microprocessors caches often occupy a significant portion of the die area, and contribute significantly to power consumption [8, 12]. Figure 1.1 shows how area, dynamic energy, leakage power, and access latency scales with cache size. The results in Figure 1.1 were generated using CACTI [16].

A different approach that has gained more popularity recently is to use compression to better utilize the existing cache space instead of increasing cache sizes. Cache compression allows caches to store more than their physical sizes, essentially acting like bigger size caches. However, it also imposes more constraints on the compression and decompression design, requiring compression to be very fast to not affect the cache performance. Previous proposals discussed and exploited data similarities and redundancies on different granularities. Some studies have found that applications can exhibit a lot of zero cache lines (i.e. cache lines with only the value zeros in them). some proposals were completely focused on zero compression [10] while others treated zero lines as a special case in their compression schemes [2, 4, 23]. Other proposals took advantage of the existence of



(a) Cache Area vs. Cache Size

(b) Dynamic Area vs. Cache Size

(c) Leakage Power vs. Cache Size

(d) Access Latency vs. Cache Size

Figure 1.1: Showing how different cache traits are affected by cache size.

data similarities of different granularities across caches. Such approaches used dictionaries for intra-line granularity [7, 17] or used deduplications [23] to take advantage of inter-line granularity similarities. Some studies opted to do data compression to decrease the data size on a line or smaller granularity rather than exploit existing cache wide data patterns [3, 4, 18].

All of the previously mentioned compression approaches improve the utilization of cache space. However, each of those approaches focused on only one of two granularities to do compression. This first granularity was intra-line, only concerned about compressing data inside one data line to make the data line smaller and thus fit more than one data line in one physical cache line. The second granularity was inter-line, focusing only on reducing the total number of cache lines by

getting rid of zero lines or deduplicating similar lines.

Some of today's applications can benefit from only one of those while others can benefit from both of them. For example, the dedup [6] benchmark is only inter-line compressible through deduplication [23], the inversek2j [26] benchmark is only intra-line compressible, while the bwaves [13] benchmark is compressible using both. Using one of the compressions on only one of the granularities is always missing some opportunity. Combining inter and intra-line compressions is the only way to gain the maximum possible compression and the highest benefits. But doing so also means that the cache organization is more complex, requires redesigning replacement policies, and more overhead. To our knowledge, no previous proposals have considered a compressed cache doing inter and intra-line compression at the same time.

In this work we pick two cache compression algorithms to cover inter and intra-line compression. We discuss their cache structures, compression, replacement policies, and their effect on different applications in Chapter 2. Then we describe how to combine both compression schemes in a cache in Chapter 3. In the same chapter we also describe the effects and constraints it imposes on cache organization and replacement policies, and establish an ideal model for evaluation. In Chapter 4 we show the performance of our implementation against the two selected algorithms, and study how the performance of our cache varies with multiple design choices and parameters. Chapter 5 describes related cache compression techniques other than the two we picked. We conclude in Chapter 6 and talk about future work.

Chapter 2

Background and Motivation

2.1 Opportunity

Caches have been observed to have a great degree of redundancy when used with real world applications. Those patterns can be observed on different granularities;on a cache line granularity, and on granularities smaller than a cache line. Some of those patterns are shown in Figure 2.1 and are described as follows:

1. **Intra-line granularity:**

   (a) **Zeros:** A lot of cache data lines are fully zeros [5, 11, 25]. This is a very common case because zeros are usually used to initialize data structures. They are also very common in applications that deal with sparse matrices. Shown in part A of the figure.

   (b) **Repeated Values:** Where a full cache line can contain the same value [1, 20]. This can be observed in applications that initialize large arrays to the same initial value, or in multimedia applications where adjacent pixels could hold the same colours. Shown in part B of the figure.

   (c) **Narrow Values:** Programmers typically code for the worst case and thus have to pick larger data types while the majority of the values can fit in smaller narrower data types [1, 15, 24]. An example is a cache line of 32 bit integers where all the integer values are less than 256. A similar case is shown in part C of the figure.

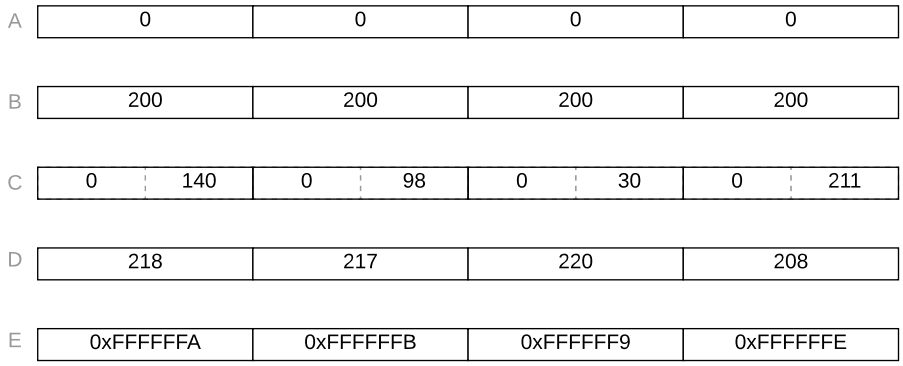


Figure 2.1: Some of the patterns observed in cache lines.

(d) **Near Values:** Lines that contain values that are somewhat similar but not exactly, those are values that have high entropy in their lower bits but have the same higher bits. An example would be a list of pointers that are in the same memory regions, or pixels of an image that have almost similar colours [22, 24]. Part D and E of the figure show examples of this case.

2. **Inter-line granularity:**

(a) **Zero lines:** Similar to the first pattern in the previous point, this case is about data lines being fully zeros. The only difference here is that we consider different zero lines instead of one line being fully zeroed.

(b) **Repeated lines:** Multiple cache lines can be exactly the same. This can happen in scientific benchmarks that have symmetry [23]. For example the benchmark bwaves [13] simulates a spherical blast wave around an origin point, which involves perfect symmetry.

Figures 2.2 and 2.3 show the percentage of cache lines where those inter-line and intra-line patterns occur. A modified version of the zsim [19] simulator was used to dump snapshots of the L3 cache every 100,000 L3 accesses, with a maximum of 10 dumps per benchmark. The figures were created by doing offline analysis on the dumps and searching line by line for all patterns.

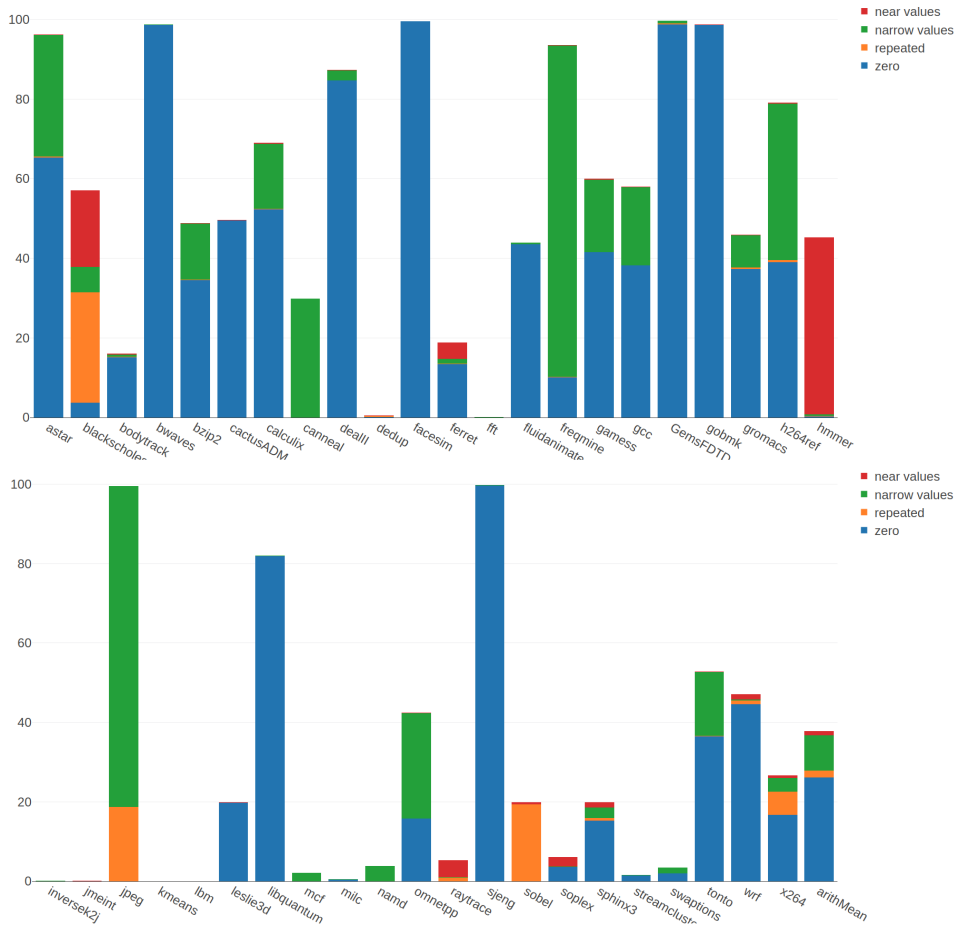


Figure 2.2: The figure shows the percentage of cache lines that have one of the following patterns: zero, repeated values, narrow values, or near values. Recreated from Pekhimenko et al.[18]

2.2 Intra-line Compression

The BDI cache introduced by Pekhimenko et al. [18] is a cache that takes advantage of similarities within data lines to compress those lines into smaller sizes. In the following section we discuss its compression strategy, structure, and operation.

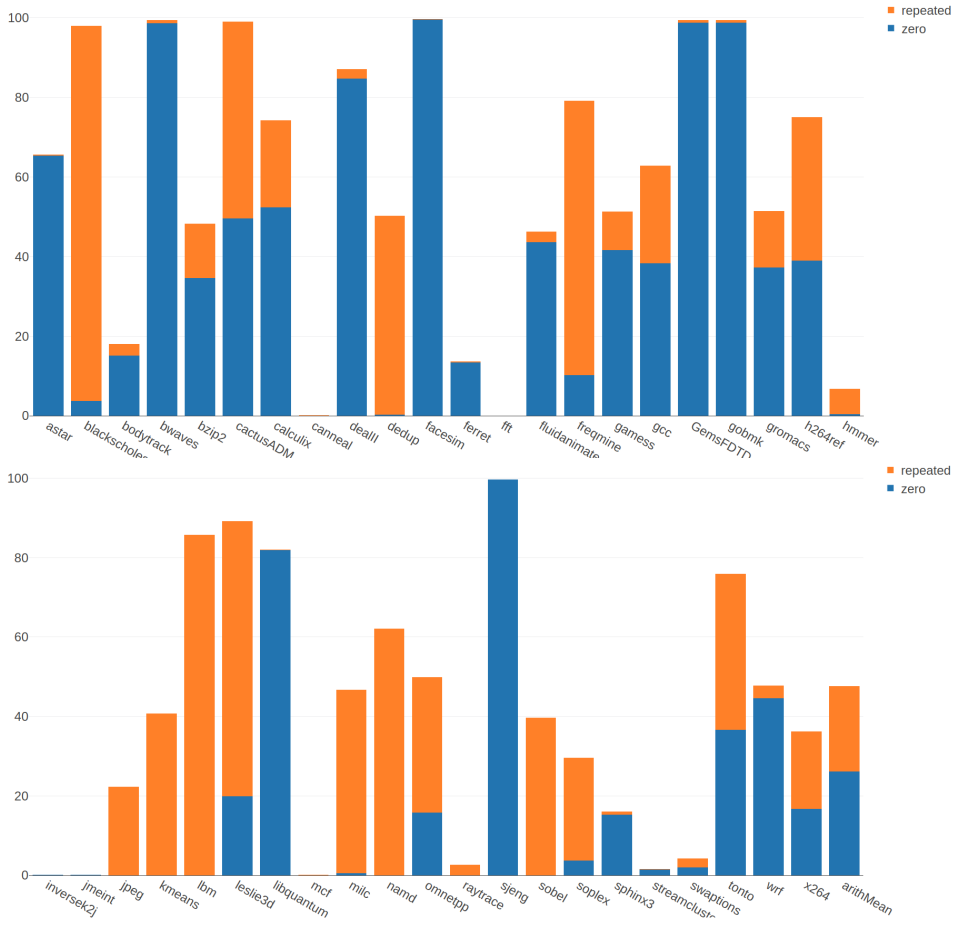


Figure 2.3: The figure shows the percentage of similar cache lines in a cache, those present an opportunity for deduplication. Recreated from Tian et al.[23]

2.2.1 Compression

Base Delta Compression

In the previously mentioned patterns, the data in a cache line granularity have a low dynamic range, and the difference between values in one cache line can be represented using fewer bytes than the original data type. A compression scheme was proposed to utilize this low dynamic range. It works by representing the data

line using a base value, and an array of delta values. Since the delta values can be smaller in size than the original data elements, this allows for a lot of savings in the data line itself. If the delta size required to represent the delta is not smaller than the original size, the line is then not compressible and is left untouched.

Finding the right base is key to compressing the data line optimally, and it happens in two steps:

- **Finding the base size:** The right base size would affect the deltas and their sizes, and thus will affect the final compression size. Since caches have no knowledge of the data types stored in them, compression is not able to identify whether a specific data line is comprised of 16 bit integers or 32 bit floats and so the base size is not directly known. Choosing one base size statically would greatly reduce the opportunity for compression, so the authors opted to allow three different base sizes: 2, 4 and 8 bytes that are used simultaneously. The one that provides the best compression is selected.

- **Finding the base value:** Once a base size is chosen, the base value itself mush be found. Ideally, selecting the base value in a compressed line should be in the middle point between the minimum and the maximum values of a cache line. To avoid the hardware implementation complexity of finding the base, the authors opted to use the first value as base.

The allowed compression schemes and compressed sizes are shown in Figure 2.4 and table 2.1.

Base Delta Immediate Compression

Although one can gain a lot of compression from the base delta compression scheme, some patterns cannot be represented just by one base value. An example would be applications that use structs comprising of different data types. Because of this, allowing the base delta compression to use more than one base might help compress such lines. Based on experiments in [18] it was clear that bases more than two do not provide much additional compression, so the authors selected two bases as their optimum number. To avoid the complexity of looking for a second base when compressing a data line, the authors opted to use the second base as an

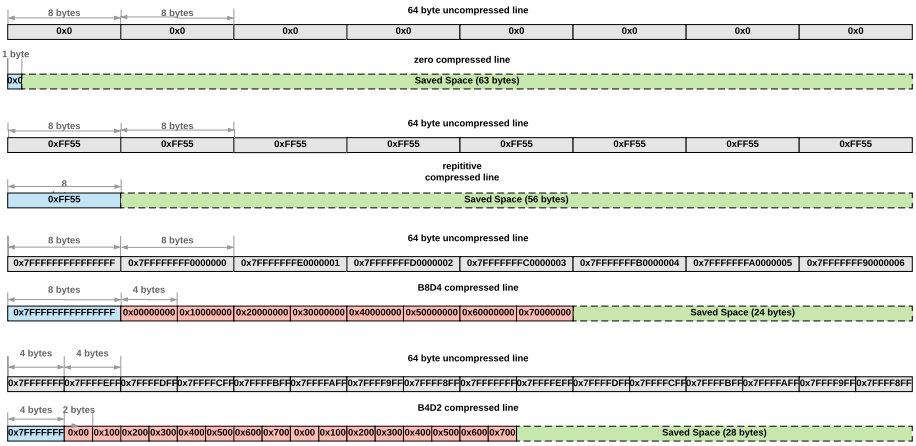


Figure 2.4: The figure shows examples of different cases of Base Delta compression.

Name	Base Size	Delta Size	Compression Size	Compression Encoding
Zero	1	0	1	0000
Rep	8	0	8	0001
B8D1	8	1	16	0010
B8D2	8	2	24	0011
B8D4	8	4	40	0100
B4D1	4	1	20	0101
B4D2	4	2	36	0110
B2D1	2	1	34	0111
Uncompressed	N/A	N/A	64	1111

Table 2.1: The table shows BDI compression sizes, all sizes are in bytes. Original line size is 64 bytes. Recreated from Pekhimenko et al.[18]

implicit zero. The intuition behind this is when structs are used in applications, they are likely to contain wide values with low dynamic range (e.g. Pointers) along with narrow values; the authors observe that using an implicit zero base captures most of the compression enabled by using an arbitrary second base.

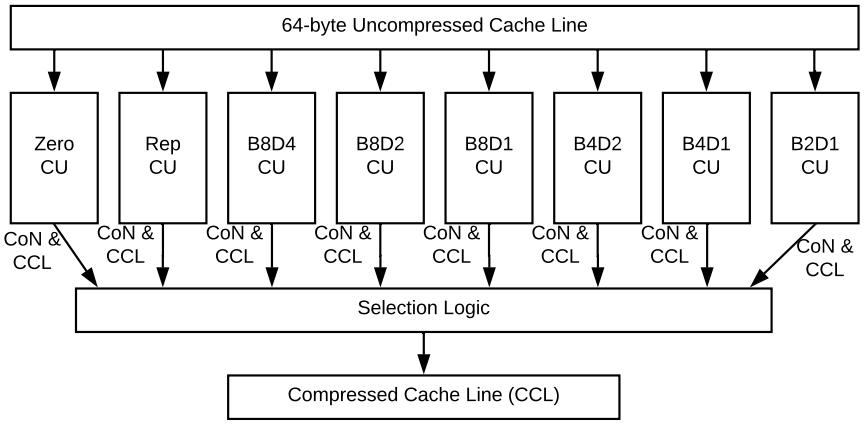


Figure 2.5: The figure shows BDI compression hardware. CU: Compression Unit, CoN: Compressed or Not bit, CCL: Compressed Cache Line. Recreated from Pekhimenko et al.[18]

Compression Hardware

The compression hardware used for BDI consists of eight units that operate simultaneously in parallel, two for the zero and repetitive compression schemes and six units for the three different bases with their deltas. Each unit corresponds for one of the compression sizes described in Table 2.1. Each unit outputs whether the line can be compressed in its scheme or not, and if it can be compressed it also outputs the compressed line. If multiple units can compress the line then the selection logic picks the one with the least compressed size. A compression unit treats the line as elements of its corresponding base size. It picks the first element as the base and then subtracts all the elements from the it. If all the results can be represented in the required delta size then the line is compressed. Figure 2.5 shows the compression hardware.

2.2.2 Structure

The BDI cache builds upon the conventional cache design by adding some modifications to allow compression. Tag and data arrays are arranged in sets, and tag sets

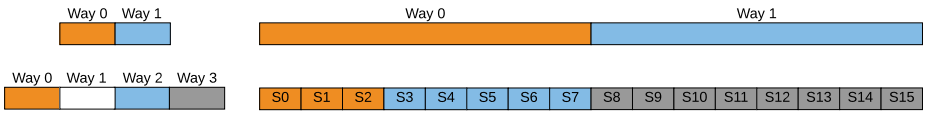


Figure 2.6: Conventional (up) vs BDI (down), showing one set of each. Tags doubled, Data size same but divided to segments from S0 to S7.

and data sets are coupled. The BDI cache structure is shown in Figure 2.6

Tag Array

Tag arrays in a BDI cache are no different than their counterparts in a conventional cache. The tags are arranged in sets and ways.

Along with their normal function, the tags also have two extra fields: a compression metadata field, and a segment pointer. The compression metadata field describes the type of compression in the corresponding data line, and contains a mask to distinguish which base is used for each delta. The segment pointer field is used to point to the first segment of the corresponding data line.

Other than the addition of compression metadata and segment pointer, no other changes to the tag array are required. There are no constraints on its organization or replacement policy. However, because BDI potentially allows more data to be stored in the same cache size, more tags are needed to index this data. The tag array has to generally have more tags than data lines, otherwise no benefits come out of compression.

Data Array

Data lines in a BDI cache are logically divided into eight fixed size segments of eight bytes each (assuming a 64-byte cache line). A compressed data line can occupy any number of segments between one and eight. The data array does not store any metadata.

This general structure means the cache no longer has coupled tag and data *lines* but it still maintains coupled data and tag *sets*, i.e. A tag entry is no longer associated with a data entry in the corresponding location in the data array, but

is associated with one of the segments in the corresponding set in the data array. The location of that segment depends on the compression encodings in the current selected tag set.

2.2.3 Operation

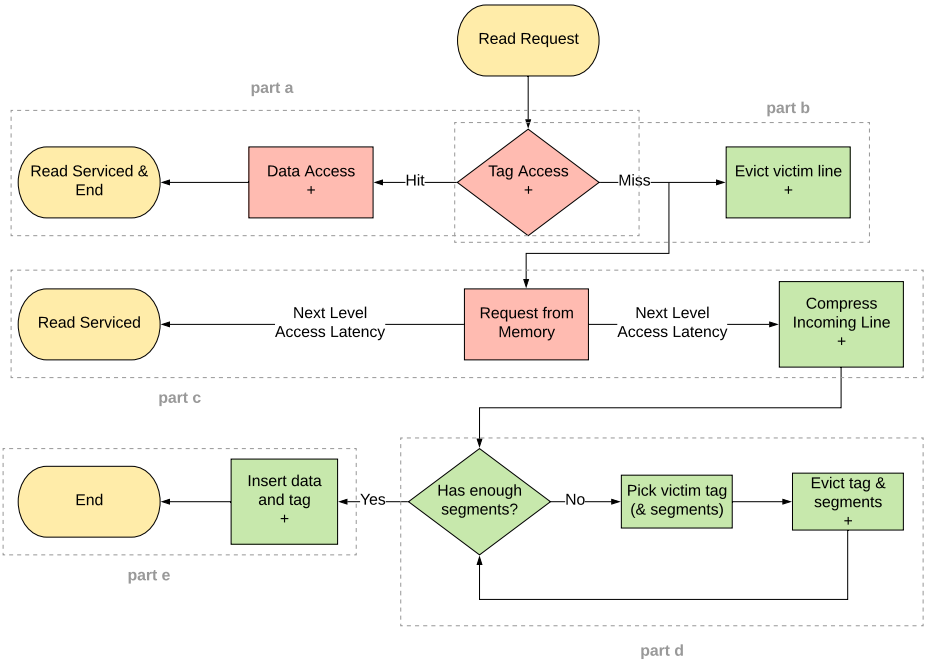


Figure 2.7: The flowchart shows the sequence of actions triggered by a read access to the BDI cache. Red blocks signify actions happening on the critical path, while green blocks mean actions happening off the critical path. Each + sign in any of the blocks signifies an extra latency for tag array access, data array access, or compression.

Cache Read

A flowchart for the BDI cache read is shown in Figure 2.7. When the request first arrives, the tag array is accessed. part (a) in the figure shows the case when the tag access results in a hit then the data line can be read, decompressed, and sent back

to the requester right away. If the tag access is a miss, as shown in part (b), the victim line is evicted. At the same time, in parallel, a request is sent to the next cache level (or main memory). Once the requested line comes back from the next level it can be used to service the requester right away. Inserting the line itself into the cache happens after that and off the critical path, as shown in part (c).

So far through the access everything is similar to a conventional cache. The insertion of the new line is where BDI differs. The insertion of the new line starts in part (c) in the figure. Once the requested line comes back from the next level, and in parallel to serving the requester, the line is BDI compressed. Then, before the line is actually written to the cache, we must verify that there is enough space in the data set for it. If there are not enough segments, we have to trigger extra evictions in the current set to free up more segments. Those evictions happen according to the tag replacement policy. The size caused evictions are shown in part (d), and the line can finally be inserted once enough segments are available as shown in part (e).

To avoid segmentation, the authors assume compaction always happens in the data array before each insertion.

Cache Write

A flowchart of write access to a BDI cache is shown in Figure 2.8. Because we use inclusive write-back caches, if a line is in a lower level cache it must also be in its parent. A cache miss on a write request thus can never happen and a tag array access on a write request will always yield a hit as shown in part (a). In parallel to the tag access, the written data line can also be compressed. Once we know the size of the compressed line, we can check whether or not it fits in its old size. If it remains the same or requires a lower number of segments, as shown in part (b), it can be written right away and the compression metadata in the tag array is also updated. If the new compressed size is bigger than the old size, insertion of the new written line will trigger size based evictions. The replacement policy will be consulted for tags and their segments to be evicted until enough space for the new data line is free; this is shown in part (c). Once there are enough free segments, the line can finally be written and its tag's metadata can be updated accordingly, as

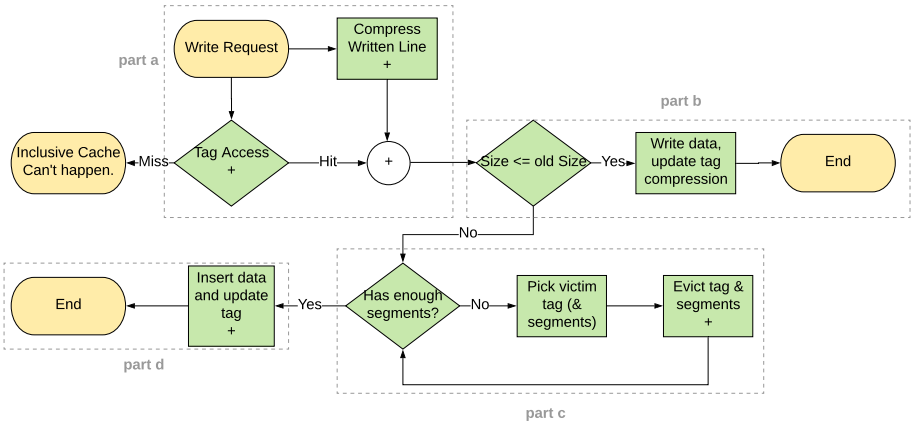


Figure 2.8: The flowchart shows the sequence of actions triggered by a write access to the BDI cache. All the blocks are shaded in green because any write request should be off the critical path of the processor regardless of its status in the cache (hit or miss). Each + sign in any of the blocks signifies an extra latency for tag array access, data array access, or compression.

shown in part (d).

As mentioned previously, the authors assume compaction always happens in the data array before each insertion.

2.3 Inter-line Compression

The Dedup cache was introduced by Tian et al. [23]. The authors found that in some applications, many lines in the cache can be similar, as shown in Figure 2.3. Previous cache compression implementations had been proposed to take advantage of duplications, but only focused on special cases like zeros [1, 9]. Tian et al. proposed and built a cache that takes full advantage of the line similarity by doing deduplication, getting rid of the redundant data copies, and keeping only one unique copy. The structure and operation of such a cache is discussed in this section.

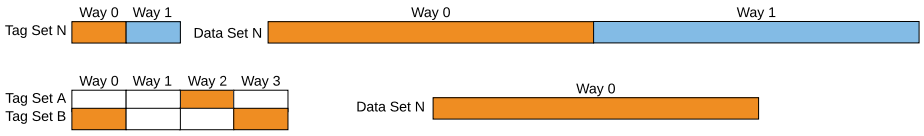


Figure 2.9: Conventional (up) vs Dedup (down), showing one set of tags and data from a conventional cache, but one set and related tags from a dedup cache. Tags doubled, Data size same but is now direct mapped.

2.3.1 Structure

The Dedup cache introduces some major modifications on top of a conventional cache to support deduplication. The tag array is arranged as normal in sets and ways. The data array is no longer coupled with the tag array, instead it has to be explicitly accessed by pointers. So the data array has no constraints or requirements on its associativity, so it is simply left to be direct mapped. To take advantage of deduplications, the tag array must generally have more entries than the data array.

To facilitate finding and locating duplicate lines, a new structure is added to the cache, called the hash array. The hash array is a storage that saves hashes of data lines. Searching and comparing in the hash array makes it faster and cheaper than searching for duplicate lines directly in the data cache.

The Dedup cache structure is shown in Figure 2.9

Tag Array

The Dedup tag array organization is no different than a tag array in a conventional cache, it is arranged in sets and ways.

To support deduplication, some other modifications to the tag array need to be made. Deduplication means more than one tag entry will share the same data entry. This poses two requirements on the Dedup cache: tags must know which data line is the one they need, and if that data line gets evicted, all tags that use it must be evicted too.

Based on these requirements, some additions to the tag array entries must be made. For the tags to be able to address their corresponding data lines, a pointer to the data line must be added to the tag entry. Similarly, to facilitate eviction of all

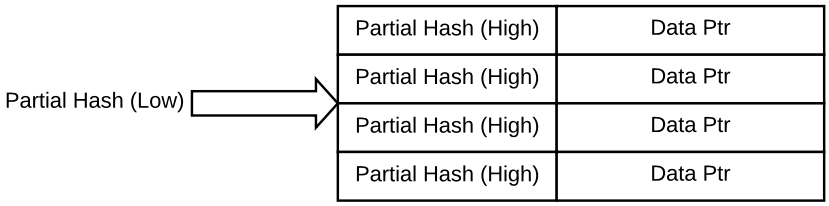


Figure 2.10: A direct mapped Dedup hash array is shown. Hashes are treated like addresses in the tag array: the lower part is used for indexing while the higher part is saved as a tag.

the tags connected to a data line when the line is evicted, those tags are organized in a linked list, so each tag entry has two previous and next pointers that point to the two tags before and after it in a linked list. Those pointers need to be only big enough to point to tag sets: since reading a tag requires reading the whole set, the required tag from that set can be resolved by comparing its data pointer.

Other than the addition of the previous/next and data pointers, there are no other changes to the tag array, As with BDI, tags need to have more entries than data to take advantage of deduplication.

Data Array

The data array in a Dedup cache is decoupled from the tag array, and thus it is not required to have the same associativity as the tag array. In fact, this means the data array is not required to have any associativity at all, and it can be directly mapped.

To facilitate the eviction of the tags associated with the data line in case it gets evicted, the data lines have an extra metadata field, a tag pointer. This tag pointer is used to point to the first tag in the linked list associated with the data line, in case the data line is evicted the linked list is walked and all the tags in it are evicted.

The data line also includes an extra counter. This counter is used to track how many tags use this data line, which is useful to decide which data lines to evict.

Hash Array

The hash array in a Dedup cache is used to store hashes of some of the data lines, providing a way to facilitate finding similar lines and deduplicating them. The hash array is a simple set associative structure. It is shown in Figure 2.10

When a data line is hashed, its hash is split into two parts. One is used to index the hash array, while the other is saved in the hash array itself as a ”tag”. This process is similar to how memory addresses access the tag array, where part of the address is used to index the array while the other part is saved in the array itself.

Along with each hash stored in the hash array, there is also a pointer to a data line that is associated with this hash, similar to the data pointers in tag arrays. The structure of a Dedup hash array is shown in Figure 2.10.

The hashes are computed through an XOR tree. Based on experiments done in [23], the number of hash entries in the array can be kept small (64 entries). Tian et al. show that a small number of hash entries is enough to keep collisions less that 1%.

2.3.2 Replacement Policies

While the tag array in a Dedup cache maintains its normal replacement policies, the data and hash array are treated differently because they are decoupled from the tag array.

Data Array

The data array replacement policy can be split into two parts. The first part uses a list to keep track of all free lines in the data array. This list is always consulted first for a free line to insert into. If the free list is empty, indicating that there are no free lines in the data array, the second part of the replacement policy is then used.

In this case, a random data line is selected. If that line is distinct (not deduplicated, has one tag), it is evicted and used for insertions; otherwise another random line is selected. This process is repeated up to four times, if after those four times no distinct line has been found, the line with the lowest deduplications is evicted and used for insertion.

Hash Array

The hash array is indexed by a part of the hash, the selection of a victim hash set is dependant on the hash that needs to be inserted. Once that set is selected, a hash entry is then selected based on the number of deduplication on the data line it point to, if this data line is not deduplicated then this hash can be used and overwritten, otherwise it is not touched. The rationale behind this is to keep hashes pointing to deduplicated lines from getting evicted and causing the cache to miss extra deduplication in favour of a newly incoming line that might not be useful for deduplication.

2.3.3 Operation

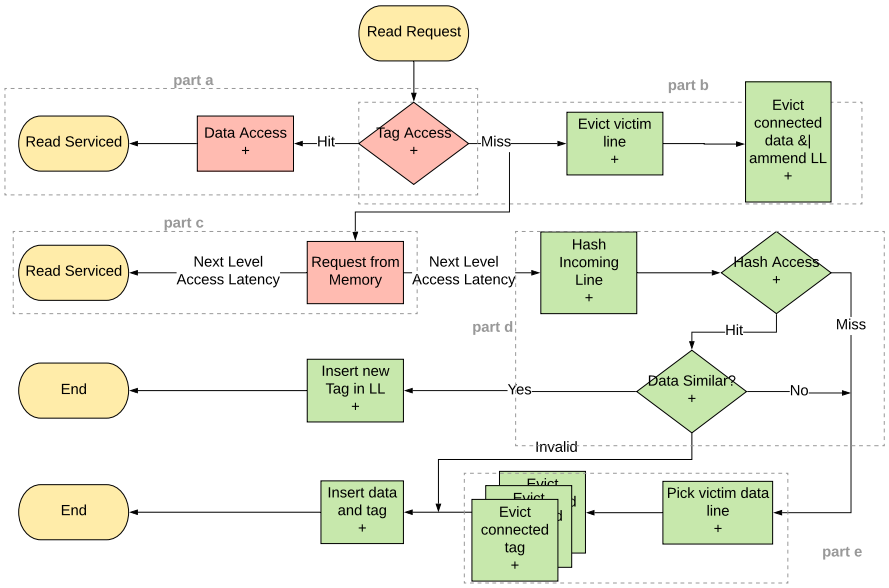


Figure 2.11: The flowchart shows the sequence of actions triggered by a read access to the Dedup cache. Red blocks signify actions happening on the critical path, while green blocks mean actions happening off the critical path. Each + sign in any of the blocks signifies an extra latency for tag array access, data array access, or compression.

Cache Read

A flowchart for the Dedup cache read is shown in Figure 2.11. When the request first arrives, the tag array is accessed. part (a) in the figure shows the case when the tag access results in a hit, and the data line can be read and sent back to the requester right away. If the tag access is a miss, as shown in part (b), a victim tag is evicted and its linked list (if any) has to be updated (i.e. The pointers of the previous and next tags have to connect to each other instead of the victim). At the same time, and in parallel, a request is sent to the next cache level (or main memory). Once the requested line comes back from the next level, it can be used to service the requester right away, as shown in part (c). Inserting the line itself into the cache happens after that and off the critical path.

So far everything is similar to a conventional cache. Only the insertion of the new line requires different actions. The insertion of the new line starts in part (d) of the figure. Once the requested line comes back from next level, and in parallel to serving the requester, the line is hashed and the hash is used to access the hash array.

If the hash access hits, then the incoming line must be compared to the line pointed to by the hash, to make sure there are no collisions. There are four outcomes of this scenario, shown in part (d):

- **Hash Miss:** No similar hash is found, either because similar lines do not exist in the data array, or because the hash array is not big enough to keep track of all data lines. In this case we use the data replacement policy to evict a victim data line. The new tag is inserted and the tag and data lines are made to point to each other; the tag will not point to other tags and will not create a linked list because no deduplication is happening yet. A new hash entry will be selected based on the hash replacement policy and will point to the newly inserted data line.

- **Hash Hit, Line Similar:** In this case the received data line can be deduplicated. It will use the same data line and hash entry and only the new tag needs to be inserted. It is inserted as the head of the linked list and points to the deduplicated data line. The data line's tag counter (deduplication counter) has to be incremented and it has to point to the new linked list head.

- **Hash Hit, Line Invalid:** Because hashes point to data lines, but data lines do not point to hashes, a croner case arises: when a data line is evicted, its associated hash might not be. In this case, we use the invalid line right away instead of consulting the data replacement policy. The tag entry is inserted: it points to the data line, but it doesn't point to any other tags because the data line is not deduplicated yet and shouldn't have a linked list associated with it. The data line in return also points to the tag entry. The hash entry is not changed because it already points to the space we used for the data line.

- **Hash Hit, Line Different:** This case happens only on a hash collision, when two different data lines can be hashed to the same value. Once a collision happens, it is treated like a hash miss with one modification: a new hash insertion is not needed. The same hash entry will be changed to point to the newly inserted data line only if the line it was previously pointing to was not deduplicated (i.e. Its tag counter is 1). Otherwise it remains untouched.

Cache Write

A flowchart of write access to a Dedup cache is shown in Figure 2.12. Because we use inclusive write-back caches, if a line is in a lower level cache it must also be in its parent. A cache miss on a write request thus can never happen and a tag array access on a write request will always yield a hit as shown in part (a). In parallel to the tag access, the written data line can also be hashed.

Once the tag access is finished, we can find out whether the line was distinct or not. If the line is distinct and had no other tags associated with it, then it can be written to right away as shown in part (b). If the line is deduplicated, then a write to the line can change it and cause that tag to lose similarity. The insertion of the written data line then can be handled similarly to a miss. It has to access the hash array and look for similar lines, this has one of the four outcomes described above as shown in parts (c) and (d).

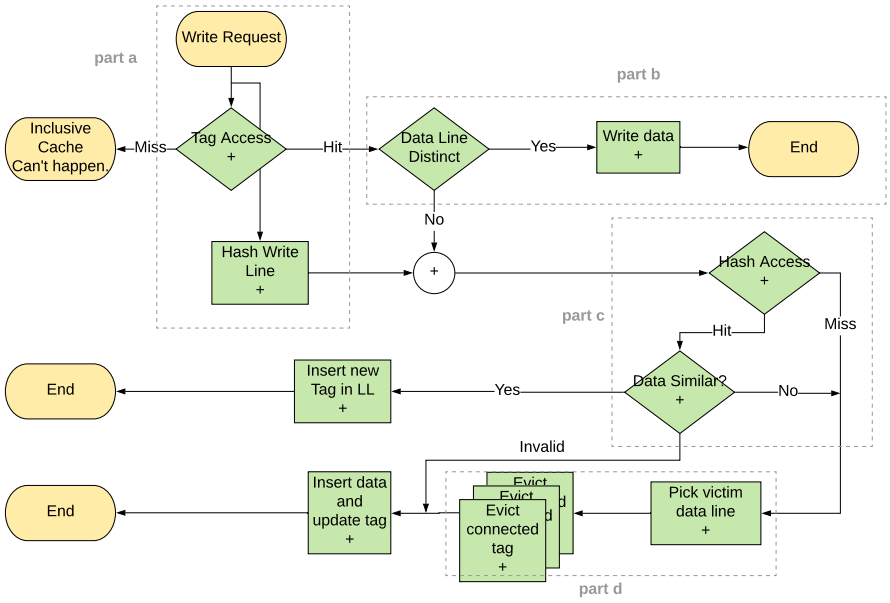


Figure 2.12: The flowchart shows the sequence of actions triggered by a write access to the Dedup cache. All the blocks are shaded in green because any write request should be off the critical path of the processor regardless of its status in the cache (hit or miss). Each + sign in any of the blocks signifies an extra latency for tag array access, data array access, or compression.

2.4 Motivation

Deduplication and base delta immediate compression work on two different granularities. One is inter-line and works on a cache line granularity while the other works intra-line with a granularity of two, four, or eight bytes. This means that they can both be combined to deduplicate similar lines and compress each of those lines at the same time. In this section, we look at four different categories of benchmarks, each with different cache patterns and behaviors. The first category is very compressible by BDI, the second is compressible using deduplication, the third is incompressible, and the fourth is compressible by BDI and deduplication. We pick one benchmark of each category as a representative. Then we do offline analysis on

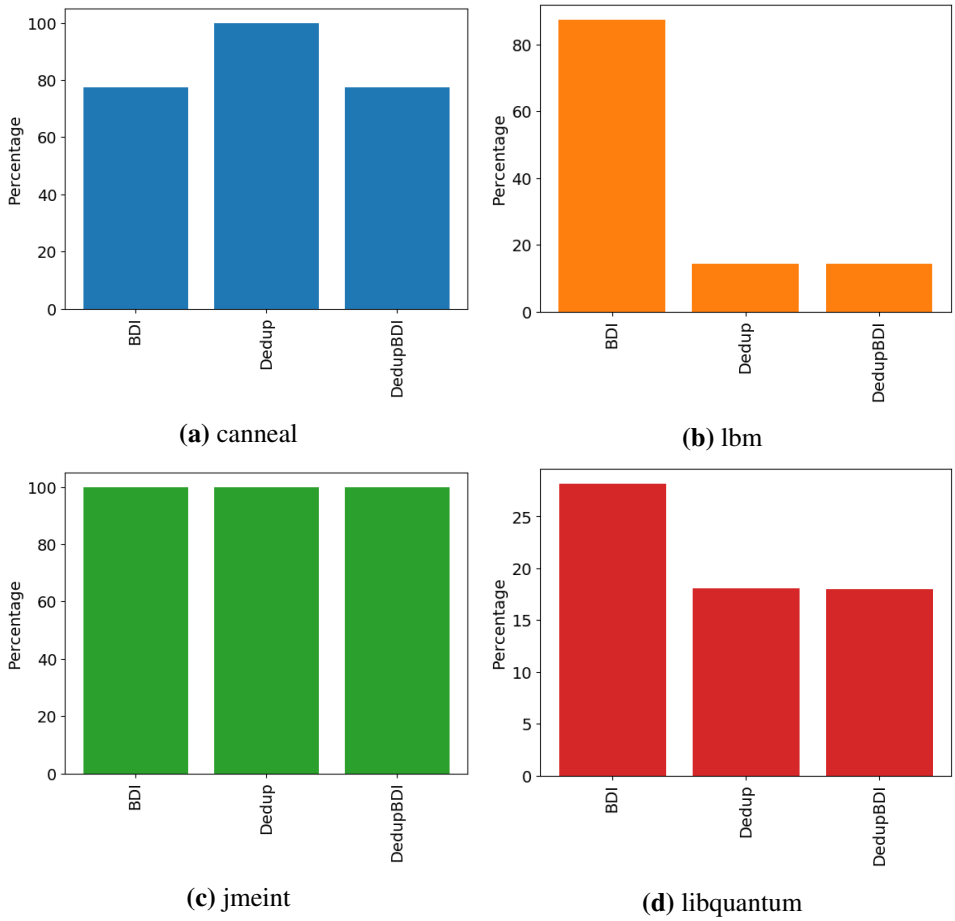


Figure 2.13: The figure shows the percentage of compressed data compared to original cache sizes.

cache dumps of these benchmarks to find how they respond to each type of compression and how a combination of both compression (Dedup+BDI) techniques can improve on them.

The four benchmarks are:

- **canneal:** The canneal benchmark [6] is a cache-aware implementation of a simulated annealing algorithm used for routing in chip designs. canneal is BDI sensitive but doesn't have enough potential to benefit from deudplication. Figure 2.13a shows the Dedup, BDI, and Dedup+BDI compressed data

size of canneal cache dumps as a percentage of the original size.

- **lbm:** The lbm benchmark [13] implements the "Lattice Boltzman Method" to simulate incompressible fluids in 3D. The output from lbm benchmark is a set of values representing 3D velocity vectors for each cell in the simulation. Because of the symmetric flow of fluids around obstacles, lbm is deduplication sensitive. Figure 2.13b shows the Dedup, BDI, and Dedup+BDI compressed data size of lbm cache dumps as a percentage of the original size.

- **jmeint:** The jmeint benchmark [26] is a triangle intersection workload that is used in 3D gaming. It takes coordinates of two triangles as its input. Most of the data in jmeint is insensitive to BDI compression and deduplication. Figure 2.13c shows the Dedup, BDI, and Dedup+BDI compressed data size of jmeint cache dumps as a percentage of the original size.

- **libquantum:** The libquantum library [13] is used to simulate quantum computers. Specifically it simulates the Shor factorization algorithm used for cryptanalysis. It contains (at least at the first few cache dumps) a lot of zero lines which are 2D compressible through deduplication and BDI. Figure 2.13d shows the Dedup, BDI, and Dedup+BDI compressed data size of libquantum cache dumps as a percentage of the original size.

Generalizing on what we found. We used the cache dumps from a modified zsim[19] to analyze the data and show the percentage of cache lines that can be compressed by BDI, the percentage of cache lines that are similar and thus deduplicable, and the percentage of cache lines that can be compressed using both schemes at the same time.

As Figure 2.14 shows, BDI and deduplications are orthogonal to some degree. The existence of some cache lines that can be compressed by only one of both compression schemes means they can work separately without affecting each other. However, because there are lines that can be compressed by both schemes, BDI can lose some of its improvements because of deduplication. For example, if ten lines were similar and compressible by BDI, using BDI will compress each of them on its own, using deduplication will reduce all ten lines to only one line. Using both

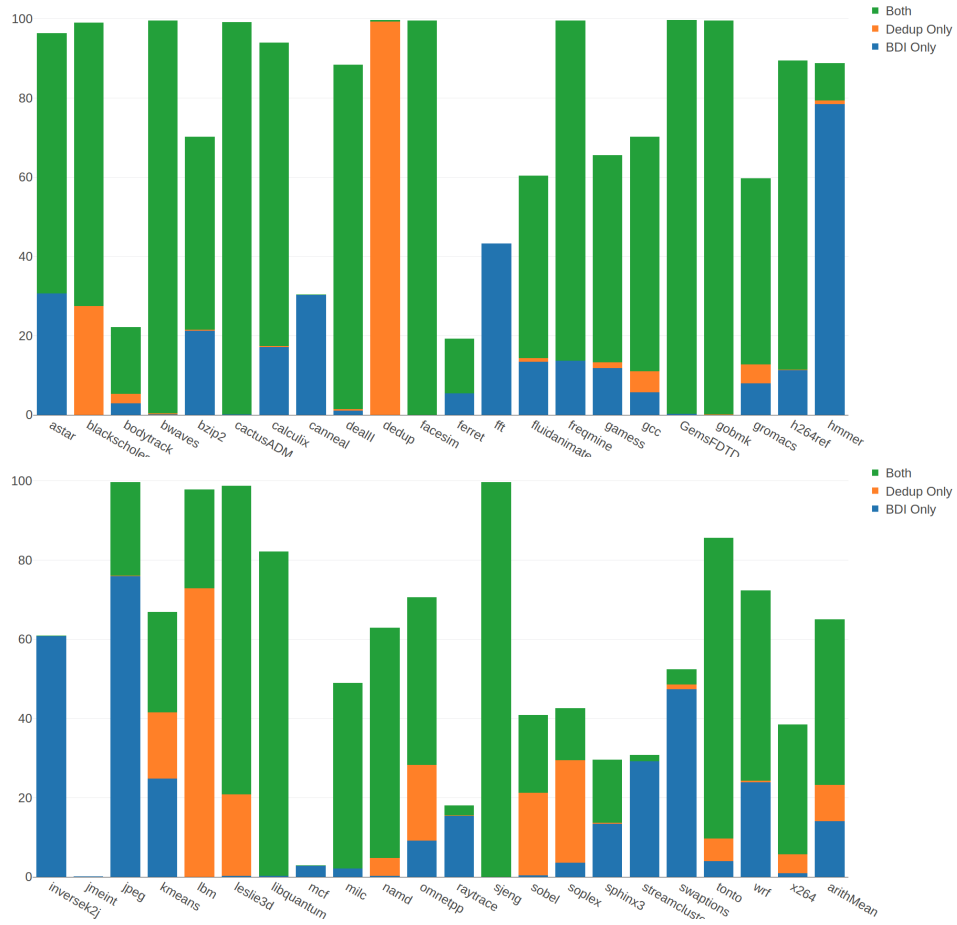


Figure 2.14: The figure shows the percentage of cache lines that can be compressed using Dedup, BDI, or both techniques combined together.

techniques means deduplication still compresses ten lines to one, but BDI now compresses only one line instead of 10. The improvements of BDI can thus be limited by deduplication.

Nevertheless, combining BDI and Deduplication can still outperform each of them on its own. Figure 2.15 shows the compressed data size when using BDI, Dedup, and Dedup+BDI. In its worst case Dedup+BDI performs the same as the best out of Dedup and BDI, in most cases it compresses even more on both. Note that the experiments used to create Figure 2.15 are done statically on cache dumps

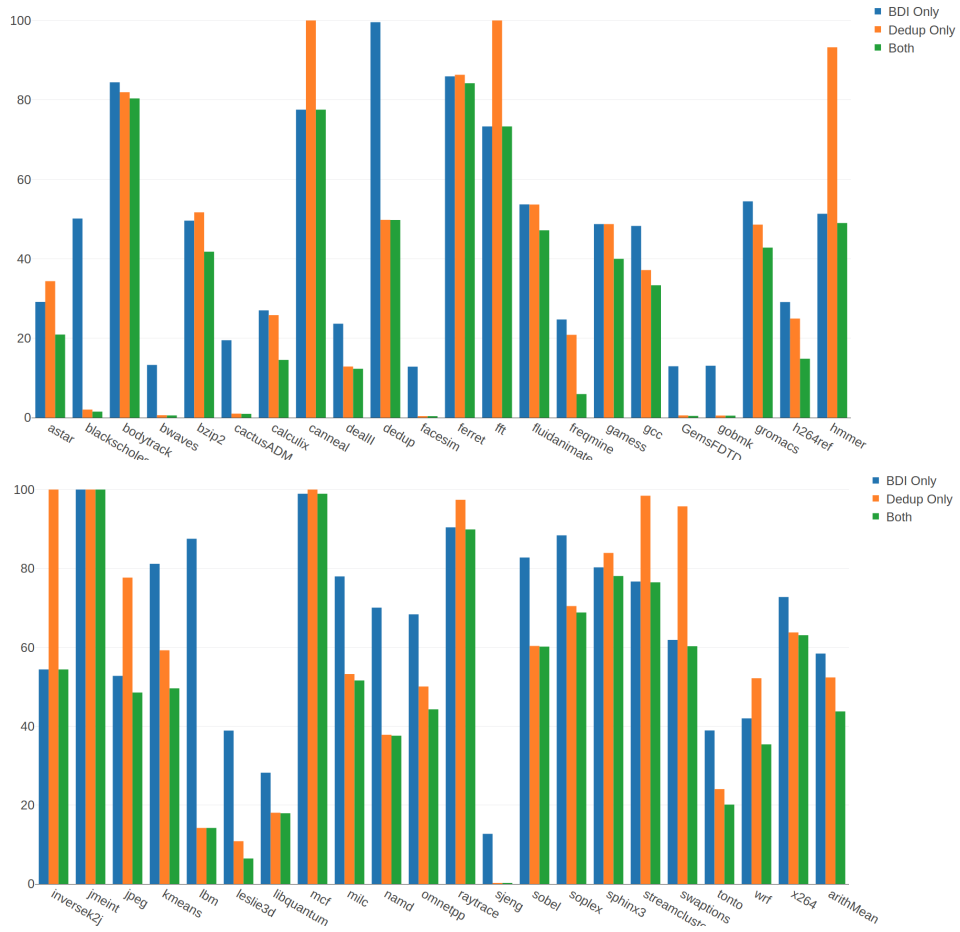


Figure 2.15: The figure shows the size of data in a cache after using compression. BDI, Dedup, and a combination of both is shown.

and thus are completely missing the time dimension and its effect on compression. As time passes more requests to data lines can occur, with more requests the probability that compression can occur increases allowing for better compression.

Chapter 3

Design

In this section, we design a cache that can do deduplication and BDI at the same time. We also describe an ideal implementation of the cache, and discuss improving different aspects of the cache design to allow it to perform more like the ideal implementation.

3.1 A Straightforward Implementation

In this section we propose an implementation of a cache that naively combines inter-line and intra-line compression techniques. This implementation is built simply by combining the Dedup and BDI caches. This cache is thus able to perform deduplication on BDI compressed blocks and uses the same infrastructure and a similar replacement policy to Dedup.

3.1.1 Structure

Like Dedup, the cache consists of three arrays: a tag array, a data array, and a hash array. Unlike a normal cache and similar to a Dedup cache, the tag and data arrays are decoupled and do not have a one-to-one mapping. Instead, related tags and data entries have to be able to point to each other.

On top of its normal operation, the tag array is also used to save compression and deduplication metadata. The hash array is needed to facilitate searching for similar data lines to enable deduplication.

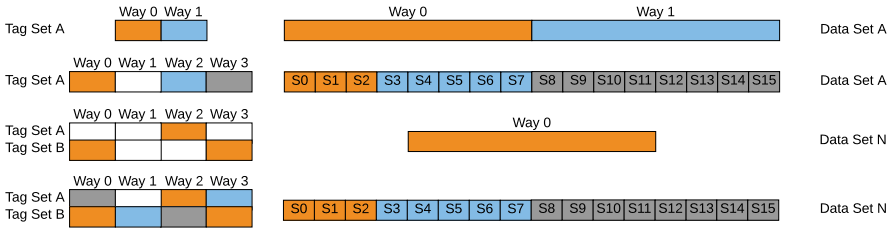


Figure 3.1: From top to bottom: Conventional, BDI, Dedup, DedupBDI. showing one set from the data array of each, and one or more sets from the tag array.

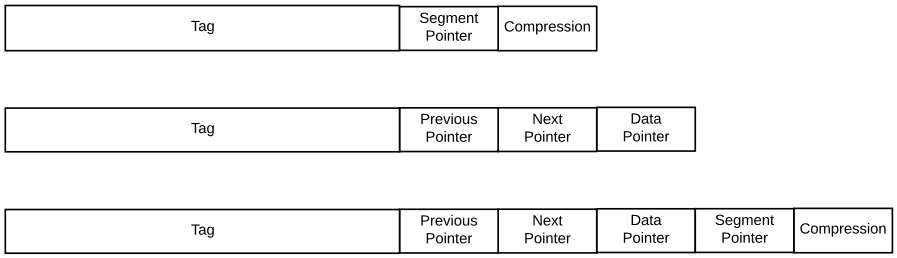


Figure 3.2: From top to bottom: the figure shows three tag entries and their metadata from the BDI, Dedup, and DedupBDI Caches.

The general structure of the cache is shown in Figure 3.1.

Tag Array

The DedupBDI tag array is a normal, set associative array with no limitations on its associativity or organization. It incorporates extra features from Dedup and BDI to support dealing with deduplicated and compressed data lines.

Deduplication in its essence means that multiple tags that have similar data lines will be allowed to only save one copy of this data. This breaks the conventional one-to-one relationship between tags and data in a normal cache making the tag and data arrays decoupled. Because of this, a tag entry needs an extra pointer to its corresponding data entry. Unlike Dedup tag array's data pointer, this pointer needs to be able to point to compressed data lines that are smaller in size than nor-

mal cache lines. The pointer needs to be larger in size to be able to address lines of smaller granularity. It is actually divided into two pointers: a data set pointer, and a segment pointer within the set, allowing the tag to point to a compressed line anywhere withing the data array boundaries.

Similar to BDI, The tag entries also need to save compression encoding to be able to resolve the size of the compressed data it points to.

Similarly to Dedup, tag entries that share the same deduplicated data line are arranged in a linked list, facilitating their eviction in case that data line is evicted. So each tag entry has two extra previous/next pointers to allow them to form a linked list. Those pointers need to be big enough to point to tag sets. Since reading a tag requires reading the whole set, the required tag from that set can be resolved by comparing its data pointer. The use of a linked list is necessary in case of data line eviction.

In general, the tag array must have more tag entries than the data array, otherwise it wouldn't be able to utilize compression and deduplication. The structure, metadata, and differences between tag entries in Dedup, BDI, and DedupBDI is shown in Figure 3.2. It shows previous and next pointers in Dedup and DedupBDI tags, segment pointers in BDI, and data pointers in Dedup and DedupBDI.

Data Array

The DedupBDI data array is similar to that of a BDI cache. Each data line in the DedupBDI data array is logically divided into eight segments of eight bytes each (assuming a 64 byte cache line). A compressed data line can occupy any number of segments between one and eight, in the best case scenario all lines will be compressed into one segment each.

The data array is decoupled from the tag array because it must also support deduplication. In case a compressed data line is evicted, all the tags associated with it must also be evicted. The compressed data line then must be able to point to the tag linked list and to know how long that list is. Accounting for the best case scenario where each line is compressed into one segment, each segment must have two metadata fields: a pointer to the head of the tag linked list that is associated with this line, and a counter of the tags in that linked list. Both those fields are

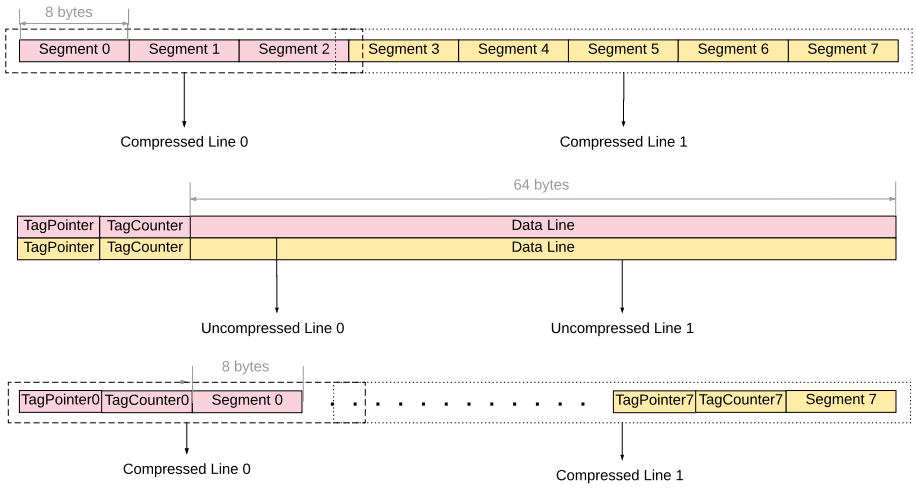


Figure 3.3: From top to bottom: the figure shows three data entries and their metadata from the BDI, Dedup, and DedupBDI Caches.

useful in case an eviction of a data line happens.

The pointer needs to be only big enough to point to a tag set; The target tag from this set can be determined by comparing its data pointer. The counter only needs to be two bits large, as it represents zero/one/two/more. In cases of eviction it can be determined if the count is less than 3 by checking only one tag entry for previous and next pointers.

The structure and differences among tag entries in Dedup, BDI, and DedupBDI is shown in Figure 3.3

Hash Array

The hash array is a simple set-associative structure. It is used to store hashes of some of the data lines, providing a way to finding similar lines and deduplicate them. It is the same as that of a Dedup cache. When a data line is hashed the hash is split into two parts: one is used to index the hash array, while the other is stored in the hash array itself, similar to the tag part of the address being stored in tag array.

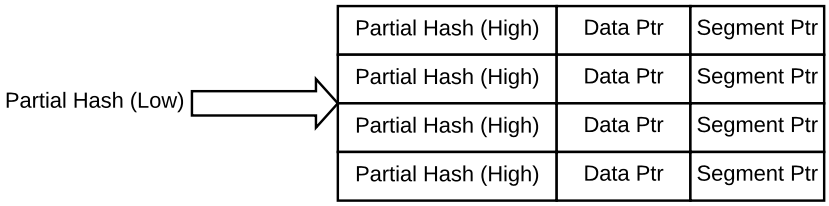


Figure 3.4: A direct mapped Dedup hash array is shown. Hashes are treated like addresses in the tag array: the lower part is used for indexing while the higher part is saved as a tag.

With each entry in the hash array there is also a pointer to a data line that is associated with this hash. Just like the data pointers in the tag array, the pointer is actually two parts: one points to the data set, and the other to the segment in that set. The structure of a Dedup hash array is shown in Figure 3.4.

The hash itself is computed like Dedup, through an XOR tree. Similar to Dedup, it uses only 64 entries.

3.1.2 Operation

Cache Read

A flowchart for the Dedup cache read is shown in Figure 3.5. When the request first arrives, the tag array is accessed. part (a) in the figure shows the case when the tag access results in a hit, and the tag points exactly to the required line. In this case, the data line can be read, decompressed, and sent back to the requester right away. If the tag access is a miss, as shown in part (b), the victim line is evicted. If the tag was connected to a distinct data line, the tag is deleted as well; otherwise the linked list containing the tag has to be updated. Updating the linked list in this case means changing the pointers of the previous and next elements to point to each other instead of the victim.

At the same time, in parallel, a request is sent to the next cache level (or main memory). Once the requested line comes back from the next level it can be used to

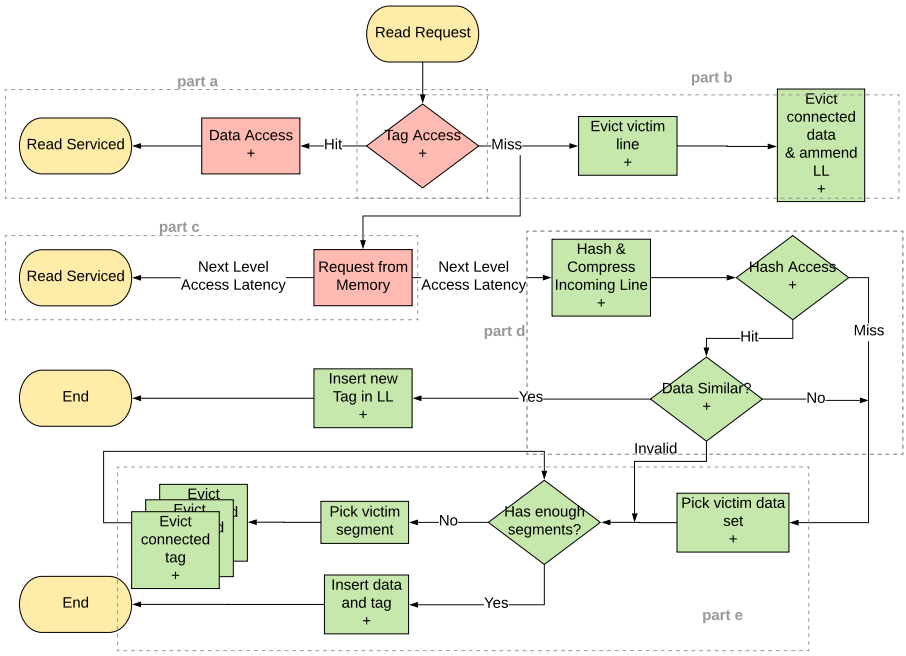


Figure 3.5: The flowchart shows the sequence of actions triggered by a read access to the DedupBDI cache. Red blocks signify actions happening on the critical path, while green blocks mean actions happening off the critical path. Each + sign in any of the blocks signifies an extra latency for tag array access, data array access, or compression.

service the requester right away, as shown in part (c). Inserting the line itself into the cache happens after that and off the critical path.

Other than updating the linked list if needed, everything in the access so far is similar to a conventional cache. The insertion of the new line is going to be different. The insertion of the new line starts in part (d) in the figure. Once the requested line comes back from next level, and in parallel to serving the requester, the line is hashed and compressed. Then the hash is used to access the hash array.

If the hash access hits, then the incoming line must be compared to the line pointed to by the hash, to make sure there are not collisions. There are four outcomes of this scenario, shown in part (d):

- **Hash Miss:** No similar hash is found, either because similar lines do not exist in the data array, or because the hash array is not big enough to keep track of all data lines. In this case we use the data replacement policy to pick a set and evict compressed data line(s) until sufficient space for the compressed line exists. The new tag is inserted and the tag and data lines have to point to each other. The tag will not point to other tags and will not create a linked list because no deduplication is happening yet. A new hash entry will be selected based on the hash replacement policy and will point to the newly inserted data line. This is shown in part (e) of the figure.

- **Hash Hit, Line Similar:** In this case the received data line can be deduplicated. It will use the same data line and hash entry. Only the new tag needs to be inserted. It is inserted as the head of the linked list and points to the deduplicated data line. The data line's tag counter (deduplication counter) has to be incremented and it has to point to the new linked list head.

- **Hash Hit, Line Invalid:** Because hashes point to data lines, but data lines do not point to hashes, when a data line is evicted, its associated hash might not be, causing a situation like this to arise.

  In this case, we utilize the invalid space right away instead of consulting the data replacement policy. This is similar to the same case in Dedup with one minor difference: if the invalid space is not enough for the compressed line, we use the data replacement policy to start evicting from the current data set until there is enough space for the compressed data line to be inserted.

  The tag entry is also inserted. It has to point to the data line, but it doesn't point to any other tags because the data line is not deduplicated yet and should not have a linked list associated with it. The data line in return also has to point to the tag entry. The hash entry is not changed because it already points to the space we used for the data line. This case is also shown in part (e).

- **Hash Hit, Line Different:** This case happens only on a hash collision. Once a collision happens, it is treated like a hash miss with one modification. A new hash insertion is not needed, so the same hash entry will be changed

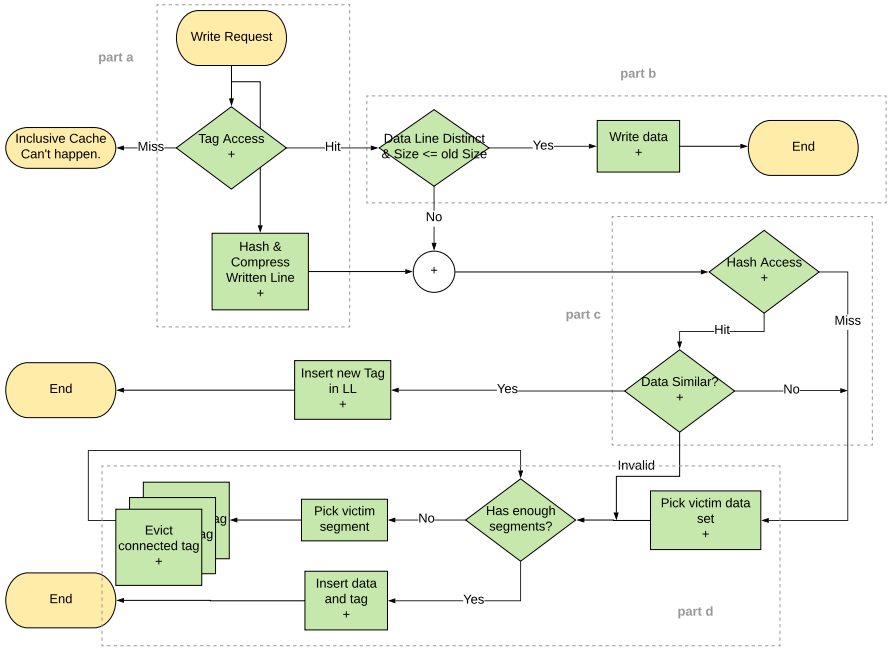


Figure 3.6: The flowchart shows the sequence of actions triggered by a write access to the DedupBDI cache. All the blocks are shaded in green because any write request should be off the critical path of the processor regardless of its status in the cache (hit or miss). Each + sign in any of the blocks signifies an extra latency for tag array access, data array access, or compression.

to point to the newly inserted data line in only if the line it was previously pointing to was not deduplicated (i.e. Its tag counter is 1). Otherwise it remains untouched. This follows the same policy used in Dedup.

Similar to Dedup, we assume compaction always happens in the data array before each insertion.

Cache Write

A flowchart of write access to a DedupBDI cache is shown in Figure 3.6. Because we use inclusive write-back caches, if a line is in a lower level cache it must also

be in it's parent. A cache miss on a write request thus can never happen and a tag array access on a write request will always yield a hit, as shown in part (a). In parallel to the tag access, the written data line can also be hashed.

Once the tag access is finished, we can find out whether the line was distinct or not. If the line had no other tags associated with it and the new size is less than or equal to the original size, then it can be written to right away as shown in part (b). If the line is deduplicated, then a write to the line can change it and cause that tag to lose similarity. The insertion of the written data line then can be handled similarly to a miss. An access to the hash array has to be made to look for similar lines, this has one of the four outcomes described above as shown in parts (c) and (d).

3.1.3 Replacement Policies

Some parts of the DedupBDI cache like the tag array can still operate in the same way with the same replacement policies. The tag array can use one of the known replacement policies like LRU. Other parts of the cache have to be treated differently. Since the data and hash array are decoupled from the tag array each one of them needs its own replacement policy.

Data Array

The data array uses a somewhat similar replacement policy to Dedup data replacement policy. A free list keeping track of free data sets (not lines) is used. Whenever needed an entry of this list is used to insert a line. If the list is empty then up to four data sets are picked at random. If any of the selected four sets have enough segments for insertion of the line then it is selected right away, otherwise the one with the least sum of deduplication counters is picked. This is similar to the data replacement policy in Dedup which victimizes the line with the least number of deduplications.

The picked data set then is used for the second part of the replacement policy, in which a segment (or group of segments representing a compressed data line) with the lowest deduplication counter is selected for eviction. Segments keep evicted as necessary until an enough space is freed for the insertion. Each segment or group of segments evicted can trigger a chain of evictions for the tag linked list associated

with it, starting from the linked list head that is pointed to by the segments, then following the next pointers in the tags.

Hash Array

The hash array is indexed by a part of the hash. The selection of a victim hash set is thus dependant on the hash that needs to be inserted. Once that set is selected, a hash entry is then selected based on the number of deduplications on the data line it points to. If this data line is not deduplicated then this hash can be used and overwritten. Otherwise it is not touched. The rationale behind this is to keep hashes pointing to deduplicated lines from getting evicted and causing the cache to miss extra deduplication for a newly incoming line that might not be useful for deduplication.

3.1.4 Interface with Memory

Deadlock Mitigation

In a system with only two queues between the last level cache and the main memory—a response queue (RespQ) which gets data from main memory to the last level cache, and a request queue (ReqQ) to send read/write requests from the last level cache to main memory—the potential need to evict multiple compressed blocks to fit a new less-compressed block introduces a new potential source of deadlock.

When a read is serviced by the main memory and enters RespQ, it may trigger an eviction. This means that one writeback line may be enqueued in ReqQ, and there must be space for it. In an uncompressed cache, this does not cause a problem: the cache just freed an entry in RespQ, which allows the memory to service another request and create a free slot in ReqQ.However, in the same case in a compressed cache, the line coming in RespQ might cause eviction of multiple victims not just one. This means before the cache can read a line from the RespQ, multiple victims have to be inserted into ReqQ. But multiple insertions to ReqQ cannot happen because main memory needs space in RespQ to service the ReqQ. The cache and memory can then enter a deadlock.

One solution for the deadlock is to split the ReqQ into two different queues,

one for blocking requests like reads, atomics, and fences (BlkQ) and another for non-blocking writeback requests (WriteQ). If the cache and memory arrive at the deadlock situation described above, the memory can always service WriteQ first (since servicing writebacks doesn't require responding to the RespQ), eventually creating enough slots for the cache to enqueue all of its writeback requests. This avoids the deadlock scenario.

3.2 Establishing an Upper Bound

To be able to evaluate the straightforward design we discussed in the previous section. We first establish an upper bound by idealizing the design choices we have selected. There are two aspects we can idealize in this design:

- **Finding a similar line:** Finding a similar line in the DedupBDI cache has two sources of possible error:
  - **Hashing:** Because hashing data lines to a smaller space means it will never be perfect and collisions are inevitable.
  - **Size of Hash Array:** Because the number of entries in the hash array is less than that of the data array, there is a chance that an opportunity for deduplication is missed because the hash array was too small and couldn't keep its hash. Finding a similar line can be idealized by directly searching through the data array and comparing each line until a similar data line is found.

  Figure 3.7 shows the percentage of cache lines in a DedupBDI cache that could have been deduplicated but are not due to imperfections in the hashing in the straightforward implementation. The results are generated by using a modified version of the zsim [19] simulator to dump snapshots of the L3 cache every 100,000 L3 accesses, with a maximum of 10 dumps per benchmark. The figures were created by doing offline analysis on the dumps for similar lines.

- **Replacement of data lines:** Imperfections in the replacement policy of data array comes from two sources:

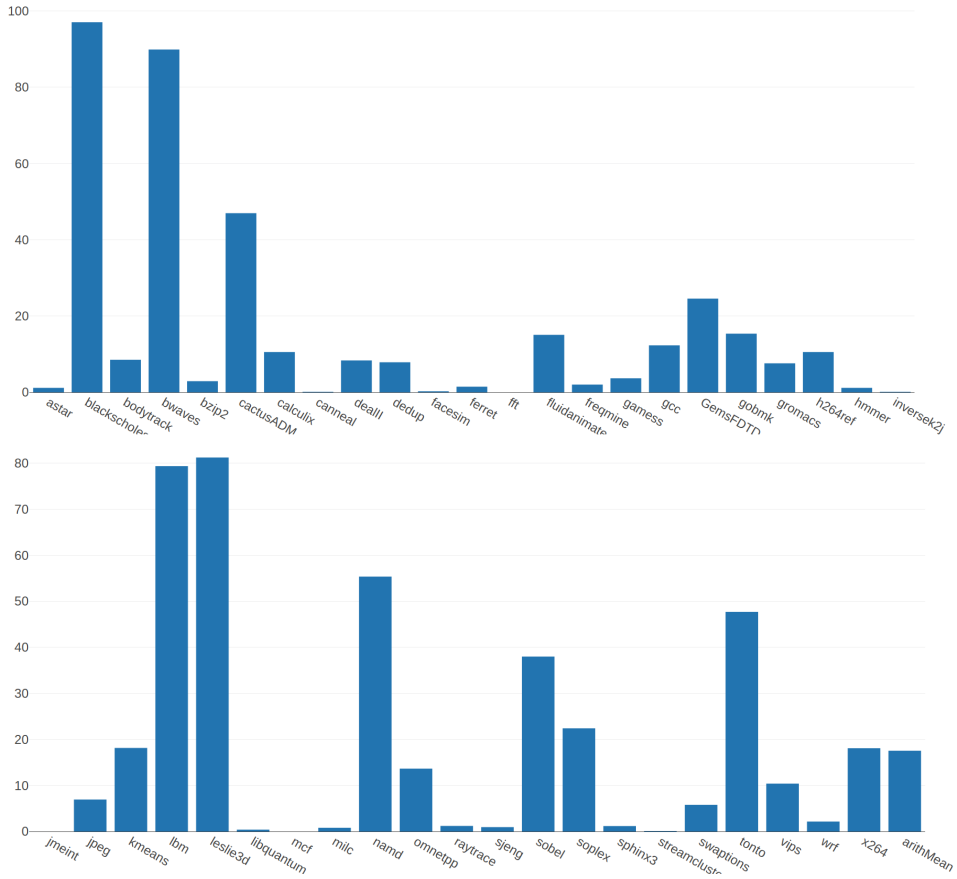


Figure 3.7: The figure shows the percentage of similar cache lines in a cache that could not be deduplicated

- **Free Lists:** We cannot make the free list keep track of all free segments in the data array because it would be impractically large. It can only keep track of free sets. That means even if only one segment is used from a set that set is not kept in the free list anymore.
- **Random Selection of Data:** Whenever the free list is empty, random selection of data lines is required. The random selection of four victim lines might not always yield free segments. But selecting more than four can be time inefficient. The random selection does not amend the harm caused by the inefficiency of the free list.

Our experiments showed that the combination of both sources can cause utilization of the data array to to reach as low as 80%. We idealize the replacement policy by directly searching the whole data cache for free segments that are enough to insert a new compressed line. This ensures a 100% utilization of the data array. If no free segments are available we use data-array-wide LRU to pick a victim segment(s) instead of doing it randomly.

With the use of perfect deduplication and data insertion, a hash array is no longer needed. And the tag array is no different than a normal cache.

3.3 An Optimized Implementation

In order to improve on the naive implementation and towards the upper bound, we propose the following realistic improvements on the baseline implementation discussed in Section 3.1.

- **Deduplication:** While any realistic deduplication will still depend on the existence of a hash array, Deciding the number of entries in the hash array can be tricky. The hash array used in the straightforward implementation was the same as Dedup, it only used 64 hashes. The hash array was enough to cover deduplications in the data array of 2.3. However, because of the use of compression in DedupBDI, each line can hold up to eight times the amount of data. The same hash array with the same number of entries might not be enough to cover that amount of deduplication efficiently. In Figure 3.8 we show how varying the number of hash array entries can affect the performance of deduplication in DedupBDI cache. Increasing the number of hash entries from 64 to 1024 only reduces the number of lines that are missed from deduplication by only 1.5%. We use the hash array with 64 entries for the rest of this work, just like the straightforward implementation. We've set the associativity of the hash array to 16. Increasing or decreasing the hash associativity in our experiments had almost no effect.

- **Data free list:** To keep track of all free segments in the data array, a modification to the free list was made. The free list is now divided into multiple free lists, each of which can keep track of data sets with a certain amount

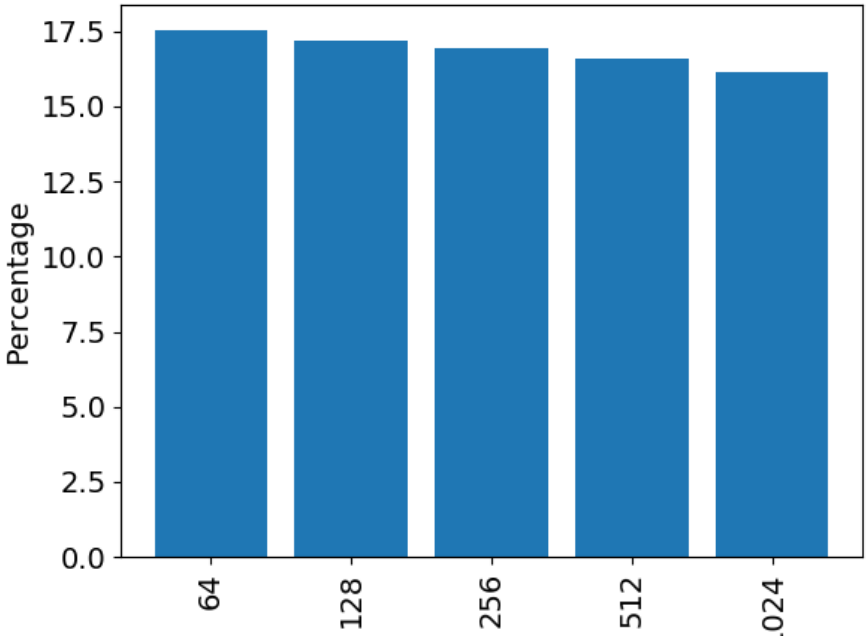


Figure 3.8: The figure shows the percentage of similar cache lines in a cache that could not be deduplicated, against the number of hash entries in the hash array

of free segments. Namely we have eight free lists: the first keeps track of data sets with one free segment and the second with two, until the last which keeps track of data sets with eight or more free segments. Whenever a data line needs to be inserted, the free list corresponding to its size is looked up first. If it is empty, then the larger ones are searched one by one. If no empty segments exist then we return to the second part of the replacement policy.

- **Replacement of data lines:** In the straightforward implementation in case of a hash hit that points to an invalid data segment(s), described in 3.1.2, we opted to victimize those data segments and use them for insertion rather than consult the replacement policies. This was following what the Dedup cache does in the same case. However, unlike the Dedup cache, the invalid segments might not be enough for insertion, causing extra evictions in the same

set without consulting the replacement policy. This caused our eviction ratio to be high and hurt performance. A better alternative is to always consult the replacement policy to find the best location for insertion.

The cache size including area overhead, leakage power, and dynamic energy is shown in Chapter 4

Chapter 4

Results

In this chapter we show results for the BDI cache, the Dedup cache, and the improved implementation caches along with the upper bound we established for DedepBDI. We show how those four cache designs affect compression, MPKI, and speedup.

4.1 Methodology

We used the zsim [19] simulator to design and implement the four target caches: BDI, Dedup, DedupBDI, and Ideal DedupBDI along with a baseline conventional (uncompressed) cache. We used an out-of-order core timing model based on a medium-size x86 core, with detailed timing for all critical-path and off-critical-path events. We simulated a system with a 3 level cache hierarchy, with private L1

Component	Configuration
CPU	x86_64, 2.6GHz, 4-wide OoO, 80 entry ROB.
L1I	32KB, 4 way, 3 cycle access lat, 64B lines, LRU.
L1D	32KB, 8 way, 4 cycle access lat, 64B lines, LRU.
L2	Private, 256KB, 8 way, 11 cycle access lat, 64B lines, LRU.
L3	Shared, 4MB (or similar data array size if compressed), 16 way, 39 cycle access lat, 64B lines, 8 banks.
Mem	DDR3-1066, 1GB.

Table 4.1: The simulated system

Compression	Cache	Data Entries	Tag Entries
Conventional	0.5MB	8192	8192
	1MB	16384	16384
	2MB	32768	32768
	4MB	65536	65536
	8MB	131072	131072
COMP-2	0.5MB	8192	16384
	1MB	16384	32768
	2MB	32768	65536
	4MB	65536	131072
	8MB	131072	262144
COMP-4	0.5MB	8192	32768
	1MB	16384	65536
	2MB	32768	131072
	4MB	65536	262144
	8MB	131072	524288

Table 4.2: All cache configurations.

and L2 caches and a shared L3 cache. The simulated system is listed in Table 4.1.

We simulated different conventional L3 cache sizes ranging from 0.5MB to 8MB. We also simulated the four compressed caches at the same data array sizes, but with tag arrays double and four times of the data lines. For example, a 4MB-2 BDI cache has the same data array as a conventional 4MB cache, but its tag to data ratio is 2 so its tag array size is double its conventional counterpart. The LLC cache configurations are shown in 4.2. The workloads used are all the benchmarks from the AxBench [26] benchmark suite, the Parsec [6] benchmark suite with simmedium inputs, and the Spec [13] benchmark suite with train inputs (mid-size inputs). All the benchmarks were simulated until end or 3 billion instructions.

The offline analysis done in Chapters 2 and 3 were done using a modified version of zsim. zsim was used to dump cache contents every 100,000 L3 accesses, then the dumps were used for data line pattern analysis.

4.2 Case Study

In this section we describe the four benchmarks we picked in 2.4. The benchmarks have different cache patterns and behaviors. Recall that during the offline cache dump analysis done in 2.4 canneal was only compressible by BDI, dedup was compressible using deduplication, jmeint was incompressible, and libquantum was compressible by both. We analyze how these benchmarks behave when simulated in a real computing system with compressed caches.

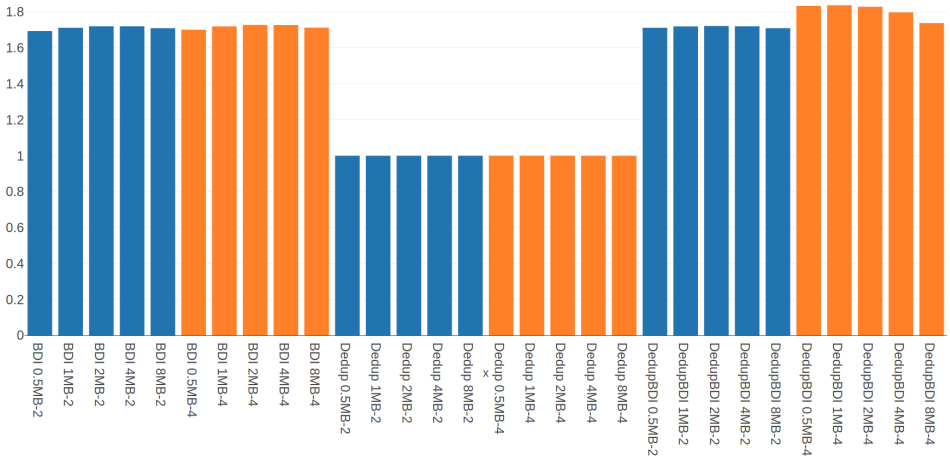
Figures 4.1 and 4.2 show the compression ratio of the four benchmarks. Compression ratio is calculated as the ratio between valid tags and valid data entries. The blue columns are caches with tag entries double the data entries, while the orange ones have tags four times the data entries. As expected, canneal shows some compression with BDI but nothing with Dedup, lbm shows compression using Dedup but not using BDI, jmeint shows no compression using any of the caches, while libquantum shows compression using all of them. The DedupBDI cache combines both compressions and either outperforms or at least does as well as BDI or Dedup.

Figures 4.3 and 4.4 show the MPKI and Speedup for the four benchmarks under all cache configurations. Apart from the incompressible jmeint, all the benchmarks show big speedups (up to 70% in case of libquantum). DedupBDI is the best performing cache in canneal, lbm and libquantum. But it is slightly worse (0.05% slowdown) than a conventional cache in jmeint.

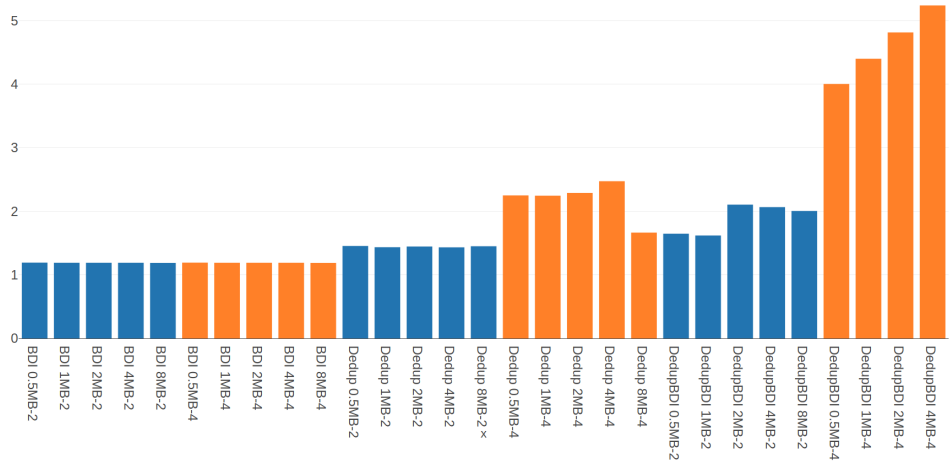
The results shown are in sync with the offline analysis we did in section 2.4. This shows how the DedupBDI cache brings together the best of both worlds. It is able to function exactly similar to BDI and Dedup in cases that are only compressible by one, and it is able to use both compressions at the same time to enhance compression and performance. Rhe cost for this is its increased complexity and access latency that causes it to perform slightly worse in incompressible cases.

4.3 Compression

Figure 4.5 shows the compression ratio for all the benchmarks when simulated with a compressed 4MB LLC with tags four times the data lines. The benchmarks are ordered by their compressibility. The first five benchmarks are hardly compressible

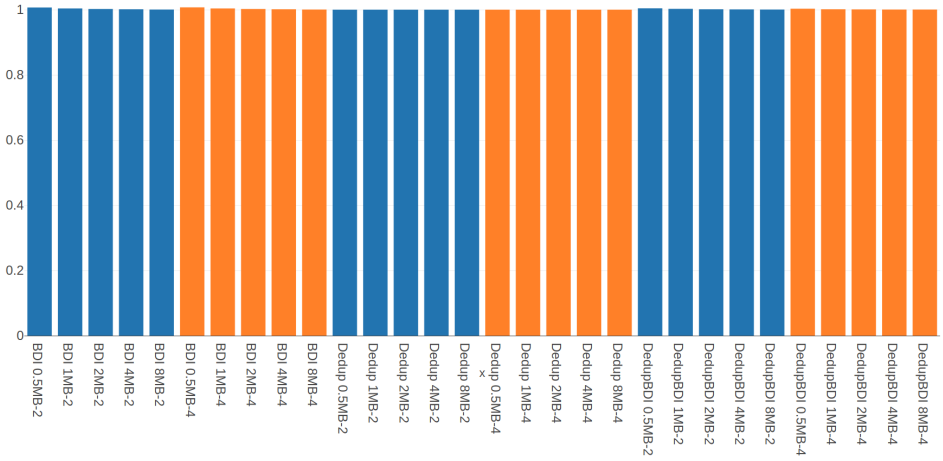


(a) canneal

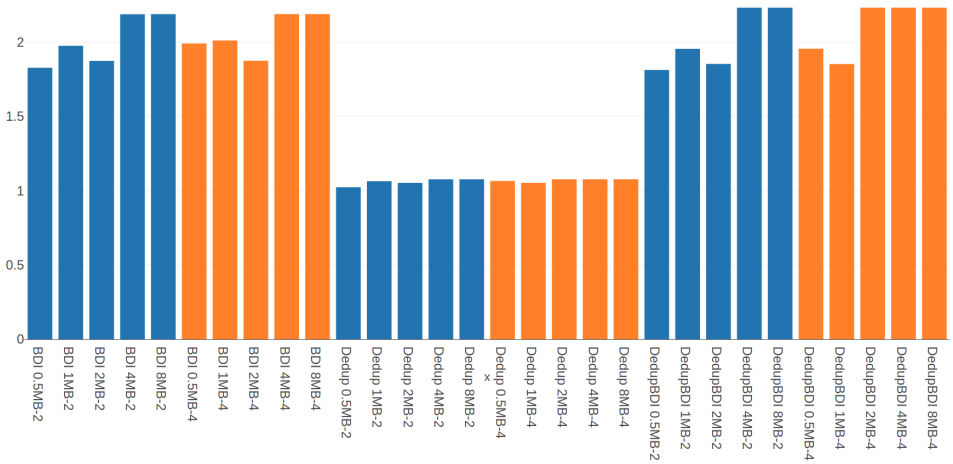


(b) lbm

Figure 4.1: Showing four benchmarks and their compression ratio using different compressed caches.

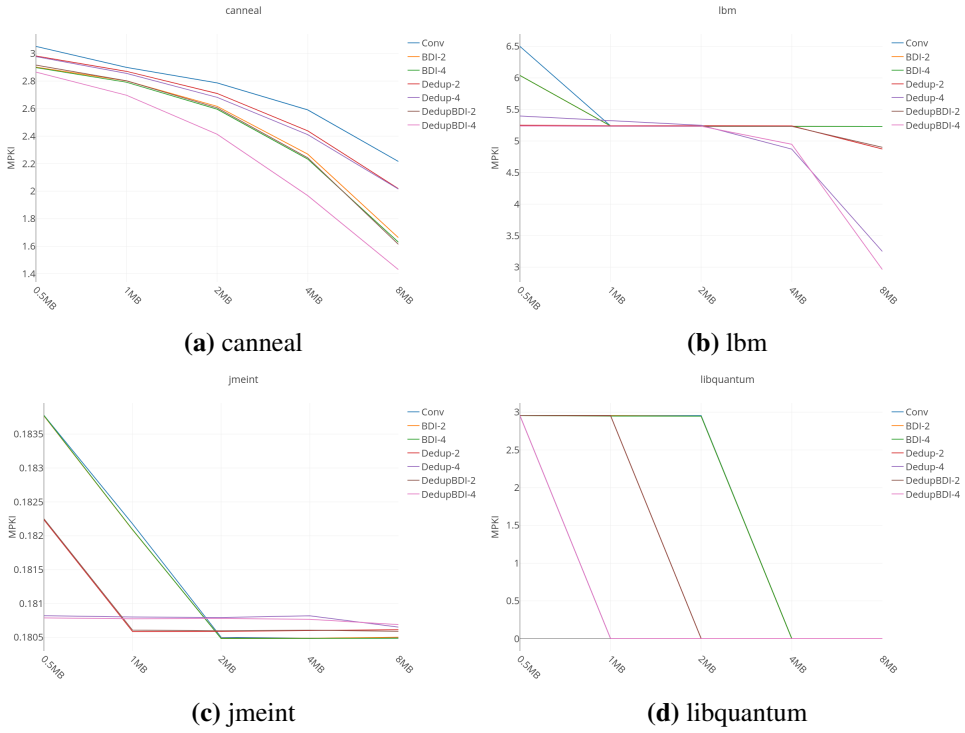


(a) jmeint



(b) libquantum

Figure 4.2: Showing four benchmarks and their compression ratio using different compressed caches.



(a) canneal **(b)** lbm

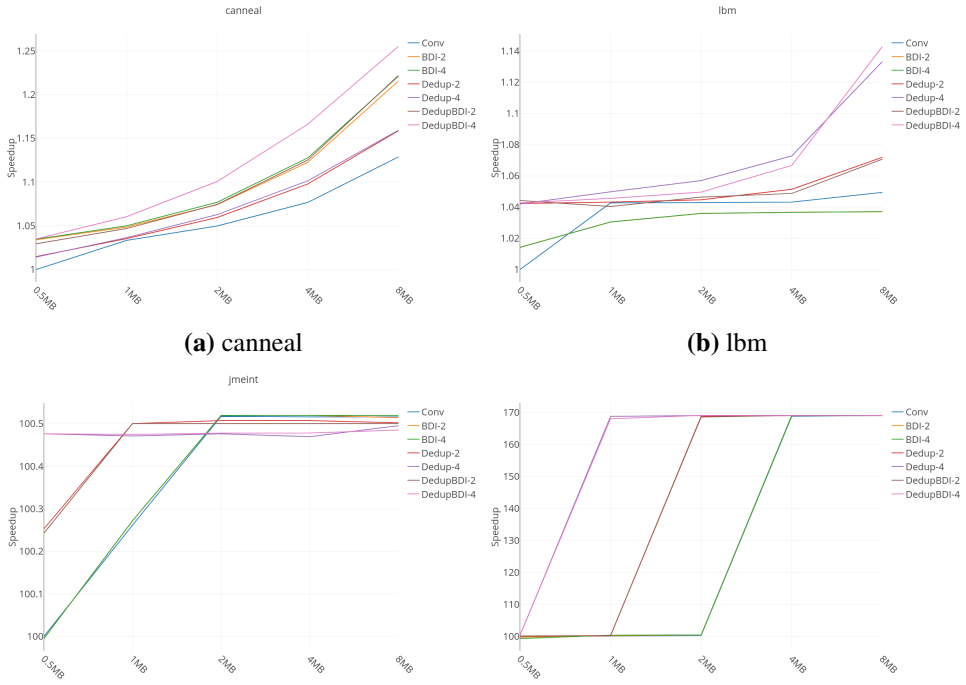
(c) jmeint **(d)** libquantum

Figure 4.3: Showing four benchmarks and their MPKI using different compressed caches.

by the BDI or the Dedup caches. The benchmarks from astar to zeusmp are only compressible by BDI. ferret and kmeans are only compressible by Dedup. The rest of the benchmarks are compressible using both techniques.

In all cases, it is shown that DedupBDI performs either better than BDI and Dedup, or at least similar to the dominant one. There are no cases in which DedupBDI compresses worse than BDI or Dedup. It can also be seen that the difference between DedupBDI and the ideal DedupBDI cache is minimal in most cases. Only in bwaves and fft benchmarks the upper bound perform a lot better than the real world implementation.

Benchmarks like bwaves, GemsFDTD, sjeng and facesim are compressible through both inter and intre-line compression but they are specifically very compressible and have a very high degree of deduplication. For example bwaves sim-



(a) canneal **(b)** lbm

(c) jmeint **(d)** libquantum

Figure 4.4: Showing four benchmarks and their speedup using different compressed caches.

ulates a spherical blast wave and facesim simulates movements of a human face giving them a high degree of symmetry. Out of those four, only bwaves suffers from a very high hash miss ratio causing it to lose opportunity for a lot of deduplication, this shown in the huge difference between DedupBDI and IDedupBDI in Figure 4.5. Other benchmarks like fluidanimate, lbm, gcc, gobmk and tonto have enough symmetry and data patterns to be compressed through both BDI and Dedup. gobmk plays a Go game, fluidanimate does fluid dynamics for animation, tonto and lbm are scientific benchmarks for simulating quantum chemistry and incompressible fluids. All those applications have a degree of similarities to merit deduplication and compression. Figure 2.2 showed the patterns existing in some of those applications. Applications like astar, calculix, canneal, freqmine, hmmer, libquantum and omnetpp are all scientific applications that do not have obvious

Figure 4.5: Showing compression ratio for all benchmarks using all types of compressed 4MB caches with #Tags = 4*#Data Blocks. The third figure is the same as the second but zoomed out.

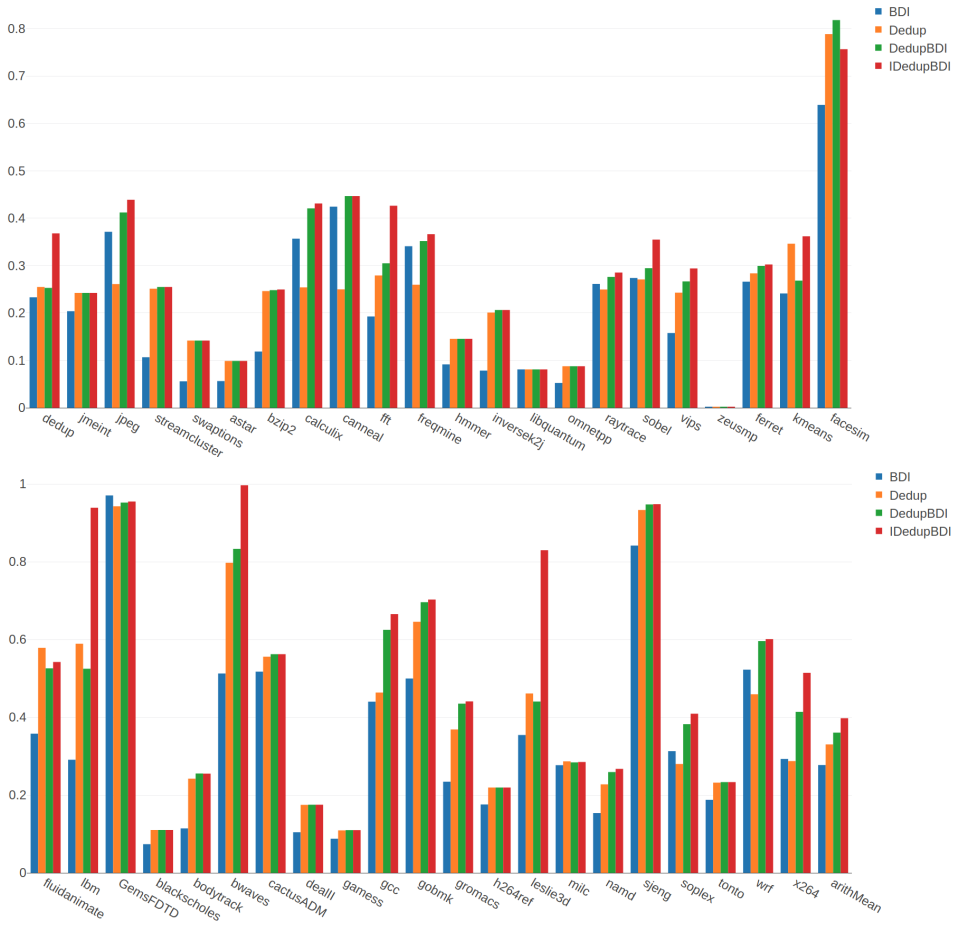


Figure 4.6: Showing tag utilization ratio for all benchmarks using all types of compressed 4MB caches with #Tags = 4*#Data Blocks.

symmetry, but they show patterns compressible through BDI. These patterns are shown in Figure 2.2.

However, the compression ratio does not tell the full story. If a cache contains 512 valid tags out of 1024, and 256 valid data lines out of 256 then it has a compression ratio of 2 but it is not being fully utilized. Normal conventional caches usually have near 100% utilization unless the working data set is very small to entirely fit inside a cache. Figures 4.6 and 4.7 shows the tag and data array utilization for all benchmarks. Most of the benchmarks can be divided into three categories:

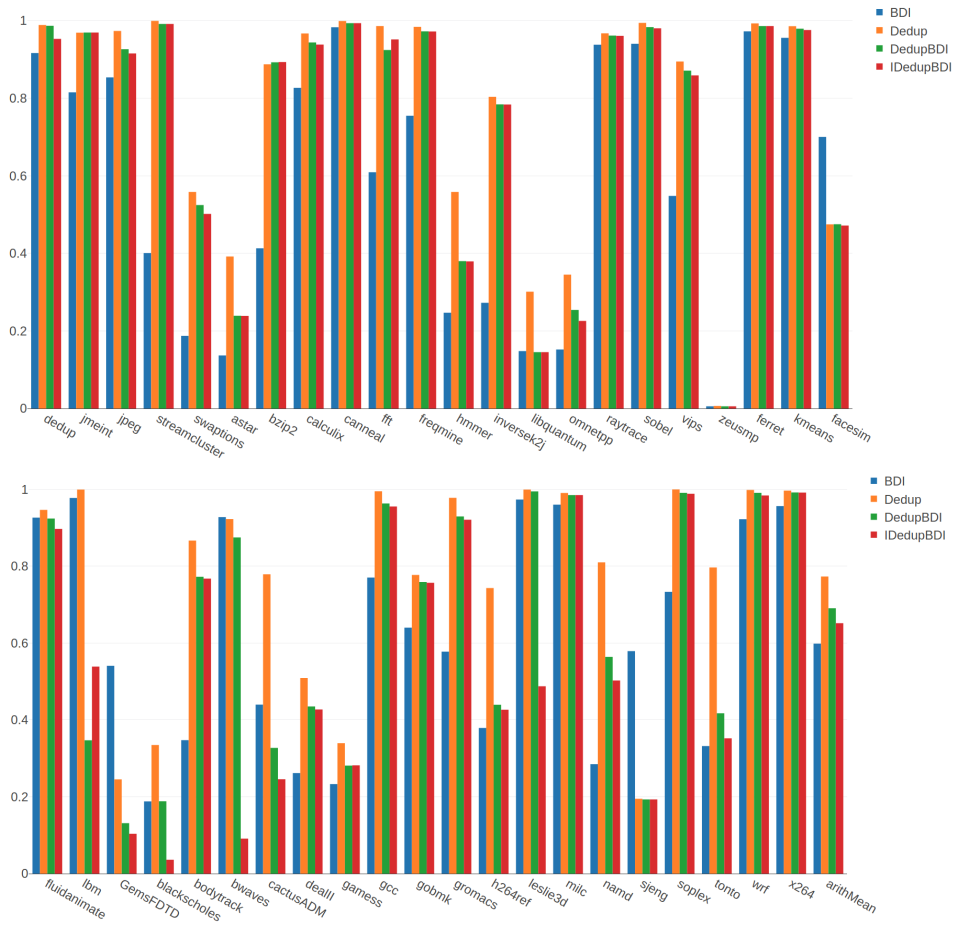


Figure 4.7: Showing data utilization ratio for all benchmarks using all types of compressed 4MB caches with #Tags = 4*#Data Blocks.

- **tag limited:** Those are benchmarks that have very high tag utilization, but lower than 100% data utilization. facesim, GemsFDTD, bwaves and sjeng fall under this category. Those benchmarks are highly compressible and thus are bound by the size of the tag array. Evictions in such benchmarks are mainly caused by the tag array replacement policy.

- **data limited:** Those benchmarks reach near 100% utilization of their data arrays before the tag array is full. This is because they have a lower degree of compressibility. For incompressible benchmarks like dedup and jmeint

Benchmark	Cache Sensitive	**Benchmark**	**Cache Sensitive**	**Benchmark**	**Cache Sensitive**	**Benchmark**	**Cache Sensitive**
fft	Big Sizes (Semi)	ferret	All (Semi)	zeusmp	-	sphinx3	Big Sizes
inversek2j	-	fluidanimate	-	gromacs	Small Sizes (Semi)	wrf	All
jmeint	-	freqmine	-	cactusADM	All (Semi)	bzip2	All
jpeg	-	raytrace	-	leslie3d	Big Sizes (Semi)	gcc	All
kmeans	-	streamcluster	-	namd	-	gobmk	Small Sizes (Semi)
sobel	-	swaptions	-	dealII	Small Sizes	hmmer	Small Sizes (Semi)
blackscholes	-	vips	Small Sizes (Semi)	soplex	All (Semi)	sjeng	-
bodytrack	-	x264	-	calculix	Small Sizes	libquantum	Mid Sizes
canneal	All (Semi)	bwaves	-	GemsFDTD	-	h264ref	All (Semi)
dedup	-	gamess	Small Sizes (Semi)	tonto	Small Sizes (Semi)	omnetpp	Small Sizes
facesim	-	milc	-	lbm	-	astar	All

Table 4.3: All benchmarks and their cache behavior

the tag array utilization sets around 25% which is equivalent to the number of data lines, behaving exactly like an uncompressed cache of the same size and ignoring the extra tags. For other benchmarks like canneal, gromacs, and leslie3d the tag utilization is in mid ranges. Evictions in those benchmarks are prominently caused by the data array replacement policy.

- **Small Working Set:** Those are benchmark with small enough data to fit entirely in the L3 cache. Those show low tag and low data utilization ratios at the same time. Examples of benchmarks that fall under this category is zeusmp, blackscholes, omnetpp, and libquantum.

Results generated from other cache sizes (ranging from 0.5MB to 8MB) are very similar, we have chosen the 4MB-4 as a representative.

It is necessary to mention that our upper bound implementation only idealizes certain aspects as discussed in Section 3.2 but not all of them. For example, victimizing and evicting data lines is still done randomly. This causes the IDedupBDI implementation to sometimes perform worse than DedupBDI, like in the case of facesim in Figure 4.5.

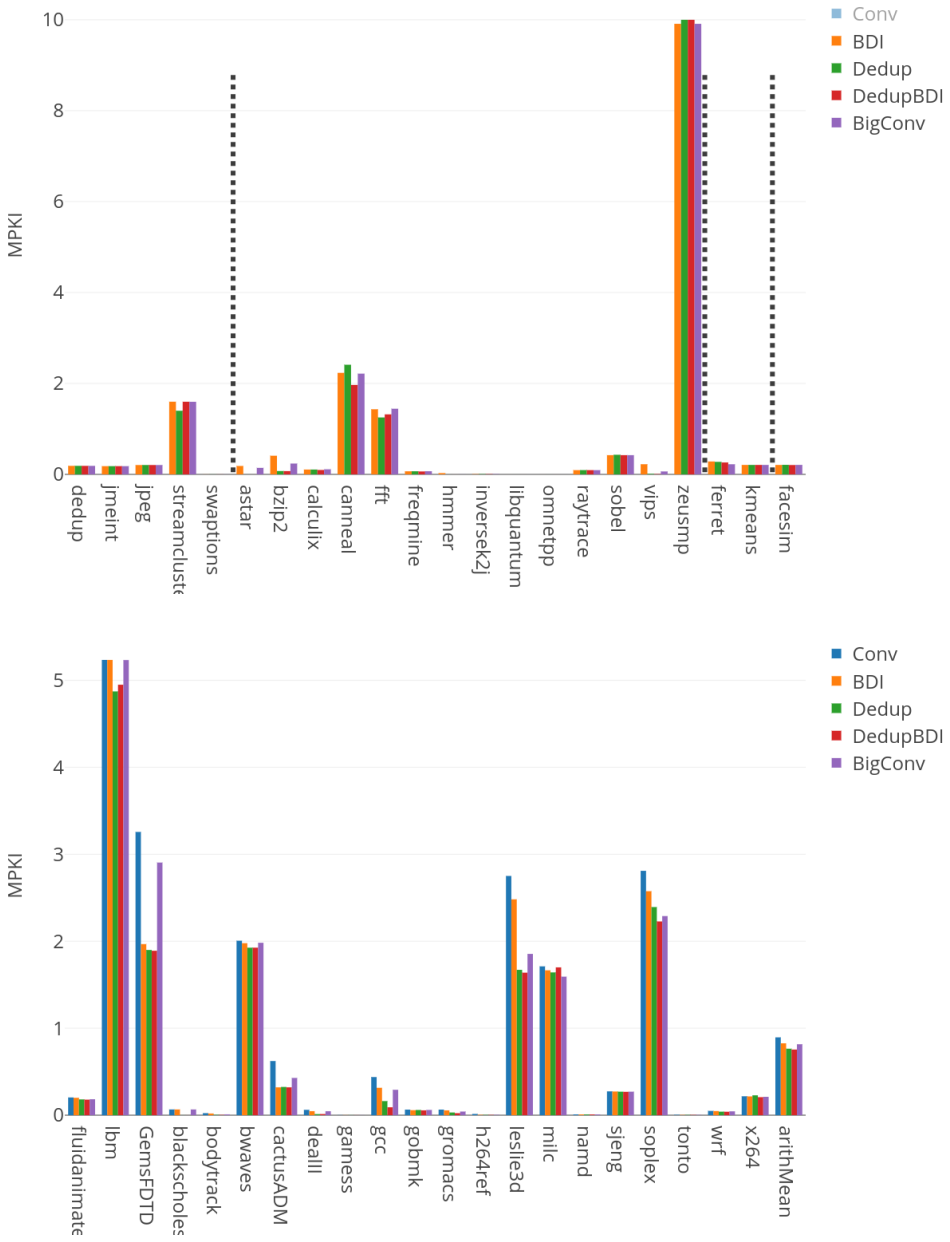


Figure 4.8: Showing MPKI for all types of compressed 4MB caches with #Tags = 4*#Data Blocks against conventional same size caches and conventional cache with double the size.

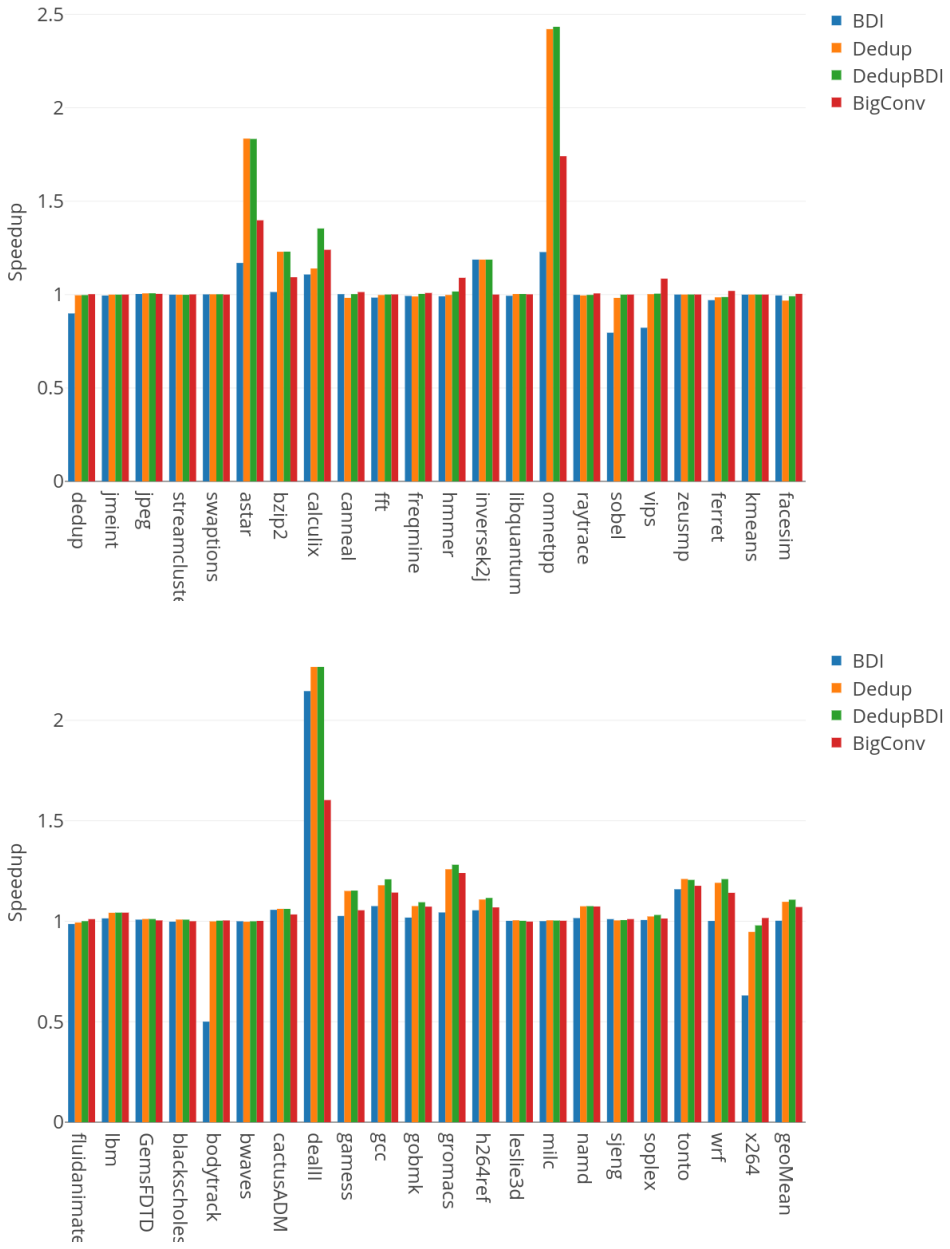


Figure 4.9: Showing Speedup for all types of compressed 0.5MB caches with #Tags = 4*#Data Blocks against conventional same size caches and conventional cache with double the size.

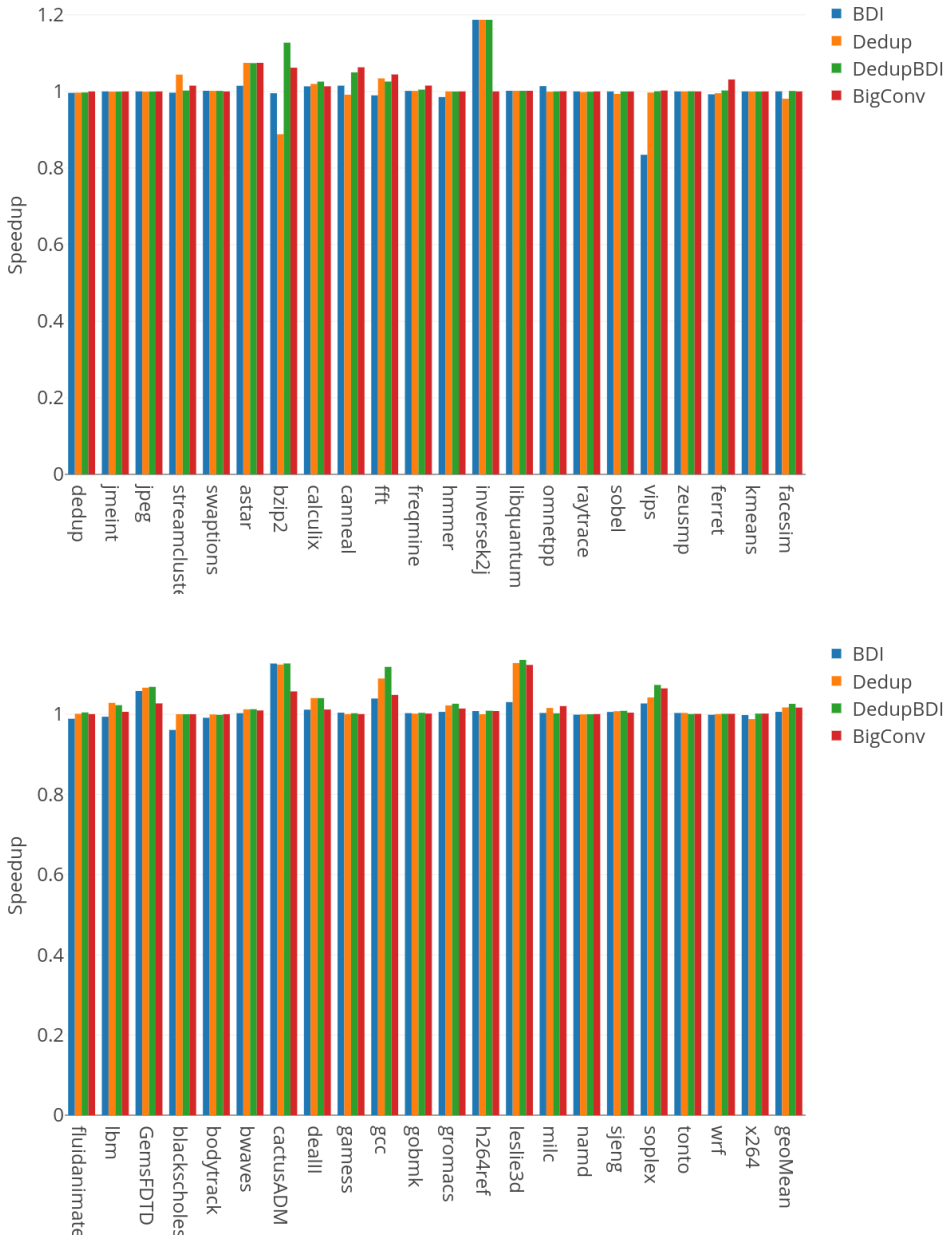


Figure 4.10: Showing Speedup for all types of compressed 4MB caches with #Tags = 4*#Data Blocks against conventional same size caches and conventional cache with double the size.

4.4 Performance

Figure 4.8 shows the MPKI for all the benchmarks when simulated with conventional and compressed 4MB LLC caches with tags four times the data lines. Benchmarks are ordered similarly to 4.5. The figure shows that compressed caches reduce MPKI in LLC. DedupBDI is either the same as Dedup or BDI or is outperforming them. In general DedupBDI performs better than a conventional cache of twice the size.

However, the decrease of MPKI doesn't always necessitate speedup. The benchmark access patterns and cache sensitivity also play a role. Table 4.3 shows the behavior of all benchmarks over a range of cache sizes. Every benchmark is classified as cache sensitive, cache insensitive, or cache sensitive at a specific cache size range. Figures 4.9 and 4.10 show the speedup for all the benchmarks when simulated with a compressed 0.5MB and 4MB LLC caches with tags four times the data lines, respectively. We have chosen those two sizes because some benchmarks are only cache sensitive with lower cache sizes while others are only cache sensitive at higher cache sizes.

For the cache insensitive benchmarks, the DedupBDI cache performs as fast as the conventional caches. It does not provide any more speedup. But it allows the data to be compressed, providing area and power savings. For the cache sensitive benchmarks, DedupBDI outperforms the conventional, Dedup, and DedupBDI caches altogether.

4.5 Tag to Data Ratio

Figure 4.11 shows the difference between two 4MB DedupBDI caches, one with tags twice the data entries (4MB-2) and the other with tags four times the data entries(4MB-4). Using the tags as four times the data entries requires compression to be highly effective. Otherwise the cache will run out of data space quickly and throttle the performance. As Figure 4.11 shows, the smaller 4MB-4 cache achieves better performance than the 4MB-2 and has better compression ratio. We've chosen caches with four times the tags to be our default compressed cache size.

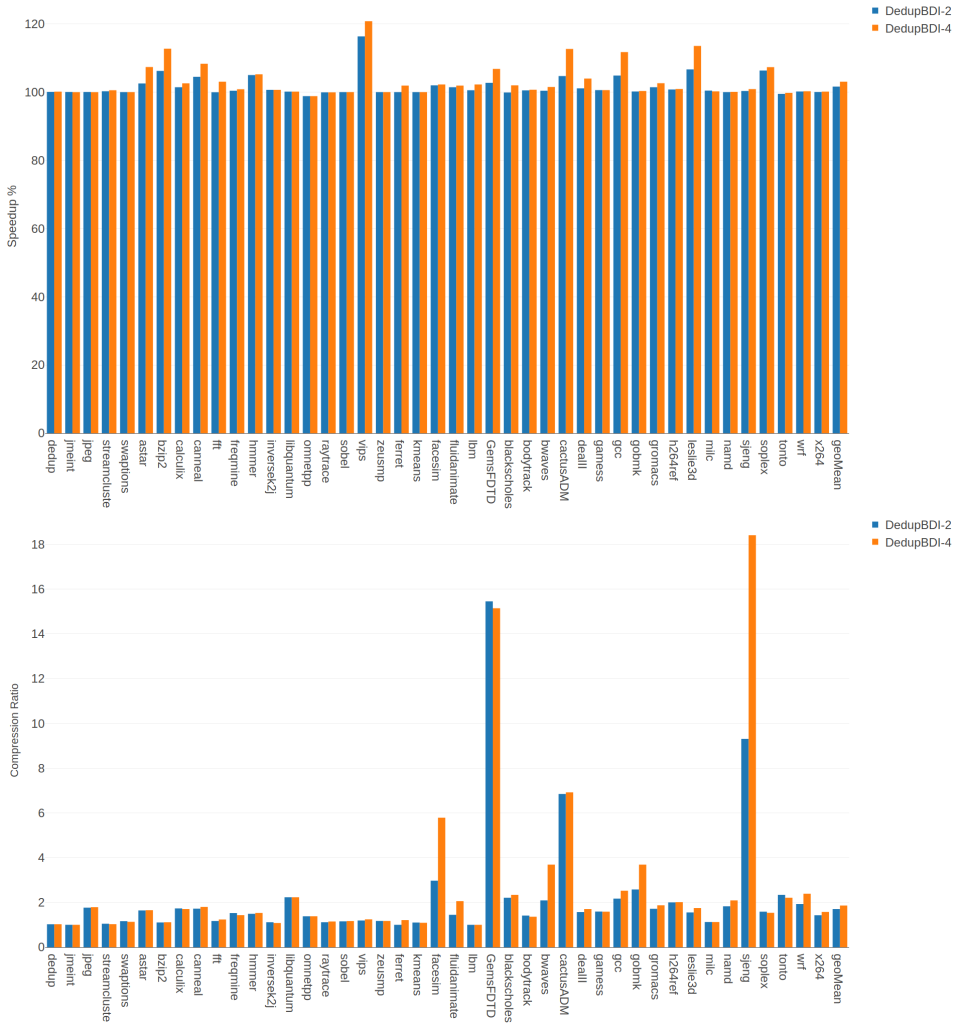


Figure 4.11: Showing performance and compression ratio for all benchmarks using DedupBDI 4MB caches with #Tags = 4*#Data Blocks and twice the data.

	Conv	BDI	Dedup	DedupBDI
Tags				
# Entries	8192	32768	32768	32768
Entry Size	49b	47b	47b	47b
Overhead	6b	15b	35b	42b
Total Size	55KB	248KB	328KB	356KB
Data				
# Entries	8192	8192	8192	8192
Entry Size	64B	64B	64B	64B
Overhead			11b	11B
FreeList			2.75KB	896B
Total Size	0.5MB	0.5MB	525.75KB	600.875KB
Hash				
# Entries			64	64
Entry Size			10b	10b
Overhead			11b	14b
Total Size			168B	192B
Total Bank Size	567KB	760KB	853.91KB	980.875KB

Table 4.4: 4MB cache size and overhead in all configurations.

4.6 Overhead Analysis

Table 4.4 shows the overheads in one bank in the four cache designs we studied, assuming an 8 banked cache, with an associativity of 16, cache size of 4MB, and LRU replacement policy. The overhead in all caches comes from four different sources:

- **Tag Overhead:**
  - **conventional:** In conventional caches, assuming LRU replacement policy, the tag overhead per entry consists of one bit for Valid, one for Dirty, and log2(associativity) bits for the LRU replacement.
  - **BDI:** In BDI there is an extra overhead of 4 bits for compression encoding and log2(associativity*#SegmentsPerLine) bits for segment pointers. Along with the normal overhead from a conventional cache. Note that in BDI the data array has half or one quarter of the associativity of the tag array.

- **Dedup:** On top of the conventional cache overhead, Dedup adds two pointers per tag entry for the next/previous tag to build a linked list, each one of those is log2(#Tags/associativity) bits. It also adds pointers to the data line associated with it. Since Dedup data arrays are direct mapped the data pointer size is log2(#Data).
- **DedupBDI:** DedupBDI adds the same overhead as the BDI cache. It also adds the same linked list pointers from the Dedup cache, but its data pointer size is different from the Dedup cache. Because the data array in DedupBDI maintains its associativity, the data pointer size is log2(#Data/associativity).

- **Data Overhead:**
  - **conventional:** A conventional cache has no overheads in its data array.
  - **BDI:** Similar to a conventional cache, A BDI cache does not have any metadata in its data array and thus does not have any overhead.
  - **Dedup:** The data lines in a Dedup cache requires a pointer to the tag, which is enough to point to the tag set so it has a size of log2(#Tags/associativity). It also requires each data line to have a deduplication counter, which we described in 2 to be 2 bits.
  - **DedupBDI:** The DedupBDI data array has the same overhead as its Dedup counterpart. Except the overhead is per segment instead of being per line. that means for each data line we have 8 Dedup data overheads.

- **Extra Data Overhead:**
  - **Dedup:** The Dedup cache maintains a free list of its free data lines. The free list is essentially a FIFO with the same number of entries as data lines, and with a width that is enough to hold a pointer to the corresponding data line (i.e. log2(#Data)).
  - **DedupBDI:** The DedupBDI cache maintains 8 different free lists. Each of them is a FIFO with the same number of sets as data sets (i.e. Data

lines/associativity), with a width that is enough to hold a pointer to the set (i.e. log2(#Data/associativity)).

- **Hash Array:** The Dedup and DedupBDI have the same hash array design. The hash array saves part of the hash as a tag while the other part is used as an index. The hash array then has the size of #Hash*(HashSize-log2(#Hash/associativity)). The Dedup and DedupBDI hash arrays then have some extra overhead:
  - **Dedup:** each hash entry has a data pointer of size log2(#Data)
  - **DedupBDI:** each hash entry has a data pointer of size log2(#Data/associativity) and segment pointer of size log2(associativity*#SegmentsPerLine) bits.

It is obvious that DedupBDI has the highest amount of overhead. But it also gains the most savings and speedups.

4.7 Area and Power

We used cacti6[16] to get power and area estimations for all the cache types and sizes we simulated. We used CACTI to only model the cache arrays, but not the compression/decompression or hashing hardware. We used the target technology as 32nm. The results are shown in 4.5. The results in the table are consistent with what we've discussed so far. DedupBDI caches have higher access latencies, higher energy and power consumption than their conventional same-size counterparts. But they are also smaller, faster, more power efficient, and allow for less MPKI and higher speedups than bigger size conventional caches. The DedupDI cache targets the right balance in the cache design tradeoff.

Cache		Access Latency (ns)	Dynamic Read Energy (nJ)	Leakage Power (mW)	Area (mm2)
CONV	0.5	1.04	0.28	353.33	12.20
	1	1.12	0.3	530.38	13.07
	2	1.29	0.33	884.79	14.77
	4	1.46	0.4	1579.64	18.13
	8	1.92	0.53	2978.40	24.88
BDI	0.5	1.13	0.29	418.39	12.54
	1	1.31	0.31	660.59	13.73
	2	1.4	0.35	1140.98	16.03
	4	1.78	0.44	2096.47	20.58
	8	2.23	0.602	3939.53	29.58
DEDUP	0.5	1.16	0.29	442.7	12.65
	1	1.40	0.31	719.35	14
	2	1.49	0.36	1273.9	16.65
	4	1.83	0.46	2355.5	21.96
	8	2.21	0.59	4583.88	32.95
DEDUPBDI	0.5	1.18	0.29	471.35	12.79
	1	1.44	0.32	778.99	14.28
	2	1.53	0.37	1403.04	17.26
	4	1.89	0.49	2628.79	23.26
	8	2.33	0.63	5160.35	35.73

Table 4.5: Area, Power, Energy, and Access Latency for all cache sizes and configurations.

Chapter 5

Related Work

In this section we describe some of the previous work in cache compression. We discuss how they are different that our picked algorithms.

5.1 Special Case Schemes

5.1.1 Zero Content Augmented Cache

The Zero Content Augmented [10] cache takes advantage of the fact that zero lines are very common in today's benchmarks. It uses a special structure called zero content cache to complement normal caches. This zero content cache only saves tags for data lines that are completely null. Special care has to taken to handle cases where a line changes and thus has to be moved from one of the caches to the other one. The choice to use an external zero content cache simplifies the design process. It allows seamless integration with any normal cache without changing its underlying structure.

5.2 Data Compression

5.2.1 Frequent Pattern Compression

Frequent Pattern Compression [2] is one of the earliest cache compression techniques proposed. It is focused on compressing data on an intra-line granularity. If

is based on the observation that some patterns are more frequent in data and can be represented by fewer bits. It takes advantage of this observation by dividing each cache line to words. Each word is represented as a 3-bit encoding prefix then a (un)compressed data form. The patterns are frequent enough to achieve good compression, but in the worst case scenario an uncompressed data work will be represented in 35 bits instead of 32. The FPC cache uses tag and data array decoupling, similar to Dedup, to allow more tags than data lines. This simple scheme provides good compression ratios and is not complex to implement.

5.2.2 SC2

The SC2 cache [3] is a statistical compression caches. It uses Huffman-coding [14] to assign variable length codes to data based on their probability of occurance. This allows saving small codes for values with high probability of occurance like zeros for example. While this cache achieves good compression ratios, its downside is that it requires a sampling phase to collect statistics on the frequency of values, and requires an extra structure for recording the frequency of values, it also requires software routines to do encoding after sampling.

5.2.3 HyComp

The HyComp cache compression [4] is a hybrid cache compression algorithm. It selects different cache compression algorithm depending on the data type. It first employs a predictor to predict the data type in each line. Then it tries to use the appropriate cache compression for this data type. It uses SC2 for integer compression, BDI for pointer compression, ZCA for zero lines, and it also introduces a new compression technique called FP-H for floating points. The combination of those different algorithms and using them with different data types allows this cache to achieve high compression ratios. However, It also increases the complexity and overhead. Because of this, we chose to only implement BDI as intra-line compression, although our implementation should be extendable to support any intra-line compression. It will only require changing the compression/decompression hardware, and the compression metadata in the cache arrays.

5.3 Dictionary Based Compression

5.3.1 CPACK

CPACK is a combination of dictionary compression and frequent pattern compression (not to be mistaken with the 5.2.1). It uses encoding for frequently observed patterns and combines this with using dictionaries or partial dictionaries for frequently encountered data values. It maintains dictionaries per data line and is able to compress multiple words in parallel at the same time.

5.3.2 Dictionary Sharing

The Dictionary Sharing cache (DISH) is a dictionary based compression cache. The authors make the observation that there is sufficient value locality in consecutive data lines to merit using dictionaries on a scale bigger than one data line. They use dictionaries on a granularity from one cache line to a super-block (four cache lines). Like other compressed caches it also increases tags over data entries, but it does so using one single tag for a whole super-block.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Cache compression allows caches to perform as if they are of bigger size. It allows the cache to store more data rather than increase its physical size. There are multiple proposals for different cache compression techniques, but they all focus on just one granularity of compression, missing the opportunity of other compression granularities.

In this work we proposed combining two different compression granularities, intra-line cache compression in the form of Base Delta Immediate (BDI) compression, and inter-line compression in the form of deduplication. The result was a compressed cache that is slightly bigger and slower than its conventional counterpart but it makes the whole system faster and allows for more performance. Our cache allows for an average speedup of 15% and a maximum speedup of 2.25X over a conventional cache, more than a bigger conventional cache of twice the size could achieve, while maintaining the area overhead around 20% and the power overhead 33%.

6.2 Future Work

Future additions or enhancements of this work include the following:

1. Investigating the effect of compression on across multiple levels of cache.

Using more than one compressed cache level can allow inter cache communication to be cheaper.

2. Researching ways to decrease the overhead of using DedupBDI. For example, it is unlikely that all data lines in the cache will be compressed to just one segment, so the overhead of metadata per segment might be unnecessary and can be reduced. Dividing the cache into smaller partitions with different compression limitations (and thus, overhead) can help reduce the overall overhead.

3. Investigating better replacement policies for the hash array. Hash arrays are critical for finding deduplication candidates. Allowing them to keep the useful hashes will positively affect the overall compression.

4. Investigating the possibility of extending this work to support dictionary compression algorithms. CPACK [7] and DISH [17] have a one line and a four line granularity, respectively. Extending this granularity to be cache wide might be useful.

Bibliography

[1] A. R. Alameldeen and D. A. Wood. Adaptive cache compression for high-performance processors. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 212–223. IEEE, 2004. → pages 4, 14

[2] A. R. Alameldeen and D. A. Wood. Frequent pattern compression: A significance-based compression scheme for l2 caches. *Dept. Comp. Scie., Univ. Wisconsin-Madison, Tech. Rep*, 1500, 2004. → pages 1, 61

[3] A. Arelakis and P. Stenstrom. Sc 2: A statistical compression cache scheme. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 145–156. IEEE, 2014. → pages 2, 62

[4] A. Arelakis, F. Dahlgren, and P. Stenstrom. Hycomp: A hybrid cache compression method for selection of data-type-specific compression methods. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 38–49. ACM, 2015. → pages 1, 2, 62

[5] S. Balakrishnan and G. S. Sohi. Exploiting value locality in physical register files. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 265. IEEE Computer Society, 2003. → page 4

[6] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011. → pages 3, 22, 42

[7] X. Chen, L. Yang, R. P. Dick, L. Shang, and H. Lekatsas. C-pack: A high-performance microprocessor cache compression algorithm. *IEEE transactions on very large scale integration (VLSI) systems*, 18(8): 1196–1208, 2010. → pages 2, 65

[8] J. Doweck, W.-F. Kao, A. K.-y. Lu, J. Mandelblat, A. Rahatekar, L. Rappoport, E. Rotem, A. Yasin, and A. Yoaz. Inside 6th-generation intel

core: New microarchitecture code-named skylake. *IEEE Micro*, 37(2): 52–62, 2017. → page 1

[9] J. Dusser, T. Piquet, and A. Seznec. Zero-content augmented caches. In *Proceedings of the 23rd international conference on Supercomputing*, pages 46–55. ACM, 2009. → page 14

[10] J. Dusser, T. Piquet, and A. Seznec. Zero-content augmented caches. In *Proceedings of the 23rd international conference on Supercomputing*, pages 46–55. ACM, 2009. → pages 1, 61

[11] M. Ekman and P. Stenstrom. A robust main-memory compression scheme. In *ACM SIGARCH Computer Architecture News*, volume 33, pages 74–85. IEEE Computer Society, 2005. → page 4

[12] C. Gonzalez, M. Floyd, E. Fluhr, P. Restle, D. Dreps, M. Sperling, R. Rao, D. Hogenmiller, C. Vezyrtis, P. Chuang, et al. The 24-core power9 processor with adaptive clocking, 25-gb/s accelerator links, and 16-gb/s pcie gen4. *IEEE Journal of Solid-State Circuits*, 2018. → page 1

[13] J. L. Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006. → pages 3, 5, 23, 42

[14] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952. → page 62

[15] M. M. Islam and P. Stenstrom. Characterization and exploitation of narrow-width loads: the narrow-width cache approach. In *Proceedings of the 2010 international conference on Compilers, architectures and synthesis for embedded systems*, pages 227–236. ACM, 2010. → page 4

[16] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. Cacti 6.0: A tool to model large caches. *HP laboratories*, pages 22–31, 2009. → pages 1, 59

[17] B. Panda and A. Seznec. Dictionary sharing: An efficient cache compression scheme for compressed caches. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–12. IEEE, 2016. → pages 2, 65

[18] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. Base-delta-immediate compression: practical data compression for on-chip caches. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 377–388. ACM, 2012. → pages 2, 6, 8, 9, 10

[19] D. Sanchez and C. Kozyrakis. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. In *ACM SIGARCH Computer architecture news*, volume 41, pages 475–486. ACM, 2013. → pages 5, 23, 36, 41

[20] Y. Sazeides and J. E. Smith. The predictability of data values. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 248–258. IEEE Computer Society, 1997. → page 4

[21] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark. Branch prediction, instruction-window size, and cache size: Performance trade-offs and simulation techniques. *IEEE Transactions on Computers*, 48(11): 1260–1281, 1999. → page 1

[22] W. Sun, Y. Lu, F. Wu, and S. Li. Dhtc: an effective dxtc-based hdr texture compression scheme. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 85–94. Eurographics Association, 2008. → page 5

[23] Y. Tian, S. M. Khan, D. A. Jiménez, and G. H. Loh. Last-level cache deduplication. In *Proceedings of the 28th ACM international conference on Supercomputing*, pages 53–62. ACM, 2014. → pages 1, 2, 3, 5, 7, 14, 17

[24] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis. The case for compressed caching in virtual memory systems. In *USENIX Annual Technical Conference, General Track*, pages 101–116, 1999. → pages 4, 5

[25] J. Yang, Y. Zhang, and R. Gupta. Frequent value compression in data caches. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 258–265. ACM, 2000. → page 4

[26] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran. Axbench: A multiplatform benchmark suite for approximate computing. *IEEE Design & Test*, 34(2):60–68, 2017. → pages 3, 23, 42