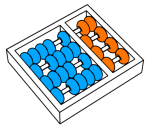
Universidade Estadual de Campinas
Instituto de Computação



Daniel Rodrigues Carvalho

Extensions To The Base-Delta-Immediate Compression

Extensões Para A Compressão Base-Delta-Imediato

CAMPINAS
2017

Daniel Rodrigues Carvalho

Extensions To The Base-Delta-Immediate Compression

Extensões Para A Compressão Base-Delta-Imediato

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Thesis presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Computer Science.

Supervisor/Orientador: Prof. Dr. Rodolfo Jardim de Azevedo

Este exemplar corresponde à versão final da Dissertação defendida por Daniel Rodrigues Carvalho e orientada pelo Prof. Dr. Rodolfo Jardim de Azevedo.

CAMPINAS
2017

Agência(s) de fomento e nº(s) de processo(s): CAPES, 1564395

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

C253e

Carvalho, Daniel Rodrigues, 1992-
Extensions to the Base-Delta-Immediate compression / Daniel Rodrigues Carvalho. – Campinas, SP : [s.n.], 2017.

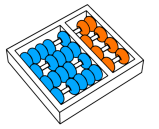
Orientador: Rodolfo Jardim de Azevedo.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Memoria cache. 2. Compressão de dados (Computação). 3. Arquitetura de computador. I. Azevedo, Rodolfo Jardim de,1974-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Extensões para a compressão Base-Delta-Imediato
Palavras-chave em inglês:
Cache memory
Data compression (Computer science)
Computer architecture
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Rodolfo Jardim de Azevedo [Orientador]
Guido Costa Souza de Araujo
Philippe Olivier Alexandre Navaux
Data de defesa: 20-07-2017
Programa de Pós-Graduação: Ciência da Computação

Universidade Estadual de Campinas
Instituto de Computação



Daniel Rodrigues Carvalho

Extensions To The Base-Delta-Immediate Compression

Extensões Para A Compressão Base-Delta-Imediato

Banca Examinadora:

- Prof. Dr. Rodolfo Jardim de Azevedo
  Universidade Estadual de Campinas
- Prof. Dr. Philippe Olivier Alexandre Navaux
  Universidade Federal do Rio Grande do Sul
- Prof. Dr. Guido Costa Souza de Araujo
  Universidade Estadual de Campinas

A ata da defesa com as respectivas assinaturas dos membros da banca encontra-se no processo de vida acadêmica do aluno.

Campinas, 20 de julho de 2017

Acknowledgements

Many thanks to Carlos Petry and Emilio Francesquini for their feedback on the writing of this thesis.

Resumo

Memórias *cache* há muito têm sido utilizadas para reduzir os problemas decorrentes da discrepância de desempenho entre a memória e o processador: muitos níveis de caches on-chip reduzem a latência média de memória ao custo de área e energia extra no *die*. Para diminuir o dispêndio desses componentes extras, técnicas de compressão de cache são usadas para armazenar dados comprimidos e permitir um aumento de capacidade de cache. Este projeto apresenta extensões para a *Compressão Base-Delta-Imediato*, várias modificações da técnica original que minimizam a quantidade de bits de preenchimento numa compressão através da flexibilização dos tamanhos de delta permitidos para cada base e do aumento do número de bases. As extensões foram testadas utilizando ZSim, avaliadas contra métodos estado da arte, e os resultados de desempenho foram comparados e avaliados para determinar a validade de utilização das técnicas propostas. Foi constatado um aumento do fator de compressão médio de 1.37x para 1.58x com um aumento de energia tão baixo quanto 27%.

Abstract

Cache memories have long been used to reduce problems deriving from the memory-processor performance discrepancy: many levels of on-chip cache reduce the average memory latency at the cost of extra die area and power. To decrease the outlay of these extra components, cache compression techniques are used to store compressed data and allow a cache capacity boost. This project introduces extensions to the *Base-Delta-Immediate Compression*, many modifications of the original technique that minimize the quantity of padding bits by relaxing the allowed delta sizes for each base and increasing number of bases. The extensions were tested using ZSim, evaluated against state-of-the-art methods, and the performance results were compared and evaluated to determine the validity of the proposed techniques. We verified an improvement of the original BDI compression factor from 1.37x to 1.58x at a energy increase as low as 27%.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Over the years, electronic hardware has become faster and more efficient, but devices do not improve at equal rates. Off-chip memories, although increasing in capacity, present lower bandwidth and latency improvements than what is requested by current microprocessors [21] [43]. Besides, these memories require huge amounts of energy when compared to local accesses (*i.e.*, register access) [38]. To try to overcome these problems, on-chip memories, lower level memories with small latency and energy cost, but small storage capacity and high cost per bit, have been created (caches) [31]. This dichotomy has led designers to try to increase the effective cache size by compressing and compacting data blocks before inserting them on caches, a technique called cache compression.

The main goal of cache compression methods is to virtually increase the cache size without the disadvantages of doing so, that is, improve power consumption and cache capacity with as low latency, area, and metadata overhead as possible. These overheads are inherent to such methods due to the extra hardware and wiring required, and the extra stages every access must go through: decompression on lookups and compression on writes. Besides, extra metadata is also necessary to inform data compression state or location. These techniques should also be lossless so as to maintain correct processor behavior (although some applications may tolerate data modification).

This work presents and describes several cache compression techniques, as well as their strengths and weaknesses. *Decoupled Compressed Caches* (*DCC*) [48], for example, allow compressed blocks with variable size, but this size freedom comes with extra costs as there must exist extra metadata to find out a block's location. This also increases latency due to the additional indirection level.

Skewed Compressed Caches (*SCC*) [46], on the other hand, assume 4 fixed compression factors and use the block's address to determine its way, set index and byte offset. By removing the need to access extra metadata they allow lower access latency, reduce die area and improve energy efficiency. SCC, however, are highly dependent on compression locality, and thus may have a high rate of internal fragmentation in case a workload's neighboring blocks do not compress similarly.

There are also methods that select the most suitable compression decision, as described in Alameldeen *et al.* [1], which enables or disables compression when advisable based on the analysis of the cache misses. Hybrid approaches like HyComp [4] also exist, which selects the most relevant technique from a pool of methods to compress a given block.

Other approaches are based on the fact that the data stored in the cache is usually redundant, as neighbor blocks commonly have identical values, and when that is not the case they are similar. The Doppelgänger method [36] approximates these close values, sacrificing data correctness to allow higher capacity. Another possibility is to store the difference between neighbor values, as in the Base-Delta-Immediate (*BDI*) Compression technique [41]. It uses a base value as a cache line guide and parses and translates the line as a series of deltas relative to it. This allows fast decompression latency, as the decompressor is a simple sequence of adders. However, this heavily relies on data similarity, as if data needs more than 2 bases, or the delta values are too far apart, compression becomes inviable or inefficient.

To try to overcome BDI's high dependence on data similarity, we propose extensions and modifications: Base-Delta-Immediate Relative to Bases (*BDI-RB*), Base-Delta-Immediate Relative to Deltas (*BDI-RD*), Base-Delta-Immediate with multiple bases (*MBDI*). These allow more flexibility on compressed data creation by either granting each base a delta size, allowing each delta entry to have its own delta size, or increasing the number of bases. By doing so less space is used with padding bits as the deltas become more independent of each other.

With BDI the delta size of a compressed line's deltas is fixed to all deltas. BDI-RB allows using a delta size for each base entry, therefore if the bases need different delta sizes the one with smaller delta size does not need to add padding to fit the bigger one. We propose three encoding schemes to this technique: *1)* one to maintain compatibility with the original scheme; *2)* one to separate base from delta size, reducing bit usage when delta size is not needed, although slighly increasing when that is not true; *3)* and one with variable opcode sizes based on frequency of the opcodes.

BDI-RD, on the other hand, provides more flexibility by giving each delta entry its corresponding delta size. This wastes more space with metadata, as although there will be no need to represent the delta size of the compressed line on the opcode and thus 1 bit can be saved, it will be necessary to add a delta size for each delta, which can represent 16 bits for a 64-byte cache line. Nonetheless, this completely removes the need for padding, and thus increases the compression factor by 3% when compared to the original approach.

In the original approach, the number of bases is limited to an explicit base and an implicit zero-base. Although having more bases would allow more lines to be compressed, having their number fixed would cause the compression ratio to drop. MBDI focuses on reducing the number of uncompressed lines by providing a field informing the number of bases in the compressed line. Therefore, besides allowing more cache lines to be compressed when two bases are not enough, whenever a single null base is sufficient to represent the line, the space used by the extra mandatory base is saved.

All proposed techniques but two variations of BDI-RB improved compression factor when compared to the original BDI. The BDI-RB variants do not differ much from each other, and had the lowest improvements, as they do not tackle the main cause of padding generation nor allow more uncompressed data to be compressed. The best compression ratio results were from an hybrid of the BDI-RD and MBDI methods, the *MBDI-RD*, with a ratio 15% better than the original's. This method generates a compression factor similar to CPack's, but with a smaller decompression latency (one cycle).

Although both 32 and 64-bytes per block cache are used in this work, none of the techniques is restricted to these sizes, so this can be changed by applying the corresponding modifications to the equations provided. We will use the terms *compression factor* and *compression ratio* interchangeably, which is given by the the original block size divided by the compressed block size.

The main contributions of this work are:

- Creation of techniques to remove padding in the Base-Delta-Immediate compression to improve its compression ratio;
- Allow using a different number of bases on BDI with marginal metadata overhead;

Chapter 2 provides a background on memories, detailing their history and manner of operation. On Chapter 3 some cache compression techniques are presented, described, and explained. Chapter 4 presents the proposed extensions and explains their modus operandi. The introduced modifications are then scrutinized and the results regarding compression ratio, area and energy usage are disclosed on Chapter 5. Finally, a conclusion is made on Chapter 6.

Chapter 2

Background

Memories are devices capable of storing information temporarily or permanently, and they are used to access data faster than using secondary (*i.e.*, HDD, SDD, FDD) or offline storage (*i.e.*, CD, USB drive). Typical memory mechanisms include Static Random Access Memory (SRAM), Dynamic Random Access Memory (DRAM), Synchronous DRAM (SDRAM), Flash Memories, and Electrically Erasable Programmable Read-Only Memory (EEPROM).

These devices are divided into volatile, that is, their contents are erased when power is interrupted (*e.g.*, random access memory), and non-volatile, which retains their contents even when the power is turned off (*e.g.*, read-only memory, flash memory, optical discs and most of the magnetic storage devices).

2.1 Off-chip memories

Computers today use variants of DRAMs as off-chip memories. They have lower access latency than disk drives, but are considerably slower than processors. Each DRAM bit is composed of a transistor and a capacitor, and as transistors always leak energy, even when not doing useful work, the capacitors discharge, and thus data must be refreshed periodically in order not to loose information.

Each DRAM chip is divided into rows and columns, therefore every access must first send a Row Address Strobe signal (RAS) to determine the row to be accessed, and then a Column Address Strobe signal (CAS) to select the desired bits [13]. Memory can also be divided in multiple banks instead of a monolithic block. Having multiple banks allows concurrent accesses, which may improve bandwidth in interleaved sequential accesses. Reading the DRAM is a destructive process, so a read row must always be written back.

2.1.1 Virtual Memory

Due to the various data storage technologies and distinct design objectives, systems usually have different amounts of memory, and thus allowing programs to map addresses to a unique physical location would require code modification for each machine. Besides, on multitasking systems multiple programs map to the same physical address, so conflicts would arise frequently.

Because of that, compilers and operating systems are designed to use a virtualization of the address space, that is, on loads and stores the addresses used are purely virtual, and must be translated to physical addresses before being accessed. Therefore, the physical addresses used can be chosen in the most suitable location by the operating system.

Virtual memory is divided into fixed-sized blocks (*pages*), variable-sized blocks (*segments*), or a mixed approach (*paged segments*), which are placed in memory when needed. This is done because programs may not fit entirely in memory. Besides, since a program does not need to be completely inside memory, the load time and the amount of I/O operations are reduced, and the memory may be used by many processes simultaneously [51].

As all virtual addresses must be translated to physical addresses before data is accessed, the access latency becomes high and impractical. However, by exploiting the principle of locality the translation step can be done more efficiently. This is done by keeping a special cache to keep track of virtual addresses and their physical counterparts, the *Translation Lookaside Buffer* (*TLB*). The TLB, like a conventional cache, is divided in tags and data, where the tag portion holds the virtual addresses, and the data portion is responsible for storing their corresponding physical page addresses and metadata (access permissions, storage type, valid bits and dirty bits), allowing faster translation of frequently/recently accessed addresses.

2.1.2 Errors and Error Handling

Bits in the memory cells are subject to spontaneous flipping due to the interference of other electrical or magnetic devices inside a computer system. Even when accessing data within memory, the internal leakage of electrical charges may make the cells cross talk and alter their contents, an effect known as RowHammer [27]. Besides, in faulty chips, bits can be stuck to a particular value, so requests to change their state will not be fulfilled. These behaviors may lead to corruption of data, crashes, and create system vulnerabilities [49, 26].

A partial solution to these problems is to add a bit to indicate if the number of 1's in a binary string is odd. Typically this parity bit is added for every byte in memory and is set to 1 if the number of 1's in the byte is odd, and 0 otherwise. For example, the parity bit for bytes 01100001, 00010000, and 10001000 are, respectively, 1, 1, and 0. As the parity bit does not tell where the error is, it is an error detection, but not error correction solution.

This parity bit, however, is not guaranteed to detect errors if the number of flipped bits is even. For example, suppose byte 10010000 has been stored in memory. Its parity bit is 0 as there is an even number of 1's. If bits 2 and 3 become faulty and get stuck to 1, that is, the byte becomes 11110000, the parity bit will still be 0, as there will are 4 1's, so it will fail to catch the error.

In order to deal with multiple bit errors, Error-Correction Code (*ECC*) memory modules have been researched [42]. These additional chips are added to the memory to monitor, prevent, detect and fix memory bit errors by generating a multi-bit code for each data word provided, instead of a single bit as in the parity bit error detection. These

modules make the memory system fault tolerant [11], that is, they reduce the amount of crashes and data corruption, at the cost of extra hardware, when compared to non-ECC memories, and reduce memory performance due to the error checking steps.

2.2 Caches

As mentioned previously, off-chip memories are a few orders of magnitude slower than the processor. Therefore, if every memory access goes through main memory, the average number of cycles per instruction becomes higher than the memory latency. Luckily, most of the time programs exhibit temporal and spatial locality [21], so if the current working memory data is stored in small buffers with lower access latency this problem can be significantly reduced. These buffers are called *caches*, and are typically made of SRAMs.

This idea can be further extended to put middle ground between cache and main memory. Usually caches are organized in hierarchies of two or three levels, but some processors, such as Intel Haswell and IBM Power8, contain a 4th level cache [19] [32]. The first level cache is the fastest and closest to the processor, but is also the smallest cache, with sizes varying from 8KB to 64KB. The second level is bigger (commonly 256KB), but slower. The third level has a notably higher capacity (a few megabytes), but is also slower than the previous levels. This pattern is repeated on the fourth level cache.

Usually, due to the locality found in programs, processors have two first level caches: one for program instructions, called instruction cache, and one for program data, the data cache. This is helpful because as instructions are always being fetched to be executed, data loads/stores are likely to be requested simultaneously, which would have to generate a stall if only one first level cache was being used. Besides, by using two caches, the different access patterns for program data and instructions can be exploited and fewer unnecessary conflicts will be generated.

Each cache entry usually consists of a *tag*, the *data block* and *flags* (Figure 2.1). The tag is part of the memory address to which the data block corresponds. The data block, also called *cache line*, stores either program data or instructions. The flags region contains a *valid bit* to determine whether or not the entry encloses valid data, a *dirty bit* to inform if the data has been changed since it was read from memory, among others.

Figure 2.1: A cache entry consists of a tag, a data block and flags.

These entries are divided and placed into two logical blocks: the tag and flags portions are placed in the tag array and the data block in the data array. The arrays are organized in *sets* and each set contains n blocks (or *ways*). Figure 2.2 presents an example of the logical organization of a 4-way cache with 8 sets.

Figure 2.2: Logical organization of a 4-way cache. A cache consists of sets, each of which contains 4 sequential blocks.

2.2.1 Block Placement

When placing a block in the cache, the location at which it will be inserted can be unique, that is, a given block is always mapped to the same spot (*direct mapped cache*); undefined, that is, a given block can be placed anywhere in the cache (*fully associative cache*); or a something between these two, *i.e.*, a block can be placed anywhere within a set (*n-way set associative cache*). Therefore, a fully associative cache can also be thought of as a n-way set associative cache where there is only one set and n is the number of blocks in the cache. Analogously, the direct mapped cache can be thought of as a 1-way set associative cache. Figure 2.3 presents an example of block placement for these policies.

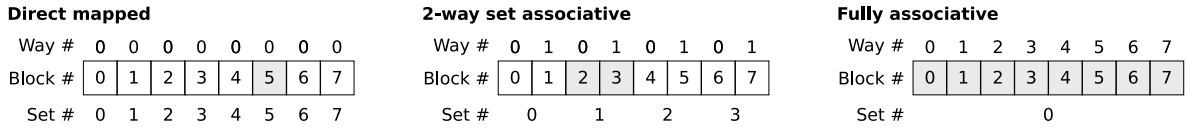


Figure 2.3: Placement of block 13 on direct mapped, 2-way set associative and fully associative caches. The caches are limited to 8 blocks. On a direct mapped cache the memory block can only be placed in block 5 (13 modulo 8). With a 2-way set associative cache the memory block is mapped to set 1 (13 modulo 4) and can be placed in either way 0 or 1. It can be placed anywhere when using a fully associative cache.

In a conventional cache the amount of ways in a set determines its *associativity*. However, Sanchez *et al.* [44] have proposed a cache design that modifies the number of replacement candidates instead of the number of ways in order to vary the associativity. That is, instead of determining associativity by the amount of locations that a block can be placed, it is determined through the number of candidate blocks to be replaced on an eviction.

For direct mapped caches, finding a block is straightforward, as there is only one location at which the block can be. For n-way set associative caches and fully associative caches all tags in a set must be parsed to determine the way at which the block can be. This makes caches with high degree of associativity expensive, and thus direct mapped, 16-way and 32-way set associative caches are more commonly used on modern architectures.

2.2.2 Data Access

Whenever a memory access is requested, the place where the entry should be, if it exists, is scanned to determine if the block is present. If the tag at that position matches the given tag then the corresponding data block lies in the data array (*cache hit*). Otherwise, the data does not reside in the cache and it must be fetched from the higher level caches (*cache miss*). If the entry cannot be found in the last level cache, the request is issued to main memory.

During a memory access, the tag and data arrays can either be parsed serially or in parallel. In the first approach the tag array is first scanned to find the location of the requested data and then the corresponding data block is fetched. By using parallel accesses, the access latency is reduced, as both the tag and data array are parsed simultaneously, but this requires more energy because all the ways of a set must be checked from the data array before being selected at the end of the request. Besides, if there is a cache miss the data will not be present and the energy used on the data path will be wasted.

2.2.3 Evictions

To *evict* a block means to remove a block from cache in order to create space for new data. Deciding which entry should be removed on an eviction is a compelling task, as it directly affects processor performance and depends solely on the *replacement policy* being used. Some common replacement policies include *Least Recently Used (LRU)*, *Most Recently Used (MRU)*, Not Most Recently Used (NMRU), *Least Frequently Used (LFU)*, and *Random*.

The LRU replacement policy has counters to keep track of entries' ages, so that the last recently accessed entry is the next to be removed. Its counterpart, the Most Recently Used replacement policy, on the other hand, evicts the most recently accessed entry. NMRU has a different assumption when compared to MRU, as it speculates that a recently accessed item has a higher likelihood of being accessed soon. Deciding when to use NMRU, MRU or LRU depends directly on the dataset, *i.e.*, when it is known that an entry that was just accessed will not be accessed for a long time, MRU would be a better fit.

Both the NMRU, MRU and LRU are expensive due to the amount of extra bits needed for the counters. In order to reduce this extra cost some implementations only update the counters every few accesses and thus the counters require less bits. Other approaches, called Pseudo-LRU (PLRU), keep a bit per entry to indirectly indicate the age. The bit-PLRU, for example, sets the bit to 1 whenever its corresponding entry is accessed, so on evictions the first cache line with status bit set to 0 is evicted. When all entries are set to 1 the process restarts using 0 as the verification bit instead. The tree-PLRU is another example of PRLU, but it uses a tree to keep track of the least recently used element, and the status bit indicates the direction to go (left or right) to find it. Whenever an element is accessed the tree is traversed and updated accordingly to point to it.

The LFU replacement policy has counters to keep track of how often an entry is accessed, so the entry with the lowest number of accesses is evicted. There are also hybrid approaches that associate different policies in order to improve efficiency where one of the policies would fail, such as the Adaptive Replacement Policy [35], which combines both

a LRU and a LFU list and dynamically decides which policy best suits the payload. The Random replacement policy is used due to its simplicity, as it randomly selects a candidate to discard, and thus does not require much hardware.

2.2.4 Writes

What happens on writes is important to determine the system's speed and complexity, and three design decisions must be accounted for: where the data should be written, what to do while writing, and what to do with the data when write misses occur.

Write Policy

Deciding on which cache level the data should be written, also called *write policy*, usually takes one of two forms: *write-through* and *write-back* [24]. A write-through cache must keep the information synchronized with the lower memory level all the time. Therefore, whenever a write happens, the data is written to the block in the cache and the corresponding block on the lower memory level. This design decision focuses on maintaining coherence between levels, and minimizing data loss in case of unexpected disruptions, at the expense of system speed, as all writes will be done at the speed of the next level. It is also easy to implement, as the cache is always up-to-date, so evictions do not require extra steps.

The write-back policy, on the other hand, only updates the lower level memory when an eviction happens and the block has been modified. It does so by keeping a *dirty bit* for every block. If the block is *clean* (not modified) it is simply removed from the cache, otherwise it is written to the lower level before being removed. This allows a reduction of overall memory traffic because multiple writes to the same block can happen before it is evicted, and only one write to the lower level will occur.

Processing

The write process can take some time, and deciding whether to try to keep processing or not can cause great impact on the system. If the processor stalls on every write (*write stall*) instructions following the miss that do not depend on the reference are penalized [28], so processing time is wasted, but there is no extra hardware complexity. However, if a buffer (*write buffer*) is used to keep track of on-going writes and the processor keeps executing until an operation directly depends on the data contained in the write buffer, the time efficiency is optimized at the cost of extra hardware to keep track of the dependencies and the buffer.

Write Misses

Regarding the data on write misses, two options can be taken: *write allocate*, and *no-write allocate* [21]. Write-allocate assumes that if the data was written, due to program locality, it is probably going to be accessed again soon, so the block should be cached in order to avoid future read and write misses. No-write allocate, on the other hand, does not change the cache, so the block is modified only in the lower memory level, and is generally used

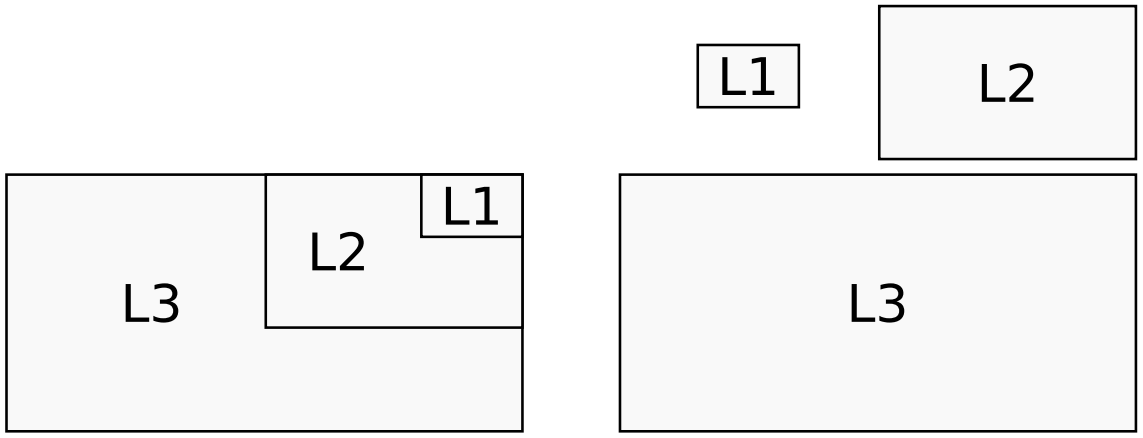
with write-through policies, as there will be no advantages to keep data in the cache when write operations happen sequentially.

2.2.5 Data Inclusion

The data contained inside the caches can relate in two manners: a block in a higher level cache is always present in the lower level caches (*inclusive cache*), or is only present in one of the caches levels (*exclusive cache*).

Baer *et al.* [7] state that a multilevel cache hierarchy has the inclusion property if the contents of a lower level cache is a superset of the contents of all its children caches. Therefore, all data contained in cache L1 is also present in L2, all data in L2 is also present in L3, and so forth (Figure 2.4a). This eases searches incoming from other processors or external devices, as they only need to search for the data in the last level cache. However, on evictions, data in all cache levels must be evicted in order to enforce the inclusion property. Also, due to the storage of duplicated data, this scheme is highly dispendious regarding cache capacity.

Exclusive caches have different address spaces (Figure 2.4b), so their use provides higher effective storage of data in the caches, but it requires more work on searches, as all cache levels must be scanned. Besides, when executing an eviction, data is swapped between cache levels, instead of copied from the lower level as in the inclusive approach, which is costlier energy-wise. This approach is generally used when the cache sizes have low discrepancy and thus storage efficiency is essential, or when there are many large inner levels in a memory hierarchy and thus cross-level replication lowers performance [17].



(a) Inclusive caches. Higher level caches are subsets of the lower level caches.

(b) Exclusive caches. All caches have different address spaces.

Figure 2.4: Data in inclusive and exclusive caches.

2.2.6 Data Consistency

When designing a many-core system, special care must be taken regarding the cache hierarchy. Generally the cache system for such architectures relies on a combination of two

approaches: keeping a private cache for each core and maintaining a shared cache between all processors. The first method may either restrict multiple caches from containing a copy of the same memory location, or allow numerous copies to occur and suffer from the *data consistency* (or *cache coherence*) problem, that is, if multiple copies of a block exist, each at a different private cache, any changes to one of the copies must be forwarded to the other copies in order to keep the computation correct.

Shared caches, on the other hand, are slower and require a significant amount of wires, as all cores must connect to them. Besides, a moderate number of processors can generate high bus traffic, which may turn the bus into the system's bottleneck. Nonetheless they allow an alternative way of communication between the processors, reduce the complexity of cache coherence protocols, and allow a more efficient space utilization, as there will be less data redundancy. Besides, in case a core goes idle, the other cores can take its space and thus reduce resource underutilization. Typical many-core architectures take advantage of both approaches and use local caches for the L1 caches in order to keep access latency low, shared caches between pairs of cores as L2 caches, and a L3 cache shared by all cores.

It is important to notice that one of the factors that generate data inconsistency is when caches update local copies of the data without updating other copies, so caches following the write-through protocol will automatically handle this cache coherence problem as they always keep data synchronized on all memory levels by broadcasting all writes to the bus.

The usage of inclusive caches greatly simplifies cache coherence. This is due to the fact that in inclusive caches searching for a block can be done by checking its presence only on the last level cache, unlike exclusive caches, which require all caches to be accessed and checked.

2.2.7 Coherence Protocols

In order to manage cache coherence, many solutions have been proposed, either through software or hardware [33]. A simple approach to deal with this problem is to broadcast the address for every memory access and let the caches listen (*snoop*) to these, so that each cache can check its contents for a copy and invalidate it, if such copy exists (*broadcast-invalidate*). Another possibility is to update all copies of the block with the new value (*broadcast-update*). Although functional, these solutions may not be practical, as they can easily saturate system traffic due to high write frequencies.

More elaborated approaches consist of having state machine cache controllers that snoop on bus transactions and take actions depending on the block's state and the data consistency protocol being used [3], or specialized controllers (*directories*) to keep track of the holders of the copies of each block [30], which remove the obligation to broadcast data. These directories can be distributed among the caches, be placed centralized, or even designed in hierarchies They can also contain a list of all caches sharing the block, or may be designed as a linked list, so each controller keeps track of the next holder.

The cache coherence protocols usually contain a subset of these six different states: *Modified*, *Shared*, *Invalid*, *Exclusive*, *Owned* and *Forward*. A block in the *invalid* state is not valid, and thus must be fetched from the lower level memories before being used. The

modified state describes a cache line that is the only up to date copy of the data and that the data in the memory is stale. A cache line in the *exclusive* state implies that its cache has the only copy of it, and that its contents match its correspondent in main memory, *i.e.*, it has not been modified. The *shared* state represents a cache that has one of the current unmodified copies of a block. The *owned* state assigns a cache line to a cache. A cache that has a cache line in the *forward* state works as if in the shared state, but is responsible to respond to all requests to that given line.

MSI

The *MSI* protocol is the state machine of a basic protocol consisting of a valid and a dirty bit. A cache line can be in either the *modified*, *shared* or *invalid* state. Figure 2.5 shows the transitions of the MSI state machine.

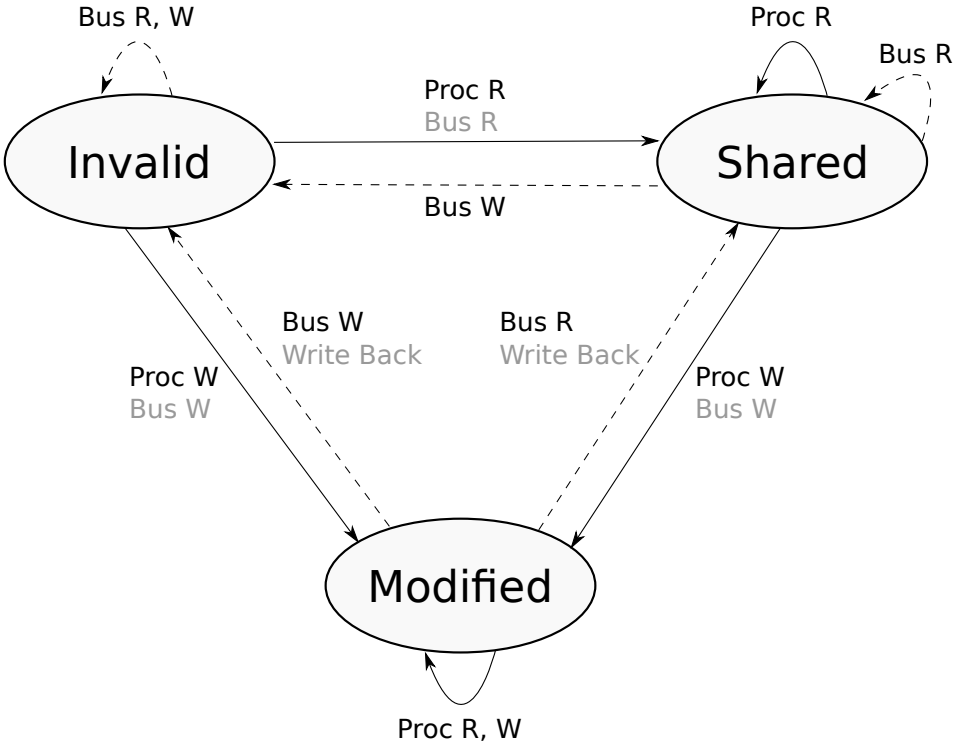


Figure 2.5: MSI state machine. Dashed lines represent bus actions, while the others represent processor actions. The action in gray is a consequence of the main action.

MESI

The *exclusive* state was added by the *MESI* protocol to reduce bus traffic generated by writes of blocks that are present in a single cache only (*i.e.*, there is only one active copy) [39]. Whenever a cache is in the exclusive state and an eviction happens, there is no need to broadcast to the other caches, as they are unaffiliated with that data. Figure 2.6 presents the MESI state machine.

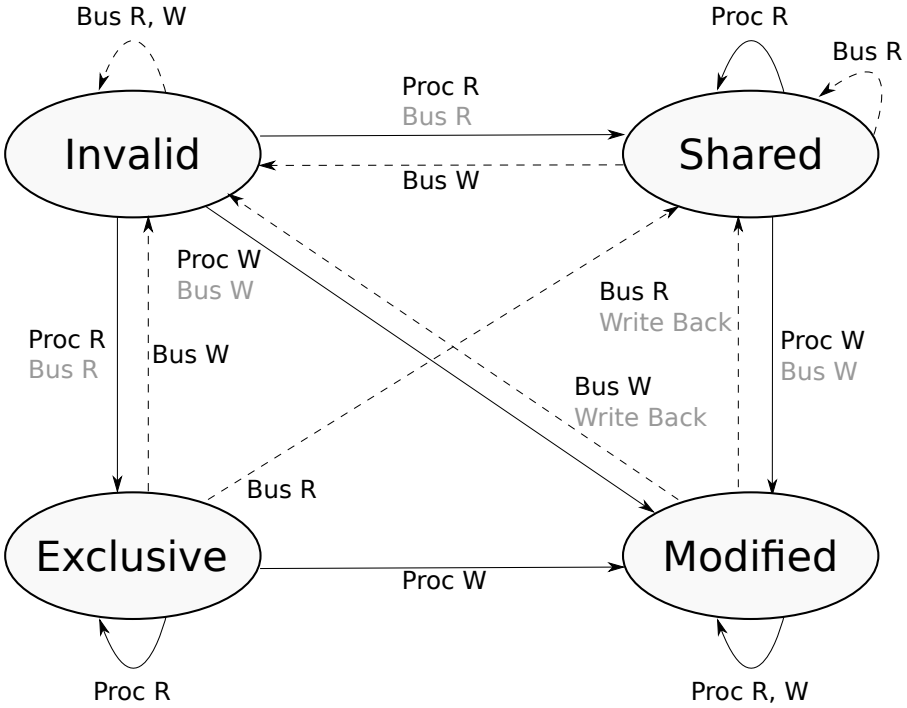


Figure 2.6: MESI state machine. Dashed lines represent bus actions, while the others represent processor actions. The action in gray is a consequence of the main action.

MOSI

The *MOSI* protocol adds the *owned* state. The owned state is similar to the shared state in a sense that it indicates that the cache line has a shared read access, but it differs from the *shared* state because its contents may be either clean or dirty. Therefore, although other caches may have valid copies of the line, only the cache in the owned state is responsible for maintaining main memory correctness, changing the value of the line, substituting main memory in data transfers, and exchanging its ownership status with other caches. Figure 2.7 shows the MOSI protocol transitions.

MESIF

The *forward* state was introduced by the *MESIF* protocol [18]. This state acts like the *shared* state, that is, the cache line is clean and being shared with other caches, but it optimizes data transfers by forwarding the data whenever a read request arrives. This allows faster transfers, as there is no need to access main memory. As in the *owned* state, at most one cache can be in the *forward* state. Whenever a read request is supplied by the cache in this state, it goes to the *shared* state, and the cache that received the line is set to be the new forwarder. The MESIF state machine is shown on Figure 2.8.

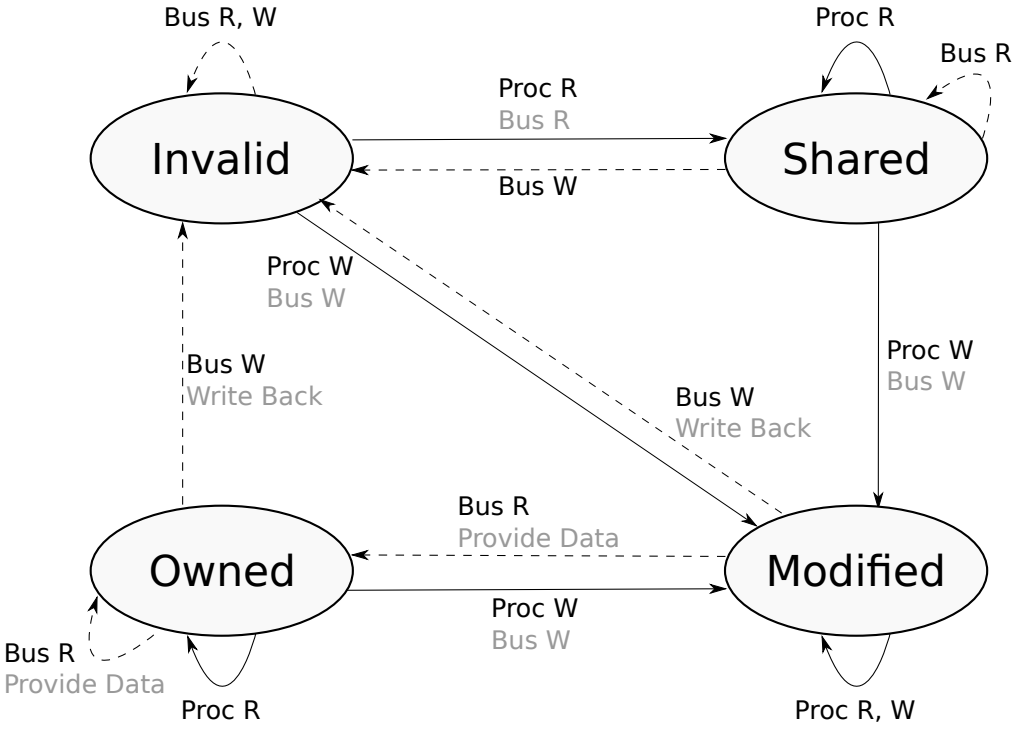


Figure 2.7: MOSI state machine. Dashed lines represent bus actions, while the others represent processor actions. The action in gray is a consequence of the main action.

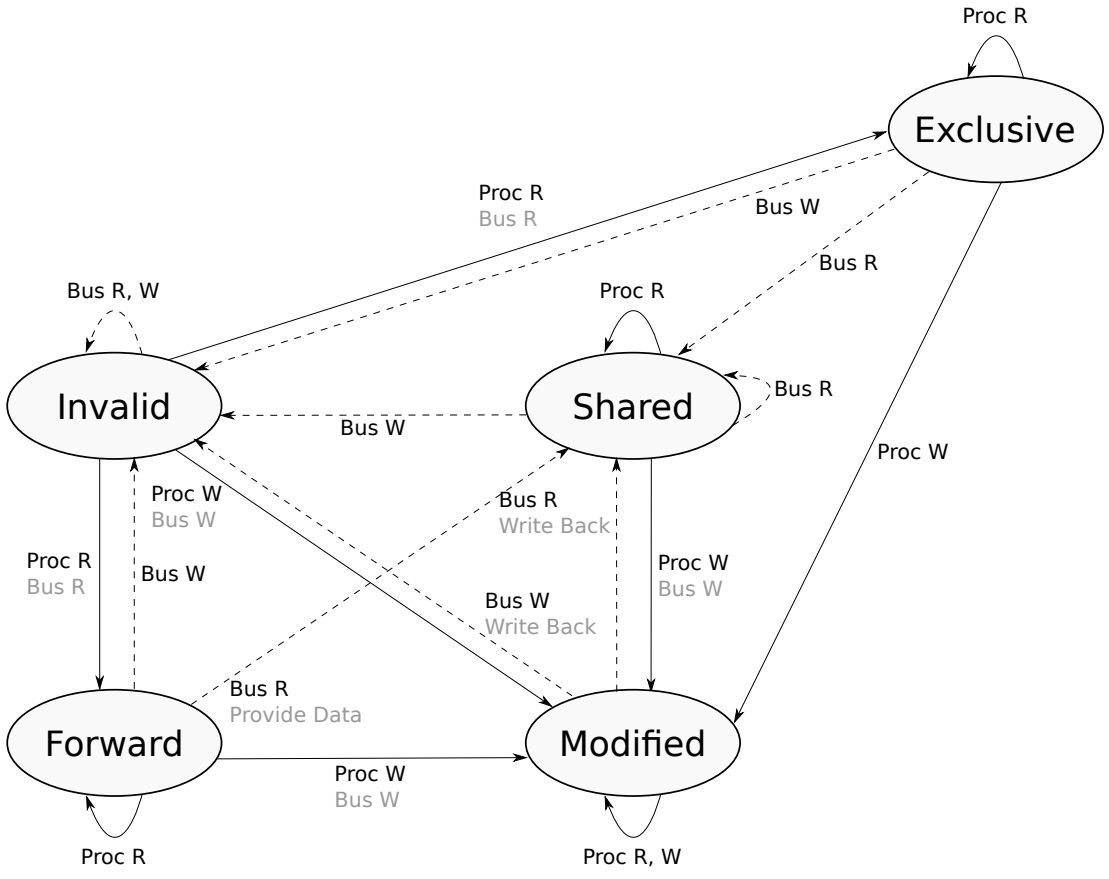


Figure 2.8: MESIF state machine. Dashed lines represent bus actions, while the others represent processor actions. The action in gray is a consequence of the main action.

2.2.8 False Sharing

A major problem when dealing with coherence is *false sharing*. False sharing occurs when multiple cores try to write to a cache line at different locations. As the addresses being written to are not equal, there's no real coherence problem, but some cache coherence protocols would mark the whole line as dirty, and thus the second access would require an update from memory. This problem can severely degrade system performance, as it will add latency to memory operations and generate unnecessary bus traffic.

For example, let us assume there are two threads, $T0$ incrementing a variable x from 0 to 100, and $T1$ incrementing variable y from 0 to 100. x and y are located in the same cache line. Assuming that the executions are interleaved, and that $T0$ is the first to execute, x is read to cache 0 and incremented to 1, and thus x's value on cache 1 is incorrect, so a signal is sent to invalidate its contents. When $T1$ requests a memory write to y, the cache line containing y is invalid, and thus it must be fetched from memory. After y is incremented, the corresponding cache line at cache 0 must be invalidated analogously. This process is repeated until the end of the loops. Figure 2.9 illustrates this example. As can be seen, there is false sharing, as both threads only use one of the variables in the line, so there would be no need to refresh its contents on every update of the variables.

Typically, false sharing can be avoided by optimizing the data layout either through programmer-aware or programmer-transparent (compiler generated) changes [52]. Examples of such modifications include allocating shared space from different heap regions in consonance with the processor that requested the space; copying the global variable that will be accessed several times to a local variable, and copying back the modified variable at the end of computation; allocating non-shared data to different cache lines; and aligning shared blocks to cache lines.

2.2.9 Cache Access

Modern caches are subdivided into banks, and architectures may be classified as *Uniform Cache Access* (*UCA*) and *Non-Uniform Cache Access* (*NUCA*) [25]. If the bank access time is the same for all banks in a cache level, that is, there is a uniform access time, the architecture is said to be an UCA. Thus, the access time of a cache level will always match the worst-case time (*i.e.*, the time to access the furthest bank). However, the hit time of a single bank is highly dependent of its distance from the processor due to the wire delays, so as cache capacity grows, so does the average access latency [20], which makes cache stalls a major bottleneck for such architectures.

A solution to this problem is to allow each bank to be accessed at a different speed, proportional to its distance to the cores. This approach negatively impacts coherence, but the overall latency is reduced when compared to a conventional UCA design. Further optimization can be achieved if data is placed so that its access frequency is inversely proportional to its distance [25].

2.2.10 Summary

Table 2.1 presents a summary of the policies detailed in this chapter.

Policy	Options	Description
Block Placement	Direct Mapped, n-way Set Associative, Fully Associative	Determine the locations at which a block can be inserted on a block placement
Tag/Data access	Serial, Parallel	Decide whether the tag and data arrays are accessed sequentially or simultaneously
Eviction	LRU, MRU, LFU	Clinch the block that should be evicted when the cache is full
Write	Write-through, Write-back	Arbitrate whether the cache levels should be kept synchronized
Processing	Write Stall, Write Buffer	Decide if the processor should wait until a write operation is complete or if it should try to keep processing
Write Miss	Write-allocate, No-write Allocate	Determine if the data written should be cached in the higher level cache or not
Data Inclusion	Inclusive, Exclusive	Resolve if the data contained in a cache level must be also present in all lower levels or not
Cache Access	UCA, NUCA	Decide whether banks in the cache have different access speeds

Table 2.1: Summary of the policies presented in this chapter.

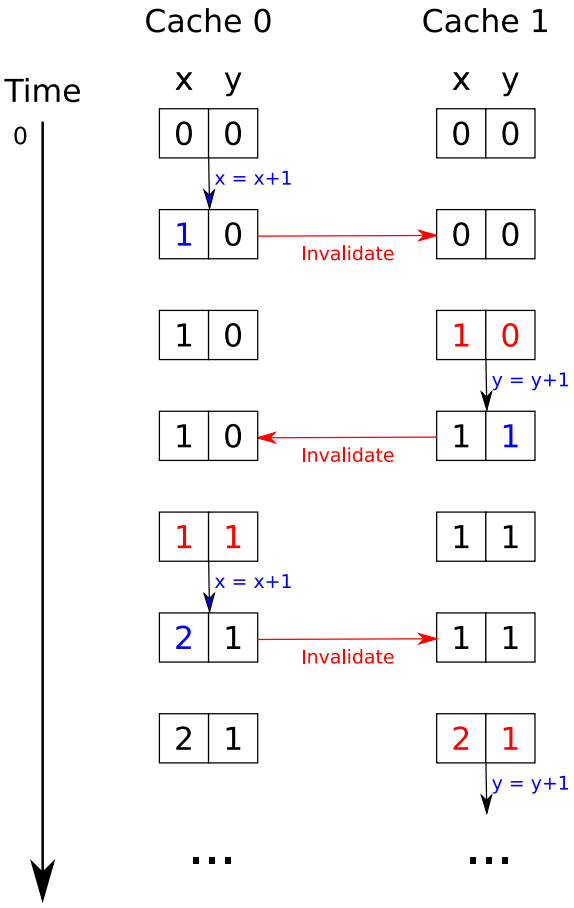


Figure 2.9: Thread 0 increments x from 0 to 100 using private cache 0, and thread 1 increments y from 0 to 100 using private cache 1. The cache line presents false sharing, because each increment of the variables in one of the caches invalidates the contents of the other cache, despite the variable being unnecessary to the opposite thread.

Chapter 3

Related Work

This chapter provides an overview of some cache compression techniques that have been proposed over the years. Table 3.3, at the end of this chapter, presents a comparison of these techniques regarding extra metadata storage, energy savings, compression ratio, matching scheme used and remarkable features. Compression ratio is calculated using Equation 3.1.

$$CR = \frac{original\ block\ size}{compressed\ block\ size} \tag{3.1}$$

3.1 Cache Compression

3.1.1 Zero-Content Augmented Caches

It has been observed that applications deal with a large amount of null data, and most of it usually shows spatial locality. Ekman *et al.* [16] experimented with 16 different benchmarks and discovered that on average 55% of all bytes in the memory are zero. Thus, caches tend to waste a lot of their limited capacity storing void data.

In order to reduce the amount of wasted space, Dusser *et al.* propose *Zero-Content Augmented Caches* (*ZCA*) [15], which consists of a conventional cache and a cache that is specialized for memorizing null blocks (Zero Content cache - ZC). On reads, both caches are checked in parallel, and if there's a hit in the main cache, a conventional cache read happens. If the hit is in the ZC, zero is returned. In the case of a miss, the read from memory happens normally, and when the data arrives at the cache, it passes through a zero detector to find out on which cache it will be stored. During writes, the zero detector is also used, and special care is taken in order to maintain coherency between both caches (*i.e.*, when data that was in the conventional cache becomes zero, or vice-versa). By keeping track of all zeros in a special cache, the ZCA manages to reduce the cache miss rate by up to 81% on the SPEC 2000 benchmarks.

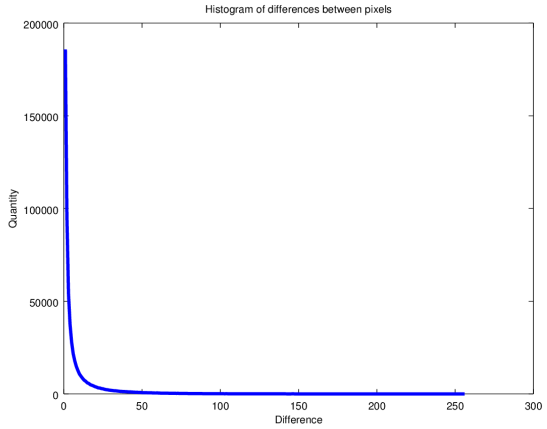
3.1.2 Doppelgänger Cache

Another important observation is that cache values usually exhibit approximate similarity, that is, they are approximately equal across many blocks. This can be seen, for example,

in images with low overall color gradient, where neighbor pixels exhibit similar values, as in Figure 3.1a. Its histogram of differences is shown on Figure 3.1c, where it can be seen that most of the adjacent pixels present similar values. Approximate similarity can also happen due to worst case scenarios, as programmers use data structures bigger than the average value, and thus the extra bytes tend to be zero. Besides that, as previously stated, zero is a frequent value in applications, as it is commonly used to initialize data and nullify pointers.

(a) Original image.

(b) Pixels whose difference with their adjacent pixels is higher than 10 units are shown in black.



(c) Histogram of differences between pixels. The horizontal axis represents the difference between the values of two neighbor pixels and the vertical axis presents the total number of pixels with such difference. Adjacent pixels tend to exhibit approximately equal values.

Figure 3.1: Original image and the contrast between its pixels.

These remarks are exploited in the Doppelgänger cache, created by Miguel *et al.* [36]. It uses an error tolerance threshold to determine values that should be considered equal and thus reduce the amount of data that the cache needs to store. In order to allow precise calculations, the Doppelgänger cache is divided into two areas: precise and approximate.

As in a conventional design, the Doppelgänger cache consists of a tag and a data array, but they are decoupled so that there can be more tags than data blocks. The tag arrays entries contain extra bits to store information regarding a doubly linked list of blocks that map to a particular data entry, and each data block contains a pointer to its

corresponding tag list's head (Figure 3.2). This way, when a block is evicted, this linked list can be accessed to find out which entries should be invalidated without applying a full tag array scan.

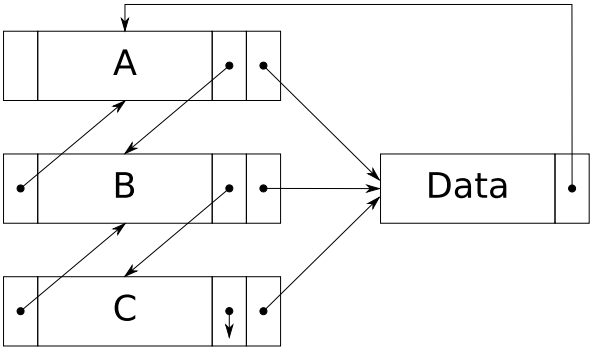


Figure 3.2: Doppelganger's doubly linked list. Each tag entry has extra metadata to inform the previous entry, the next entry and the map tag of its corresponding data block. The data block contains a pointer to the head of its corresponding tag list.

The data array stores approximate values in a way that if two blocks in the cache exhibit similar values, their tags are mapped to the same data block using a specialized hash function. When a block is inserted or there is a lookup, the block's map value is generated and the cache is searched for matches, first in the tag array, and then in the map tag array (hash value), to find the corresponding data block position.

During writes and insertions, special care must be taken as the removal of a block implies on the removal of several tag entries. Therefore, previous to removing an entry, the cache is searched for similar values, and if there's a match, it is simply appended to the match's linked list. If not, all tag entries referencing the data block to be evicted must be invalidated and cleared.

3.1.3 Base-Delta-Immediate Compression

Also based on the observation that cache values tend to exhibit approximate similarity, the Base-Delta-Immediate compression (BDI) [41] stores arrays of relative differences between blocks instead of real values, which allows high compression ratio and low decompression latency. Unlike most of the compression techniques, it has a cache line granularity, that is, it compresses whole cache lines when their values have low dynamic range. When that is not the case it leaves the data uncompressed.

Each compressed cache line in BDI is composed of metadata, one base value and an array of deltas (differences between base and a block's value) (Figure 3.3). There is also an implicit base with null value (which we will call *Base Zero*). The use of this second base is essential to improve efficiency, as one base can be used to store the common value of the line, and the other to store values approximately equal to zero, due to their high frequency on the cache. The deltas relative to Base Zero can be thought of as immediate values, which completes the naming of the method.

During data compression, several factors must be accounted for: delta sizes, as choosing sizes greater than a block's size would not be beneficial; compression factor, as setting

it fixed would narrow the possibilities; and hardware complexity to find the best base value, as applying a maximum or minimum logic operation would decrease performance and increase latency and complexity. In order to fulfill these requirements, BDI compression chooses the first value found to be the base, since it only reduces the average compression rate by 0.4%, and *a compressor for each combination of base size and delta size is created*. Note that the *delta size* must be smaller than the *base size* in order to have any advantage over an uncompressed line. Data decompression for this scheme is as simple as an adder operating on a base and its corresponding delta value.

The metadata area consists of an encoding and a bitmask. The encoding value can be one of the following: *Zeros*, that is, a cache line consisting only of null data; *Rep Values*, a cache line consisting only of a repeated 8-byte value; *Uncompressed*, which states that the cache line could not be compressed; $Base8\Delta1$ informs that the base size is 8 bytes and the delta size (size of each delta entry) is 1 byte; and $Base8\Delta2$, $Base8\Delta4$, $Base4\Delta1$, $Base4\Delta2$, and $Base2\Delta1$, which are all analogous to $Base8\Delta1$. As there is more than one base, there must be a way to inform to which base a delta refers. This is done in the bitmask area, where a set bit means that the delta uses the saved base, and a clear bit that the delta refers to Base Zero.

Figure 3.4 shows an example of a cache line being compressed using BDI, as seen from the compressor that parses the cache line as 4-byte entries with a delta size of 1 byte. The first block is selected to be the base value, and all delta values are calculated based on it and the zero-base, *i.e.*, the first block is the base, and thus its difference is 0x00 (Figure 3.4b), the second block's difference from the base (delta) is 0x02 (Figure 3.4c), the third block value's is 0x10 higher than the base (Figure 3.4d), and so on. Entry 6, however, has a delta value of 0x04 relative to the zero-base, and so its metadata is set accordingly (Figure 3.4g). As the delta values don't require more than 1 byte to be represented this encoding manages to save approximately 18 bytes.

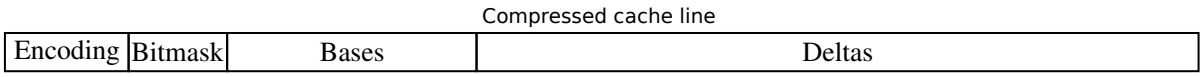
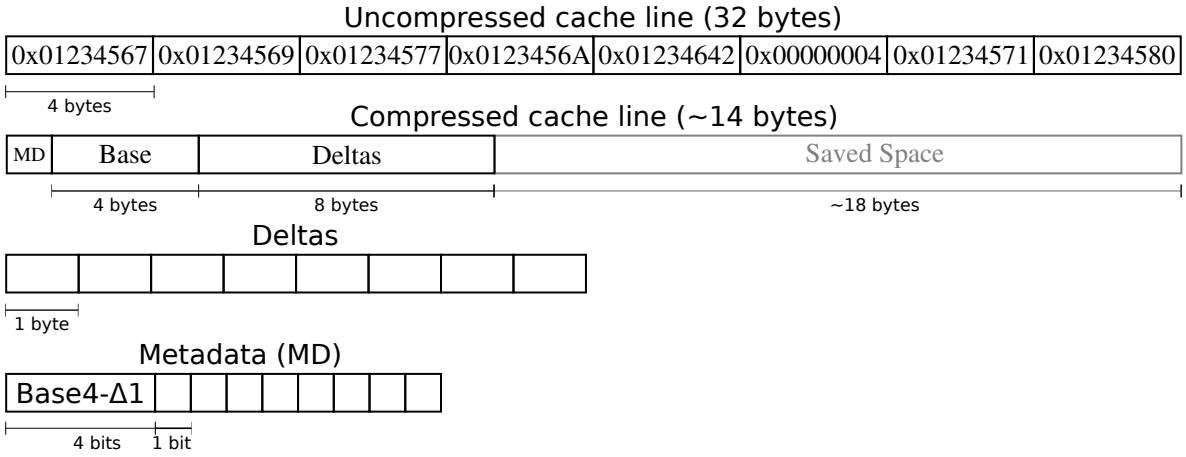


Figure 3.3: Overview of the Base-Delta-Immediate compressed data.

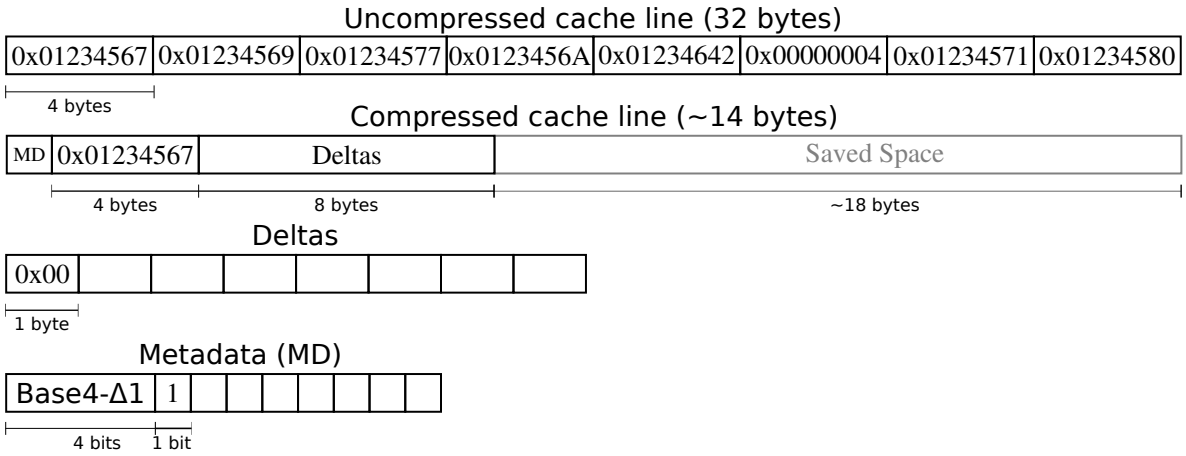
3.1.4 Frequent Pattern Compression

Frequent Pattern Compression (*FPC*), proposed by Alameldeen *et al.* [2] is a simple cache compression technique that takes into account the significance of the values stored in blocks to compress the cache in a word-per-word basis, detecting and compressing patterns based on their frequency on common workloads. For example, it is common to store in words data that could possibly fit in a smaller data structure, and thus its high order bits are usually set to zero (or one if sign-extended). Therefore space can be saved by encoding words that match these special patterns.

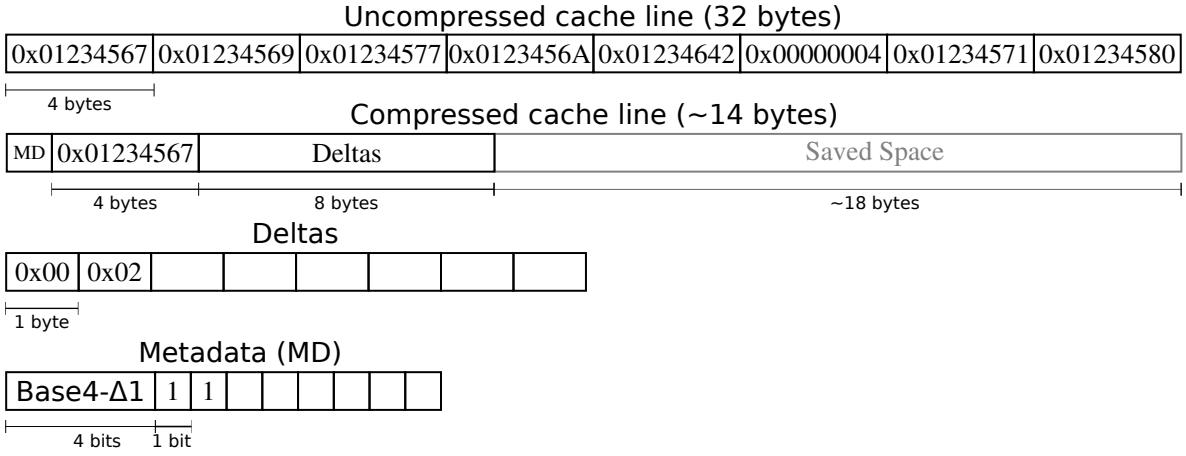
They have shown that the patterns listed on Table 3.1 have high frequency on integer workloads, so if a block matches a pattern, its relevant portion is extracted and a



(a) Initial state.

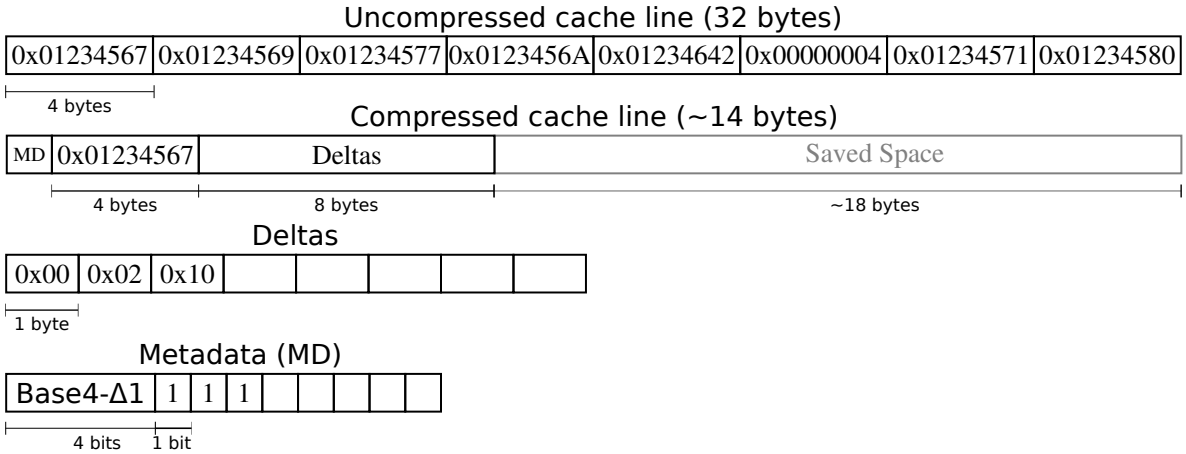


(b) Entry 0x01234567 is parsed and set as the base. Delta is set relative to base with a value of 0x00.

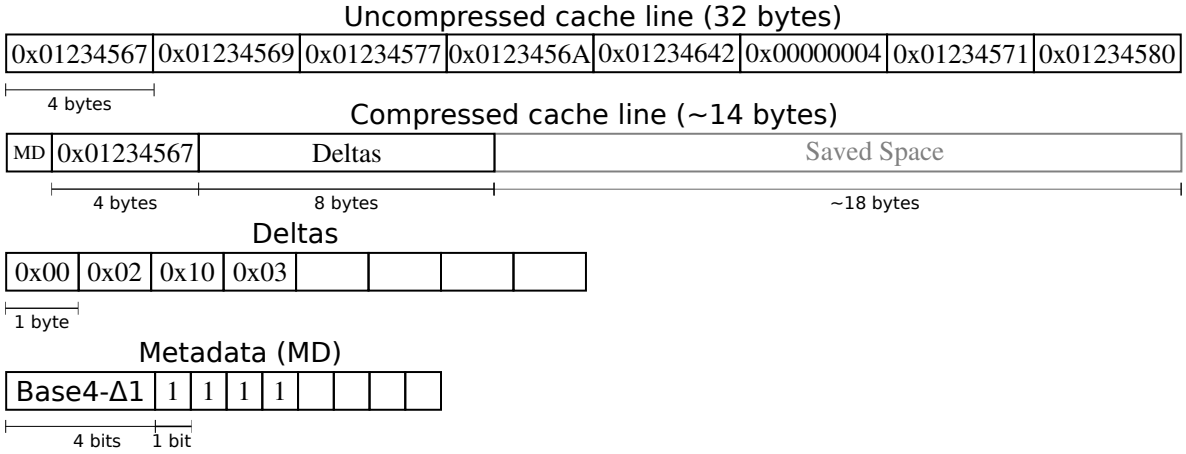


(c) Entry 0x01234569 is parsed. Delta relative to base is 0x02.

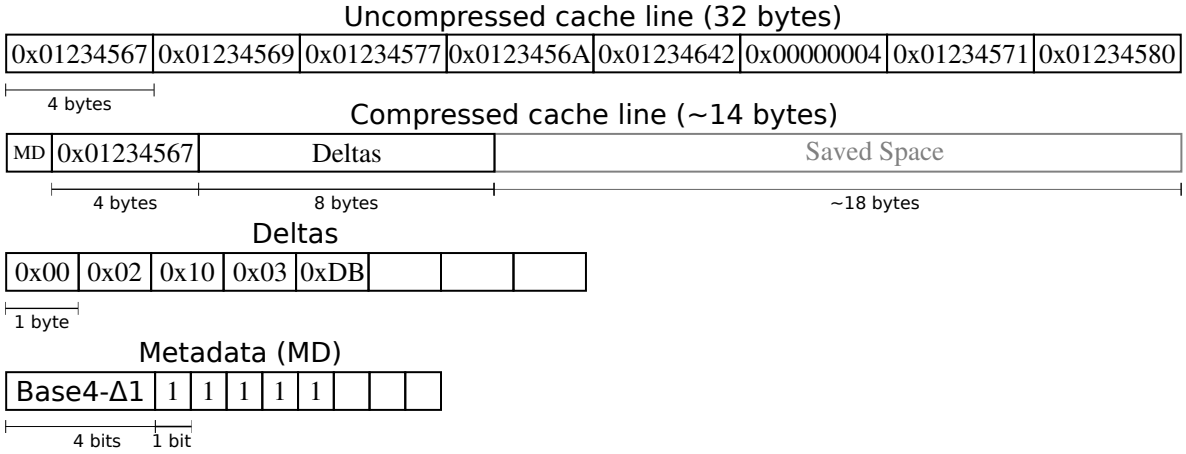
Figure 3.4: Example of Base-Delta-Immediate compression. Output of the compressor that parses the cache line as a sequence of 4-byte entries and uses a delta size of 1 byte. The first block in the line is selected to be the base, and all delta values are calculated accordingly.



(d) Entry 0x01234577 is parsed. Delta relative to base is 0x10.

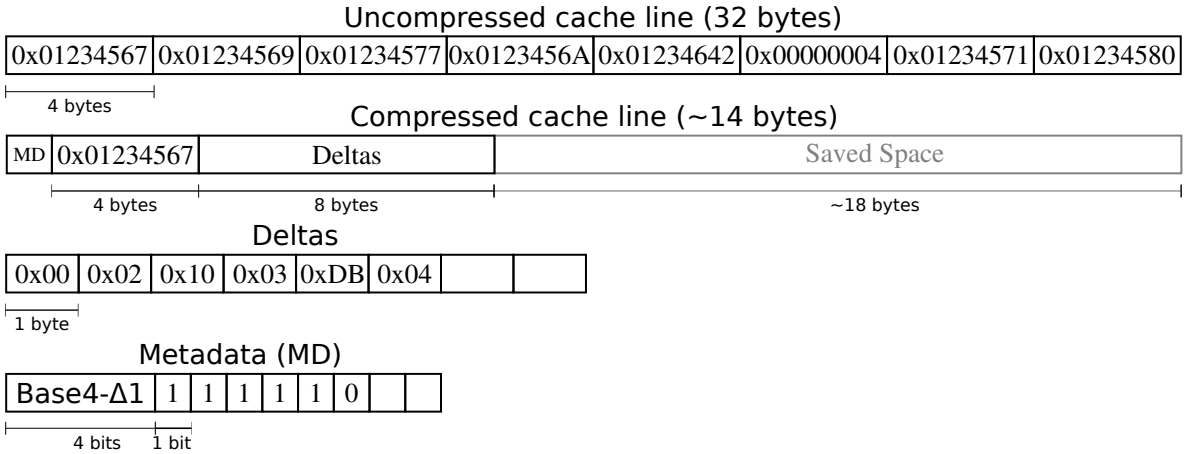


(e) Entry 0x0123456A is parsed. Delta relative to base is 0x03.

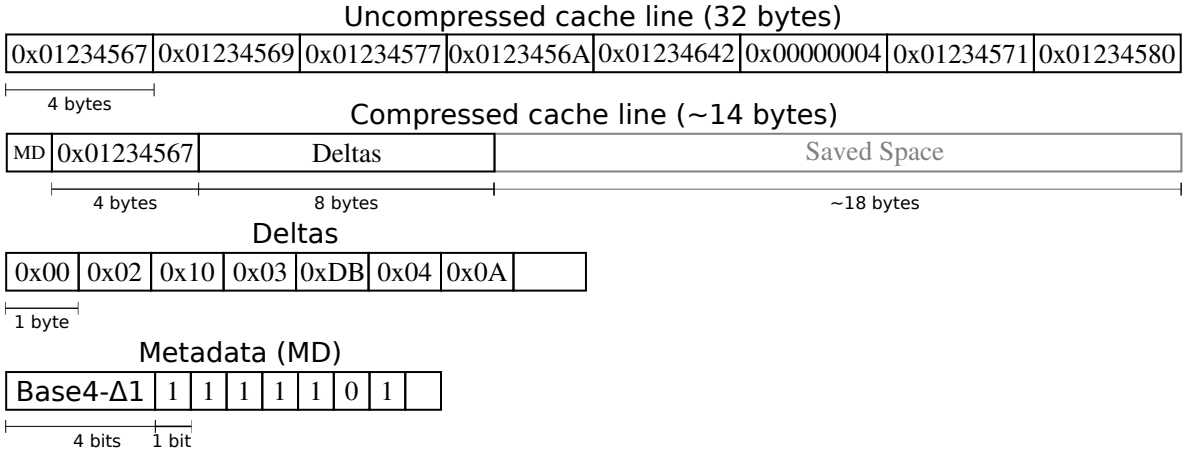


(f) Entry 0x01234642 is parsed. Delta relative to base is 0xDB.

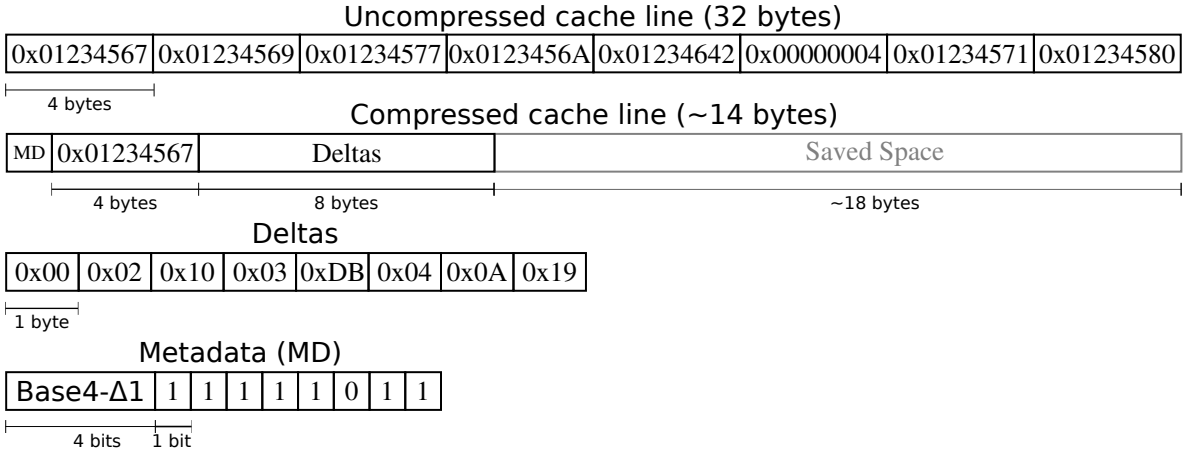
Figure 3.4: (*cont 1.*) Example of Base-Delta-Immediate compression. Output of the compressor that parses the cache line as a sequence of 4-byte entries and uses a delta size of 1 byte. The first block in the line is selected to be the base, and all delta values are calculated accordingly.



(g) Entry 0x00000004 is parsed. Delta relative to zero-base is 0x04.



(h) Entry 0x01234571 is parsed. Delta relative to base is 0x0A.



(i) Entry 0x01234580 is parsed. Delta relative to base is 0x19. Compressor successfully finished compressing the line.

Figure 3.4: (*cont 2.*) Example of Base-Delta-Immediate compression. Output of the compressor that parses the cache line as a sequence of 4-byte entries and uses a delta size of 1 byte. The first block in the line is selected to be the base, and all delta values are calculated accordingly.

compressed block is created. Otherwise the code for uncompressed word is used, and the block is stored uncompressed.

Prefix	Pattern	Data Size	Example
000	Zero run (one or more zero words)	3 bits[1]	0x00000000
001	4-bit sign extended	4 bits	0x00000002
010	1-byte sign-extended	8 bits	0xFFFFFF8D
011	Halfword sign-extended	16 bits	0xFFFFABCD
100	Halfword padded with zero	16 bits	0x0000A43B
101	Two halfwords, each 1-byte sign-extended	16 bits	0x0010FF99
110	Repeated bytes	8 bits	0x15151515
111	Uncompressed word	32 bits	0x12345678

Table 3.1: Frequent pattern encoding. The first column represents the code to be prefixed to the stored data, the second is the pattern found, and the third column presents the size of the data after compression (without the code). The last column shows examples of words with the patterns.

The compression and decompression steps are simple. During compression, each data block in the cache line is analyzed to determine if it matches a pattern, its corresponding encoding is generated and saved in a prefix array, and the compressed block is stored. The decompression stage requires more time, as the words can only be parsed sequentially due to the variable data size. First the prefix array is checked to find the data sizes. Then the data word position is determined using an accumulator on these previously calculated sizes. A register array is then shifted to match the positions of the blocks. Finally, a pattern decoder decodes the contents of each of the registers based on their respective prefixes. Due to its simplicity this approach allows a low compression and decompression latency (up to 5 cycles) while keeping as high as a 2.4 compression ratio. Figure 3.5 shows an example of cache line compression using FPC.

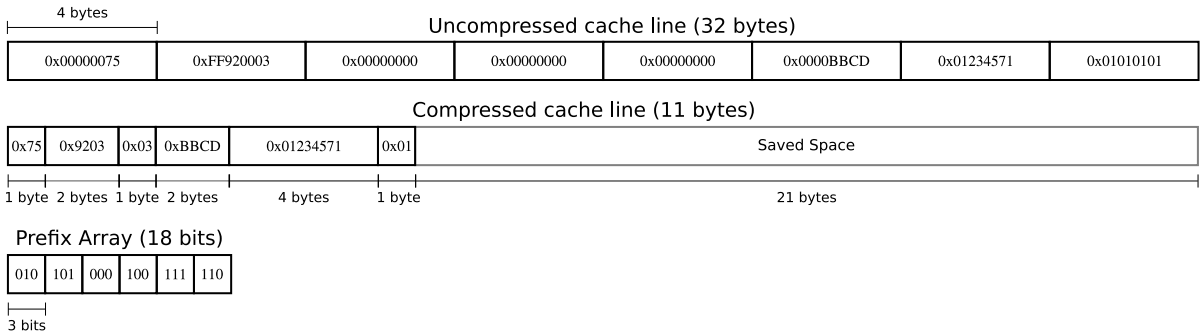


Figure 3.5: Example of Frequent Pattern Compression.

[1]For a cache with 8 words/cache line

3.1.5 Statistical Compressed Cache

Most of the previously mentioned schemes are based on simple pattern matching compression algorithms, and thus are limited to low compression ratios [5]. Arelakis *et al.* [6] propose Statistical Compressed Cache (SC^2), a cache compression technique that uses Huffman coding [23] to increase compressibility at the cost of greater compression and decompression overheads, in order to achieve higher cache utilization.

Due to its statistical nature, the SC^2 requires a short phase to collect statistics about the code before code compression can be started. It does so by storing the frequency of all unique values in a Value-Frequency Table (VFT). After the sampling phase is done the VFT is frozen, and the codewords are generated. When a block is inserted, the codewords are checked, and if there is no match, the value is left uncompressed, and a special codeword is used to inform the decompressor.

In case the compression ratio becomes unsatisfactory (*i.e.*, it becomes lower than a threshold), another sampling stage can be started concurrently using a second decoder, as the data inside the cache may have two encodings, and whenever a compressed block using the old encoding is decompressed, the new encoding is applied to it. The authors have shown that the sampling stage does not have to be frequent, due to the little variation in value locality over time, even if multiple applications are being executed, and thus it can be managed by software to reduce compression latency. Due to the higher compression ratios achieved on the SPEC 2006 benchmarks (2.2X, on average) it can achieve better performance than simpler methods, despite its higher decompression latency.

3.1.6 C-Pack

C-Pack+Z is a pattern matching and partial dictionary decoding, *i.e.*, an entry in the dictionary can be matched completely or partially matched, that detects frequently appearing data and does not generate much hardware complexity while keeping an effective compression ratio (61%) and a low decompression latency [12], and zero-block detection [15].

Table 3.2 presents the patterns used on the original paper. The first column indicates the code generated if the respective pattern in the second column is found. The Data Size column shows the block's size after compression without code and position index. The last column presents the generated compressed block with code and position index. The size of the position index is directly proportional to the dictionary size, *i.e.*, if the dictionary has 16 entries, the position index is 4 bits long. It is important to notice that the compressed block doesn't need to be decompressed to find out the compression factor, as it can be determined from parsing the code, and therefore it can be done in 1 cycle.

Whenever a word is compressed it is parsed to find a pattern. If a $ZZZZ$ or $ZZZX$ pattern is found, where Z denotes a zero byte and X denotes no dictionary match, its code is generated and the dictionary remains intact. Otherwise, there's either a dictionary match (denoted by M) or no match, and the dictionary must be updated with the new word. When there's a match, the position of the match within the dictionary is indicated through a position index placed between the code and the data. Figure 3.6 demonstrates how C-Pack effectuates word compression. For the sake of simplicity we assume the

Code	Pattern	Data Size	Output - (Code)Data
00	ZZZZ	0	(00)
01	XXXX	32	(01)XXXX
10	MMMM	0	(10)p
1100	MMXX	16	(1100)pXX
1101	ZZZX	8	(1101)X
1110	MMMX	8	(1110)pX

Table 3.2: Pattern encoding for C-Pack. Z is a zero byte, X is a byte that does not match any dictionary entries, M is a dictionary match, and p is the index of the position of the match.

dictionary supports 8 entries and thus the position index is 3 bits long and that it initially contains 4 entries. The outputs of the compressor are concatenated to generate the compressed cache line.

During decompression the code is read to define the pattern. If the pattern is $ZZZZ$, $ZZZX$ or $XXXX$ the decompression is straightforward. When there's a dictionary match in the pattern, the dictionary must be consulted using the position index to discover the data that is not present in the compressed word.

3.1.7 Manycore-Oriented Compressed Cache

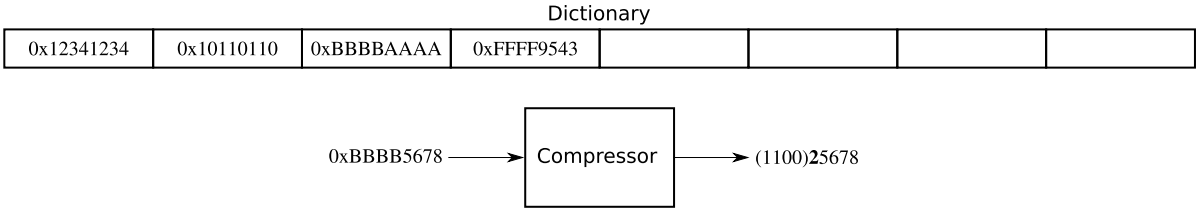
All previous compression schemes are single-thread oriented, that is, they focus on improving performance for single stream applications. MORC, proposed by Nguyen *et al.*, aims on future manycore architectures, which will have thousands of cores and smaller bandwidth per core, and thus is focused on improving throughput [38].

MORC was designed based on the insight that "compressing larger blocks of data [...] is likely to yield higher compression ratios". While conventional designs exploit intra-line compression, that is, each block is isolatedly compressed, regardless of its surrounding blocks, the *Manycore-Oriented Compressed Cache* adds inter-line compression by using pointers to previously compressed data in order to avoid compression duplication. To achieve this, MORC uses a log-based cache, on which data is appended based on data commonality patterns, instead of data address, as in typical set-based caches. Besides, tags are also compressed, by using Base-Delta encoding, so that higher compression ratios can be attained.

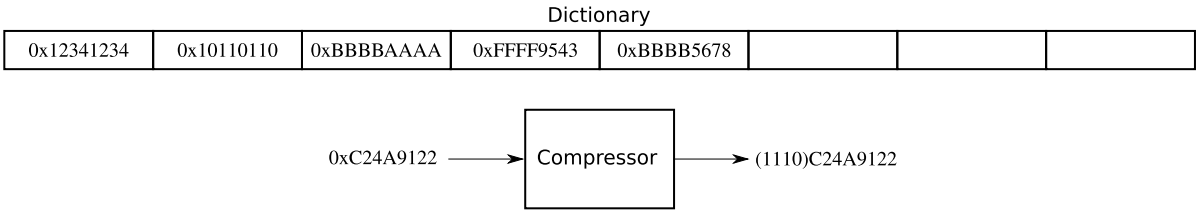
Even though its decompression step requires more energy and access latency, less cache misses are generated because of the improved compression ratio, that is, the number of accesses to off-chip memories are reduced, and thus this method manages to save bandwidth and lower overall energy.

3.2 Selective Cache Compression

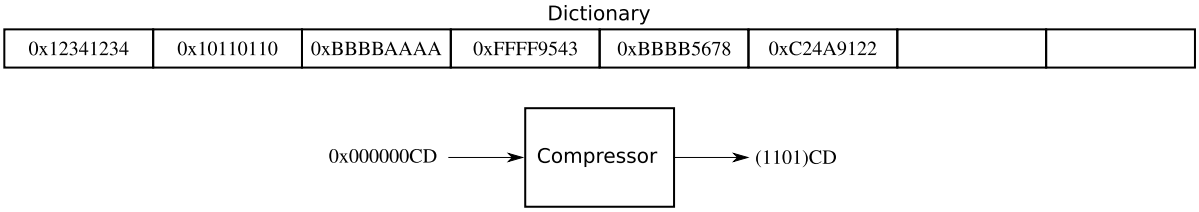
Cache compression has the advantage of increasing cache capacity without proportionally increasing energy and area, but its decompression stage may degrade cache performance



(a) Input word 0xBBBB5678 is parsed for patterns. It matches MMXX at dictionary entry 2, so the compressor outputs 0x(1100)25678. The word is appended to the dictionary and the compressed cache line now contains: 0x(01)12341234(01)10110110(01)BBBBAAAA(01)FFFF9543(1100)25678.



(b) Input word 0xC24A9122 is parsed for patterns. It matches XXXX (no match with dictionary entries), so the compressor outputs 0x(1110)C24A9122. The word is appended to the dictionary and the compressed cache line now contains: 0x(01)12341234(01)10110110(01)BBBBAAAA(01)FFFF9543(1100)25678(1110)C24A9122.



(c) Input word 0x000000CD is parsed for patterns. It matches ZZZX, so the compressor outputs 0x(1101)CD. The word is *not* appended to the dictionary and the compressed cache line now contains: 0x(01)12341234(01)10110110(01)BBBBAAAA(01)FFFF9543(1100)25678(1110)C24A9122(1101)CD.

Figure 3.6: Example of word compression using C-Pack. The outputs of the compressor are concatenated to generate the compressed cache line. Initial compressed cache line is: 0x(01)12341234(01)10110110(01)BBBBAAAA(01)FFFF9543. Numbers within parenthesis are in binary representation, and are used so that non-4-bit offsets are not applied, which makes data easier to read.

due to an increase in access latency. Selective methods turn on and off compression by analyzing overall system performance.

3.2.1 Adaptive Cache Compression

Alameldeen *et al.* [1] proposed an adaptive compression scheme for a two-level cache hierarchy where L1 is uncompressed to avoid hits overhead, as they are usually critical, and L2 can be compressed. L2's compression state is decided dynamically, depending whether its advantageous to allow compression for a line or not.

Although this approach allows a performance speedup of up to 26%, the extra bits needed for each tag to store the lines' compressed size and compression status along with the small compression factor (a compressed line can store twice as much as an

uncompressed one) may outweight its usage if memory performance is not the limiting factor.

3.2.2 Selective Code Compression

Selective Code Compression methods selectively choose regions of the cache to be compressed based on execution profile [8] [14] [53] and instruction frequency [29] in order to minimize this decompressor impact. The Dual Selective Code Compression scheme [37], for example, creates a dictionary for frequently executed instructions (called Dynamic Dictionary by the authors), such as in instructions within loops, and a dictionary for instructions based on their frequency (called Static Dictionary), that is, the remaining instructions.

This idea is based on the fact that inner-loop instructions are generally different from the ones surrounding them, and thus they will produce different dictionaries. Then the code is scanned and, for each dictionary, the instructions are selected in order to find out which ones should be stored compressed, and which shouldn't as to increase processor performance.

3.2.3 Hybrid Methods

Hybrid methods also exist, such as HyComp [4], which was created based on the observation that a cache compression method is not the best for every workload, and thus it should be chosen from a pool of methods. It does the compression in a two phase approach: first it inspects the block to predict its contents and then it selects the most suitable compression technique.

3.3 Cache Organization

Compressed data has to be placed properly, as if the data is compressed but allocated to the same amount of space the compression would be useless. This is called *cache organization* or *cache compaction*. Some techniques focus on determining where to put compressed data, and how organizing the compressed cache in order to benefit from the compression. Note that although some of the previously mentioned techniques may address this problem, its not their main proposal.

3.3.1 Decoupled Compressed Cache

The Decoupled Compressed Cache [48] uses decoupled super-blocks, a group of four aligned neighbor 64-byte cache blocks that share a single tag, where each block can be divided in a variable number of 16-byte sub-blocks (from 0 to 4). The DCC consists of a super-block tag array, a sub-blocked back pointer array and a sub-blocked data array. Each entry in the back pointer array maps to a single entry in the data array, and consists of a tag id representing the way number and a block number.

When a block is compressed, a variable number of sub-blocks is generated, and due to the decoupled nature of the method, they do not need to be placed contiguously (Figure 3.7). This creates extra area and latency overhead, but removes the need to relocate the block whenever its compression changes. On lookups both the tag and back pointer arrays are accessed in parallel, and the sub-blocks corresponding to the block are determined by the back pointer array.

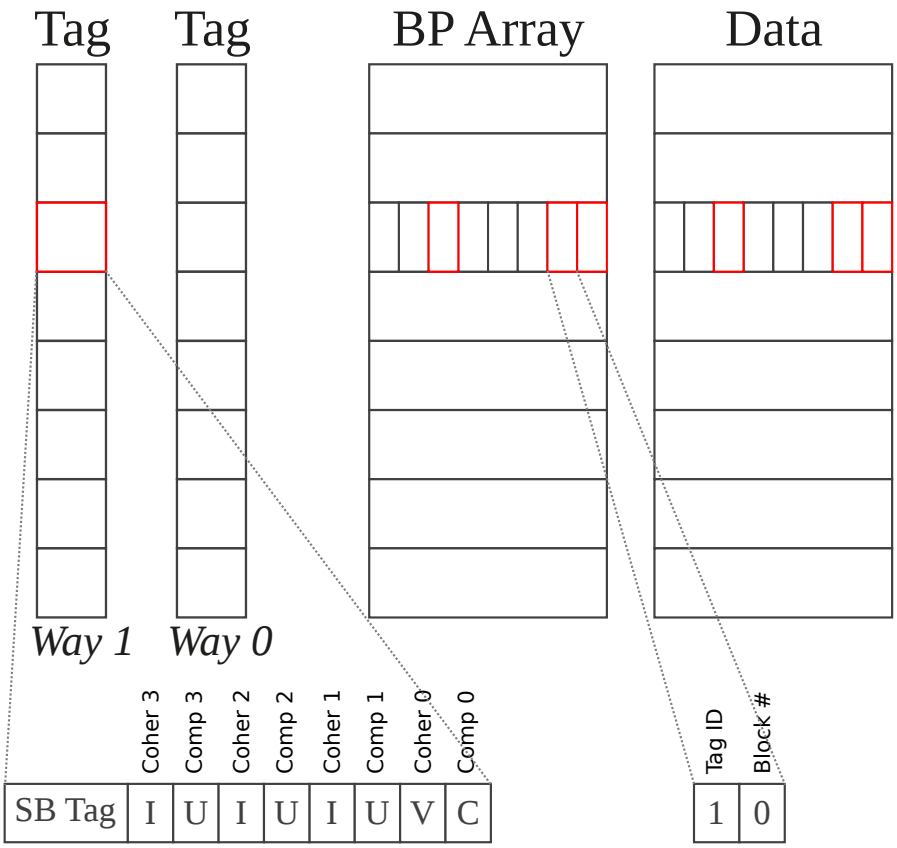


Figure 3.7: Set 2's super-block contains a single valid compressed block at sub-block 0 (its respective coherence state is valid, and compression bit is set). The back pointer array contains 3 entries related to this sub-block: 0, 1 and 5. The tag ID of each of these BP entries is set to 1 to match the way at which its respective sub-block can be found, and the block number is set to the number of the sub-block. The corresponding data entries are highlighted.

3.3.2 Skewed Compressed Caches

Skewed Associative Caches (SAC) [50] are caches that allow overcoming one of the problems of cache associativity: conflict misses. Conflict misses happen when a quantity of blocks higher than the number of ways map to the same set. These misses can be reduced by increasing the number of ways or the number of replacement candidates [44], but this is costly, both latency and energy-wise.

By indexing each way with a different hash function, conflict misses are reduced without the drawbacks of increasing associativity. *i.e.*, without using Skewed Associative

Caches, blocks that conflict in a way will conflict in all other ways, as the hash function will be the same. If Skewed Associative Caches are used, blocks that conflict in a way most likely won't conflict in other ways. For example, in a conventional cache, a block A would be mapped to set 1 on all ways (Figure 3.8), but in a SAC each hash maps A to a different set (Figure 3.9).

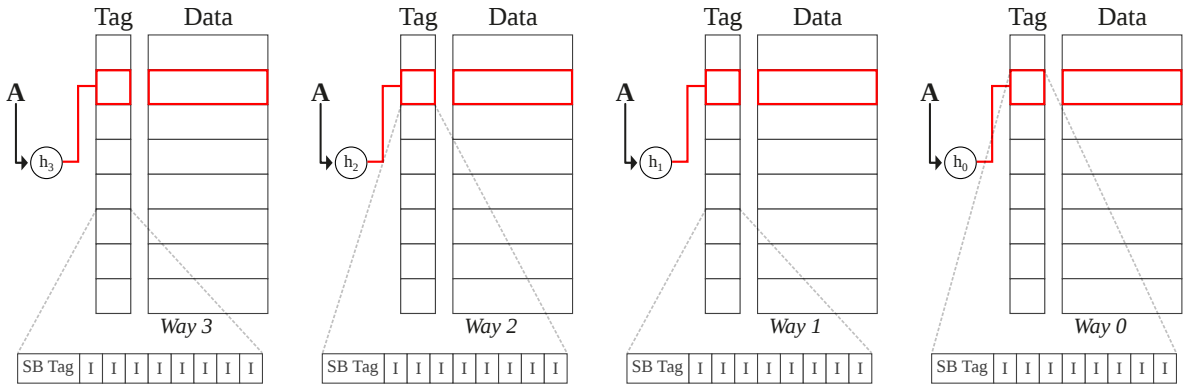


Figure 3.8: Conventional cache mapping. Blocks are mapped to the same set even on different ways due to the usage of a single hash function for all ways.

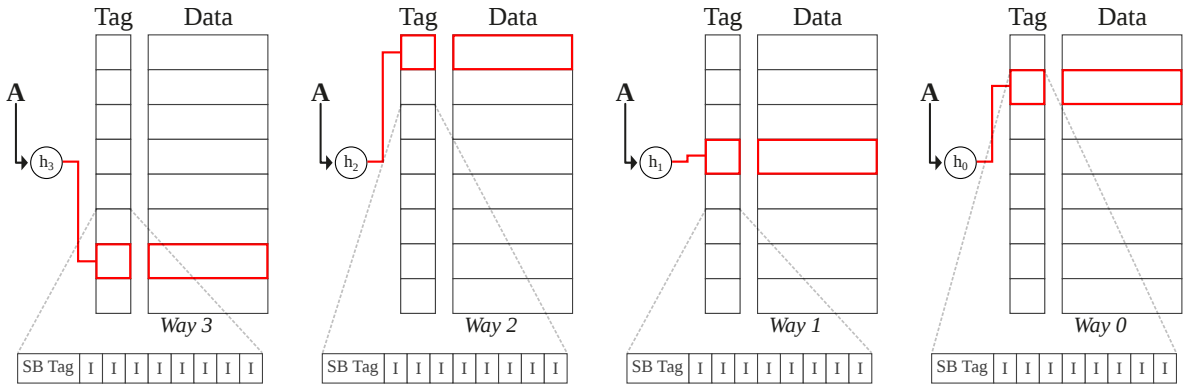


Figure 3.9: Skewed Associative Caches have different hash functions for each way.

Skewed Compressed Caches are based on Skewed Associative Caches and use the C-PACK + Z compression algorithm [12]. SCC use the fact that most workloads tend to have compression and spatial locality, that is, neighboring blocks tend to have similar compressibility and be inside the cache simultaneously, respectively, to gather neighboring blocks with the same compressibility in a single physical entry.

It does so by grouping up to 8 adjacent blocks (*super-blocks*), each with its corresponding super-block tag. These super-blocks are sparse, so they can be distributed along many data entries, according to its blocks' compressibilities (blocks with the same compressibility are placed on the same data entry). When they are placed in a data entry, a state tag informs the validness of each of the data entry's 8 sub-blocks (8 bytes each, totaling 64 bytes).

To execute cache lookups, first the compression factor must be determined, which is done using Equation 3.2 with all cache ways. Then, for each way, Equations 3.3 and 3.4

are used to find out the set index and byte offset, respectively. h_i represent the hash functions used for each compression factor. The tags of each of the possible data entries are then checked to verify if the super-block tag matches and the offset is valid. By using the state tag and the address of the block to determine the offset information, the block's location within the data entry will be known without using many extra bits, as in DCC [48]. Figure 3.10 shows an example of possible mappings for block A for all its different compression factors using SCC.

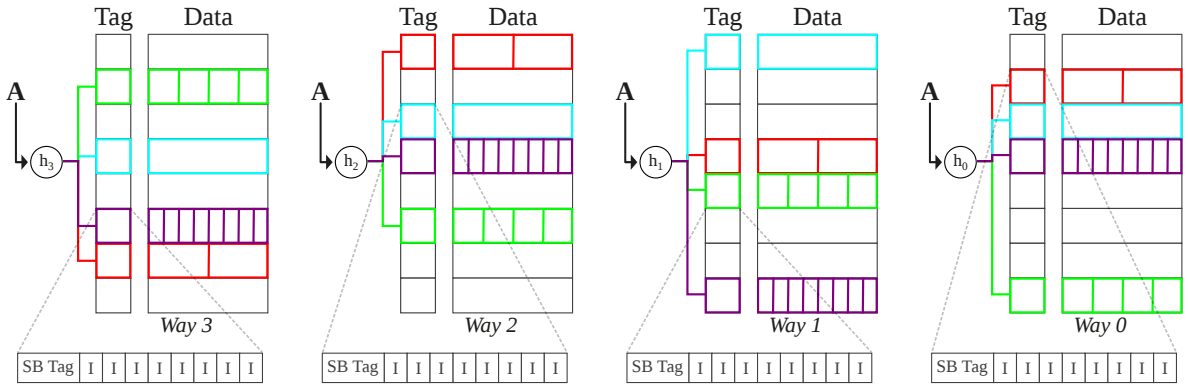


Figure 3.10: Different mappings for block A for all its different compression factors.

$$CF_1CF_0 = A_{10}A_9 \wedge W_1W_0 \tag{3.2}$$

$$SetIndex = \begin{cases} h_0(\{A_{47} - A_{11}, A_8, A_7, A_6\}); & CF == 0 \\ h_1(\{A_{47} - A_{11}, A_8, A_7\}); & CF == 1 \\ h_2(\{A_{47} - A_{11}, A_8\}); & CF == 2 \\ h_3(A_{47} - A_{11}); & CF == 3 \end{cases} \tag{3.3}$$

$$ByteOffset = \begin{cases} none; & CF == 0 \\ A_6 << 4; & CF == 1 \\ A_7A_6 << 3; & CF == 2 \\ A_8A_7A_6 << 2; & CF == 3 \end{cases} \tag{3.4}$$

For cache writes the computation of the set index, byte offset and way can be done in parallel using Equations 3.3, 3.4 and 3.5, respectively. As can be seen, unlike regular skewed associative caches, which allow a block to be allocated to any way, the compression factor of a block determines the way to be used.

$$W_1W_0 = A_{10}A_9 \wedge CF_1CF_0 \tag{3.5}$$

3.3.3 Yet Another Compressed Cache

Skewing the cache may lead to complicated tag array design, due to the need of extra decoders and wiring. Besides, it limits the choice of replacement policies. Based on these

observations, and trying to overcome the problems in the SCC design, Sardashti *et al.* propose *Yet Another Compressed Cache* (*YACC*) [47].

As in the SCC, YACC uses sparse super-block tags and relies on spatial and compression locality, but it does not skew the cache, therefore a block can be easily found by its index. Each super-block can track up to 4 16-byte blocks, and there are only 3 valid compression factors: 4 (each block is compressed to 16 bytes), 2 (the block is compressed to 2 bytes), and 1 (the block is uncompressed).

Each tag entry consists of a super-block tag, a CF4 bit and an optional CF2 bit informing, respectively, if the compression factor of the blocks is 4 and 2. It also contains the coherence state for each of the valid entries, and, if the compression factor is not 4, an extra entry is needed to reference the index of the super-block block stored. This last information allows non consecutive blocks of a super-block to be mapped to the same tag, as opposed to SCC's behavior.

Assume blocks ABCD belong to a super-block, and A, C and D compress to 16 bytes, and B compresses to 32 bytes. When allocating these blocks, first the super-block tag is checked to find the set to which it belongs. Then A, C and D are mapped to a free way (in this case, 0), and its tag's CF4 bit is set and their corresponding coherence states are set. B is mapped to another way (in this example, 2) due to its different compression factor, so the entry's tag's CF4 is set to 0, the CF2 bit is set to 1, the block index is set to 2 (B's index within the super-block) and its corresponding coherence state is set. Figure 3.11 illustrates this example.

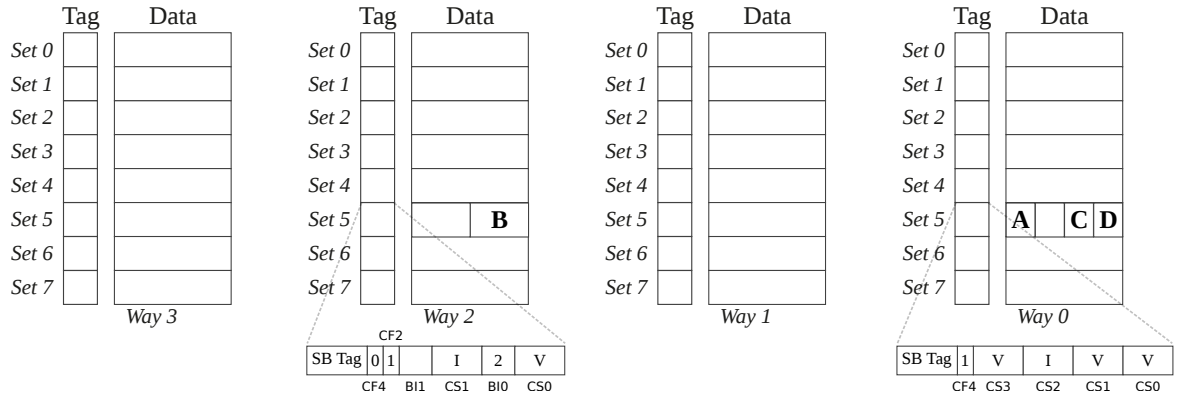


Figure 3.11: Yet Another Compressed Cache example. Blocks A, B, C, D are consecutive blocks in a super-block. A, C and D compress to 16 bytes and B compressed to 32 bytes.

Reading and writing data in the YACC design is similar to what happens in a conventional cache. The super-block tag is searched for on all possible ways, and if there is an entry with the given tag the coherence state and block index must be scanned for a match. Recompressions are kept simple, as if the block is the only valid entry it only requires a tag and data update. Otherwise, the previous entry is invalidated and the block is written to another place in the same set.

Technique	Metadata	Decompression Latency (cycles)	Compression Ratio	Other features
ZCA	$< 1\%$	1	-	Multicore aware
Doppelganger	High	6	1.43x	Decoupled tag and data
BDI	Low	1	1.4x	Multicore aware
FPC	Low	5	1.0-2.4x	Significance based
C-Pack	Low	8	1.61x	Dictionary based
MORC	17%	Highly Variable	3x	Multicore oriented
SC^2	10%	14	2.2x	Huffman matching scheme
Adaptive	11%	5	1.25-1.75x	Selective compression
Dual Selective	High	1	1.7x	Selective code compression
DCC	16.7%	9	2.6x	Decoupled tag and data
SCC	3.5%	9	1.8x	Superblocks
YACC	2%	9	1.84x	Flexible replacement policies

Table 3.3: Comparison of the techniques described in this chapter. "-" has been inserted on cells at which data is not known or has not been explored by the authors of the method. The compression ratios were extracted from the original papers and were calculated using different benchmarks, therefore they should not be compared between themselves.

3.4 Summary

Table 3.3 presents a summary of the techniques presented in this chapter. Values were extracted from original papers using various benchmarks, so they should not be compared between themselves.

Chapter 4

BDI Compression Extensions

The Base-Delta-Immediate compression constrains all bases and deltas to a single delta size. This means that even if almost all deltas can be represented using a small delta size, if there exists one difference value that needs a large delta size, then all deltas will be padded to accommodate that size.

An example of compression that wastes space with padding is given on Figure 4.1. For simplicity the cache line consists of 32 bytes, but the idea can be applied to any cache size. All entries of the uncompressed cache line but the third can be compressed to a delta size of 1 byte, and the exception needs 4 bytes due to its higher discrepancy. As in BDI all deltas are compressed using a single delta size we have to waste 3 bytes in padding for every other entry, totaling 9 wasted bytes (in red). We evaluated BDI using the SPEC2006 benchmarks, and the mean amount of space wasted with padding corresponds to 28% and 9% of the space used by the delta array for the entries with base size 8 and 4 bytes, that is, 28% of the compressed line generated by a base 8 compressor's data, on average, is padding.

In this work we make two proposals that remove the paddings by allowing multiple delta sizes, instead of a fixed one: BDI Relative to Bases (BDI-RB) keeps a delta size for each base, and BDI Relative to Deltas (BDI-RD) has a delta size for each delta entry. We also present a technique to allow BDI to use multiple bases, the MBDI.

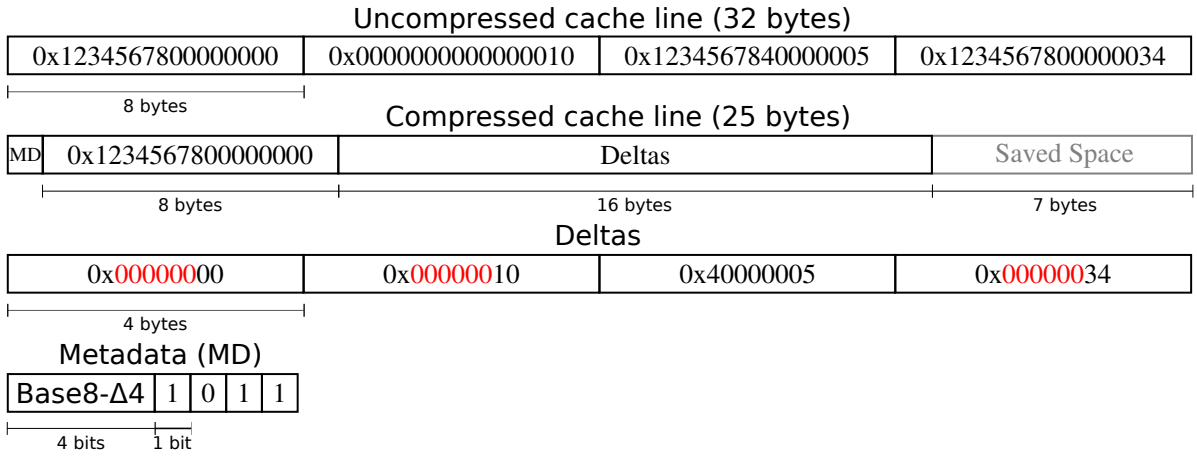


Figure 4.1: Example of wasted space using Base-Delta-Immediate Compression.

Cache Organization

We maintain the original approach regarding cache organization: use twice as many tags per data entry, and divide each data entry in segments. Pointer are also added for each tag entry to provide the starting segment of its corresponding compressed cache line. Therefore, the storage cost for the tag-store is the same as BDI's.

4.1 Flexible Base-Delta-Immediate Compression

We propose two approaches to solve the padding problem, all of them increasing the compression flexibility by allowing more delta sizes. Figures 4.2 and 4.3 present an overview of the compressed data using the solutions that will be presented in this section. The encoding, bitmask, bases and deltas fields keep the values as in the original BDI, but the provided expansions use different encodings, and the deltas are not restricted to a single delta size.

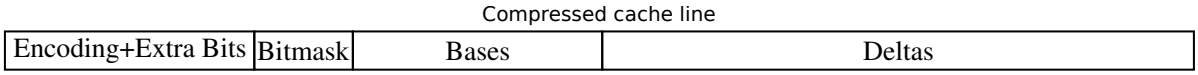


Figure 4.2: Overview of the compressed data using Base-Delta-Immediate Relative to Bases (BDI-RB).

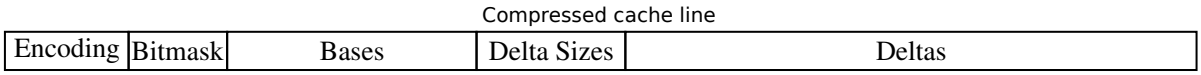


Figure 4.3: Overview of the compressed data using Base-Delta-Immediate Relative to Deltas (BDI-RD).

4.1.1 Delta sizes relative to bases (BDI-RB)

One solution to this problem is to set a delta size for each base. Therefore every entry that refers to a base uses its delta size. This extra information can be stored in the metadata area by adding new encodings (discussed in Section 4.1.4).

In the example given in Figure 4.1, if each base was mapped to a delta size then Base Zero would have a delta size of 1 byte, as all differences referring to it (immediate values) are smaller than 1 byte, and all entries referring to the stored base would be stored in 4 bytes. This would save 3 extra bytes when compared to the original compression, as seen in Figure 4.4.

Whenever Base Zero does not have entries, its delta size is set to be the same as the saved base's delta size. This is done because some of the encodings proposed require more metadata to be stored in case delta sizes differ.

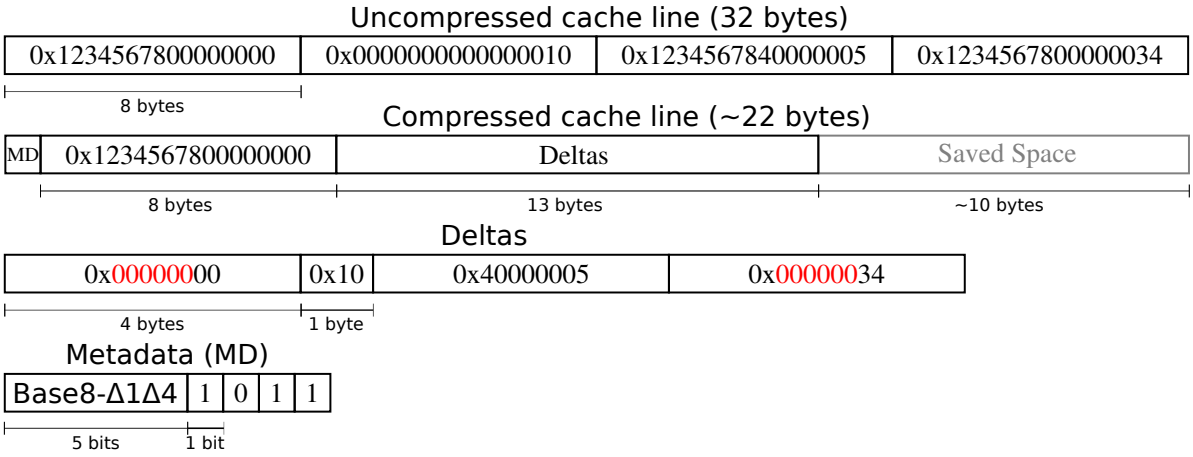


Figure 4.4: Saved space using Base-Delta-Immediate Compression with delta sizes relative to bases.

4.1.2 Delta sizes relative to deltas (BDI-RD)

Other viable approach is to associate the delta sizes with the delta values themselves, that is, every delta entry has a corresponding delta size entry. This technique wastes more space with metadata, but has a better compression ratio than the previous methods when entries referring to a base are not constrained to a single delta size. For example, when using the uncompressed cache line of Figure 4.1 this approach manages to save 8 extra bytes when compared to the original compression, as seen in Figure 4.5

Again, whenever Base Zero does not have entries, its delta size is set to be the same as the first delta size. This is done because the encoding proposed for this approach requires more metadata (*i.e.*, the delta size for each delta) to be stored in case delta sizes differ. To complement that, whenever all delta sizes are equal the delta sizes array is substituted by a single delta size entry, so the cases covered by the original encoding are not worsened by more than 2 bits. In the case of a compression with a 2-byte base, there is only one possible delta, so there is no need to store delta size.

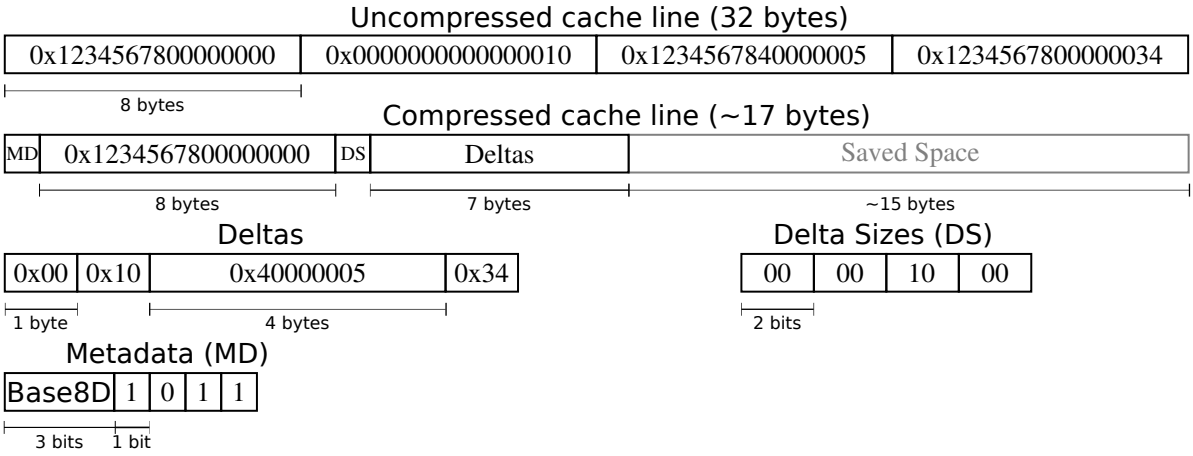


Figure 4.5: Saved space using Base-Delta-Immediate Compression with delta sizes relative to deltas.

4.1.3 Implementation

So far we discussed how deltas should be stored, but not how the compressors work. As in the BDI idea, the proposed method consists of having multiple compressors, each with a different combination of base and delta sizes. Each compressor operates on entries with size given by base size, and the delta limits are given by its delta size. However, in the flexible delta size BDI solutions the delta size is not fixed, and thus the compressors try all possible delta sizes up to the compressor's delta size.

For example, the compressor with base size equal to 8 and maximum delta size of 4 has to test if the entry is within limits using delta sizes 1, 2 and 4. If it is not out of bounds the respective delta and delta size are inserted in the compressed data according to the aforementioned ways. When neither is possible, if the maximum amount of bases has not been reached the entry is added as a new base. If the base array is full, the compressor fails. If all compressors fail then the line is marked as incompressible. Otherwise, the compressed line of the compressor that generates the lowest compression size is used.

Number of Compressors

By using Flexible BDI each compressor operates at multiple delta sizes, so instead of creating 6 compressors, as in the original approach ($Base8\Delta1, Base8 - \Delta2, Base8 - \Delta4, Base4 - \Delta1, Base4 - \Delta2, Base2 - \Delta1$), one might be tempted to use 3 compressors ($Base8 - \Delta4, Base4 - \Delta2, Base2\Delta1$). Although this is viable, it reduces the compression ratio.

This happens because when the compressor still has not acquired its first non-zero base, if the difference between the current entry and the Base Zero is less or equal to the maximum delta of the compressor, it will add this delta with a high delta size, instead of using an empty base slot for it.

Let us assume the cache line of Figure 4.6. The first four entries require a delta size of 1 byte relative to the Base Zero. The fifth entry, however, has a delta 2 bytes away from zero. The dynamic compressor will then add this entry as a new entry relative to Base Zero instead of using the unused base slot. Consequently all entries relative to zero must now use 2 bytes instead of one.

Nonetheless, if more compressors are used, all using different maximum delta sizes, the compressor with smaller maximum delta size will not be able to use Base Zero as the base for the fifth entry, and therefore will add the entry to the base slot, resulting in a better compression (Figure 4.7).

We compare the effects of using different number of compressors in Chapter 5.

4.1.4 Encoding

BDI-RB

The original encoding for the Base-Delta-Immediate compression (presented in Table 4.1) must be expanded or modified to allow the extra metadata to be read correctly. We propose 3 new encodings for the Base-Delta-Immediate Relative to Bases and 1 for the Base-Delta-Immediate Relative to Deltas.

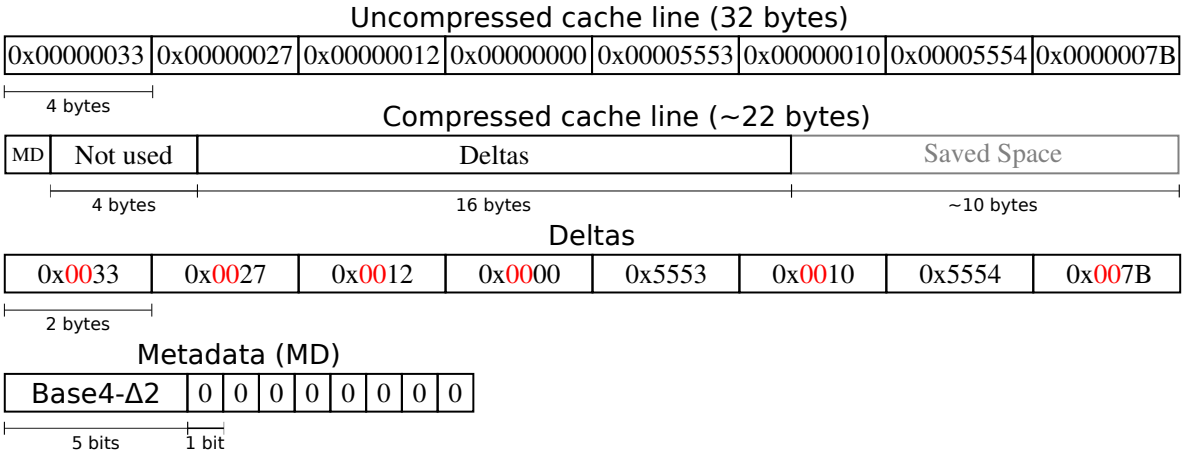


Figure 4.6: Example of compression using one compressor per base size.

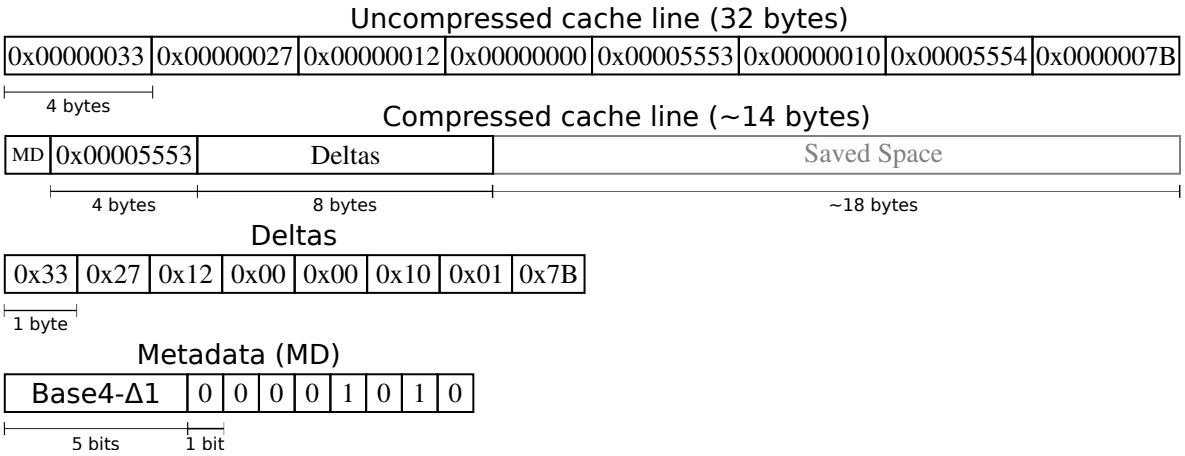


Figure 4.7: Example of compression using multiple compressors per base size.

The variables used in this section are described as follows: N is the number of entries in the cache line, and is given by $N = CacheLineSize/BaseSize$; BS is the base size; DS is the list of delta sizes, each entry respective to a base when using BDI-RB, and the possible values for delta sizes when using BDI-RD; The *Min Size* and *Max Size* entries take into account metadata bits, represent the minimum and maximum compressed data sizes in bits, respectively, and are relative to a 32/64-byte cache line. *Value* is the encoding opcode.

If only two bases are used, the number of possible combinations of bases and deltas is 14: 9 relative to base 8, 4 relative to base 4, and 1 relative to base 2. There are also 3 encodings that are not represented by the previous combinations: Zeros, Rep Values and Uncompressed. Therefore, 17 encodings are needed to represent all these possibilities. The original encoding uses 4 bits, but there are 7 unused entries, so we can use these entries to store the new encodings. In order to simplify comprehension, the encodings given in table 4.2 were chosen.

All possibilities are represented using four bits, but whenever a compression uses a 4-byte base and different delta sizes, an extra bit is used to inform whether Base Zero has a delta size of 1 byte and the other base of 2 bytes (0), or if the opposite is true (1).

Name	BS	DS	Min Size	Max Size	Value
Zeros	0	0	4/4	4/4	0000
Rep Values	64	0	68/68	68/68	0001
Base8-Δ1	64	8	104/140	104/140	0010
Base8-Δ2	64	16	136/204	136/204	0011
Base8-Δ4	64	32	200/332	200/332	0100
Base4-Δ1	32	8	108/180	108/180	0101
Base4-Δ2	32	16	172/308	172/308	0110
Base2-Δ1	16	8	164/308	164/308	0111
Uncompressed	64	0	260/516	260/516	1000

Table 4.1: Original BDI encoding.

This is done due to the fact that there is one more encoding than possible values in 4-bit representation. The equations to calculate *Min Size* and *Max Size* are given by Equations 4.1 and 4.2, respectively.

Name	BS	DS	EB	Min Size	Max Size	Value
Zeros	0	0	N/A	4/4	4/4	0000
Rep Values	64	0	N/A	68/68	68/68	0001
Base8-Δ1	64	8	N/A	104/140	104/140	0010
Base8-Δ2	64	16	N/A	136/204	136/204	0011
Base8-Δ4	64	32	N/A	200/332	200/332	0100
Base4-Δ1	32	8	N/A	108/180	108/180	0101
Base4-Δ2	32	16	N/A	172/308	172/308	0110
Base2-Δ1	16	8	N/A	164/308	164/308	0111
Uncompressed	64	0	N/A	260/516	260/516	1000
Base8-Δ1Δ2	64	8, 16	N/A	112/148	128/196	1001
Base8-Δ2Δ1	64	16, 8	N/A	112/148	128/196	1010
Base8-Δ1Δ4	64	8, 32	N/A	128/164	176/308	1011
Base8-Δ4Δ1	64	32, 8	N/A	128/164	176/308	1100
Base8-Δ2Δ4	64	16, 32	N/A	152/220	184/316	1101
Base8-Δ4Δ2	64	32, 16	N/A	152/220	184/316	1110
Base4D	32	8, 16	0	117/189	165/301	1111
Base4D	32	16, 8	1	117/189	165/301	1111

Table 4.2: Encoding for the BDI relative to bases, version 1.

$$MinSize \begin{cases} 4 + BS & Zeros, RepValues \\ 4 + N + BS + min(DS) * (N-1) + 1 * max(DS) & BaseZ - X \\ 4 + 1 + N + BS + min(DS) * (N-1) + 1 * max(DS) & Base4D \\ 4 + BS * N & Uncompressed \end{cases} \tag{4.1}$$

$$MaxSize \begin{cases} 4 + BS & Zeros, RepValues \\ 4 + N + BS + max(DS) * (N-1) + 1 * min(DS) & BaseZ - X \\ 4 + 1 + N + BS + max(DS) * (N-1) + 1 * min(DS) & Base4D \\ 4 + BS * N & Uncompressed \end{cases} \tag{4.2}$$

Another possibility is to encode the base sizes and whether the delta sizes are equal or different, and represent the delta sizes apart from the encoding. This allows the encoding size to be reduced to 3 bits, as there will only be 8 possible encodings: *Zeros*, *Repeated Values*, *Uncompressed*, base 8 with equal deltas (*Base8E*), base 4 with equal deltas (*Base4E*), base 2 with equal deltas (*Base2E*), base 8 with different deltas (*Base8D*), and base 4 with different deltas (*Base4D*).

In order to represent the delta size both base 8 and base 4 encodings must have extra bits (EB). This is not necessary for base 2 because it only has one possible delta size value: 1 byte. There are 2 delta sizes for Base4E, which can be represented by 1 bit. Base8E, however, has 3 possible delta sizes, so it needs 2 extra bits. Base4D has 2 possible combinations of delta sizes: Base Zero with a delta size of 1 byte and the other base with a delta size of 2, and its inverse, therefore it also only needs 1 extra bit. Lastly, Base8D has 6 possible combinations and therefore needs 3 extra bits. These encodings and combinations are shown in Table 4.3, and Equations 4.3 and 4.4 demonstrate how the compressed sizes were calculated.

By using 3 bits for the encoding, all occurrences of Zeros, Rep Values, $Base2 - \Delta 1$ and Uncompressed are compressed more (by 1 bit). As the Base4X encodings need 1 extra bit for delta size representation, its size will always be less or equal to the original BDI's $Base4 - \Delta 1$ and $Base4 - \Delta 2$. Base8E, however is the equivalent of $Base8 - \Delta 1$, $Base8 - \Delta 2$ and $Base8 - \Delta 4$ and uses 1 extra bit when compared to it. The 2 extra bits needed by the Base8D are compensated by the bytes saved by the dynamic compression.

$$MinSize \begin{cases} 3 + BS & Zeros, RepValues \\ 3 + EB + N + BS + min(DS) * (N-1) + 1 * max(DS) & BaseZDiff, BaseZEqual \\ 3 + BS * N & Uncompressed \end{cases} \tag{4.3}$$

$$MaxSize \begin{cases} 3 + BS & Zeros, RepValues \\ 3 + EB + N + BS + max(DS) * (N-1) + 1 * min(DS) & BaseZDiff, BaseZEqual \\ 3 + BS * N & Uncompressed \end{cases} \tag{4.4}$$

From Figure 4.8 we can see that the samples that remain uncompressed and that consist of zero entries represent 68% of the total encoding frequency for the original BDI. Therefore, if we substitute the fixed-length encodings by variable-length ones that favor these encodings, we can increase the overall compression ratio.

Table 4.4 explores that observation by using two bits to represent the encoding of these two most frequent encodings. We maintain the 4 bits needed by the original base-delta encodings to simplify comparison and add the encodings with different deltas with 7 bits-long opcodes. Equations 4.5 and 4.6 calculate the minimum and maximum size for each encoding, respectively.

Name	BS	DS	Min Size	Max Size	Value	Num EB	EB
Zeros	0	0	3/3	3/3	000	0	N/A
Rep Values	64	0	67/67	67/67	001	0	N/A
Base8D	64	8, 16	114/150	130/198	010	3	000
Base8D	64	16, 8	114/150	130/198	010	3	001
Base8D	64	8, 32	130/166	178/310	010	3	010
Base8D	64	32, 8	130/166	178/310	010	3	011
Base8D	64	16, 32	154/222	186/318	010	3	100
Base8D	64	32, 16	154/222	186/318	010	3	101
Base4D	32	8, 16	116/188	164/300	011	1	0
Base4D	32	16, 8	116/188	164/300	011	1	1
Base8E	64	8	105/141	105/141	100	2	00
Base8E	64	16	137/205	137/205	100	2	01
Base8E	64	32	201/333	201/333	100	2	10
Base4E	32	8	108/180	108/180	101	1	0
Base4E	32	16	172/308	172/308	101	1	1
Base2E	16	8	163/307	163/307	110	0	N/A
Uncompressed	64	0	259/515	259/515	111	0	N/A

Table 4.3: Encoding for the BDI relative to bases, version 2. *Num EB* is the number of extra bits needed to store the delta sizes and *EB* is the value of these extra bits.

Name	BS	DS	Min Size	Max Size	Value
Zeros	0	0	2/2	2/2	00
Uncompressed	64	0	258/514	258/514	01
Rep Values	64	0	68/68	68/68	1000
Base8-Δ1	64	8	104/140	104/140	1001
Base8-Δ2	64	16	136/204	136/204	1010
Base8-Δ4	64	32	200/332	200/332	1011
Base4-Δ1	32	8	108/180	108/180	1100
Base4-Δ2	32	16	172/308	172/308	1101
Base2-Δ1	16	8	164/308	164/308	1110
Base8-Δ1Δ2	64	8, 16	115/151	131/199	1111000
Base8-Δ2Δ1	64	16, 8	115/151	131/199	1111001
Base8-Δ1Δ4	64	8, 32	131/167	179/311	1111010
Base8-Δ4Δ1	64	32, 8	131/167	179/311	1111011
Base8-Δ2Δ4	64	16, 32	155/223	187/319	1111100
Base8-Δ4Δ2	64	32, 16	155/223	187/319	1111101
Base4-Δ1Δ2	32	8, 16	119/191	167/303	1111110
Base4-Δ2Δ1	32	16, 8	119/191	167/303	1111111

Table 4.4: Encoding for the BDI relative to bases, version 3.

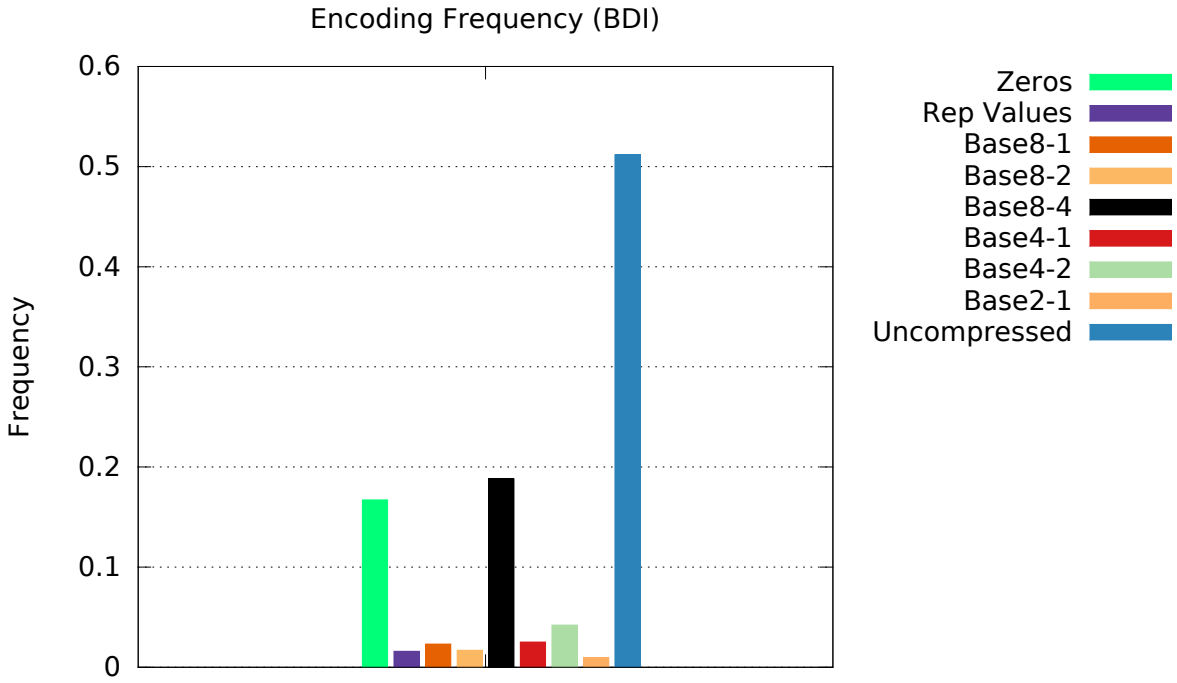


Figure 4.8: Original BDI encoding frequency (in %).

$$MinSize \begin{cases} 2 & Zeros \\ 4+BS & RepValues \\ 4+N+BS+min(DS)*(N-1)+1*max(DS) & BaseZ-\Delta X \\ 7+N+BS+min(DS)*(N-1)+1*max(DS) & BaseZ-\Delta X\Delta Y \\ 2+BS*N & Uncompressed \end{cases} \quad (4.5)$$

$$MaxSize \begin{cases} 2 & Zeros \\ 4+BS & RepValues \\ 4+N+BS+max(DS)*(N-1)+1*min(DS) & BaseZ-\Delta X \\ 7+N+BS+max(DS)*(N-1)+1*min(DS) & BaseZ-\Delta X\Delta Y \\ 2+BS*N & Uncompressed \end{cases} \quad (4.6)$$

BDI-RD

Table 4.5 presents the encoding used by the BDI relative to deltas. Only 3 bits are needed for the encoding representation. The minimum size for a encoding is given by Equation 4.7 and the maximum size is given by Equation 4.8, where hDS is the highest possible delta size and $shDS$ is the second highest possible delta size for that given encoding. For example, for Base8Diff the hDS is 32, and $shDS$ is 16.

$$MinSize \begin{cases} 3+BS & Zeros, RepValues \\ 3+N+BS+DSS*NumDS+(NumDS-1)*8+1*16 & BaseZDiff \\ 3+N+BS+DSS+N*8 & BaseZEqual \\ 3+BS*N & Uncompressed \end{cases} \quad (4.7)$$

Name	BS	DS	DSS	Num DS	Min Size	Max Size	Value
Zeros	0	0	0	0	3/3	3/3	000
Rep Values	64	0	0	0	67/67	67/67	001
Base8D	64	8/16/32	2	8	119/163	191/331	010
Base4D	32	8/16	1	16	123/203	171/323	011
Base8E	64	8/16/32	2	1	111/141	207/333	100
Base4E	32	8/16	1	1	108/180	172/308	101
Base2E	16	8	0	0	163/307	163/307	110
Uncompressed	64	0	0	0	259/515	259/515	111

Table 4.5: Encoding for the BDI relative to deltas. *DSS* is the size used by a delta size. *Num DS* is the number of delta size entries needed by the encoding.

$$MaxSize \begin{cases} 3 + BS & Zeros, RepValues \\ 3 + N + BS + DSS * NumDS + (NumDS - 1) * hDS + 1 * shDS & BaseZDiff \\ 3 + N + BS + DSS + N * hDS & BaseZEqual \\ 3 + BS * N & Uncompressed \end{cases} \tag{4.8}$$

4.1.5 Operations

Compression

The compressors for the BDI-RB technique are more complex than the ones used in BDI, as the delta size of each delta is not known until the last delta is parsed. They can be divided in two steps: *delta calculation* and *output generation*. *Delta calculation* is done in N steps, N being the number of entries in the cache line. If an entry is within one of the possible delta sizes then it is added to the delta array, and the lowest of the successful delta sizes is compared to the current delta size of the successful base, and if it is bigger the current delta size is updated with the new delta size. If the entry is not within the delta limits the base is added to the base array, if there is still a vacant slot. Otherwise the compressor fails.

When all entries are parsed the *output generation* step starts by using one of the following two approaches: use N cycles and a shift adder to insert the deltas obeying their delta sizes, or use a mapper to simultaneously insert all deltas in the compressed output.

BDI-RD is simpler than BDI-RB, and its compressors are similar to BDI's. Their main difference is that when parsing the last entry it has to check whether the deltas are equal or not to decide if the delta sizes array will contain one or N entries. The deltas are set using a shifter during compression.

Decompression

As in the original BDI, the decompressor is composed of adders that, for each entry, adds the corresponding base and delta. However, the location of the deltas is not fixed according to the encoding in the BDI-RB, so the position of each delta must be mapped

from the sum of delta sizes up to its position. This can be done using a shift adder and N cycles, where N is the number of deltas, or by using more hardware in a single cycle. BDI-RD's compressor is as simple as BDI's, so decompression can easily be done in a single cycle. A comparison of the hardware needed both approaches will be shown in Chapter 5.

4.2 Multiple bases (MBDI)

As noted in Figure 4.8, 51% of the data is still left uncompressed using BDI. This happens because applications have data from different types and sizes spread all over the cache lines, so a single cache line is not constrained to a single representation with a variation. The original BDI authors mentioned that if BDI is applied to multiple bases then the applications for which that condition exists can have improved compressibility, however their results showed that having more than two bases did not further improve compression.

The main problem with their approach is that all compressed data had the same amount of bases for all cases. Therefore, applications that could use more than two bases had a higher compression ratio, but the ones that did not suffered with the extra space used by the unnecessary bases. This can be overcome by allowing the amount of bases to be dynamic, that is, if we store only the bases used we can have all the advantages of multiple bases without the unwanted wasted space.

To do so extra metadata is needed to store the amount of bases, and this fixed space used is dependent of the number of entries present in the cache line, as calculated by Equation 4.10. Besides, as bitmask width is dependent of the number of bases, its size is not fixed anymore and is given by Equation 4.11. Figure 4.9 presents an overview of the MBDI compressed data.

As before, we maintain the implicit Base Zero, so in case only references to Base Zero exists in a cache line, there are no bitmask entries and the number of bases is set to 0. If there is a single non-zero base and no references to Base Zero, the number of bases is set to 1, and each bitmask entry has a width of 1 bit. For a higher number of non-zero bases the bitmask width is increases accordingly, but the space used by the number of bases is always the same.

$$NumEntries = CacheLineWidth/BaseSize \tag{4.9}$$

$$NumBasesWidth = \log(NumEntries) \tag{4.10}$$

$$BitmaskWidth = \log(NumBases) * NumEntries \tag{4.11}$$

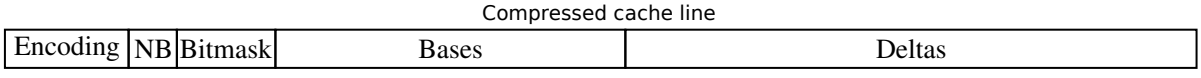


Figure 4.9: Overview of the Base-Delta-Immediate compressor with multiple bases.

4.2.1 Number of Compressors

As in the Flexible BDI solution, we can have a compressor for each base size vs delta size combination, however, due to the fact that the compressors are not constrained to two bases, we can reduce this extra hardware cost by having a compressor for each base size that uses the minimal delta size (1 byte). This saves the space used by the hardware compressors using higher delta sizes at the cost of compression ratio.

4.2.2 Encoding

Table 4.6 presents MBDI's encoding, and Table 4.7 presents the encoding used by MBDI using less compressors. The minimum and maximum size in bits of each encoding are given by Equations 4.12 and 4.13, respectively, where *EncodingWidth* is the width in bits of the encoding field, *BS* is the base size, *DS* is the delta size, *N* is the number of entries in the cache line, *NBW* is the width in bits of the field that stores the number of bases. *BW* is the maximum possible value for the width of the bitmask field in bits, and *NumBases* is the number of bases.

$$MinSize \begin{cases} EncodingWidth + BS & Zeros, RepValues \\ EncodingWidth + NBW + N * DS & BaseX \\ EncodingWidth + BS * N & Uncompressed \end{cases} \tag{4.12}$$

$$MaxSize \begin{cases} EncodingWidth + BS & Zeros, RepValues \\ EncodingWidth + NBW + BS * NumBases + BW + N * DS & BaseX \\ EncodingWidth + BS * N & Uncompressed \end{cases} \tag{4.13}$$

Name	BS	DS	NBW	BW	Min Size	Max Size	Value
Zeros	0	0	0	0/0	4/4	4/4	0000
Rep Values	64	0	0	0/0	68/68	68/68	0001
$Base8 - \Delta1$	64	8	3/4	12/32	39/72	307/616	0010
$Base8 - \Delta2$	64	16	3/4	12/32	71/136	339/680	0011
$Base8 - \Delta4$	64	32	3/4	12/32	135/264	403/808	0100
$Base4 - \Delta1$	32	8	4/5	32/80	72/137	360/729	0101
$Base4 - \Delta2$	32	16	4/5	32/80	136/265	424/857	0110
$Base2 - \Delta1$	16	8	5/6	80/192	137/266	473/970	0111
Uncompressed	64	0	0	0/0	259/515	259/515	1000

Table 4.6: MBDI Encoding. *NBW* is the width in bits of the field that stores the number of bases. *BW* is the maximum possible value for the width of the bitmask field in bits.

Notice that for both methods the maximum possible sizes for the base encodings are bigger than the uncompressed cache line, therefore they will never use an amount of bases equal to the number of entries. Tables 4.8 and 4.9 show the maximum number of bases that still allow the compressor to have a better compression size than the original cache line.

Name	BS	DS	NBW	BW	Min Size	Max Size	Value
Zeros	0	0	0	0/0	3/3	3/3	000
Rep Values	64	0	0	0/0	67/67	67/67	001
Base8	64	8	3/4	12/32	38/71	306/615	010
Base4	32	8	4/5	32/80	71/136	359/728	011
Base2	16	8	5/6	80/192	136/265	472/969	100
Uncompressed	64	0	0	0/0	259/515	259/515	101

Table 4.7: MBDI encoding using less compressors. *NBW* is the width in bits of the NumberBases field. *BW* is the maximum possible value for the width of the bitmask field in bits.

Name	MaxNumBases
Zeros	N/A
Rep Values	N/A
$Base8 - \Delta1$	3/6
$Base8 - \Delta2$	2/5
$Base8 - \Delta4$	1/3
$Base4 - \Delta1$	5/9
$Base4 - \Delta2$	3/6
$Base2 - \Delta1$	4/7
Uncompressed	N/A

Table 4.8: Maximum number of non-zero bases so that the compressed size is still better than the uncompressed data for the MBDI.

Name	MaxNumBases
Zeros	N/A
Rep Values	N/A
Base8	3/6
Base4	5/9
Base2	4/7
Uncompressed	N/A

Table 4.9: Maximum number of non-zero bases so that the compressed size is still better than the uncompressed data for the MBDI-LC.

Therefore, the sizes of the NBW and BW fields should change to accommodate these maximum values in order not to waste space, as shown in Tables 4.10 and 4.11.

Name	BS	DS	NBW	BW	Min Size	Max Size	Value
Zeros	0	0	0	0/0	4/4	4/4	0000
Rep Values	64	0	0	0/0	68/68	68/68	0001
$Base8 - \Delta1$	64	8	2/3	8/24	38/71	238/479	0010
$Base8 - \Delta2$	64	16	2/3	8/24	70/135	206/479	0011
$Base8 - \Delta4$	64	32	1/2	4/16	133/262	201/470	0100
$Base4 - \Delta1$	32	8	3/4	24/64	71/136	255/488	0101
$Base4 - \Delta2$	32	16	2/3	16/48	134/263	246/503	0110
$Base2 - \Delta1$	16	8	2/3	48/96	135/263	247/471	0111
Uncompressed	64	0	0	0/0	259/515	259/515	1000

Table 4.10: Encoding for the MBDI with fixed maximum base sizes. NBW is the width in bits of the NumberBases field. BW is the maximum possible value for the width of the bitmask field in bits.

Name	BS	DS	NBW	BW	Min Size	Max Size	Value
Zeros	0	0	0	0/0	3/3	3/3	000
Rep Values	64	0	0	0/0	67/67	67/67	001
Base8	64	8	2/3	8/24	37/70	237/478	010
Base4	32	8	3/4	24/64	70/135	254/487	011
Base2	16	8	3/3	48/96	134/262	246/470	100
Uncompressed	64	0	0	0/0	259/515	259/515	101

Table 4.11: Encoding for the MBDI using less compressors and fixed maximum base sizes. NBW is the width in bits of the NumberBases field. BW is the maximum possible value for the width of the bitmask field in bits.

Note that the dynamic base-delta-immediate relative to deltas and the multiple base-delta-immediate solutions could be combined to further improve compression. We call it MBDI-RD, and its results are shown in the next chapter.

Chapter 5

Experimental Results

5.1 Methodology

In order to emulate typical working conditions, ZSim [45], a x86-64 simulator, was used. ZSim is a fast, scalable and accurate multi-core architectural simulator that uses dynamic binary translation to perform instruction decoding and timing analysis during the instrumentation phase and thus reduce the amount of tasks to be repeated during simulation. We chose ZSim because it outperforms current popular simulators like Sniper [10], which uses approximation techniques to do timed interval instead of instruction-wise simulation, and gem5 [9], which emulates the instructions, that is, it decodes and invokes functional and timing models for every instruction.

All the compression methods were implemented in C++ and Verilog, and these implemented classes and modules provide ways to perform compression and decompression of cache lines. The validation stage was performed at the Computer Systems Laboratory of the Institute of Computing at Unicamp, as it provides ZSim and a cluster, which allows high parallel computation, and thus faster results.

The processor design was the same for all compression schemes: single core, two cache levels, compressors only on LLC. L1 and L2 cache configuration was 32KB and 2MB, respectively, and both caches consist of 16 ways, as in the original BDI paper. The systems were simulated using ZSim and tested using Simpoints [40] on the benchmarks from SPEC 2006 [22]. Tests focused on common computer operations under the same environmental conditions.

As a final step the results from the systems were compared regarding *1)* data compression, that is, which system compressed more; *2)* power efficiency, that is, which system used the least amount of power to finish the tasks; *3)* chip die area used, in other words, how much space is required to synthesize each compressor; *4)* performance: difference in IPC and MPKI for simulated executions; and *5)* implementation complexity, a comparison between the difficulties and trade-offs of the systems. Usage validness of the Dynamic Base-Delta-Immediate Compressor was determined via an analysis of these factors to verify at which conditions it could be a better scheme than the ones being used in processors nowadays.

To test data compression, the benchmarks from SPEC2006 were executed using SimPoints, and for every cache access, all techniques were applied simultaneously to the cache

line in order to find their compression ratio. For area and power efficiency evaluation we used Cadence RTL Compiler to synthesize the verilog modules implemented. We left the synthesis and evaluation of the MBDI-RD module to future work, as we only have a behavioral model of the technique. We used a 15nm technology [34] and a clock frequency of 500MHz (although most of the methods can achieve higher clock frequencies, BDI-RB is limited to this value, so we used it as default), and compared the techniques regarding number of compressors, and evaluated BDI-RD using slow and fast decompression, that is, same amount of cycles as compression vs 1-cycle. We only evaluated BDI-RB using a slow (multiple-cycle) decompression due to the higher complexity of the faster approach and the low compression ratio improvement presented by this technique. To test the performance improvements (IPC and MPKI), we ran ZSim with all SimPoints using each of the compression methods and compared the results. In order to calculate system performance, all SimPoints were executed for each of the compression techniques, and the resulting MPKI and IPC were taken.

The benchmark means were calculated by applying a geometric mean among the benchmark's Simpoint results. The general means were calculated by applying a geometric mean among the benchmark mean values.

5.2 Data compression

Figure 5.1 presents the encoding frequency for all the encoding versions proposed for the Base-Delta-Immediate Relative to Bases. They do not differ much from each other, as the likelihood of changing encoding due to the saved/extra 1-3 bits difference is low. Notice that although the real encoding values for version 2 is completely different from the others, it has been translated to allow comparison. This was done by applying the delta sizes directly to the encoding, *i.e.* a cache line with encoding $Base8D$, and delta sizes of 16 and 32 is mapped to $Base8 - \Delta2\Delta4$.

There has been a significant reduction on the frequency of $Base8-\Delta2$, $Base8-\Delta4$, and $Base4-\Delta2$ when compared to the original results (Figure 4.8), and a minor reduction for the $Base4-\Delta1$ and $Base2-\Delta1$ encodings. As expected, *Zeros*, *Rep Values*, $Base8-\Delta1$, and *Uncompressed* did not suffer any modifications due to the fact that $Base8 - \Delta1$ is already the best compression with a base for a 64-byte cache line and that the other three encodings are not affected by the method (except for a minor encoding size change).

The transitions from the original encodings to the dynamic encodings are shown on Figures 5.2, 5.3, and 5.4, and represent the percentage of original opcode (horizontal axis) that became another opcode using the proposed encoding. For example, in Figure 5.2, 19% of the compressed lines that originally used the $Base8 - \Delta2$ became $Base8 - \Delta2\Delta1$ using BDI-RB version 1, as the second base did not need the second byte to represent its deltas.

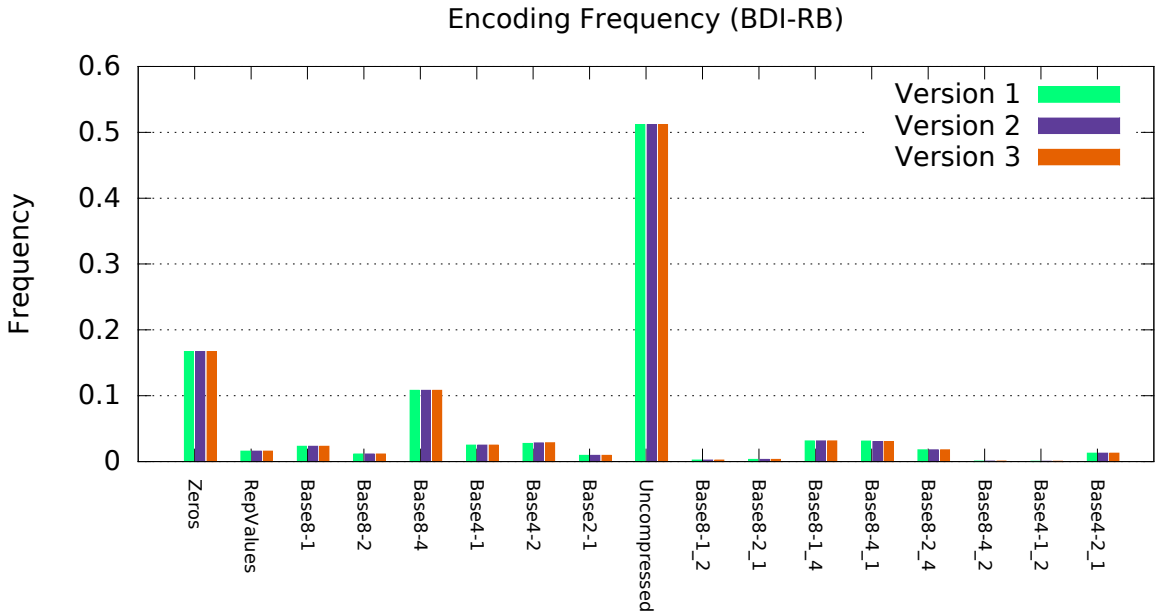


Figure 5.1: BDI-RB encoding frequency (in %).

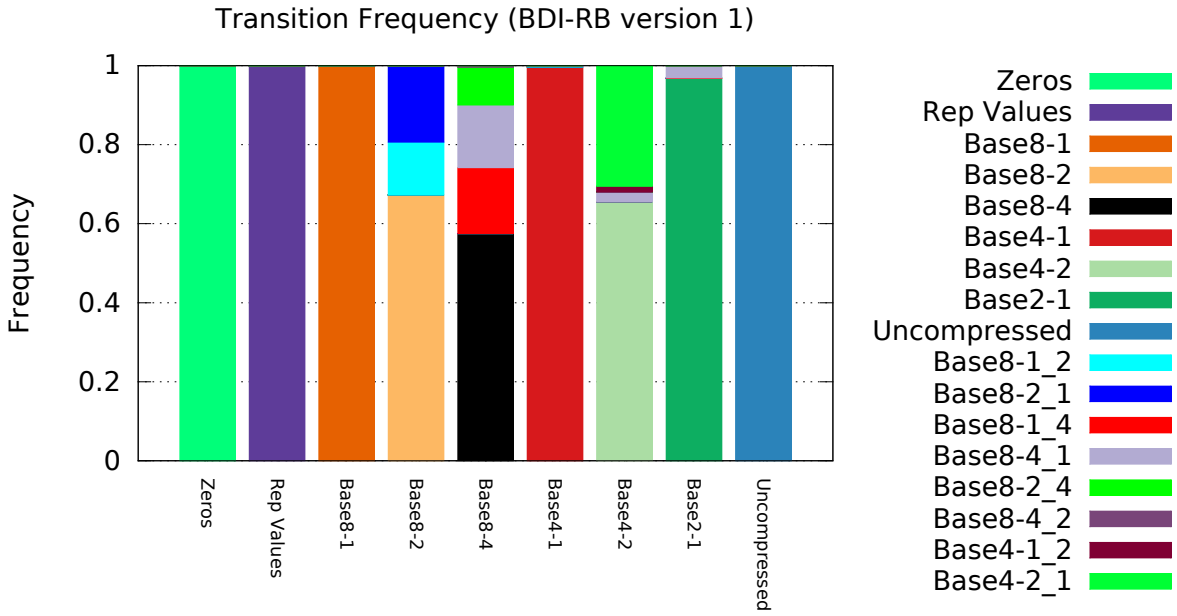


Figure 5.2: BDI-RB version 1 transition frequency (in %).

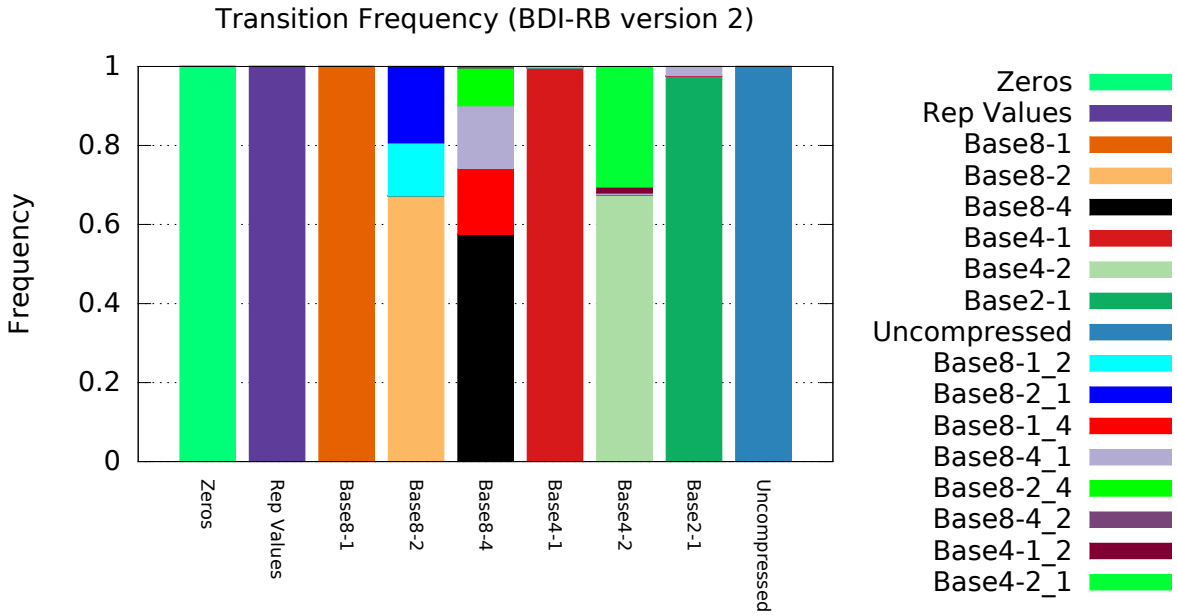


Figure 5.3: BDI-RB version 2 transition frequency (Translated to make it easier to compress to other encodings) (in %).

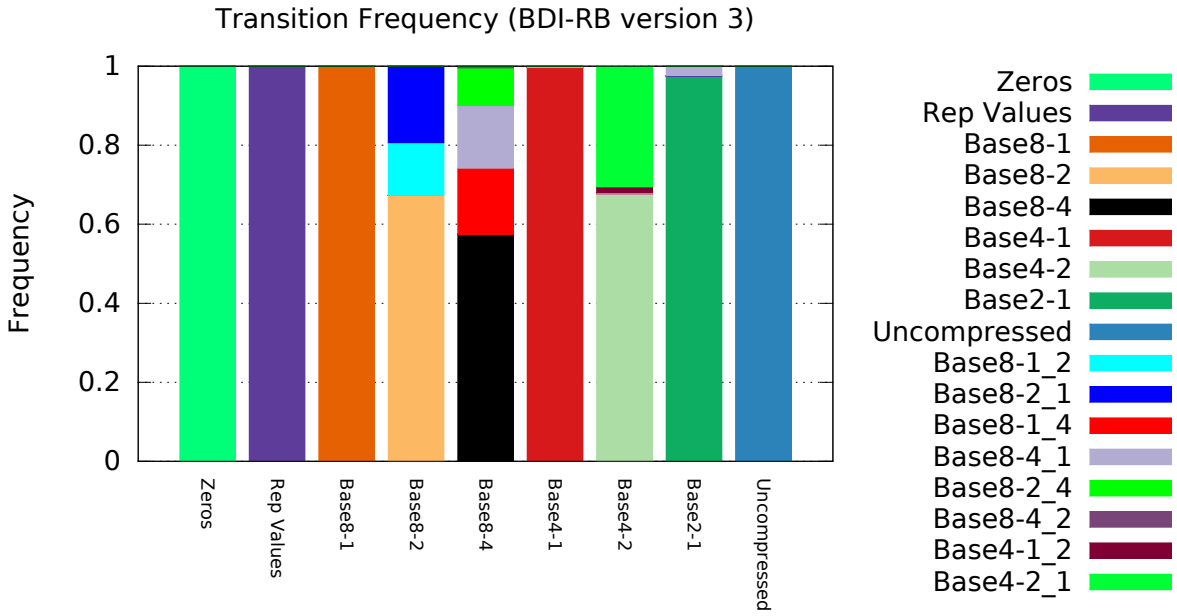


Figure 5.4: BDI-RB version 3 transition frequency (in %).

BDI-RD takes advantage of its smaller encoding width and improves compression of all *Zeros*, *Rep Values*, *Base*2Δ1 and *Uncompressed* entries by 1 bit. Besides, by allowing each delta size to be stored separately, only the necessary space for a delta entry

is used, which removes all the extra padding created by BDI. Rare $Base4\Delta2$ entries become $Base2\Delta1$ because in the original compression, when compression sizes are equal for multiple compressors, the compressor with higher base size is preferred due to its faster compression. As $Base2\Delta1$ is always at least 1 bit smaller in the BDI-RD encoding for a 64-byte cache line, there is no tie between these encodings anymore. The cache lines that were compressible by both the $Base4\Delta1$, $Base4\Delta2$, $Base8\Delta2$ and $Base8\Delta4$ can now use the compression benefits of the $Base8\Delta1$ compressor if there are few entries with a higher delta size. All transitions from BDI to BDI-RD for the benchmarks are shown in Figure 5.5.

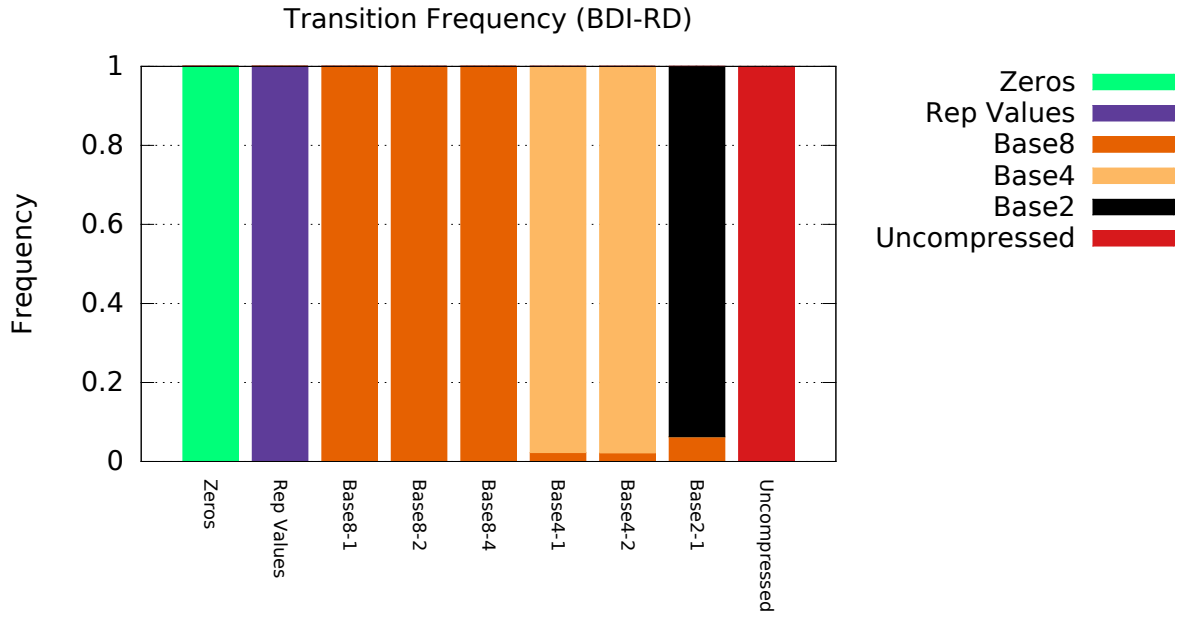


Figure 5.5: BDI-RD transition frequency (in %). Transitions from $Base4\Delta2$ to $Base2\Delta1$ correspond to only 0,00004% of the $Base4\Delta2$ transitions.

The previous methods aim on improving compression for the lines that were already compressible. MBDI, however, tackles the cache lines that remain uncompressed using the original Base-Delta-Immediate compressor. This can be seen on Figure 5.6, where the frequency of uncompressed data reduced 36%.

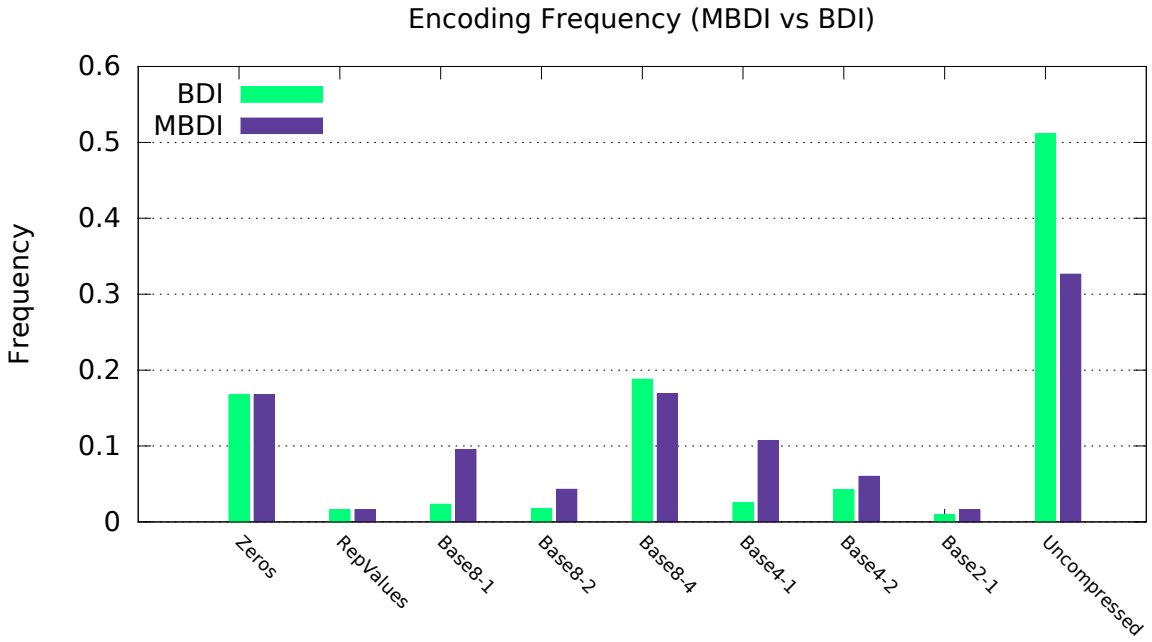


Figure 5.6: MBDI and BDI encoding frequency compared (in %).

As mentioned previously, MBDI and BDI-RD can be combined to improve compression ratio (we call it *MBDI-RD*). This reduces the number of uncompressed samples by 38% (Figure 5.7) when compared to the original BDI.

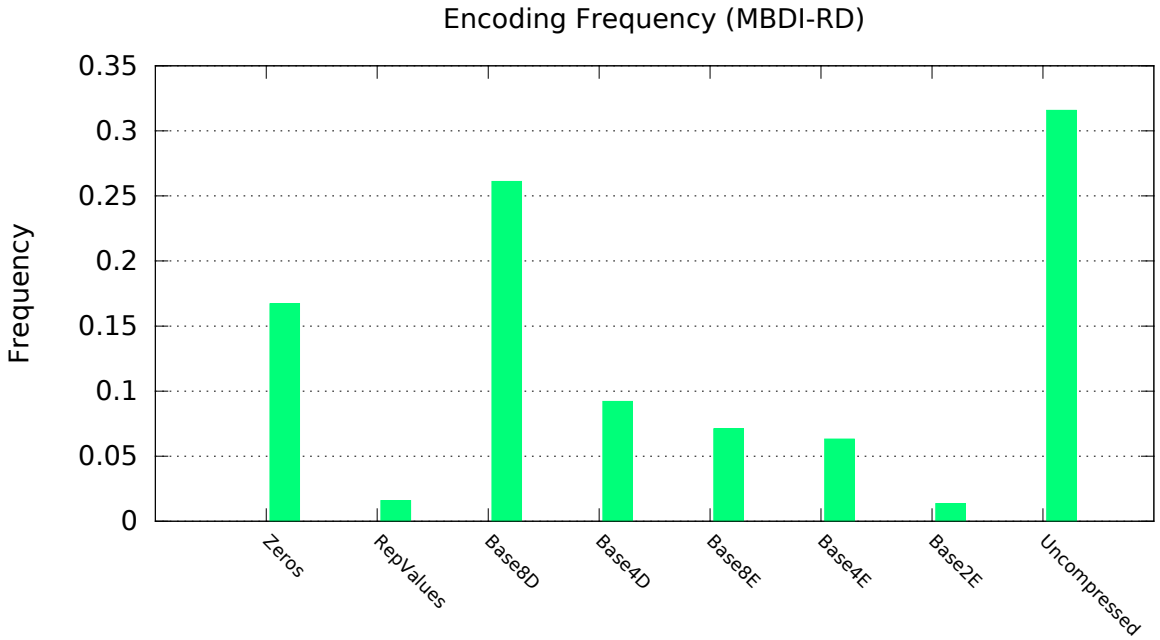


Figure 5.7: MBDI-RD encoding frequency (in %).

5.2.1 Number of compressors

As stated previously, using less compressors reduces the compression ratio. This can be seen on Figure 5.8, which represents the transitions from the original BDI compression to the BDI-RBLC version 3. The entries that were originally compressed to $Base8\Delta1$ were compressed to $Base8\Delta2$, $Base8\Delta4$, and even $Base4\Delta1$. This happens because the wrong base was saved, so the base8 compressor failed when the base that really needed to be stored was limited by the lack of base space.

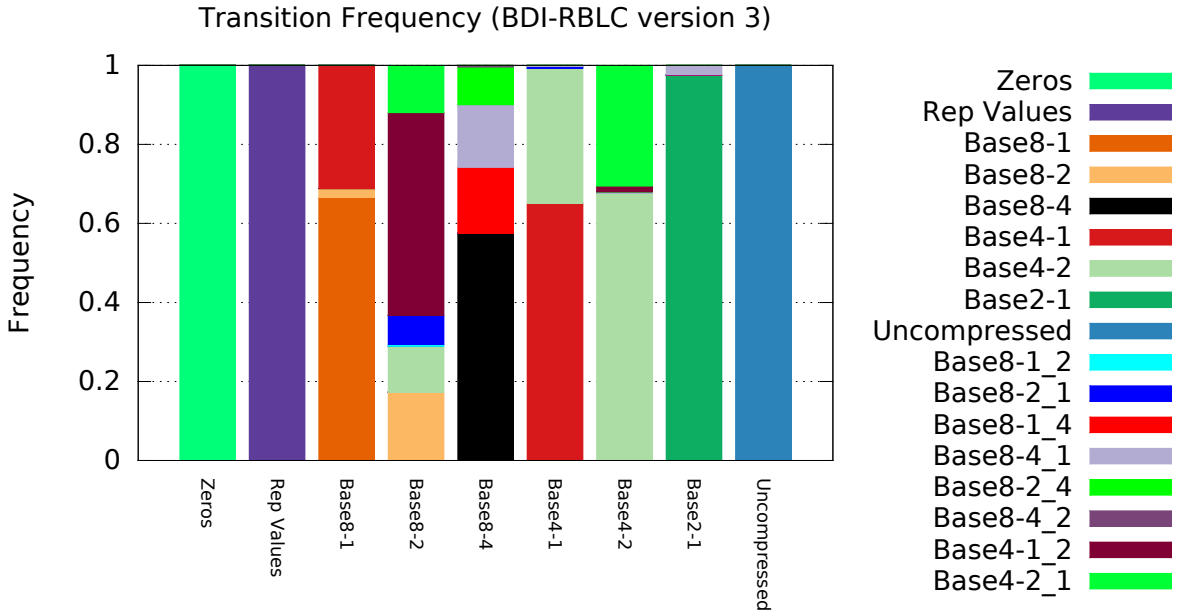


Figure 5.8: BDI-RBLC version 3 transition frequency (in %).

Varying the number of compressors does not significantly affect the compression ratio for the BDI-RB and BDI-RD (less than 1%), as can be seen on Figures 5.9 and 5.10, but it produces a difference of almost 7% for the MBDI technique (Figure 5.11).

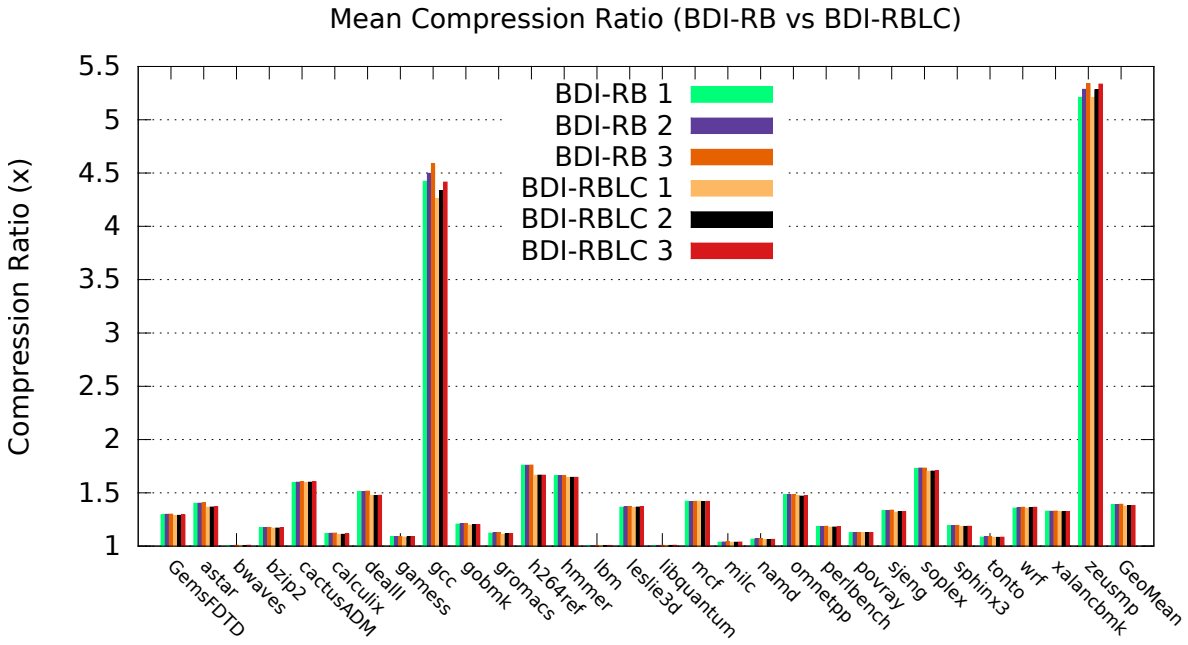


Figure 5.9: Mean compression ratio for BDI-RB using more (BDI-RB) and less (BDI-RBLC) compressors.

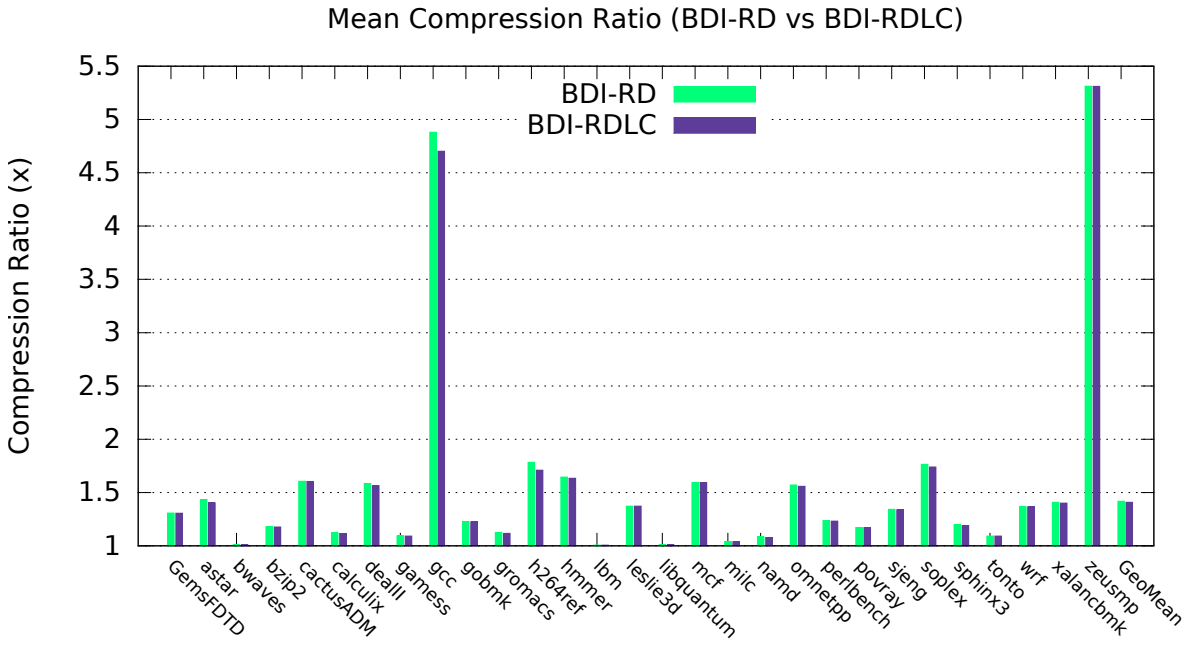


Figure 5.10: Mean compression ratio for BDI-RD using more (BDI-RD) and less (BDI-RDLC) compressors.

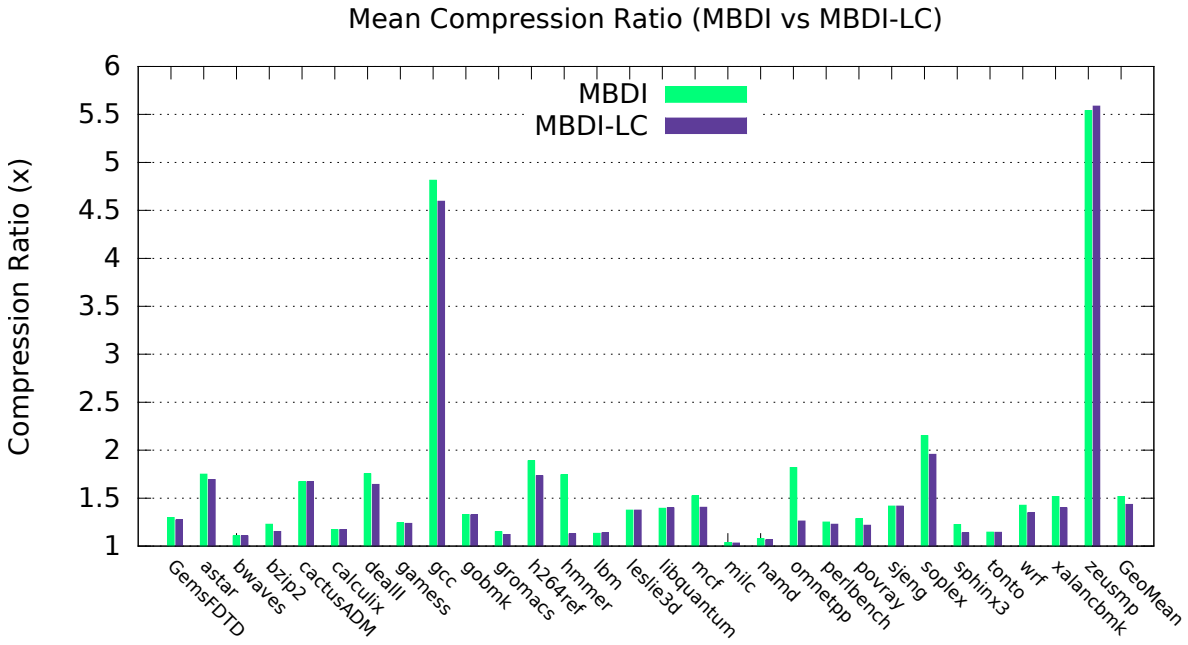


Figure 5.11: Mean compression ratio for MBDI using more (MBDI) and less (MBDI-LC) compressors.

5.2.2 Compression ratio

Figure 5.12 presents the compression ratio for CPack, FPC, BDI and the proposed techniques. As can be seen, all the methods, except BDI-RBLC_1 and BDI-RBLC_2, manage to outperform the original Base-Delta-Immediate compressor. Among the proposed algorithms, MBDI-RD exhibit the best results, with a mean compression ratio of 1.58x, and the BDI-RB variants have the lowest improvements, increasing the original compression factor from 1.37x to 1.39x.

Cache organization techniques sometimes try to fit the compressed lines in fixed sized blocks [46][47]. The Skewed Compressed Cache, for example, stores the compressed data in superblocks of 64 bytes divided in either one 64-byte, two 32-byte, four 16-byte, or eight 8-byte blocks. This means that the compressed data is padded to fit these blocks, loosing some of its compression ratio.

Therefore, if we assume that a compressed data is padded to fit these block sizes, that is, a 24 bit compressed data uses a 8-byte block, a 100 bit compressed line uses a 16-byte block, and so on, the compression ratio for all algorithms decrease (Figure 5.12). MBDI-RD manages to keep a good compression ratio when compared to the other techniques for both maximum and block compression, so we compare it to the original BDI and C-Pack, which presented the best compression results, in Figure 5.13.

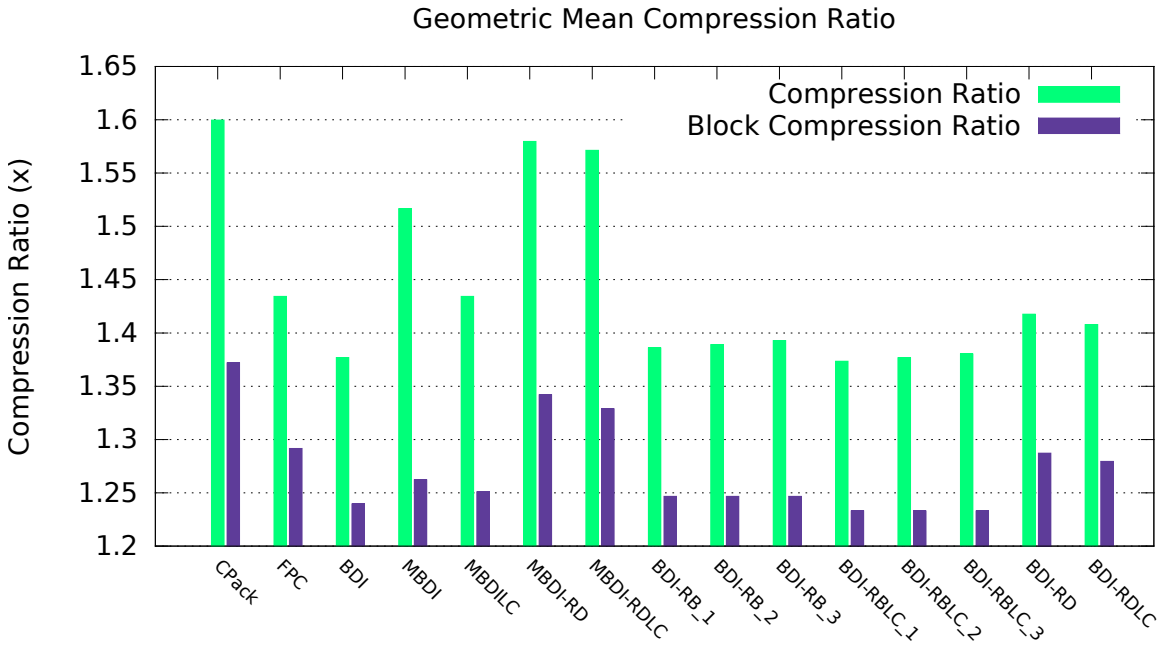


Figure 5.12: Mean (geometric) compression ratio of the techniques.

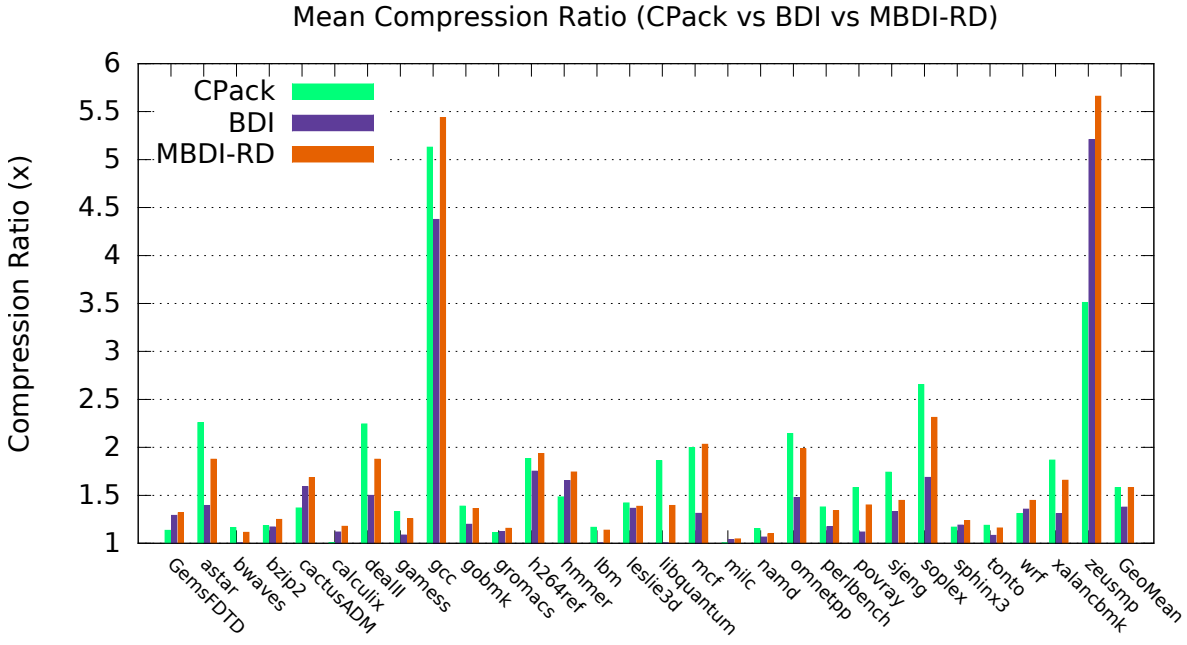


Figure 5.13: Comparison of BDI, MBDIRD and CPack for all benchmarks (geometric mean).

5.3 Power efficiency

We implemented each compression method in Verilog and used Cadence RTL Compiler tool to evaluate the power consumption. The techniques increase power usage when compared to BDI, but by using less compressors both MBDI and BDI-RD manage to stay under 2x the original power (Figure 5.14). Figure 5.15 presents the power consumption ratio of using the techniques over BDI, that is, how much compression ratio a microwatt provides compared to the original BDI. MBDI with less compressors has the lowest power cost increase without using a slow decompressor.

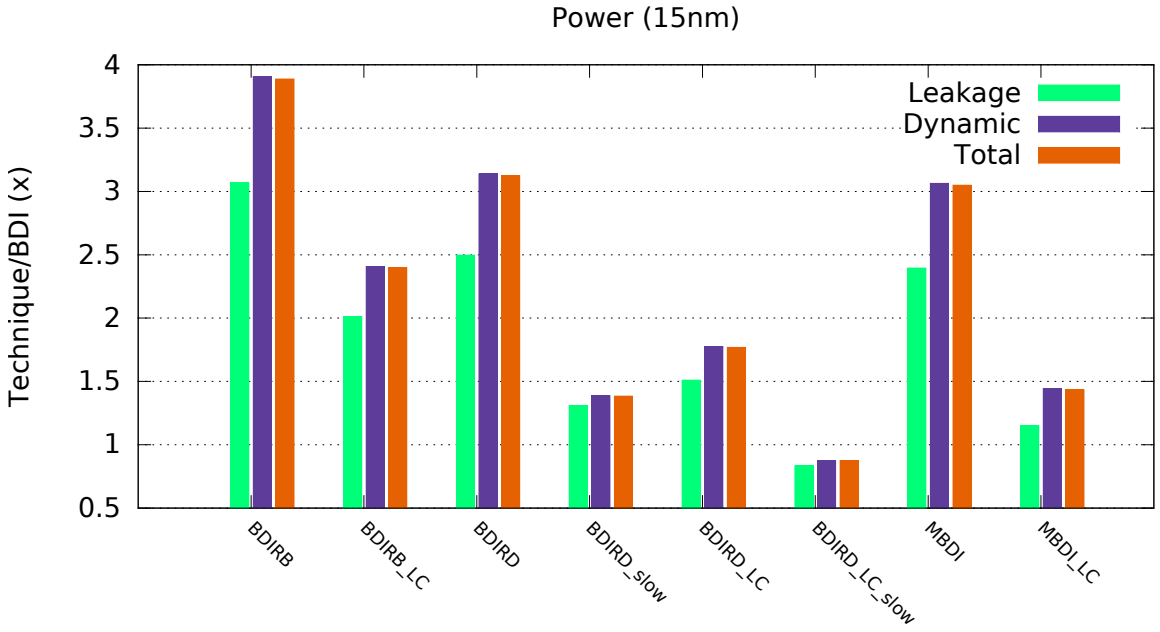


Figure 5.14: Comparison of leakage, dynamic and total power usage. Baseline is BDI power. $\_LC$ is the variant with less compressors, and $\_slow$ is the slow variant.

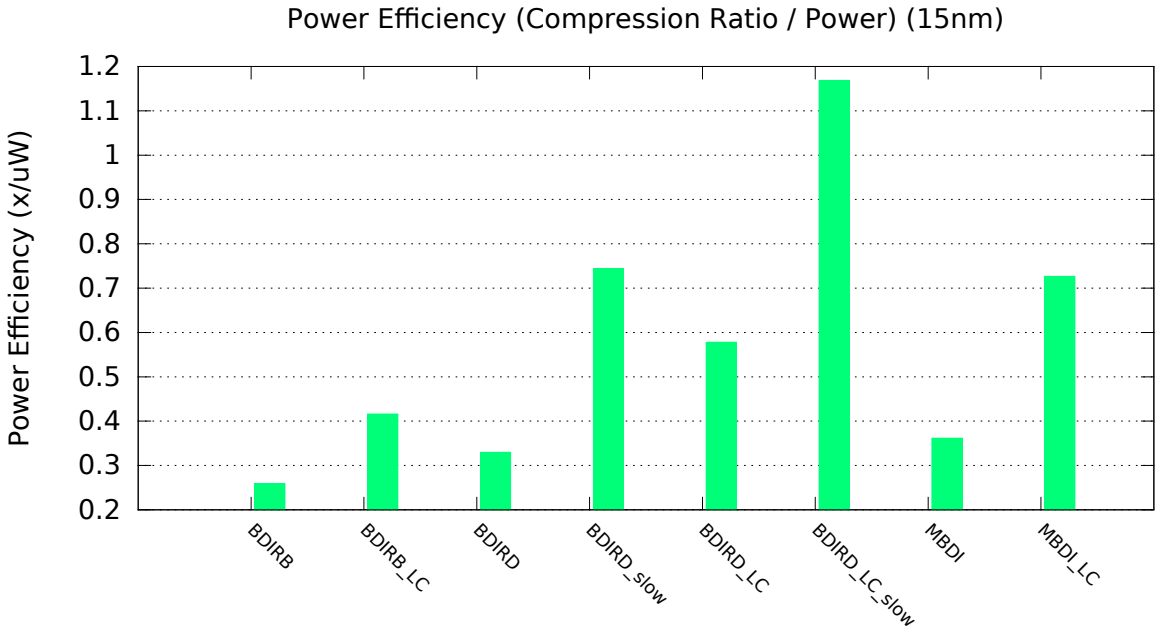


Figure 5.15: Power efficiency, that is, compression ratio divided by power consumption relative to original results. $\_LC$ is the variant with less compressors, and $\_slow$ is the slow variant.

5.4 Area

Figure 5.16 presents a comparison of area usage by CPack, FPC, BDI, BDI-RB and BDI-RD. As expected, using less compressors significantly reduces area usage, and doing a multi-cycle decompression on BDI-RD reduces die area, as operations can be done serially, so the integrated circuits can be reused. Figure 5.17 presents the area improvement ratio of using the techniques over BDI, that is, the amount of compression ratio per area unit compared to the original BDI. Both BDI-RD and MBDI using less compressors provide an area cost close to the original results without sacrificing system performance as opposed to the slow variants, which may even be better at the cost of extra latency in the critical decompression step. It is important to notice that although the compressors/decompressors can be up to 3 times the size of the original BDI compressors, **the original BDI compressors represent an increasal of only 2.3% to the area of a 16-way 2MB cache**.

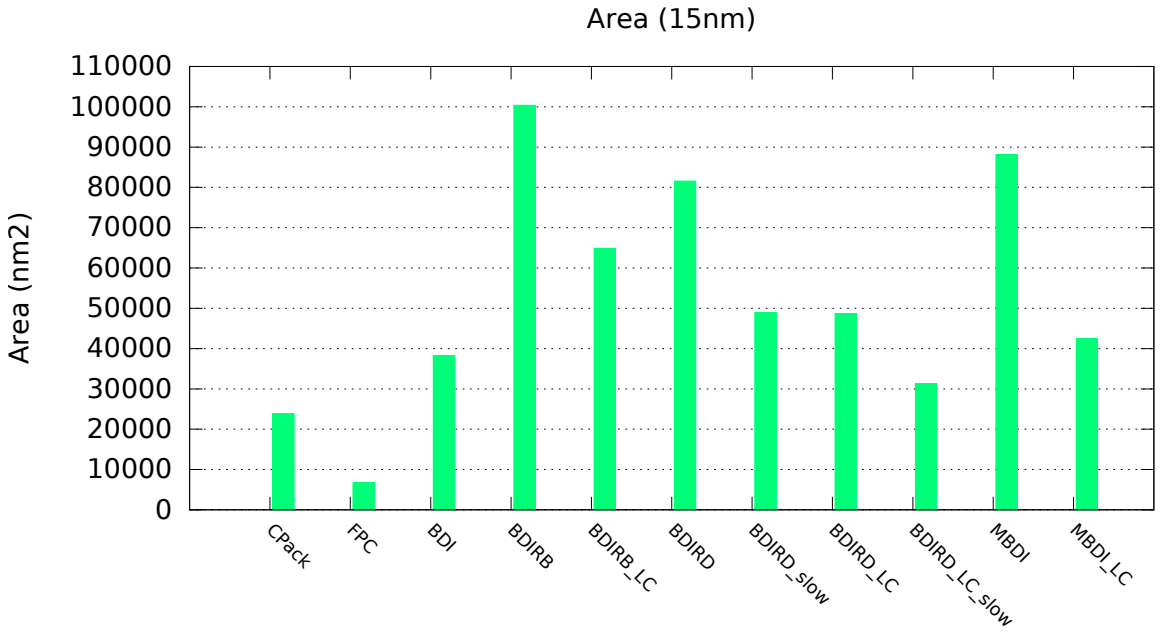


Figure 5.16: Area usage of the techniques. *_LC* is the variant with less compressors, and *_slow* is the slow variant.

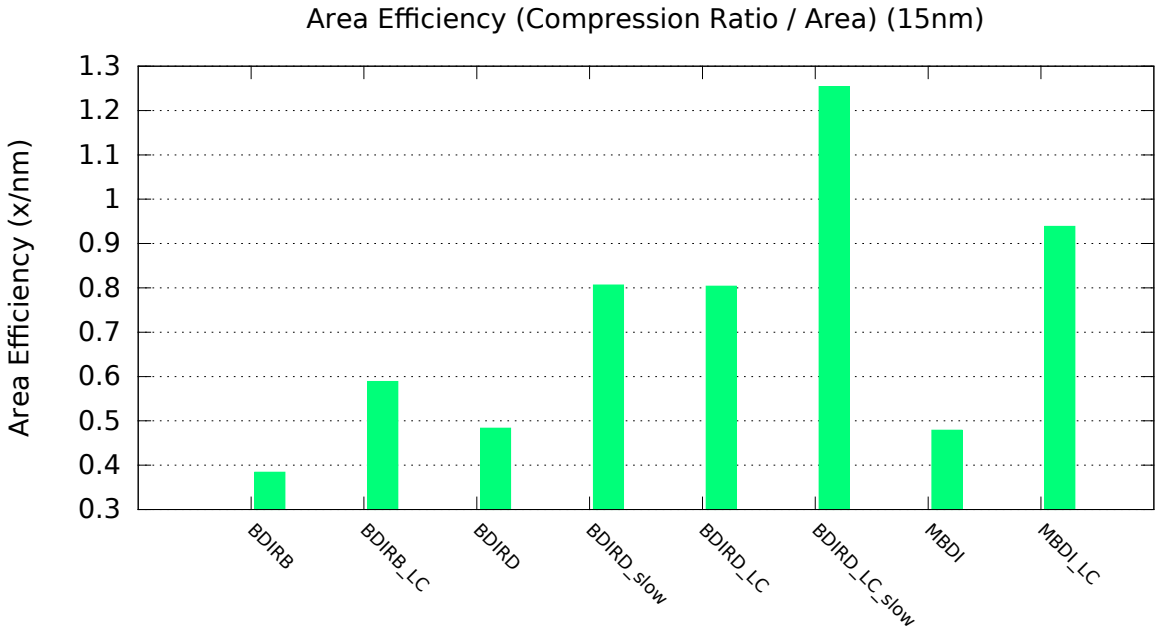


Figure 5.17: Area efficiency, that is, compression ratio divided by area usage relative to original results. *_LC* is the variant with less compressors, and *_slow* is the slow variant.

5.5 Performance analysis

Figures 5.18 and 5.19 show the performance improvements of the techniques relative to a baseline 2MB L2 cache without compression, simulated on ZSim running single-core SimPoint benchmarks. All proposed methods, except for the BDI-LC and BDIRB-LC manage to increase system IPC and MPKI, when compared to the original BDI. MBDIRD provides 95.2% of the IPC and 12.85% more MPKI than an uncompressed cache with the double of its capacity (4MB), at a minimal area and energy cost.

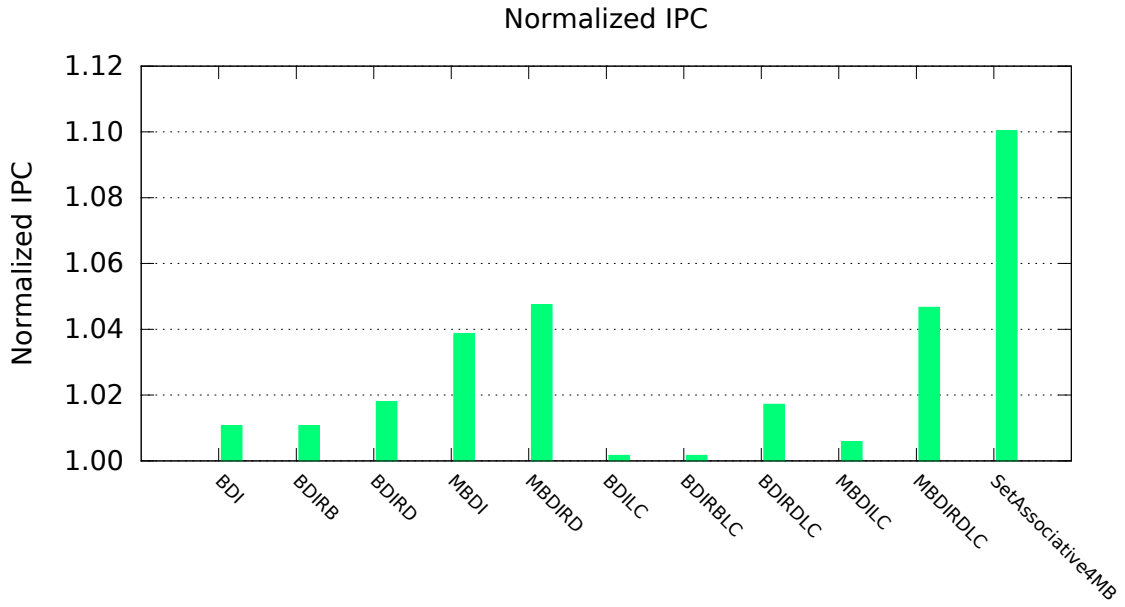


Figure 5.18: Geometric mean of IPC for all techniques normalized on a 2MB baseline cache without compression. $\_LC$ is the variant with less compressors.

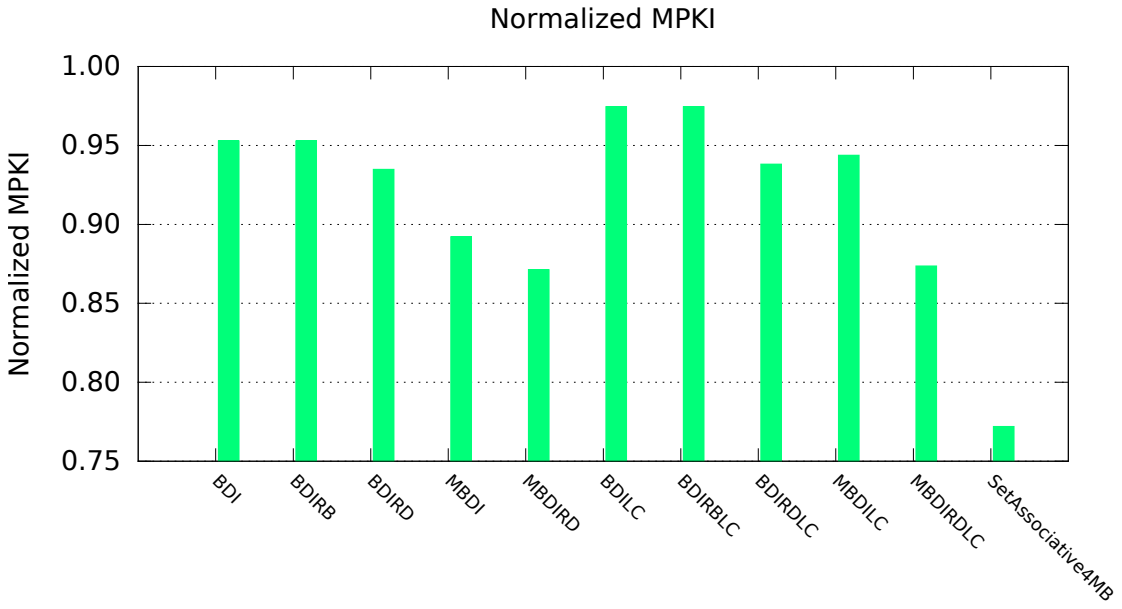


Figure 5.19: Geometric mean of MPKI for all techniques normalized on a 2MB baseline cache without compression. $\_LC$ is the variant with less compressors.

5.6 Complexity analysis

BDI-RB, due to its high power consumption and low improvement on compression ratio, is not efficient for any of its variants, even using a slow decompressor. MBDI and BDI-RD, however, manage to increase compression ratio with a small area and power cost, so they can be used to improve the original BDI. The BDI-RD with less compressors manages to keep a similar compression ratio to the full BDI-RD compressor while lowering its power consumption to 57%. MBDI-RD provides the best compression ratios among the BDI extensions, achieving results similar to CPack with a 1 cycle decompression latency.

5.7 Summary

All proposed techniques but BDI-RBLC versions 1 and 2 manage to improve the compression ratio when compared to the original BDI. BDI-RB, however, does not provide significant improvement. MBDI-RD has the best compression factor, comparable to CPack, with the advantage of having a one-cycle decompression. MBDI presents the best cost-efficiency in system performance. The less compressor (LC) variants manage to greatly reduce hardware and power needed by their full version, at the cost of compression ratio.

Chapter 6

Conclusion

In this dissertation we propose extensions and modifications to the Base-Delta-Immediate cache compression technique. Due to the diversity of data and delta sizes generalization, that is, using the highest delta size needed for compression for all delta values, BDI generates a high padding rate. The expansions presented in this work remove this delta padding by using flexible delta sizes and multiple bases to improve data compaction at a low-moderate power cost.

The main contributions of this work are: *1)* creation of techniques to remove padding in the Base-Delta-Immediate compression to improve its compression ratio; and *2)* allow using a different number of bases on BDI with marginal metadata overhead.

The first technique, BDI with delta sizes Relative to the Bases (BDI-RB), keeps a different delta size for each base instead of keeping a single delta size that is valid for all bases. Therefore, removing dependency between the bases. We suggest three encoding possibilities: one to maintain compatibility with the original encoding, which adds a single extra bit due to the number of possible combinations base-delta; one to dissociate base from delta size in the encoding area, therefore removing one bit from the encodings that do not need delta size representation; and one scheme with a variable opcode size based on opcode frequency.

In order to completely remove the need of padding we suggest keeping a delta size for each delta entry, the BDI with delta sizes Relative to Deltas (BDI-RD). This method requires extra metadata to store the delta size, but compensates with the area saved from the unnecessary paddings, resulting in a compression ratio of 1.42x (3% better than the original BDI results).

The last technique focuses on allowing the use of a number of bases different from two: Multiple bases BDI (MBDI). By adding a field to keep track of the number of bases, the compression ratio becomes slightly worse (due to the 4 extra bits) for compressed lines that in fact need two bases, but it may save up to 70 bits when only one base is needed, and allows compression of previously incompressible lines. An hybrid of the MBDI and the BDI-RD, the MBDI-RD, has also been simulated and compared to the previous techniques.

The formerly mentioned techniques, as expansions of the original Base-Delta-Immediate compression, require more hardware, and thus consume more power and die area. In order to reduce these undesired effects, we propose using a single compressor for each base

size, totaling three, instead of the original six compressors. Although this marginally reduces the compression factor due to the reduction of number of compression possibilities, hardware usage is almost halved.

Most of the presented methods manage to increase compression ratio, with the worst reaching an 1% improvement (BDI-RB), and the best 15% (MBDI-RD). The power consumption varies from 0.87x to 3.89x the original results. It has been validated that using some of the extensions with a reduced number of compressors manages to increase the compression factor at a low cost, mainly MBDI if a low overall cost design is intended.

We leave applying the techniques to a multi-core processor, and synthesizing MBDI-RD and verifying its power and area consumption for future work.

Bibliography

[1] Alaa R Alameldeen, David Wood, et al. Adaptive cache compression for high-performance processors. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 212–223. IEEE, 2004.

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

[3] James Archibald and Jean-Loup Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems (TOCS)*, 4(4):273–298, 1986.

[4] Angelos Arelakis, Fredrik Dahlgren, and Per 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.

[5] Angelos Arelakis and Per Stenstrom. A case for a value-aware cache. *Computer Architecture Letters*, 13(1):1–4, 2014.

[6] Angelos Arelakis and Per Stenstrom. Sc2: A statistical compression cache scheme. In *Proceeding of the 41st annual international symposium on Computer architecuture*, pages 145–156. IEEE Press, 2014.

[7] J-L Baer and W-H Wang. *On the inclusion properties for multi-level cache hierarchies*, volume 16. IEEE Computer Society Press, 1988.

[8] Luca Benini, Alberto Macii, Enrico Macii, and Massimo Poncino. Selective instruction compression for memory energy reduction in embedded systems. In *Proceedings of the 1999 international symposium on Low power electronics and design*, pages 206–211. ACM, 1999.

[9] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[10] Trevor E Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 52. ACM, 2011.

[11] Chin-Long Chen and MY Hsiao. Error-correcting codes for semiconductor memory applications: A state-of-the-art review. *IBM Journal of Research and Development*, 28(2):124–134, 1984.

[12] Xi Chen, Lei Yang, Robert P Dick, Li Shang, and Haris Lekatsas. C-pack: A high-performance microprocessor cache compression algorithm. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 18(8):1196–1208, 2010.

[13] Vinodh Cuppu, Bruce Jacob, Brian Davis, and Trevor Mudge. High-performance drams in workstation environments. *IEEE Transactions on Computers*, 50(11):1133–1153, 2001.

[14] Saumya Debray and William Evans. Profile-guided code compression. In *ACM SIGPLAN Notices*, volume 37, pages 95–105. ACM, 2002.

[15] Julien Dusser, Thomas Piquet, and André Seznec. Zero-content augmented caches. In *Proceedings of the 23rd international conference on Supercomputing*, pages 46–55. ACM, 2009.

[16] Magnus Ekman and Per Stenstrom. A robust main-memory compression scheme. In *ACM SIGARCH Computer Architecture News*, volume 33, pages 74–85. IEEE Computer Society, 2005.

[17] Jayesh Gaur, Mainak Chaudhuri, and Sreenivas Subramoney. Bypass and insertion algorithms for exclusive last-level caches. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 81–92, New York, NY, USA, 2011. ACM.

[18] JR Goodman and HHJ Hum. Mesif: A two-hop cache coherency protocol for point-to-point interconnects (2004). 2004.

[19] Per Hammarlund, Rajesh Kumar, Randy B Osborne, Ravi Rajwar, Ronak Singhal, Reynold D'Sa, Robert Chappell, Shiv Kaushik, Srinivas Chennupaty, Stephan Jourdan, et al. Haswell: The fourth-generation intel core processor. *IEEE Micro*, (2):6–20, 2014.

[20] Nikos Hardavellas, Ippokratis Pandis, Ryan Johnson, Naju Mancheril, Anastassia Ailamaki, and Babak Falsafi. Database servers on chip multiprocessors: Limitations and opportunities. In *Proceedings of the Biennial Conference on Innovative Data Systems Research*, number DIAS-CONF-2007-008, 2007.

[21] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

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

[23] David A Huffman et al. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[24] Norman P. Jouppi. Cache write policies and performance. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 191–201, New York, NY, USA, 1993. ACM.

[25] Changkyu Kim, Doug Burger, and Stephen W Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Acm Sigplan Notices*, volume 37, pages 211–222. ACM, 2002.

[26] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 361–372. IEEE Press, 2014.

[27] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Rowhammer: Reliability analysis and security implications. *arXiv preprint arXiv:1603.00747*, 2016.

[28] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th annual symposium on Computer Architecture*, pages 81–87. IEEE Computer Society Press, 1981.

[29] Haris Lekatsas, Jörg Henkel, and Venkata Jakkula. Design of an one-cycle decompression hardware for performance increase in embedded systems. In *Proceedings of the 39th annual Design Automation Conference*, pages 34–39. ACM, 2002.

[30] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. *The directory-based cache coherence protocol for the DASH multiprocessor*, volume 18. ACM, 1990.

[31] Nihar R Mahapatra and Balakrishna Venkatrao. The processor-memory bottleneck: problems and solutions. *Crossroads*, 5(3es):2, 1999.

[32] Christian Märtin. Multicore processors: challenges, opportunities, emerging trends. In *Proc. Embedded World Conference*, pages 1–9, 2014.

[33] Milo MK Martin, Mark D Hill, and Daniel J Sorin. Why on-chip cache coherence is here to stay. *Communications of the ACM*, 55(7):78–89, 2012.

[34] Mayler Martins, Jody Maick Matos, Renato P. Ribas, André Reis, Guilherme Schlinker, Lucio Rech, and Jens Michelsen. Open cell library in 15nm freepdk technology. In *Proceedings of the 2015 Symposium on International Symposium on Physical Design*, ISPD '15, pages 171–178, New York, NY, USA, 2015. ACM.

[35] Nimrod Megiddo and Dharmendra S Modha. Arc: A self-tuning, low overhead replacement cache. In *FAST*, volume 3, pages 115–130, 2003.

[36] Joshua San Miguel, Jorge Albericio, Andreas Moshovos, and Natalie Enright Jerger. Doppelgänger: a cache for approximate computing. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 50–61. ACM, 2015.

[37] Eduardo B Wanderley Netto, Eduardo A Billo, and Rodolfo J Azevedo. Dual selective code compression.

[38] Tri M Nguyen and David Wentzlaff. Morc: a manycore-oriented compressed cache. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 76–88. ACM, 2015.

[39] Mark S Papamarcos and Janak H Patel. A low-overhead coherence solution for multiprocessors with private cache memories. *ACM SIGARCH Computer Architecture News*, 12(3):348–354, 1984.

[40] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 2–11. ACM, 2010.

[41] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, and Todd 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.

[42] Feng Qin, Shan Lu, and Yuanyuan Zhou. Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs. In *11th International Symposium on High-Performance Computer Architecture*, pages 291–302. IEEE, 2005.

[43] Brian M. Rogers, Anil Krishna, Gordon B. Bell, Ken Vu, Xiaowei Jiang, and Yan Solihin. Scaling the bandwidth wall: Challenges in and avenues for cmp scaling. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 371–382, New York, NY, USA, 2009. ACM.

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

[45] Daniel Sanchez and Christos Kozyrakis. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 475–486, New York, NY, USA, 2013. ACM.

[46] Somayeh Sardashti, André Seznec, David Wood, et al. Skewed compressed caches. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 331–342. IEEE, 2014.

[47] Somayeh Sardashti, André Seznec, and David A Wood. *Yet Another Compressed Cache: a Low Cost Yet Effective Compressed Cache*. PhD thesis, Inria, 2016.

[48] Somayeh Sardashti and David A Wood. Decoupled compressed cache: Exploiting spatial locality for energy-optimized compressed caching. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 62–73. ACM, 2013.

[49] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. Dram errors in the wild: a large-scale field study. In *ACM SIGMETRICS Performance Evaluation Review*, volume 37, pages 193–204. ACM, 2009.

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

[51] Abraham Silberschatz, Peter B Galvin, Greg Gagne, and A Silberschatz. *Operating system concepts*, volume 4. Addison-Wesley Reading, 1998.

[52] Josep Torrellas, HS Lam, and John L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, 1994.

[53] Yuan Xie, Wayne Wolf, and Haris Lekatsas. Profile-driven selective code compression. In *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1*, page 10462. IEEE Computer Society, 2003.