ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Συμπίεση στο δίαυλο μνήμης σε πολυπύρηνες αρχιτεκτονικές

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Χλόη Ν. Αλβέρτη

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π

Αθήνα, Ιούλιος 2014

Εθνικο Μετσοβιο Πολυτεχνειο
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Συμπίεση στο δίαυλο μνήμης σε πολυπύρηνες αρχιτεκτονικές

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Χλόη Ν. Αλβέρτη

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 24η Ιουλίου 2014.

.....	.....	.....
Ν. Κοζύρης Καθηγητής Ε.Μ.Π	Δ. Σούντρης Επίκουρος Καθηγητής Ε.Μ.Π	Γ. Γκούμας Λέκτορας

Αθήνα, Ιούλιος 2014.

.....

Χλόη Ν. Αλβέρτη
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π

Περίληψη

Το τέλος της εκθετικής αύξησης της απόδοσης των μονοεπεξεργαστικών συστημάτων σηματοδότησε το τέλος της κυριαρχίας των απλών μικροεπεξεργαστών στην πληροφορική. Η μελλοντική εξέλιξη των υπολογιστικών επιδόσεων μπορεί και αναμένεται να προέλθει από τον παραλληλισμό κάθε είδους. Τα θετικό νέο είναι ότι η τεχνολογική κλιμάκωση θα συνεχιστεί και η ενσωμάτωση εκατοντάδων πυρήνων σε ένα τσιπ αναμένεται στο εγγύς μέλλον. Τα αρνητικό είναι ότι η κλιμάκωση της επίδοσης στις πολυπύρηνες αρχιτεκτονικές δεν είναι εύκολη ούτε προφανής. Ένα από τα σημαντικότερα προβλήματα που προκαλούν μείωση της επίδοσης είναι το περιορισμένο εύρος ζώνης της κύριας μνήμης. Καθώς ο αριθμός των πυρήνων ανά τσιπ αυξάνεται, οι απαιτήσεις σε εύρος ζώνης μνήμης αυξάνουν σχεδόν γραμμικά. Δυστυχώς, η κλιμάκωση του εύρους ζώνης υπολείπεται σημαντικά της τεχνολογικής κλιμάκωσης και οι διαμοιραζόμενοι πόροι μνήμης και εύρους ζώνης αναδεικνύονται σε κρίσιμα σημεία συμφόρησης που περιορίζουν σημαντικά την επίδοση. Οι όροι *Memory* ή *Bandwidth* wall χρησιμοποιούνται συχνά για την αναφορά σε αυτό το ζήτημα. Ο στόχος της παρούσας διπλωματικής είναι η μελέτη και η αξιολόγηση της συμπίεσης στο δίαυλο μνήμης ως μιας τεχνικής για την αύξηση του αποτελεσματικού εύρους ζώνης και τη βελτίωση της επίδοσης σε σύγχρονους πολυπύρηνους επεξεργαστές. Η λογική πίσω από τη συμπίεση στο δίαυλο μνήμης είναι να καταφέρουμε να μειώσουμε την ποσότητα των δεδομένων που μεταφέρονται από/προς τη κύρια μνήμη με την αποστολή και τη λήψη των δεδομένων σε συμπιεσμένη μορφή. Στο σχήμα συμπίεσης που μελετήσαμε, πειραματιστήκαμε με διάφορους αλγόριθμους που έχουν προταθεί στο παρελθόν για συμπίεση στο υλικό και ανήκουν σε διαφορετικές "οικογένειες". Παρακινούμενοι από την κακή απόδοση που έχουν συνήθως τα σχήματα σε επιστημονικά δεδομένα, εφαρμόσαμε και τον αλγόριθμο FPC, ένα σύγχρονο λογισμικό συμπίεσης για δεδομένα κινητής υποδιαστολής διπλής ακρίβειας. Για τη μοντελοποίηση του συστήματος και την πειραματική αξιολόγηση χρησιμοποιήσαμε τον Sniper, ένα σύγχρονο προσομοιωτή πολυπύρηνων αρχιτεκτονικών. Πειραματιστήκαμε με τρεις εφαρμογές που η κλιμάκωση τους περιορίζεται από τη μνήμη και με σύνολα ακέραιων δεδομένων και δεδομένων κινητής υποδιαστολής. Τα πειραματικά μας αποτελέσματα δείχνουν ότι η συμπίεση στο δίαυλο μπορεί να μειώσει σημαντικά τις απαιτήσεις σε εύρος ζώνης και υπό ορισμένες προϋποθέσεις, να βελτιώσει την επίδοση των πολυπύρηνων επεξεργαστών.

Λέξεις-Κλειδιά: περιορισμένο εύρος ζώνης μνήμης, συμπίεση στο υλικό, συμπίεση στο δίαυλο μνήμης, πολυπύρηνοι επεξεργαστές (CMPs), προσομοιωτής Sniper

Abstract

The end of exponential growth in single-processor performance marks the end of the dominance of the single microprocessors in computing. Future growth in computing performance is expected and must come from parallelism. The good news is that technology scaling will continue and the integration of hundreds of cores in a single die is the near future. The bad news is that performance scaling in the multicore era is not easy nor obvious. One of the major issues that causes performance degradation in Chip Multiprocessor systems is unsustainable off-chip memory bandwidth. As the number of cores per die increases, the demands in off-chip memory bandwidth increase almost linearly too. Unfortunately, bandwidth scaling typically lags significantly behind and the shared resources of memory and bandwidth emerge as critical performance and throughput bottlenecks. *Memory* or *Bandwidth* wall are terms usually used to refer to this performance constraint. The goal of this diploma is to study and evaluate memory-link compression as a technique to increase the effective memory bandwidth in modern CMPs and overcome the "wall". The rational behind link compression is to manage to reduce the amount of data communicated from/to off-chip memory by sending and receiving it in a compressed form. In the link compression scheme we modeled, we've experimented with various compression algorithms previously proposed for hardware compression, belonging to different families. Motivated by the poor performance that schemes usually have with scientific workloads, we've also applied FPC, a modern software compression scheme for double precision floating-point data. For the scheme's modeling and experimental evaluation we've used Sniper, a modern multicore simulator. We've experimented with three scientific memory bandwidth bound applications and both integer and hard-to-compress scientific floating point datasets. Our experimental results show that link compression can significantly reduce off-chip bandwidth demands and under certain conditions improve CMP performance.

Keywords: Memory Wall, hardware compression, memory-link compression, CMP systems, Sniper multicore

Ευχαριστίες

Αρχικά θα ήθελα να ευχαριστήσω τον καθηγητή κ.Νεκτάριο Κοζύρη για την ευκαιρία που μου έδωσε να εκπονήσω τη διπλωματική μου εργασία στο συγκεκριμένο εργαστήριο.

Ιδιαίτερα θα ήθελα να ευχαριστήσω το Λέκτορα Γεώργιο Γκούμα και το Μεταδιδακτορικό Ερευνητή Κωσταντίνο Νίκα, για τη συνεχή τους καθοδήγηση κατά την εκπόνηση αυτής της εργασίας. Εκτός από τις πολύτιμες γνώσεις που μου παρείχαν, με το ενδιαφέρον, την αισιοδοξία, την ενθάρρυνση, το σεβασμό και την υπομονή τους μου έδιναν συνεχή ώθηση για να τα καταφέρω. Η συνεργασία μαζί τους έκανε τη διπλωματική μου μια πραγματικά όμορφη εμπειρία.

Επίσης, επιβάλλεται να ευχαριστήσω τη Νικέλα, το Νίκο, το Βασίλη και πολλούς άλλους που σαν παργματικά καλοί φίλοι με βοήθησαν πολύ μέχρι και την τελευταία στιγμή, υπομένοντας την απαισιοδοξία και το άγχος μου.

Για τα φοιτητικά χρόνια που πέρασαν, θέλω να ευχαριστήσω τους φίλους μου, τους συμφοιτητές και τους γνωστούς που τα έκαναν να κυλήσουν τόσο όμορφα και ξεχωριστά, χωρίς να προλάβω να το καταλάβω. Θέλω όμως να ευχαριστήσω και μια πολύ ξεχωριστή 'γνωριμία' αυτών των χρόνων, τους Ανεξάρτητους Αριστερούς Φοιτητές Ηλεκτρολόγους. Γιατί με έκαναν να δω τον κόσμο μ΄ άλλα μάτια, γιατί κοντά τους έμαθα την ομορφιά του να ονειρεύεσαι και να αγωνίζεσαι για μια καλύτερη κοινωνία, γιατί μαζί τους γνώρισα τη συλλογικότητα, τη συντροφικότητα και την αλληλεγγύη, γιατί με δίδαξαν ότι οι ζωές μας κρίνονται στο δρόμο. Για τους αγώνες που δώσαμε και τους αγώνες που έρχονται...

Τέλος, δε γίνεται να μην ευχαριστήσω την οικογένεια μου και τους γονείς μου. Χωρίς την υποστήριξη, την κατανόηση, το ενδιαφέρον και την αμέριστη αγάπη τους όλα αυτά τα χρόνια, τα πράγματα θα ήταν πολύ διαφορετικά..

Χλόη Αλβέρτη

Contents

Chapter 1

Introduction

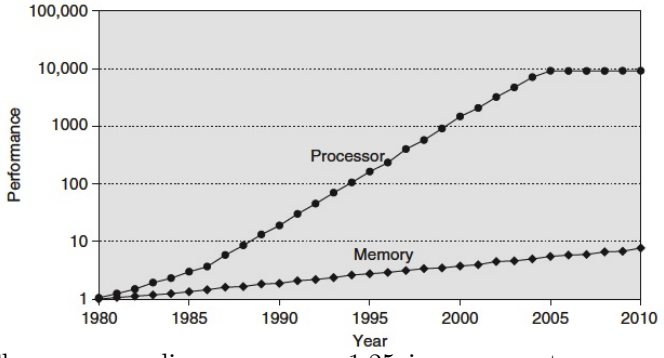
In 1965 when Gordon E. Moore was posing his famous "law"[1] he probably didn't know what he was about to trigger. His observation that on chip transistors will constantly double every 18 months has been a fundamental driver of the global hardware industry. For decades, processor manufacturers took advantage of this increasing transistor density to achieve higher clock speeds. In fact there was a conventional wisdom that thought of frequency scaling as the primary, if not the only, method to improve processor performance. In 2004 though, as per-core clock rates exceeded 3 GHz, this method tested its limits. It was not Moore's law that failed, but the so called Dennard's scaling law. Robert H. Dennard in 1974 claimed in his paper that, in general terms, as transistors would be getting smaller, power density would stay constant. What happened and overturned this theory, was the rapid increase of static power losses as the component size was getting too small. This lead to high temperatures and to the threat of a thermal runaway. The "law" of clock speed hit a wall due to power consumption, and performance could not increase without heroic and expensive cooling.

This new fact lead hardware industry to the revolutionary idea of multicore processors. Since 2005 the new way to exploit Moore's law scaling is to increase the number of cores on chip and improve computational throughput, rather than focus on a single core's performance. Processors parallelism is already a primary method of performance improvement and multicores an integral component of every high performance computing system. It seems that supercomputers are the new trend and parallel computing the future. These new architectures, though, are not a troubleless answer to the question of constant performance scaling. There is a desperate need for a different approach to hardware and software design in order to take full advantage of them and overcome some inevitable difficulties. This essay is an attempt to study one of these difficulties, the "Memory Wall" issue, and experiment with possible improvements.

1.1 The Memory Wall

In 1994, Bill Wulf and Sally McKee published a paper named: "Hitting the Memory Wall: Implications of the Obvious"[2]. In this article they predicted that increasing divergence between core and memory speed improvement would inevitably lead performance to be completely dominated by memory activity and eventually constrained. At the time, the article received controversial response by the scientific community. The following years, though, it became clear that "Memory Wall" was not just a pessimistic hypothesis but a real threat. We could say that processors became victims of their own speed. In 1996, Richard Sites stated that "across the industry, todays chips are largely able to execute code faster than we can feed them with instructions and data"[3]. Figure 1 shows the increasing performance gap between CPU and memory on a logarithmic scale.

Figure 1: The performance of memory and CPU plotted over time, starting with 1980 performance as a baseline.



The processor line assumes a 1.25 improvement per year until 1986, 1.52 until 2000 (frequency scaling), 1.20 between 2000 and 2005 and no improvement between 2005 and 2010 (this graph examines performance in a per-core basis). The memory line assumes a 1.07 per year performance improvement, and the 1980 baseline is 65KB DRAM. *Hennessy and Patterson (Computer Architecture, 4th edition)*

To evaluate this problem properly, a thorough comparative study of architecture evolution and the Memory Wall would be of great importance. This is difficult to accomplish, though, in a single essay and since our experiment is on multicore processor architectures, we will focus on them.

At this point we must make an observation. In their study, Wulf and McKee didn't make a distinction between latency and bandwidth as memory performance parameters. Although there is a subtle relationship and strong dependence between them, this distinction is important to understand real applications performance, especially in parallel architectures. Memory latency is the time needed

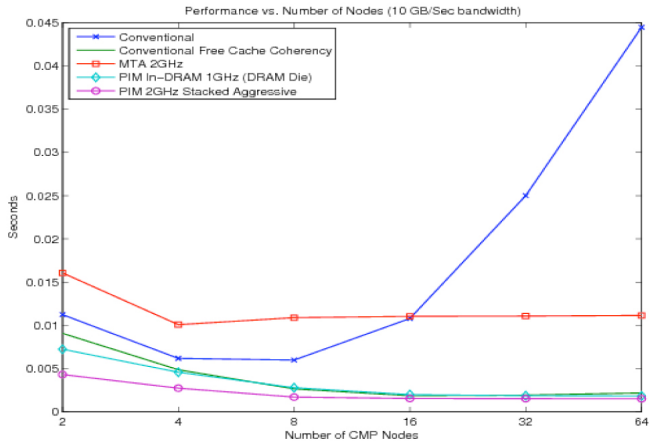
for a processor's memory request to be completed, while memory bandwidth is the rate at which data can be transfered to/from memory. High performance is achieved when memory latency is eliminated and bandwidth is maximized.

1.2 Motivation: Multicore hit the wall?

When multicore was introduced for the first time, there was a cultivating sense that we found the way to exponentially improve processors performance. Craig Barrett's statement, Intel's CEO in 2005, is indicative. Trying to explain the shift from high frequencies to multiple cores he claimed that "it's the way the industry is going to continue to follow Moore's Law going forward - to increase the processing power in an exponential fashion over time"[4].

Today, the claim that multicore architectures are indeed capable of improving performance and, in some cases, achieving outstanding speedups is already history. The belief, though, that this architectural innovation would achieve linear scaling was quickly proven wrong. The behavior of data intensive applications was an alarming indication and the simulation results published by a US Sandia Labs team were revealing. They simulated key algorithms deriving knowledge from large data sets on potential multicore computers and they noticed that increasing the number of on chip cores beyond a threshold unexpectedly worsens performance! Their results are plotted in Figure 2.

Figure 2: Data intensive application performance on four potential multicore computers. Simulation results by Sandia National Laboratories.



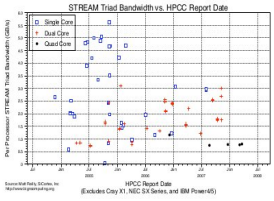
The "conventional" computer adds more standard cores to a single processor socket, MTA looks like the processor used the exotic Cray XMT supercomputer, PIM is based on Sandias X-caliber processor design and includes memory tightly integrated with the processor. The fourth line simulates a conventional processor that represents a theoretical ideal.

The simulations show a significant increase in speed going from two to four multicores, but an insignificant one from four to eight. Exceeding this number causes a decrease in speed and sixteen multicores perform barely as well as two. After that, a steep decline is registered as more cores are added. The Sandia team yielded this behavior to memory bandwidth saturation: "The problem is the lack of memory bandwidth as well as contention between processors over the memory bus available to each processor"[5].

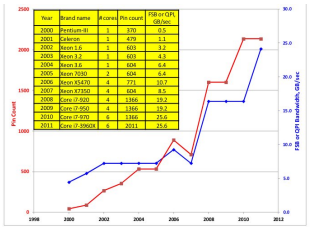
A dominant factor that affects sustainable memory bandwidth is the performance of the bus. Front Side Bus (FSB) or memory bus are both terms used to refer to the bidirectional data link between processor and main memory. In shared memory architectures (symmetric multiprocessor systems - SMPs) this resource is shared among multiple processors while in a single chip multiprocessor (CMP) it is shared among cores of the same die. Consequently it becomes a potential bottleneck. A common example used to describe contention on the memory bus and the issue of insufficient memory bandwidth in multicore, is the "cook example". Imagine there is a difficult meal to be prepared and only one cook to do all the work. If we split the preparation into tasks and increase the amount of cooks to work in parallel, we hope that the meal will get ready sooner. The ingredients needed, though, are in a single refrigerator with only one door and all the cooks will probably have to use it. So adding more cooks to the kitchen will speedup the preparation only up to a certain point where they all still coexist efficiently using the refrigerator. If we increase further their number, they will periodically stack in a queue waiting to get their ingredients since only one cook can access the refrigerator at a time. So some of them will suffer from idle periods unable to do anything or even worse they will all constantly get to each other's way. Even if a simplistic way to approach multicore behavior, this real world analogy, with cooks as processors and the refrigerator as the main memory shared resource, reflects the main idea of the problem. To be useful, each core on die must be fed by the external memory system.

Figure 3a depicts McCalpin Stream TRIAD memory bandwidth for a collection of single core and multicore processors. The graph shows something generally expected, integrating more cores on die comes with the cost of reducing the available bandwidth per core. So it becomes essential for the overall performance that as the number of cores per die grows, the pin bandwidth per die grows too. Unfortunately, pin bandwidth cannot easily improve at Moore's law rate.

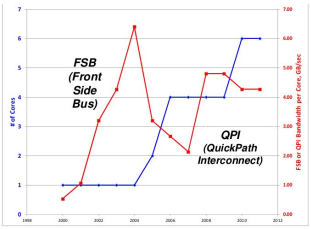
According to 2004 ITRS roadmap[6], pin count per chip will increase at approximately 11% over the next fifteen years while cores will continue to double every 18 months. And even if pin frequency is going up increasing the effective bandwidth, it is unclear at what rate since it is "pushing" up against the Power Wall due to growing heating in higher frequencies. At the same time on-chip frequency is



(a) Stream Triad Bandwidth on multicore.



(b) Pin Count and FSB/QPI bandwidth.



(c) Number of cores and per core bandwidth.

Figure 3: Insufficient memory bandwidth and especially off chip pin bandwidth becomes a potential multicore performance bottleneck.

increasing too, thus the offset between them tends to be sustained. This growing gap contributes to the perception of off chip pin bandwidth as a critical resource and as a dominant performance parameter for multicore. Figure 3b depicts the evolution of pin bandwidth and Figure 3c the evolution of per core bandwidth.

Things become more complicated if we take into account what Burger et al [7] described in their paper. They noticed that several techniques developed over the years to reduce memory latency commonly succeed at the expense of effective bandwidth. The success of latency-tolerance techniques leads to a higher instructions retirement rate, increasing the memory operands needed per unit of time. But the most important thing is that some of these techniques involve transaction to/from memory of data not requested by processor, increasing memory interconnection network's traffic. The hardware prefetching example is quite representative. Figure 4 depicts how some techniques affect both bandwidth and latency.

1.3 Facing the problem

Designing deep cache hierarchies has been for years the most popular way industry came up against high memory latencies.The use of high-speed on chip memory devices reduces both the average latency of a memory access and the off chip demands increasing the memory bandwidth. In general, it has proven to be a very successful technique when it comes to applications whose datasets fit

nicely in caches. Modern applications though, tend to become more and more data intensive (web applications, databases, scientific algorithms etc.) with working sets larger than the LLCs and this technique yields diminishing returns. In multicore architecture things are even worse since coherence protocols introduce a new type of misses and extra traffic from/to the external memory system and the network. Bigger caches could be a possible improvement and used to be an industry trend. The constant increase of the on chip area devoted to caches, though, comes with the cost of less available space for extra cores on the die and of greater power demands. We should notice that caches already occupy a great percentage of the silicon area of most CPUs. Also, for memory bound applications with really large working sets or with no significant data reuse behavior, this technique barely improves the overall performance.

Several techniques have been introduced for the increase of the effective bandwidth too. Integrating memory controllers on chip and designing non uniform memory access (NUMA) architectures are two of the most important innovations. We can notice a peak in Figure 3c's graph when Intel's Quick Path Interconnect (QPI - NUMA compatibility) replaced FSB. This mainly concerns SMPs but even for a single multicore chip's performance adding more memory controllers on die can increase available memory bandwidth. Sun's Niagara has 4 controllers on chip and approximately 20 GB/sec memory bandwidth. Nevertheless, this adds complexity to the chip and reduces the available space for extra cores. Other known techniques are the fully buffered DIMM architecture which increases memory's width without increasing the pin count and the multi-channel memory architecture which increases bandwidth by adding extra channels between the memory and the on chip controller.

Figure 4: Architectural trends and their effect on different execution time factors.

A. Latency reduction	f_P	f_L	f_B
Lockup-free caches	?	↓	↑
Intelligent load scheduling	↑	↓	↑
Hardware prefetching	?	↓	↑
Software prefetching	↑	↓	↑
Speculative loads	↑	↓	↑
Multithreading	?	↓	↑
Larger cache blocks	?	↓	↑
B. Processor trends	f_P	f_L	f_B
Faster clock speed	↓	↑	↑
Wider-issue	↓	?	↑
Speculative (Multiscalar)	↓	?	↑
Multiprocessors/chip	↓	↑	↑
C. Physical trends	f_P	f_L	f_B
Better packaging technology	↑	↓	↓
Larger on-chip memories	↑	↓	↓

f_P corresponds to processing time, f_L to latency time and f_B to stall time due to insufficient bandwidth

A radical technique explored by industry and scientific community as a potential solution for the Memory Wall problem is the integration of memory and processors on a single chip (e.g [8] [9] [10]). This is much more difficult though, than it may sound. Integrating both of them on the logical chip die has proven to be extremely expensive and thus not an acceptable solution. On the other hand the integration on the memory chip die requires the overcome of some severe difficulties. The main problem is design compatibility. DRAMs use only three layers of metal when processors use 10 to 12 to enable connections between logical gates that preserve their functionality.

Chapter 2

Hardware Compression

In this dissertation we will study and evaluate hardware compression as a potential technique to deal with the Memory Wall issue. We use the term "hardware compression" to refer to hardware implementations of known or novel compression algorithms and their intervention to the memory hierarchy or networks of a conventional computer system. It is a modern area of research that has the potential to exploit compression in order to eliminate the latencies introduced by message or data transferring, increase the effective storage capacity of memory modules, reduce power consumption, cost etc. Researchers have proposed multiple compression schemes applied to a system's main memory, cache hierarchy, memory bus, networks on chip etc both for uniprocessor and multiprocessor or multicore systems. In this dissertation we focus on the memory bandwidth/pin bandwidth constraints of multicore systems and we simulate and evaluate a memory-link compression scheme. Memory-link compression has the potential to increase the effective memory bandwidth by reducing the amount of data communicated from/to off-chip memory by sending and receiving it in a compressed form.

In this chapter though, we present related research work on hardware compression applied in both memory, cache hierarchy and bus in order to gain a global view of the subject and present the different kinds of algorithms used in the various schemes. Moreover, main memory, cache and link compression are techniques that mainly interact positively and are usually combined.

2.1 Compression

Compression is in general a very popular technique used to reduce the size of stored or communicated data. Data compression requires the identification and extraction of source redundancy. In other words, compression seeks to reduce the number of bits used to store or transmit information. In terms of storage, the effective capacity of a device can increase with methods that compress a body of data on its way to be stored and decompress it when it is retrieved.

In terms of communication, the effective bandwidth of a link can increase if the data to be transferred is compressed in the sending end and decompressed at the receiving end. There are numerous algorithms implemented in software that are widely known and used for image compression, sparse data compression, web indexes compression, databases compression etc.

Compression can be classified in lossy and lossless. If the decompressed data are identical with the compressed then the compression technique is referred to as lossless, otherwise the compression is lossy. Since we are interested in compressing memory data only lossless compression can be considered, otherwise the validity of a program's result would be at risk.

2.2 Main Memory Compression

The basic goal of a compressed main memory system is to increase memory's effective storage space, effective bandwidth and the overall cost efficiency. Despite its potential benefits, the high complexity and cost this kind of system may introduce (potential intervention to the OS, mapping of addresses etc.) has made main memory compression a research area considered only by few developers, unexploited yet to its full potential. The compressed memory systems can be classified in software and hardware based. For a description of software based systems please refer to a study of Irina Chihaia Tuduce and Thomas Gross [11], a study of Jordi Torres et al [12] or Yang et al 's CRAMES project[13].

IBM in the early '00s designed and built a memory subsystem for real-time hardware main memory contents compression, called MXT (Memory Expansion Technology)[14]. To the best of our knowledge, this is the only memory compression scheme implemented in industry. A performance study of MXT[15] indicates that this new memory architecture effectively doubles the physical available memory by achieving most of the time a 2:1 compression ratio without significant added cost. MXT's hardware design includes a compressed main memory and a large tertiary shared L3 cache (32MB) that contains wide uncompressed cache lines and is used to hide the high latencies of a memory access. This cache appears as the main memory to the upper layers of the memory hierarchy and its operation is transparent to the rest of the hardware including the processors and I/O[15].

A basic component of MXT is the Main Memory/L3 Cache controller. The controller compresses and decompresses the cache lines transferred to/from main memory using a parallel variation of the Lempel-Ziv (LZ77) algorithm implemented in hardware with CMOS ASIC technology. This compression scheme is based on the idea of constructing shared dictionaries with the cooperation of multiple compressors [16]. The cache line is partitioned equally and the parts are compressed in parallel by independent compression engines with the use of the shared dictionary. Compressed main memory is divided into storage units of 256KB -sectors- and

the compressed cache line may occupy from one to four sectors. There is a simple sharing mechanism to reduce fragmentation overhead and improve compression efficiency. The controller is responsible for the necessary translation of real addresses (processors' chip) to physical addresses (main memory) using a translation table apportioned from the main memory. MXT also requires support from the operation system, to manage the compressed memory. One of the factors that cause performance losses in MXT system, is the high latency of shuttling in and out cache lines to the main memory (at least 64 cycles[14]) significantly increased by the use of the memory resident translation table. High decompression latency also, when L3 cache is full, can cause severe performance damage as it is in the critical path.

Magnus Ekman and Per Stenstrom[17] described a memory compression scheme focusing on three common performance degradation factors that affect MXT too: a) the high decompression latency, part of the memory access critical path, b) the time consumed by translation mechanisms used for the needed mapping between logical and compressed address space, c) the fragmentation overhead introduced by the variability in the size of the compressed data blocks, reducing the freed-up storage space. To achieve low decompression latency they propose the use of zero-aware compression algorithms. These algorithms are computationally lightweight and their potential is to eliminate the redundancy of the observed high density [18] of the zero value in blocks, words or even bytes of a program's data. For their experiment, they used variations of Alamledeen and Wood's Frequent Pattern Compression algorithm(FPC)[19] and they established that this kind of algorithms is efficient when applied to in-memory data. While unable to achieve the compression ratio of LZ-like or other complex/demanding algorithms, these algorithms can fairly compete with them and their simplicity and small overheads provide a good performance trade-off. The proposed compression scheme includes an entirely compressed main memory and decompressed caches and disk. For the address mapping Stenstrom et al propose a TLB-like structure residing in the processors' chip, avoiding a memory access and allowing the translation to occur in parallel with the L2 cache access. The experiment results indicate that tis novel main memory level compression scheme can free up 30% of the memory resources -on average- (LZ based schemes can free up 50% on average) with a negligible performance overhead of only 0.2%.

2.3 Cache Compression

Caches, as mentioned in the previous chapter, are very fast on-chip storage devices, widely used to hide the main memory access latencies. Their size is limited by die area, power consumption and cost. Cache compression - the compression of the data stored in caches - has been proposed as a potential technique to a) reduce power consumption and b) improve the memory system performance [20].

The use of compression simply to achieve power/energy saving is based on the idea that the occupation of less space for data storage leaves memory cells and wires unused, thus not consuming energy. System performance improvement may occur by the utilization of the freed-up storage space, increasing cache's effective capacity without increasing its physical size. This "bigger" cache may reduce the miss rate and the off-chip bandwidth demands avoiding the disadvantages (cost, energy etc.) of a physically larger cache. The compression schemes, though, even if compressing effectively, remain beneficial only as long as the compressed cache access latency is smaller regarding to the penalty of a miss to the lower level of the memory hierarchy. So compression and especially decompression overheads must be taken into account. Data decompression occurs every time a hit to a compressed cache component occurs and data has to be transfered to a higher uncompressed memory level or to the CPU (from L1). This makes decompression a very frequent "event" and places it in the memory access critical path, imposing that its overhead must be negligible. This alone rules out must of the complex compression algorithms, such as those used for main memory compression, or demands their radical readjustment and hardware redesign. In general, while descending down the memory hierarchy, components become less sensitive to the introduced latencies and the design of the compression algorithms and schemes can be more complex and demanding, hence sometimes more efficient.

We will briefly demonstrate some cache compression schemes that have been proposed in the past. Three of the many challenges that all cache compression schemes have to face in order to be efficient are the following:

- The compression/decompression algorithm must be very fast (light decompression overhead).
- The amount of data to be compressed is very small, typically a single cache line (i.e 64 bytes), thus the design of the algorithms is quite challenging and the effective compression difficult to achieve due to limited redundancy.
- Compression creates variable-sized data blocks and consequently variable-sized freed-up storage space and both have to be managed. A compressed cache has to be able to hold and access more compressed lines than the uncompressed that fit in the same space. Conventional cache structures cannot support this kind of management, thus they cannot store compressed data and new cache design is needed.

We will refer both to schemes that apply known compression algorithms to L1,L2,L3 caches and to schemes that use significance-based algorithms. At this point we must note that we will only refer to data caches compression. Code compression and instruction caches compression [21][22][23] are also

known research areas, especially for embedded systems, but they have different characteristics and intervene differently in an instruction pipeline.

Hallnor et al [24] propose a cache compression scheme that uses the dictionary-based LZ77 compression algorithm, the algorithm used in MXT. The main contribution of this paper is the compressed cache structure design, the Indirect Index Cache with Compression (IIC-C). Based on IIC [25] this cache organization scheme uses a segmentation technique. The cache and the compressed lines are divided in small fixed-size sub-blocks, segments, and the data and address tag array are decoupled. Each tag entry holds multiple pointers that associate tags with variable number of sub-blocks anywhere in the data array. These pointers are used for indirect indexing to locate all the segments of a compressed line and this indirection provides the ability to easily manage a fully associative cache. Wood et al [26] recently described another cache organization scheme based on segmentation, the Decoupled Compressed Cache (DCC), that exploits spatial locality to improve both the performance and energy-efficiency of cache compression. This scheme uses decoupled super-blocks and non-contiguous sub-block allocation to decrease tag overhead without increasing internal fragmentation. It has numerous optimizations, it is mainly independent of the compression algorithm in use and it can potentially increase the normalized effective capacity of a cache to a maximum of 4 while most of the earlier proposals aim to a maximum of 2.

Frequent value Encoding. Yang et al [27] [28] studied the SPECint95 suite and observed that most of the programs tend to share a specific characteristic. A small amount of distinct values, different each time, appears very frequently in memory locations and is therefore involved in a large fraction of the program's memory accesses. In fact, they observed that in six out of eight programs in SPECint95 test suite, ten distinct values occupy over 50% of all referenced memory locations through-out the program's execution. They named this new kind of value locality in a program's data, frequent value locality, and in [27] they proposed the design of a new cache structure that exploits this characteristic to store data in a compressed form. Their FVC (Frequent Value Cache) caches only the lines that contain frequent values. These values are stored encoded using fewer bits while the non-frequent values of the line are simply indicated and not stored. It is a simple and small direct-mapped cache and in the paper it is used to improve the performance of a conventional direct-mapped cache by augmenting it.

In [29] Yang et al describe the design of a conventional sized level one compression cache (CC) using frequent value encoding, where each cache line can hold either one uncompressed line or two cache lines that have been compressed to at least half of their size. Frequent values of a line are encoded and non-frequent values are stored uncompressed. This scheme also preserves the valuable ability to randomly access individual data items of the line. The design includes the hardware division

of a cache line to sub-blocks in order to achieve storage and management of the compressed lines. Both [27][29] schemes reduce the miss rate and in [29] power consumption is evaluated but both schemes require the prior knowledge of a program's frequent values in order to construct a "dictionary". The authors propose profiling as a technique to identify them, producing each time a fixed frequent value set. This is discussed more thoroughly in [28] where the disadvantage of the non adaptive behavior of the fixed set is mentioned but is claimed as unavoidable, since the encoding "dictionary" for cache compression must stay fixed during a program's execution for the encoding/decoding to be consistent. In [28] the authors describe and propose a method for bus compression were the frequent values set is renewed continuously during a program's run and a dynamic dictionary is constructed. In a following chapter, we will study this in detail. Keramidas et al in [30] discuss the need for dynamic frequent value dictionaries for the needs of cache compression, they process the matter of encoding/decoding consistency and they describe a way to construct a fast dictionary. They proposed a scheme for the compression of the L1 cache but the definition of the introduced latencies is quite abstract.

A disadvantage that this kind of compression algorithms share (key performance factor of all dictionary-based algorithms) is the latency introduced by the required search in the dictionary table. This table may be memory resident or implemented in hardware as a separate storage device and an increase of its size (potentially better dictionary) leads to an increase of its access time too. We will discuss this issue later in this essay.

Significance-Based Encoding. Significant-based compression algorithms form a different category. Their operation is based on the observation that many words stored in memory occupy more bytes than actually needed for their representation. Typical examples are zero or small integer values that may occupy 4, 8 or more bytes (with the sign extension mechanism) when they actually need only a few bits to be accurately represented. These algorithms, unlike the dictionary-based, do not require dictionary table accesses or a per line dictionary overhead, thus they are simpler and they have smaller overheads. This is why they are generally preferred for cache compression and link compression schemes.

Dusser et al [31] propose the augmentation of a conventional cache with a specialized cache that stores only compressed null blocks (zero cache lines) in a way that exploits a potential spatial locality that they may present. Their scheme, the Zero Content Augmented cache (ZCA), has a very simple hardware implementation with very fast compression and decompression circuits. However, only applications that operate on a large number of zero cache lines can benefit from this design. Kim et al [32] propose a technique that uses sign compression to reduce energy dissipation in caches. In particular, the scheme encodes words whose upper half bits are either all zeros or ones by compressing them into half-words with a sign-bit.

The compressed cache structure holds either uncompressed lines or compressed lines with some tolerance to uncompressed words. The authors propose a low energy cache architecture using cache line bisection and an alternative architecture with additional tags and compression flags to store additional data in the unused cache space remaining after compression. Pujara and Aggarwal [33] propose a similar scheme for L1 caches with some optimizations.

Alaa Alameldeen and David Wood in [34] describe and propose a significance-based compression scheme for L2 caches. This scheme is based on the observation that some data patterns, compressible to a fewer number of bits, appear frequently in integer and commercial benchmarks. Runs of zeros (one or more all-zero words) along with sign-extended words are some of the patterns used by the scheme. The Frequent Pattern Compression (FPC) compresses cache lines on a word by word basis by matching them with the patterns that have statically decided compact encodings mapped to them (prefixes). The compressed cache structure proposed for hardware implementation by the authors, for the diminishing of the scheme's complexity and the access overheads, includes the division of the cache line in sub-blocks, the segmented compression of the data and the doubling of the available address tags of each set (potential doubling of the effective capacity). For the management of the cache they choose the approach of decoupling the cache access, based on similar previous proposals [24], adding a level of indirection between the address tag and the data storage of each set. The data segments are stored contiguously in address tag order. A serious problem of this scheme is the emerging need to continuously compact the sets to exploit efficiently their freed-up storage space. This operation combined with the preservation of the contiguous storage becomes quite expensive. The authors try to address this issue with some optimizations in [35]. Another disadvantage of the scheme is that it requires serial decompression of each line, because the starting location of a compressed data item is determined by the compressed size of the previous. To mitigate the decompression latency of FPC, the authors design a five-cycle decompression pipeline. They also propose an adaptive scheme which avoids compressing data if the decompression latency nullifies the benefits of compression [35].

Chen et al [36] also propose a pattern-based compression scheme for LLCs named C-Pack, with the optimization of using an additional small dynamic frequent value dictionary that enables the partial adaptation of the patterns used for encoding. For the organization of the compressed cache they propose the idea of pair-matching, combining pairs of compressed lines into one cache line, trying to overcome the disadvantages of the segmentation techniques (compressed data access latencies, hardware complexity, area overhead etc). Another novel feature of their algorithm is the ability to compress multiple words in parallel. C-Pack, though, suffers from serial decompression latencies too, since a potential parallel

implementation of the decompression algorithm would require a great area occupation and hardware complexity. This scheme offers a slightly better compression ratio than FPC and a similar decompression latency, but the estimations are based on a register transfer level hardware implementation and are probably more accurate.

Pekhimenko et al [37] propose a new algorithm named Base-Delta-Immediate compression (BΔI) that looks for data redundancy and compression opportunities at a cache line granularity. Particularly, the algorithm is based on the assumption that it is highly likely for the data values stored in a cache line to have low divergence, to have a small relative difference. We will study thoroughly this algorithm in a following chapter, since we use it for our experiment too. BΔI offers a degree of compression similar to FPC and is quite fast with low decompression latencies, as it allows compression and decompression to be done in parallel. Regarding to the proposed cache structure, the scheme is a decoupled cache based on segmentation techniques.

Many researchers combine cache compression with main memory compression or bus compression and study the overall system performance improvement. Another interesting research area is the combination of cache compression with the hardware prefetching mechanism. In [38] Zhang et al describe a scheme of partial cache line prefetching where corresponding words of sequential cache lines are fetched simultaneously in the line if they are both compressible. In [39] Alameldeen et al propose an adaptive prefetching mechanism that uses compression's extra cache tags and simple heuristics to throttle prefetching when it replaces more useful lines than it brings in. This essay studies and proves that compression and prefetching can have a bidirectional positive interaction.

2.4 Link compression

As we discussed in the previous chapter, off-chip bandwidth -i.e the rate of off-chip requests- has emerged as a critical resource especially for CMPs. Since 2000, pin bandwidth tends to be more critical than memory access latency and, as mentioned in [40], there are two fundamental ways to deal with the off-chip bandwidth problem: i) reducing the bandwidth requirements of threads and ii) augmenting the available off-chip bandwidth [40]. Cache compression may contribute to i) by increasing the capacity of the caches. Link compression or otherwise communication bandwidth compression, both terms that we read about in [41], has the potential to reduce the traffic on the memory link and increase the effective off-chip bandwidth by compressing the address, communication or data messages that are transferred through the link. Link compression schemes apart from a performance improvement also aim to reduce the power consumption.

Since the scheme that we study in this dissertation is a link compression scheme, we will discuss this subject in more detail in the following chapter. In this section we present some previous research proposals on this area. We must notice that link compression schemes share similar difficulties with the schemes of cache compression: i) (de)compression algorithms must be fast and simple. If the link compression is not combined with a compressed main memory or a compressed cache structure then both compression and decompression latencies are in the memory access critical path. ii) The block of data to be compressed is small sized and thus compression is more difficult. Thus, the algorithms that are usually preferred are significance-based and search for redundancy at a cache line granularity.

There are several proposals for link compression schemes applied on external address buses [42] [43] [44]. Values transferred on address buses have different characteristics compared to the values transferred on data buses, thus compression schemes of general purpose tend to lack of efficiency. Mussol et al [42] propose a successful scheme, the Working-Zone Encoding technique (WZE), for address bus compression that exploits the locality of the memory references. They base their scheme on the fact that applications favor a few working zones of their address space at each instant of time. Thus compression on the address bus can be achieved by sending only the offset of a reference with respect to the previous reference of the same working zone. Compression on the data buses, on the other hand, tends to be a more difficult task since there are no obvious special characteristics shared by the values transferred on these buses. Our experiment concerns data bus compression techniques, thus we chose to demonstrate only relative proposed schemes.

Benini et al [45] assume a system with compressed main memory and they propose a scheme where data compression and decompression occurs on the fly between main memory and caches. Particularly, a cache line is compressed when it is evicted from the LLC to be stored in the main memory and is decompressed when cache refills takes place. This scheme has the potential to reduce energy consumption in the cache-to-memory path of core-based embedded systems. The authors explore and evaluate two classes of compression methods: 1) a profile-driven method with the use of a static frequent value dictionary, 2) a differential method based on the assumption that it is highly possible for data words appearing in the same cache line to have some significant bits in common. So compression may occur by storing once the common bits and the remaining bits of all the data words. The authors propose three variations of the differential algorithm. We will use one of them in our experiment, thus in the following chapter we will study these algorithms in detail.

Stenstrom et al [46] propose a scheme where data compression occurs before a data block is transferred on the memory link and decompression occurs before the block is installed in the on-chip cache or written back to memory. The authors

study the value locality of the data transferred on cache/memory links and propose three lightweight compression methods that exploit three different forms of locality: 1) small value locality. The suggested compression algorithm is named Significance-width compression (SWC) and encodes the sign extension bits, thus it is suitable for small integer values. 2) clustered value locality. By studying integer, commercial and media applications the authors noticed that a program's larger values are not distributed uniformly. Instead they seem to form concentrations, i.e. clusters, in the value space. The proposed compression algorithm is named delta encoding and is based on a Hammerstrom and Davidson's proposal [47]. The main idea is the initialization and usage of a dynamic dictionary containing cluster values that allows compression to be done by encoding each data word as the difference from its closest value in the dictionary along with the respective dictionary index. The implementation of this scheme requires value caches on both sides of the link that must be kept consistent. 3) isolated value locality. Apart from the clusters there are values dispersed across the entire value space and the authors describe and propose the Citron scheme [48] for their compression. This scheme is also based on the use of a dynamic dictionary implemented with value caches. This time, though, only the 16 mst significant bits of data words are stored and the compression is achieved by matching the significant bits of each word to be compressed with the dictionary entries. If there is a hit the word is encoded as the respective dictionary index and the remaining bits, otherwise it remains uncompressed and renews the dictionary. This scheme seems like a combination of frequent value encoding and differential algorithms or delta encoding. At the end, the authors explore the results of schemes that combine these compression methods aiming to attack all three value locality properties.

Yang et al [49] applied the frequent value encoding (FVE), mentioned in 2.3, to a scheme implementing data bus compression. The main potential of their scheme is to reduce switching activity on the external bus and consequently reduce its energy consumption. Compression and decompression in their scheme occurs respectively before and after the transfer of the data through the memory link. The fact that compression and decompression happens on the fly allows the construction of a dynamic frequent value dictionary this time. Two value caches exist on both sides of the link and compression is achieved by matching each word to be compressed with all the dictionary entries. If there is a hit the word is encoded as the respective dictionary index, otherwise it remains uncompressed and it renews the dictionary. So the dictionary is dynamically refreshed and a program's new frequent values are identified. It is critical for this scheme to be functional that both value caches remain consistent containing the same dictionary. This is already guaranteed though by the fact that compression and decompression of a cache line occurs sequentially, thus no extra mechanism is needed to keep the

caches consistent. The authors describe in detail the hardware implementation of this scheme, the design of both the encoder and the decoder, estimating the latencies they introduce and the energy they consume quite accurately. Stenstrom and Thuresson [50] combine the FVE bus compression scheme with the use of a dynamic block-size changing mechanism of a system's last level cache. The objective of this scheme is to diminish the increasing off-chip bandwidth demands introduced by cache strategies that use larger block sizes (both static and dynamic) to improve performance. FVE has also been used in a scheme for compression in packet-based NoC architectures [51]. We will use this scheme in our experiment too.

Chapter 3

Link Compression on Chip Multiprocessors

In recent years, multiple performance and power limitations have driven a shift from uniprocessor systems to Chip Multi-Processor designs. Integrating multiple cores on a single die is a new system design developed to exploit the increasing density of the available transistors on a single semiconductor chip (predicted by Moore's Law). CMP systems can provide the increased throughput required by multi-threaded applications while reducing the overhead incurred due to sharing misses in traditional shared-memory multiprocessors. A chip multiprocessor design is typically composed of multiple processor cores having private high-level cache hierarchies and sharing a lower-level cache LLC (e.g., L2 or L3).

The increasing number of on-chip processor cores provides the potential for throughput and performance gains. Ideally, designers would like to extract gains proportional to the increase in the number of cores at each technology generation. However, as discussed in Chapter 1, one of the major obstacles to this goal is the limited bandwidth to off-chip memory. Generally, the doubling of on-chip cores results in a corresponding doubling of off-chip memory traffic, since each core generates additional misses that must be serviced by the memory subsystem. Thus, to maintain a balanced design, the off-chip bandwidth should increase at a corresponding rate. Otherwise, if the provided bandwidth cannot sustain the rate of the generating memory requests, an additional queuing delay will be introduced. This delay will force the performance of the cores to decline until the rate of memory requests matches the available bandwidth. At this point, integrating more cores to the die no longer provides additional throughput. Unfortunately, as mentioned in 1.2, memory bandwidth doesn't follow the scaling rate of transistor density. The severe performance limitation of the unsustainable bandwidth is referred to as the *bandwidth wall* and is a new "wall" that designers have to overcome.[52]

The approach considered in this dissertation to deal with the *bandwidth wall* issue is hardware link compression. Link compression is a technique that aims to increase the effective memory bandwidth by reducing the amount of data transferred from/to memory by sending/receiving it in a compressed form. A general research background of the subject is presented in Chapter 2. In this chapter we describe in detail the link compression scheme modeled and evaluated in this essay. We also present the four different compression algorithms we used for our experiment, trying to focus on their special characteristics, studying both their compression technique and their hardware implementation.

3.1 Link Compression Scheme

The link compression scheme modeled in this essay is the same with the scheme proposed in [46]. The scheme's potential is to free up bandwidth by transferring data over the memory link in a compressed form. Thus, data is compressed before it is transferred on the link and decompressed before the block is installed in the on-chip cache or written back to memory (as shown in Figure 5). Therefore, a compressor and a decompressor unit are both required in both sides of the link. Data could be stored in a compressed form in main memory removing the need of compression and decompression support on the memory side. This approach, though, is not evaluated in this study.

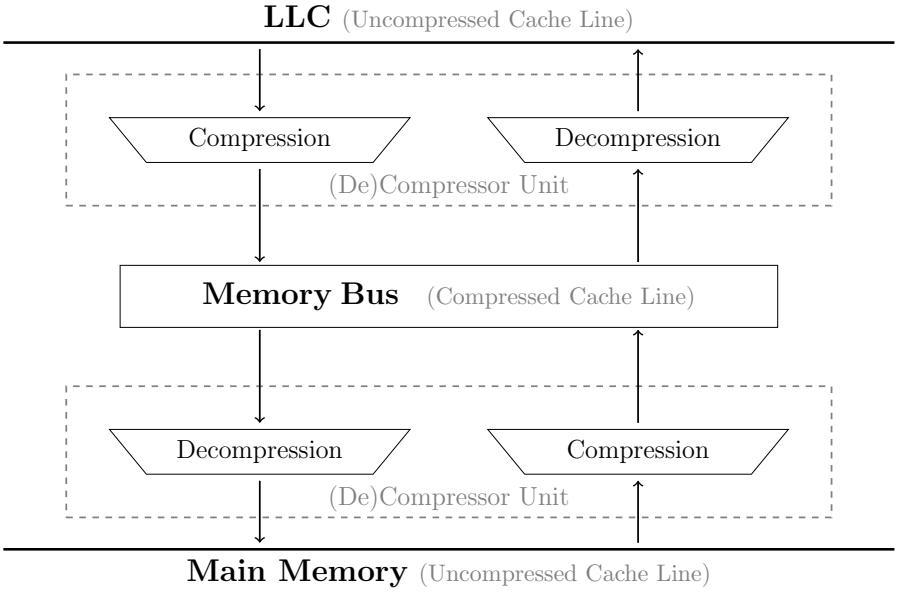


Figure 5: Link Compression scheme.

This link compression scheme could be applied in a uniprocessor system. In this essay we modeled and simulated it as part of a simple Chip Multiprocessor design. Figure 6 depicts the intervention of the scheme in the system design. The CMP consists of multiple processors with private L1 caches sharing a level-two (L2) cache. Compression and decompression occurs on the fly. A L2 cache line eviction triggers a transmission of data from chip to memory. To make use of the reduction in off-chip bandwidth due to compression, the on-chip Memory Controller must be able to compress the data before they are passed out on the memory link. Since the main memory is uncompressed, the off-chip Memory Controller must be capable of decompressing the receiving compressed data. Symmetrically, a L2 miss event triggers a transfer of data from memory to the chip. Therefore, the off-chip memory controller must be also capable of compressing data and the on-chip memory controller of decompressing them. Thus a (de)compression unit is required in both sides. For simplicity reasons, we assume a simple memory bus for the memory link.

At this point we must notice that we assume only a single on-chip Memory Controller and a single corresponding off-chip Memory Controller. This simplifies the design and allows the use of various compression schemes that require one-to-

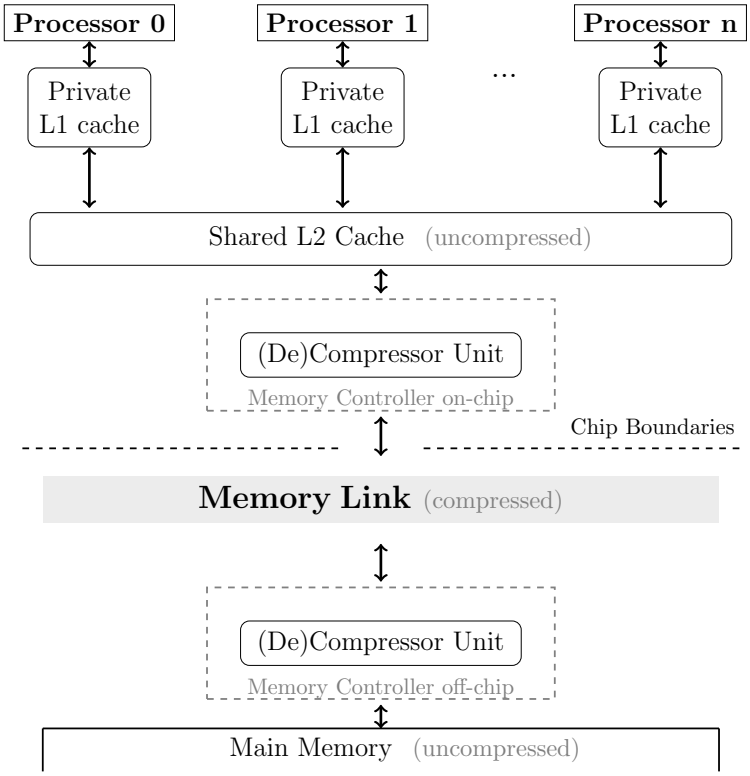


Figure 6: Link Compression Scheme on a CMP

one compression and decompression of data by the same units, due to the use of dictionaries whose consistent update is critical for the algorithm's functionality. We will come back to this matter in the following section.

Link compression affects both bandwidth and latency in a CMP. If successful, it can increase the effective pin bandwidth of the system, leading to a significant performance improvement especially of "bandwidth-bound" applications. Apart from freeing up bandwidth, it can also reduce the L2 miss penalty by reducing the transfer time, since a smaller amount of data needs to be transferred. However, both compression and decompression latencies are added in the transfer latency, as they are part of the critical memory access path. This fact dictates that only extremely light-weighted algorithms can be taken into account.

3.2 Compression Algorithms

An ideal link compression scheme would be fast, simple and effective. Thus, the compression logic implemented should:

- achieve a large average compression ratio, enough to provide the potential for performance gains (effective)
- introduce low compression and decompression latencies so that its overheads do not eliminate or overcome its benefits (fast)
- have a simple hardware implementation with low power and area overheads (simple)

The second prescription is perhaps the most important, since both compression and decompression latencies are in the memory access critical path. Thus, for the scheme to be beneficial, the (de)compression overheads must be small compared to a memory access penalty. This is feasible, though, since a memory access penalty is typically measured in hundreds of cpu cycles.

The design goals for a fast, simple and effective link compression scheme are usually conflicting. A simple scheme with low overheads may achieve modest or bad compression ratios, while a scheme of high hardware complexity and large latencies will probably be effective. Thus, the challenge of designing a link compression scheme is to find the right balance between these goals.

In this subsection we present the four different compression algorithms we applied in the link compression scheme for our experiment. A great difficulty that link compression faces, as mentioned in Chapter 2, is the fact that the block of data to be compressed is very small (a cache line) and therefore the compression opportunities are significantly reduced. In this essay we experiment with significant-based algorithms that search for redundancy at the cache line granularity and attempt to exploit the variance of the line's stored values and their

	Characteristics			Compressible data patterns			
	Decomp. Lat.	Complex.	C.Ratio	Zeros	Rep. Val.	Narrow	LDR
ZCA[31]	**Low**	**Low**	Low	✔	✗	✗	✗
FVC[29]	High	High	Modest	✔	Partly	✗	✗
FPC[19]	High	High	**High**	✔	✔	✔	✗
BΔI	**Low**	Modest	**High**	✔	✔	✔	✔

Table 1: Qualitative comparison of BΔI with prior work. LDR: Low dynamic range. Bold font indicates desirable characteristics. Pekhimenko et al's Base-Delta-Immediate Compression: Practical Data Compression for On-Chip Caches [37].

binary representation to compress them. We also study algorithms that base their functionality on dictionaries and require memory usage.

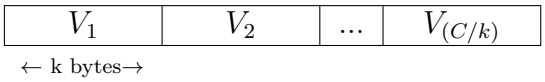
3.2.1 Base-Delta-Immediate Compression

Base Delta Immediate Compression (BΔI) is a technique described and proposed in [37] by Pekhimenko et al for cache compression. It is a lightweight significance-based compression algorithm, with no memory usage, ideal for compression on the fly since it is looking for data redundancy and compression opportunities at a cache line granularity. The key observation behind the algorithm, is that, for many cache lines, the data values stored within the line have a low dynamic range: i.e., the relative difference between values is small. In such cases, the cache line can be represented in a compact form using a common base value plus an array of relative differences (deltas), whose combined size is much smaller than the original cache line (the authors call this the base & delta encoding). Moreover, many cache lines intersperse such base & delta values with small values – BΔI technique efficiently incorporates such immediate values into its encoding. Finally, the algorithm gives special treatment to repeated values and runs of zeros, since they appear frequently in memory data. As shown in Table 1, BΔI manages to achieve a good trade-off between compression ratio, hardware complexity and (de)compression latencies. Table 1 compares BΔI with other acknowledged hardware compression algorithms presented in chapter 2, on the basis of their performance characteristics and the range of compression opportunities they cover.

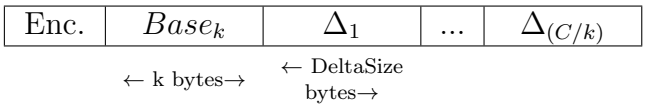
3.2.1.1 Compression algorithm

For our description, let us assume that the size of the block of data to be compressed (the cache line) is C bytes and the size of each stored value is k bytes. So the cache line is divided in $n = C/k$ number of values forming the value set S to be compressed ($S = \{V_1, V_2, V_3, ..., V_n\}$). Let us also assume that the chosen base is B. Then the encoded set of values becomes

$E = \{\Delta_1 = B - V_1, \Delta_2 = B - V_2, ..., \Delta_n = B - V_n\}$. For this algorithm a cache line is thought compressible only if $sizeof(\Delta_i) < k \ \forall i \ \in \{1, 2, ..., n\}$, where $sizeof(\Delta_i)$ is the minimum bytes needed for the Δ_i to be accurately represented. Actually, at the final implementation the criteria of compressibility will become even stricter for the sake of the algorithm's speed and simplicity as explained below.



Uncompressed Cache Line (C bytes)



Compressed Cache Line

The definition of k. One of the particularities of compressing memory resident data is that their storage type is unknown. Concerning BΔI , this becomes a crucial problem, since the algorithm searches for redundancy in the correlation of the values stored in a cache line. For example, imagine that a program's data set is an array of integer type values. The cache lines, then, should probably be divided into sets of values having the size of an integer (k → 4bytes), in order to detect and exploit the optimum (lowest) dynamic range of the values. Similarly, if the array held doubles, then probably the optimum size of values would be 8 bytes. So, the definition of k evolves as a key challenge for BΔI and, as it seems, a static definition would reduce significantly the algorithm's ability to detect redundancy and compress efficiently. Thus, the authors designed BΔI to "view" a cache line with three different value sizes, k = 2, 4 and 8 bytes (the typical sizes of most data types for the majority of the programming languages). The algorithm must then select the optimal definition of k that provides the maximum compressibility of the line.

The definition of B. Another key challenge for BΔI is to define the optimum base for the encoding. In fact, the optimum base value is either the maximum/minimum value stored in the cache line, or the value exactly in between them. This definition of B, though, requires an additional "scanning" of the entire line to compute its boundary values (min,max), and thus introduces a significant delay to the encoding and adds complexity to its hardware implementation. These drawbacks and especially the additional latency, may cause even the algorithm's performance degradation, eliminating all the benefits of the better compression ratio achieved by the use of the optimum base. In order to avoid these drawbacks the authors propose the use of the first value stored in the line as the base for the encoding.

Implementation details. To summarize the algorithm's main idea, compression is achieved by dividing in parallel each cache line into words of 2,4 and 8 bytes (k parameter) forming three different sets of values to be compressed and then by computing for each set the differences of its stored values with the relative

Name	Base	Δ	Size	Enc.	Name	Base	Δ	Size	Enc.
Zeros	1	0	1/1	0000	Rep.Values	8	0	8/8	0001
Base8-Δ1	8	1	12/16	0010	Base8-Δ2	8	2	16/24	0011
Base8-Δ4	8	4	24/40	0100	Base4-Δ1	4	1	12/20	0101
Base4-Δ2	4	2	20/36	0110	Base2-Δ1	2	1	18/34	0111
No. Compr.	N/A	N/A	32/64	1111					

Table 2: BΔI encoding. All sizes are in bytes. Compressed sizes (in bytes) are given for 32-/64- byte cache lines. *Pekhimenko et al's Base-Delta-Immediate Compression: Practical Data Compression for On-Chip Caches [37].*

base (the set's first stored value). In order to reduce the algorithm's complexity and achieve a very fast decompression procedure, the authors use for the encoding predefined compression "cases" whith the following characteristics:

(i) the possible storage sizes of the "deltas" are specific, 1,2 or 4 bytes ($sizeof(\Delta_i) \in \{1, 2, 4\}$),

(ii) all the "deltas" of a cache line's encoding have the same size (the $sizeof(\Delta_i)$ remains constant $\forall i$ in a cache line).

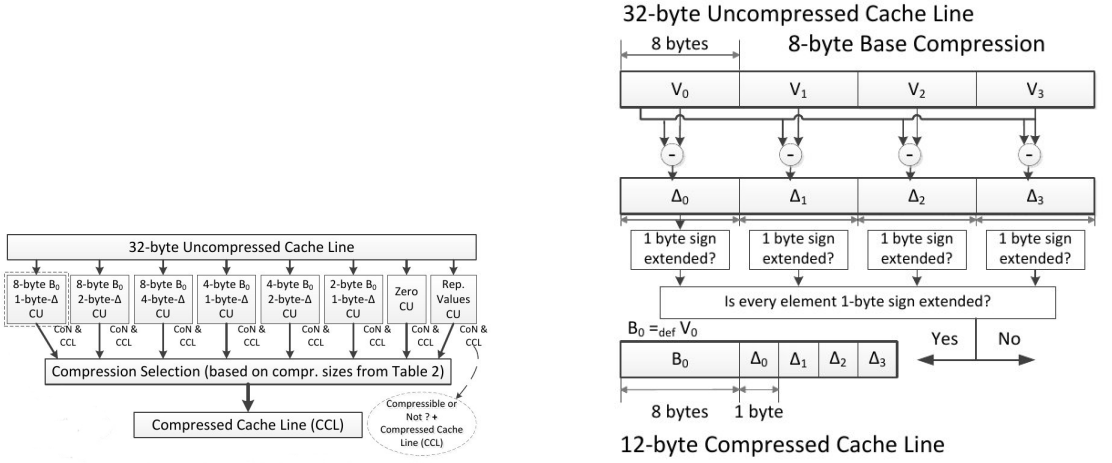
The (ii) characteristic may prevent the maximum feasible compression of a line, since all the "deltas" are stored using the number of bytes needed by the biggest (in terms of storage) "delta". On the other hand though, due to (ii), the decompression of the words does not need to be performed serially, since the starting location of an encoded value is not determined by the size of the previous compressed value. So, (ii) allows the decompression to be performed in parallel and thus to be really fast. The specific "delta" sizes (ii) and their combination with the different base sizes (k parameter) produce six specified compression "cases" that along with the zero run case and the repeated values case form the BΔI encoding as shown in Table 2.

The compression logic, demonstrated in Figure 7a, consists of eight distinct compression units that work in parallel and are in charge for the 8 different compression cases (Table 2). Every compressor unit takes the uncompressed cache line as an input, and outputs whether or not this cache line can be compressed with this unit. If it can be, the unit outputs the compressed cache line. The selection unit chooses the most effectively compressed cache line (the one with the smallest size). We should notice that the sizes of the compressed lines generated by each unit are predefined (due to specific base and delta sizes) as shown in Table 2. All compressor units can operate in parallel. Figure 7b demonstrates the encoding procedure followed by a specific compression unit. In general, the steps followed are: a) divide the line in k-sized values, b) read the first value and set it as base, c) compute the differences of the stored values with the base (in parallel), d) check if the "deltas" produced can be accurately represented (using sign-extension encoding) using the bytes allowed by the unit's predefined DeltaSize and if they

are: e) form the compressed line that consists of the base followed by the array of "deltas" in order. A pseudocode describing the basic steps of the base & delta encoding technique can be found in Algorithm 1 on page 33.

The compressed cache line finally selected, the one the selection unit outputs, has an extra 4-bit compression tag, as shown on page 26, holding the encoding (Table 2) that indicates which compression unit was used for this line's compression. This tag is essential for the decompression.

The additional zero base optimization. As expected, this compression algorithm does not always succeed. One common reason of failure is that some applications can mix data of different types in the same cache line, e.g., structures of pointers and 1-byte integers. This fact reinforces the reasonable assumption that base & delta encoding would perform better if multiple bases where used for the encoding of a cache line. The usage, though, of multiple bases comes along with the hardware overhead of their storage. This overhead is a drawback for the compression of a cache line, since the extra bases have to be stored at their original size in the compressed line. Their use, in some cases, may even worsen the average compression ratio achieved if this additional overhead gets higher than the benefit of compressing more cache lines. Another drawback of using multiple bases results from the process of their definition. The additional search that is probably required introduces a significant latency to the compression, something that, as explained before, is undesirable.



(a) Compressor design. CU: Compressor unit.

(b) Compressor unit for 8-byte Base (k=8), 1-byte DeltaSize.

Figure 7: BΔI compression. *Pekhimenko et al's Base-Delta-Immediate Compression: Practical Data Compression for On-Chip Caches [37].*

To exploit the benefits of multiple bases and simultaneously avoid the previously mentioned disadvantages, the authors propose the use of only one additional implicit base that will always be set to zero. This second base doesn't have to be stored and its definition doesn't introduce any latencies to the encoding since it is statically predefined. The selection of the zero value is based on the observation that narrow values[1] commonly coexist with large values in a cache line. Thus, the zero base allows to exploit simultaneously a second separate dynamic range close to zero and compress a cache line's small values. (The deltas relative to zero can be thought of as small immediate values, which explains the last word of the BΔI name.)

Concerning the implementation, a few changes are required. At the process of the encoding each compression unit attempts to compress all the values of a cache line using the zero base. If a value is not compressible this way then the arbitrary base (the base selected from the cache line) is used. The difference this time is that the selected base is not the first value of the set but the first value that failed to get compressed using the zero base. Finally, an extra compression tag is needed to indicate which base (zero or arbitrary) was used for the encoding of each value in the cache line.

3.2.1.2 Decompression algorithm

The decompression of a compressed line requires the generation of the original set of values $S = \{V_1, V_2, V_3, ..., V_n\}$ from the stored set of differences $\{\Delta_1, \Delta_2, ..\}$ and the used $Base\,(B)$. So, the values of the original cache line can be simply computed in parallel using a SIMD-style vector adder, since $\Delta_i = B - V_i \Rightarrow V_i = B + \Delta_i$. The B$\Delta$I decompression algorithm can be easily implemented in hardware and is extremely fast evolving as one of the major advantages of this hardware compression technique. The decompression logic is demonstrated in Figure 8.

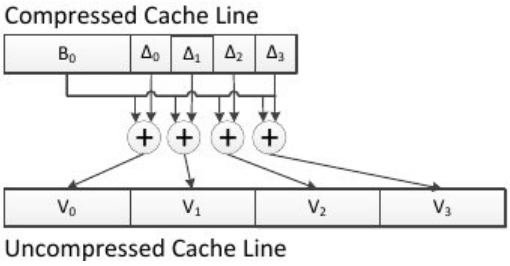


Figure 8: BΔI Decompression logic.

[1]Narrow values are values that contain top unnecessary bits introduced by sign extension. Thus, they are typically small values.

3.2.1.3 Hardware implementation and latency restraints (overhead).

Lei Fan and Martyn Romanko proposed a hardware implementation of BΔI and made its energy analysis in [53]. For their study, the authors mainly followed the design laid out by Pekhimenko, et. al [37]. We will shortly describe their design and its hardware and delay overheads since we use them as performance parameters of the compression scheme in our experiment .

Romanko and Fan assume cache lines with the original size of 64-bytes. They base their hardware design on the use of an 64-byte adder that performs both addition and substraction, the fundamental operations of this compression scheme.

Compression. As described in the previous subsection, during compression each cache line is passed through the BΔI data compression logic and the resulting compressed cache line comprises a non-zero base and the differences between the original value and either the non-zero base or zero, if compressable. A bitmask is also produced, representing which base was used for every stored value, something needed for the process of decompression.

Concerning the implementation of the compression logic Pekhimenko et al., as mentioned in 3.2.1.1, propose the use of eight distinct compression units, each of which is in charge for a different compression scheme (Table 1). The units work in parallel and in the end the most optimal compression is chosen (Figure 7a). Romanko et al., though, decided to slightly derogate from this design and sacrifice its parallel operation for the sake of the implementation's lower complexity, cost and hardware overhead. Thus, they propose the use of only one 64-byte adder (instead of one adder per compression scheme - distinct unit) shared between compression and decompression. The access to the adder from the compression blocks is time-multiplexed and priority is given to the most efficient compression schemes, ranked by the size of the compressed cache line they generate. For each compression request, the compression schemes applied at first are the ones using bases with the size of 8 bytes, since the most compressed results can be derived with an 8-byte base, then the ones with 4-byte bases etc (Table 1).

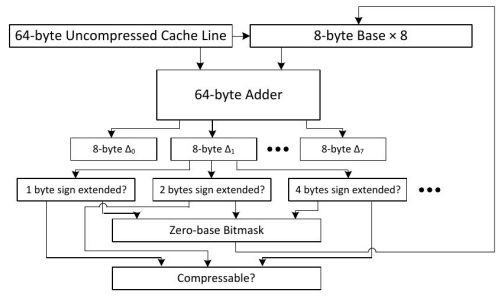
The 64-byte adder is responsible both for the substractions required for the compression and the additions required for the decompression. It is composed of eight 8-byte adders which in turn are composed of four 2-byte carry-lookahead adders, each with its own carry-in and carry-out signals (Figure 9b). This fragmented design gives the adder the ability to perform operations with 2-byte, 4-byte and 8-byte values, which are the different base widths used by the algorithm. Obviously, due to its overall width, the adder can process an entire cache line at a single cycle. Figure 9a demonstrates the compression logic centered around the adder for the schemes that use 8-byte wide bases. As it is shown, the compressibility is determined by sign extension. Every difference (delta) produced

is examined whether it is x-bytes sign-extended[2] with x defined by the applied compression scheme (x = DeltaSize). If this is true then the line is thought compressable and the deltas are stored without their unnecessary top bytes. This operation can be simply implemented using xor units.

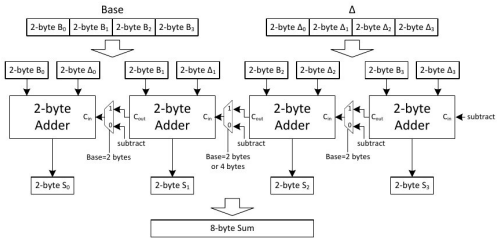
Regarding the delay introduced by compression in terms of cycles: In the first cycle there is an attempt to compress the cache line using a base of all zeros and all the non-compressible sections are saved. In the second cycle, the first non-compressed 8-byte value is set as base and there is an attempt to compress the rest of the cache line using the zeros, repeated values, or base8-Δ1 compression schemes. If this attempt is successful then the compression is completed and the compressed result is outputted. If it fails then in the third cycle compression is attempted using the base8-Δ2 scheme and this algorithm is repeated, unless compression succeeds, for all the possible schemes using 8-byte, 4-byte and at last 2-byte bases (Table1). Therefore, the compression of a cache line can last from 2 to 6 cycles.

At this point we must mention that this implementation is made for a cache compression scheme. Thus, the time overhead of the compression is underestimated since it is not a part of the critical path, as mentioned in 2.3. Therefore, reducing speed to achieve less area occupation and energy consumption is preferred in Romanko et al's design. In the case of a link compression scheme though, as

[2] Sign extension is the operation of increasing the bits of a binary number while preserving the number's sign and value. It is a technique used for the storage of small values in larger data types.



(a) Compression logic centered around the 64-byte adder.Shown with example 8-byte wide base.



(b) Internal design of an 8-byte adder. Each 8-byte adder is composed of four 2-byte adders for added flexibility.

Figure 9: BΔI implementation's adder. *Romanko et al's Implementation and Energy Analysis of Base-Delta-Immediate Compression [53].*

the one we are studying, compression is in the critical path and eliminating its time overhead is essential for the scheme's performance. Thus, an implementation of BΔI compression logic that follows strictly the description of Pekhimenko et al, where compression is performed by separate units working in parallel (Figure 7a), would be preferable. Such an implementation could probably achieve to reduce the latency introduced by compression to 1 cycle.

Decompression. This step is performed in a single cycle. The compressed cache line is provided, along with the BΔI encoding that determines the size of each segment and the bitmask that allows the decompressor to select the appropriate base for each segment, and the 64-byte adder will provide the decompressed cache line (compute all the required additions) within the same cycle, subject to the intrinsic latency of the adder.

3.2.2 Differential compression

In [45], Benini et al describe the idea of a differential compression scheme and propose three possible variations of it. The main idea behind this scheme is that it is usual for data words residing in the same cache line to have some bits in common. In particular, the proposed algorithms are based on the assumption that usually, due to the nature of many applications and their datasets, the data words stored in the same cache line take on values from a limited range and thus they may have some of their most significant bits (MSBs) in common. Therefore, compression can be achieved by storing these common bits only once. Benini et al propose the use of this scheme for hardware link compression, highlighting its advantages of low hardware complexity and low time and energy consumption overhead. We must notice that this scheme, like BΔI (3.2.1), searches for compression opportunities at a cache line granularity and doesn't use/constructs dictionaries.

3.2.2.1 Compression algorithm

For the following description of the three variations of the differential compression algorithm we assume an uncompressed cache line with the size of C bytes and n stored data words (W_i) with the size of k bits ($n = \frac{C}{(k/8)}$). It is noteworthy that the differential compression scheme is operating at a bit and not a byte granularity.

Diff1 Compression scheme. Diff1 is the basic variation of the differential scheme and its compression logic is better exemplified in Figure 10 where the form of the compressed cache line is displayed. Diff1 attempts to compress the cache line detecting the common most significant bits (MSBs) of all data words (W_i) stored in the line. The *cnt* field of the compressed line indicates the number of these common bits and since the words are k-bits wide, the maximum value that *cnt* can take is k. Therefore, the size of the *cnt* field must be $\log_2 k$ bits. The

Algorithm 1 The Base & Delta cache line compression (basic steps)

```
// k ∈ {2, 4, 8}
//Parallel For
for k ← 2, k ≤ 8, k ← k * 2 do
    // DeltaSize ∈ {1, 2, 4} & DeltaSize < k, Parallel For
    for DeltaSize ← 1, DeltaSize < k, DeltaSize ← DeltaSize * 2 do
        //The block of code in this loop is the encoding procedure performed
        //by every compression unit for a different
        //(k,DeltaSize) combination

        // "View" the line as a set of k-sized values
        CacheLine ← (k*)CacheLine
        Base ← CacheLine[0]
        Compressible_(k,DeltaSize) ← true
        Store Base in CompressedCacheLine_(k,DeltaSize)
        for i ← 1, i < CacheLineSize/k, i ← i + 1 do
            Δ_i ← Base − CacheLine[i]
            Δ_i encoded ← Remove Δ_i 's sign extension bytes
            if sizeof(Δ_i encoded) > DeltaSize then
                Compressible_(k,DeltaSize) ← false
                Break
            else
                Store Δ_i encoded in CompressedCacheLine_(k,DeltaSize)
            end if
        end for

    end for
end for
return CompressedCacheLine_(k,DeltaSize) with the miniumum
                sizeof(CompressedCacheLine_(k,DeltaSize))
```

compression is achieved by storing the first word of the line (W_0) in its original form, containing the common *cnt* MSBs, and by storing the rest of the words in a compressed form. The compressed words (WC_i) consist of the remaining $K{=}k - cnt$ least significant bits (LSBs) of the corresponding original words.

In [45] link compression is combined with main memory compression. Particularly, the authors define a fixed-size memory area called compressed memory. In order to deal with the fragmentation introduced to the compressed memory by the changing and various sizes of the compressed cache lines, the au-

thors decide to define a fixed size for the compressed cache lines equal to $S < OriginalCacheLineSize$ bytes. They refer to this size as compressed memory slot. Therefore, an extra rule is set to decide whether a line is compressable or not:

$$|W_0| + cnt + |WC_1| + |WC_2| + |WC_3|... + |WC_n| \leq S * 8 \ \ bits \qquad (3.1)$$

Our scheme does not include a compressed main memory and we will not set a predefined compressed size for the cache lines, since this would remove the possibility of optimum compression from the lines that can be compressed to fewer than S bytes. We will follow though the idea of an extra compressibility rule (3.1) in order to prevent the production of compressed lines that have bigger sizes even than the original lines ($S = OriginalCacheLineSize$). This threat will become clearer for the rest of the variations of the scheme. We will refer to S as threshold.

Diff2 Compression scheme. This variation is an optimized version of the basic Diff1 method. The rationale behind the optimization applied is that the agreement (the common bits) between pairs of data words in a cache line may be much higher than the agreement between all data words in the line. So, Diff2 method attempts to compress a line by detecting the common most significant bits (MSBs) of the sequential pairs of the line's words. Therefore, the $cnt_{0,1}$ field of the compressed cache line (Figure 10) indicates the number of the common MSBs between W_0 and W_1 while the compressed word WC_1 is composed by the $K_{0,1} = k - cnt_{0,1}$ remaining LSBs of W_1. Similarly the $cnt_{1,2}$ field indicates the number of the common MSBs between W_1 and W_2 and WC_2 is composed of the $K_{1,2} = k - cnt_{1,2}$ remaining LSBs of W_2 etc.

The storage of all the $cnt_{i,j}$ is a great overhead that increases the minimum size of a compressed line and may complicate the successful compression increasing the threat of producing compressed lines having bigger sizes than the original. It is also possible, though, that this storage overhead is compensated by the fact that the number of bits upon which the various pairs of words do agree make the size

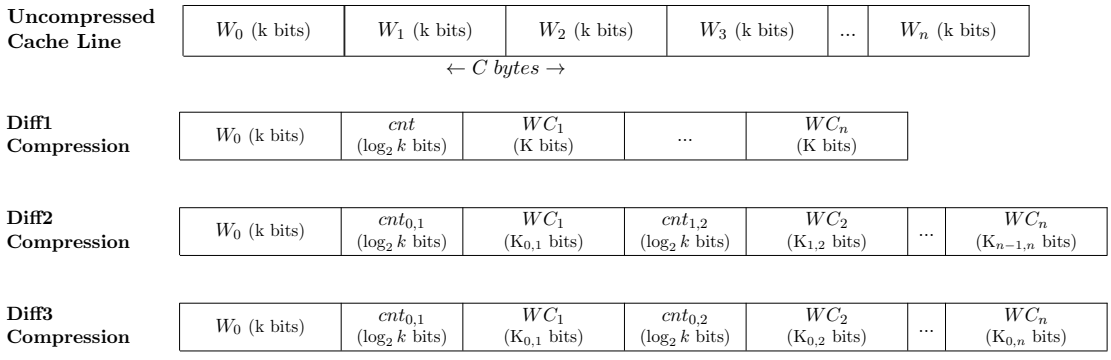


Figure 10: Differential Compression Schemes

of the $WC_{i,j}$ much shorter. Nevertheless, to avoid the worst cases we resort to the extra compressibility rule. The equation that determines the compressibility of a cache line for the Diff2 method is the following:

$$|W_0| + cnt_{0,1} + |WC_1| + cnt_{1,2} + |WC_2| + ... + cnt_{n-1,n} + |WC_n| \leq S * 8 \ \ bits \quad (3.2)$$

Diff3 Compression scheme. This last variation of the differential scheme, also shown in Figure 10, is very similar to Diff2. The only difference is that in Diff3 the agreement (the common bits) of each line's word is always calculated with respect to the left-most word in the cache line (W_0). This variation, compared to Diff2, is simpler to implement in hardware, offering lower complexity and hardware overhead, but mostly it offers a faster decompression, as explained below. Therefore, we chose to use and study only the Diff3 method in our link compression scheme. The basic steps of the Diff3 algorithm are described by the pseudocode below (Algorithm 2).

Algorithm 2 The Diff3 differential cache line compression method

```
//"view" the cache line using k bits width of words
CacheLine ← (k*/8)CacheLine
W_0 ← CacheLine[0]

//Parallel For
for i ← 1, i < CacheLineSize/(k/8) = n, i ← i + 1 do
    W_i ← CacheLine[i]
    Xor_i ← W_i xor W_0
    cnt_{0,i} ← count the leading zero bits of Xor_i
    WC_i ← the (k − cnt_{0,i}) LSBs of W_i
    Store cnt_{0,i} and WC_i in the CompressedCacheLine buffer
end for

//check if the compressed size exceeds the threshold, and if it does
//keep the line uncompressed
if sizeof(CompressedCacheLine) > S then
    CompressedCacheLine ← CacheLine
end if
return CompressedCacheLine
```

3.2.2.2 Decompression Algorithm

For all the three variations of the differential method, the decompression includes the reconstruction of the original cache line by decompressing each of its words using its stored LSBs, that form the compressed word, and its removed MSBs that are common with one or more words of the line.

Diff1. Diff1 decompression logic is very simple since it only requires to expand all the line's compressed words with the leading *cnt* bits of the first stored word (W_0). This can be simply implemented with OR logic and, as soon as the compressed words are read, it can be done in parallel.

Diff2. In this case, the reconstruction of the original cache line is a bit more perplexed, since each word's decompression requires the $cnt_{i-1,i}$ leading bits of its previous word. Consequently, a word's decompression can start only when the decompression of the previous word is completed. So, the decompression logic remains the same with Diff1, but the decompression must be done serially for all the cache line's words and thus it is slower. This important drawback of Diff2 (high decompression latency) makes it less appealing for using in a link compression scheme, even if it can achieve better compression ratios than Diff1.

Diff3. In this method the decompression of each word requires the $cnt_{0,i}$ leading bits of the first stored word (W_0). Thus, Diff3 requires a different number of W_0's MSBs for each word's decompression (difference with Diff1 - higher hardware complexity) but the decompression procedure can be done in parallel for every word (difference with Diff2) and thus it is fast. The fact that Diff3 can achieve better compression ratios than Diff1 (it is an optimized version) while preserving the advantage of a fast enough decompression procedure lead us to select it for our experiment.

3.2.2.3 Hardware implementation and latency restraints (overhead).

In this section we will attempt to describe roughly a hardware implementation of the Diff3 compression and decompression logic and estimate the corresponding delay (latencies-time overhead) in terms of CPU cycles. We assume cpu frequency equal to 1GHz, since this is the clock rate of our experiment's simulated cores. Therefore, 1 cycle equals to 1ns.

Compression. As described in 3.2.2.1, the compression logic of Diff3 consists of compressing every word of the line by detecting its MSBs that are in common with the first (the left-most) word of the line (W_0) and by removing them, storing only its remaining LSBs - the "effective" bits. Figure 11 depicts the basic steps of this compression logic. Step 1 includes xoring each word of the cache line with W_0 to find the common bits. This step could be implemented in parallel for all the line's words using multiple exor gates and it could probably last less than a cycle. Step 2 and Step 3 include the encoding and the storage of the compressed words. The $cnt_{0,i}$ prefix of each compressed word WC_i, which indicates the number of the MSBs that W_i has in common with W_0, can be calculated by

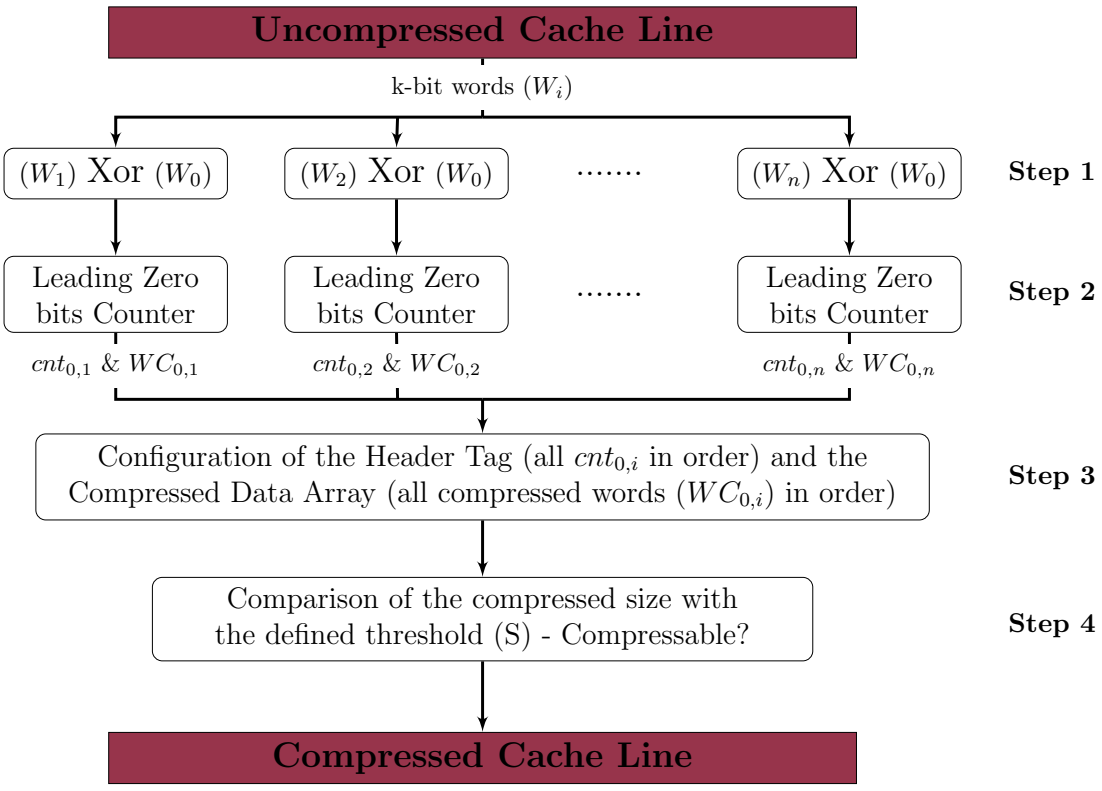


Figure 11: Differential scheme Compression Logic.

counting the leading zero bits of each xor result. This could be implemented with the use of a Leading Zero Counting unit (LZC) [54] or also known as Leading Zero Detector (LZD) [55]. Leading zero counting is the procedure of encoding in binary representation the number of consecutive zeros that appear in a word before the first more significant bit that is equal to one. It is commonly used for the normalization operation of a floating-point unit. In [54], the minimum achieved delay of the implemented 64-bit wide LZC unit is 6.5 FO4 (1 FO4 = 63ps) which is approximately 0.4ns $<$ 1 cycle. Thus, if multiple instances of a k-bits wide LZC unit were used, the leading zero bits counting step could complete in parallel for all the line's words in less than 1 cycle. Finally, for the storage of only the "effective" bits of every word (compression) and the corresponding *cnt* prefixes, multiple variable length shifters could be used combined with OR logic and multiplexing. At this point we propose an alternative layout of the compressed data, shown in Figure 12, that could facilitate decompression. The difference with the layout of Figure 10 lies in the fact that all the $cnt_{0,i}$ prefixes are stored sequentially at the left side (beginning) of the compressed line, forming a compression tag. Regarding the delay of this step, even if we can not describe in detail the circuit that could output this compressed line, we assume that it could be a single cycle. We must notice that an extra complexity is added due to the fact that the sizes of $cnt_{0,i}$ and $WC_{0,i}$ are nor predefined neither byte quantized. Finally, a comparator could be used for the decision whether the compressed size is within the predefined boundaries (threshold) forming the 5th and final step of the compression. The delay of a comparator circuit can be less than a cycle [56]. In conclusion, we estimate that the delay of a hardware implementation of the Diff3 compression logic could be of the order of 4 to 5 cycles.

Decompression. Figure 13 depicts the basic steps of the Diff3 decompression logic. The first step includes the computation of the starting bit address of each compressed data word stored in the line. Using the *cnt* tag the length of each compressed word is known (k-$cnt_{0,i}$ bits) and theese lengths could be used as an input to a multistage carry-lookahead adder network. These adders could compute in parallel every word's starting address by adding the length of the preceding words in a hierarchical fashion. A similar stage is part of the decompression pipeline of the FPC hardware implementation, as it is described in Alameldeen's thesis [41], and its estimated delay is 2 cycles. The second step includes the reading and stor-

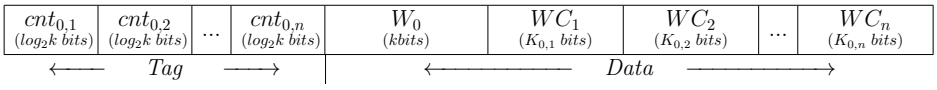


Figure 12: Alternative layout of the Compressed Line (*Diff3*)

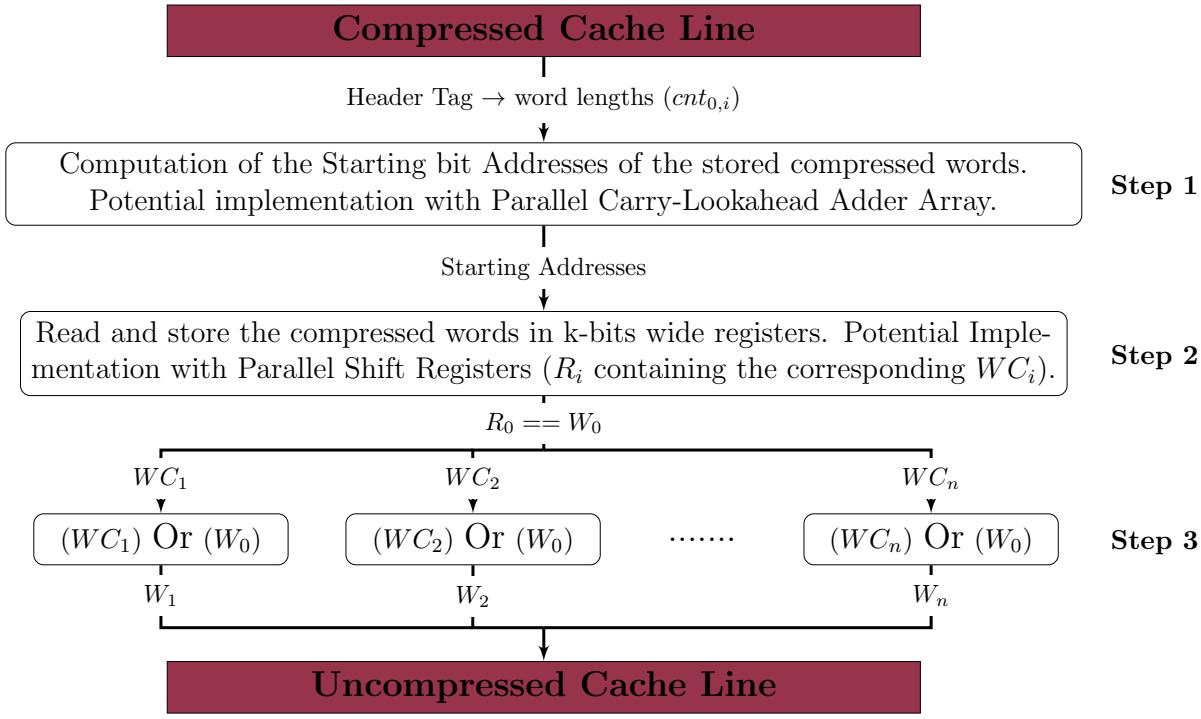


Figure 13: Differential scheme Decompression Logic.

ing of the compressed words in k-bits wide registers. For the implementation of this stage multiple parallel shifters could be used [41], and the delay of this stage could be less than 1 cycle. The third and final step of the decompression logic includes the logical OR of all compressed words with W_0 for the recreation of their common MSBs. This final stage could also complete in a single cycle. In conclusion, we estimate that the overall latency of decompression could be 4 to 5 cycles.

3.2.3 FPC double-precision floating-point data compression

FPC is a lossless, single-pass, linear-time floating-point compression algorithm recently developed by Burtscher et al.[57]. Its primary objective is to maximize the throughput while still delivering competitive compression ratio. FPC targets streams of double-precision floating-point data with unknown internal structure, such as data seen by the network or a storage device in scientific and high-performance computing systems. Its compression logic, as we will explain in detail later, is based on sequential prediction of floating-point values.

FPC is an application, a software scheme, and thus its design is not oriented towards hardware implementation. In this dissertation we attempt to apply FPC's

compression logic to a hardware link compression scheme and study its performance. Our motivation was the poor performance that most of the proposed hardware cache and link compression schemes appear to have when it comes to floating-point data. A cause of this compression failure could be found in the binary representation of floating-point. This type of data involves a high level of entropy (i.e. high irregularity), particularly in the less significant bits of the mantissa. Thus, redundancy on the bit level is hard to find while exploiting the variance of floating point values offers poor opportunities for compression too.

This difficulty to compress, though, is not unexpected. Lossless floating-point compression remains a hard problem even for complex software schemes. We chose to experiment with the FPC algorithm for the following reasons:

- FPC works well on hard-to-compress scientific datasets.
- FPC delivers average compression ratios comparable to the ratios of well-known and complex lossless compression algorithms while it is simpler and faster. Even if its speed and complexity are evaluated in software terms, it is reasonable to expect that FPC, due to these characteristics, will offer the opportunity of a less complex hardware implementation too, compared to other algorithms. It is true, though, that a hardware implementation, as light-weight as needed to be beneficial, isn't obvious nor guaranteed.
- FPC's functionality and performance does not depend on knowing the specific application domain of the data as it happens with other floating-point compression algorithms (geometric data[58], image data[59] etc.)
- FPC is a single-pass algorithm and thus the data can be compressed and decompressed on the fly. This feature is essential for a link compression scheme.

3.2.3.1 Compression algorithm

FPC compresses linear sequences of IEEE 754 double-precision floating-point values by sequentially predicting each value, xoring the true value with the predicted value, and leading-zero compressing the result. It uses variants of an fcm [60] and a $dfcm$ [61] value predictor to predict the doubles. Both predictors are effectively hash tables. The more accurate of the two predictions, i.e., the one that shares more common most significant bits with the true value, is xored with the true value. The xor operation turns identical bits into zeros. Hence, if the predicted and the true value are close, the xor result has many leading zeros. FPC then counts the number of leading zero bytes, encodes the count in a three-bit

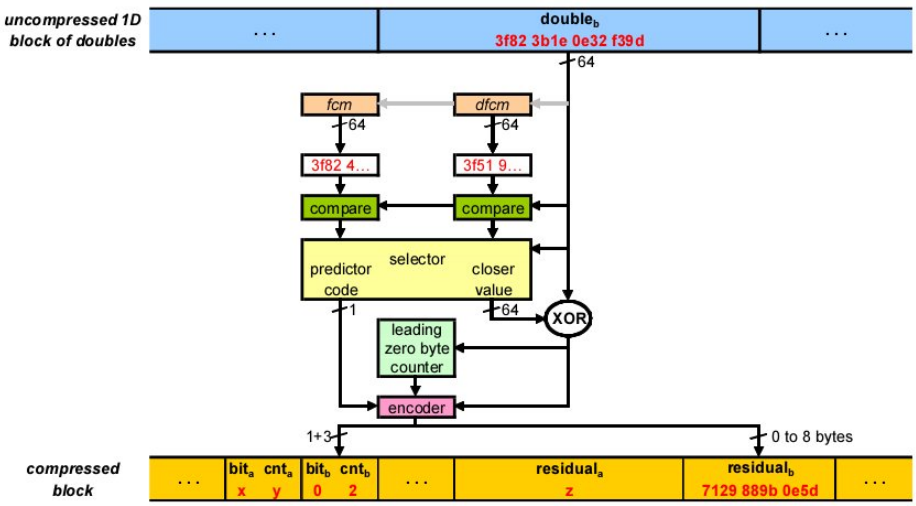


Figure 14: FPC compression algorithm overview

value, and concatenates it with a single bit that specifies which of the two predictions was used. The resulting four-bit code and the nonzero residual bytes,which form the compressed word, are written to the output. The latter are emitted verbatim without any encoding. Figure 14 depicts FPC 's compression algorithm.

Prediction. Before the start of the compression procedure both predictor tables are initialized to zero. After each prediction they are updated with the true double value. Both fcm and $dfcm$ are based on history information to predict. Particularly, the fcm hash value, the "pointer" to the fcm table, represents the sequence of most recently encountered doubles and the hash table stores the double that follows this sequence. Hence, making an fcm prediction is tantamount to performing a table lookup to determine which value followed the last time a similar sequence of previous doubles was seen. The $dfcm$ predictor operates in the same way but it predicts integer differences between consecutive values rather than absolute values. The pseudocode of Algorithm 3 demonstrates the fcm prediction logic.

The adjustment of the compression algorithm to the demands of a link compression scheme. In general terms, the compression logic of FPC, as described before, can be used unchanged in our link compression scheme. FPC interprets all doubles as 64-bit integers and uses only integer arithmetic, mainly bitwise operations, avoiding the special arithmetic operations of floating-point. This feature makes hardware implementation less difficult to design and the scheme capable to compress (less or more successfully) other types of data except floating-point (all types of data transfer over the link). We made several small changes to

Algorithm 3 The operation of the fcm predictor

unsigned long long $TrueValue$, $Prediction$, $HashValue$, $FcmTable[table\_size]$

$Prediction \leftarrow FcmTable[HashValue]$ //read predicted value from the table

$FcmTable \leftarrow TrueValue$ //update the table with the true value

$HashValue \leftarrow ((HashValue \ll 6)^{(}TrueValue \gg 48)) \,\&\, (table\_size - 1)$
//update $HashValue$. Left shift the old $HashValue$ to phase out bits from
//older values, Right shift $TrueValue$ to eliminate random mantissa bits

FPC's original source code but most of them regard implementation details. These changes aim to the algorithm's simplicity and do not alter the main compression idea and design, thus we considered them unworthy to mention.

An intervention to FPC's original design worth noting concerns prediction and the use of the corresponding tables. FPC is a software application that processes big streams of double-precision floating-point data and outputs the compressed data in blocks. During compression the predictor tables are constantly updated, having at first a small inevitable mis-prediction period until they get populated with the input stream's doubles, until they get "warmed-up". When compression is completed both tables are destroyed. In a link compression scheme the block of data to be compressed is very small, having the size of a single cache line. The idea of constructing and deconstructing prediction tables or reseting them to zero for every cache line compression can't exploit the existing compression opportunities since the tables would constantly stay in their "warm-up" stage. Thus, we propose the use of both predictor tables as dynamic dictionaries updated by every word transfered over the link, modules of the compressor unit implemented with the use of a memory device.

So, the basic steps of a cache line's compression using FPC algorithm are described in the pseudocode of Algorithm 4 and depicted on Figure 17 (page 46). Figure 15 depicts the layout of a compressed cache line. We must notice that all word's compression codes ($code_i$) are stored in the beginning of the line, forming a compression tag that facilitates decompression.

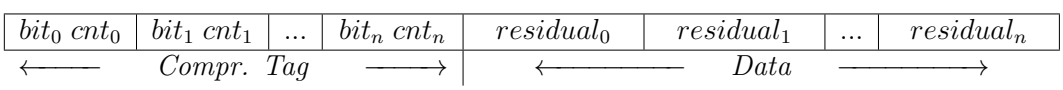


Figure 15: FPC Compressed Cache Line

Algorithm 4 FPC cache line compression

//"view" the cache line using 64-bit wide words
$CacheLine \leftarrow (double^*)CacheLine$

for each $Word$ in $CacheLine$ **do**
$TrueValue \leftarrow Word$
//Read predictions
$PredictedValueFcm \leftarrow FcmTable[HashKeyFcm]$
$PredictedValueDfcm \leftarrow DfcmTable[HashKeyDfcm]$
//Update the Prediction Tables and the "pointers"
$FcmTable[HashKeyFcm] \leftarrow TrueValue$
$DfcmTable[HashKeyDfcm] \leftarrow TrueValue$
Update $HashKeyFcm$ and $HashKeyDfcm$ history pointers using $TrueValue$ for the next prediction
//Xor the $Predictions$ with the $TrueValue$
$XorFcm \leftarrow TrueValue\ xor\ PredictedValueFcm$
$XorDfcm \leftarrow TrueValue\ xor\ PredictedValueDfcm$
//Select the $Prediction$ closest to the $TrueValue$
$XorSelected \leftarrow$ the smallest of $XorFcm$ and $XorDfcm$
Based on the above selection $bit_i \leftarrow Fcm\ or\ Dfcm$
//Leading zero Compress the $XorSelected \rightarrow$ encode the number of its leading //zero bytes to a 3-bit code
$cnt_i \leftarrow LZC(XorSelected)$
//Form the final 4-bit compression code of the $Word$, the concatenation of bit_i //and cnt_i
$code_i \leftarrow bit_i\ cnt_i$,
//Keep the $residual$ nonzero bytes
$residual_i \leftarrow$ truncate the cnt_i leading zero bytes of $TrueValue$
Output both $code_i$ and $residual_i$ to the $CompressedCacheLine$
end for

return $CompressedCacheLine$

3.2.3.2 Decompression algorithm

Decompression is the exact reverse process of compression and it uses its own fcm and $dfcm$ predictors. The decompression of a word starts by reading its corresponding four-bit code, decoding the three-bit field, reading the specified number of residual bytes, and zero-extending them to a full 64-bit number. Based on the one-bit field, this number is xored with either the 64-bit fcm or $dfcm$ prediction to recreate the original value of the word. This $TrueValue$ is used

to update both predictor tables and their corresponding "pointers" (the *Hash Keys*). This lossless reconstruction is possible because xor is reversible. Figure 19 of the following subsection depicts the basic steps of the decompression logic and their potential hardware implementation.

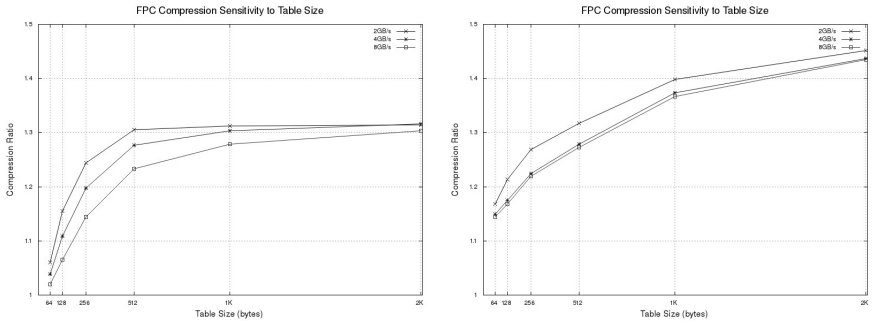
3.2.3.3 Hardware implementation and latency restraints (overhead)

FPC compared to the BΔI and Differential algorithms has the disadvantage that the cache line's words cannot compress or decompress in parallel. Serial compression and decompression are required due to the use of the predictor tables (the predictor dictionaries). In this section we will attempt to describe roughly an FPC compression and decompression pipeline and estimate the corresponding delays (latencies-time overhead) in terms of CPU cycles. We assume cpu frequency equal to 1GHz and therefore, 1 cycle equals to 1ns.

Compression. Compression, as mentioned in the previous subsections, is based on the use of prediction tables. In general terms, a word's compression requires a reading operation on the prediction table and a xor operation between the predicted value read and the true value. The compression is achieved by truncating the result's leading zero bytes that indicate the common bytes between the word and the predicted value. Thus, higher prediction accuracy leads to better compression. For better performance, two different predictors are used and the most accurate is selected for every word's compression.

At this point we must recall how prediction works. Prediction is based on history information regarding the sequence of the values recently compressed. Particularly, the "pointer" used for the table read operation represents this sequence and the table stores the value that followed the same sequence the last time it was "seen". Thus prediction is based on the assumption that specific sequences of values are repeated. For the prediction's functionality, the true value of the word being compressed at a time must update both the predictor table and the "pointer" before the next word's compression begins. This dependency makes it impossible to compress in parallel all the cache line's words. This is better shown in the Algorithm 3 on 42.

Regarding the hardware implementation of the predictor tables, content addressable memory (CAM) modules could be used. The same implementation is proposed and described in detail [49] for the frequent value table (dictionary) of the Frequent Value Encoding scheme (FVE). The use of memory is the main delay factor of the FPC compressor/decompressor and CAM is a fast memory module. An important performance parameter is the size of the CAM (the tables) that affects both compressibility and (de)compression latencies. A bigger size probably leads to better compression ratios but to higher latencies too. Thus, the right balance should be found. Figure 16 depicts the sensitivity of FPC compression



(a) LU algorithm *(b) Fw algorithm*

Figure 16: Fpc's Table Size performance parameter.

The compression is applied on data transferred over the memory link of a simulated 8-core Chip MultiProcessor for various bus bandwidths. The running applications have the same data input obtained from a scientific workload.

to the predictors' table. These partial results indicate that the algorithm can be highly sensitive to this parameter. However, a significance increase can be found from 256 to 512 bytes. Therefore, even if bigger tables can occasionally perform better we decided to use in our experiment 512 bytes tables, considering it a good trade-off taking into account the scheme's area, power and delay overheads.

Figure 17 depicts the basic steps of a word's (W_i) compression using FPC. Step 1 includes reading the predicted values from the fcm and $dfcm$ tables and updating the tables with $W_i's\ TrueValue$. In [49] the (search + update) delay of a 128 bytes (32 x 32 bits) CAM is less than 1.5ns and according to [62] a 2K bytes CAM (128 x 128 bits) has a search delay of 1.07ns, while the write operation delay is mentioned as less critical. Therefore we assume that the delay introduced to the FPC encoder by a read + update operation of our 512 bytes (64 x 64 bits) CAM could be equal to 2ns $\rightarrow$ 2 cpu cycles. This assumption grows stronger by the fact that FPC compression, unlike FVE, requires indexed CAM accesses and not CAM look-ups (the most time-consuming CAM operation). Step 1 also includes the parallel update of the tables' pointers, $HashKeys$, for the next word's compression. These keys could be stored in registers and get updated with the use of parallel shifters and xor units. Nevertheless, the 1st step's critical path is the CAM access followed by the CAM update operation and thus, the step's delay is considered to be 2 cycles. Step 2 includes the xor operation between the $W_i's\ TrueValue$ and the fcm and $dfcm$ predicted values. The delay of this step is assumed to be 1 cycle. Step 3 includes the selection of the most accurate predictor between fcm and $dfcm$. This step could possibly be implemented with the use of a comparator unit having a single cycle delay. Step 4 includes the Leading Zero byte compression of the selected xor result and one possible implementation is the use of a LZC [54] unit

having 1 cycle delay, as described in 3.2.2.3. At this point, we must notice that the cnt_i result of the LZC unit is described to be 3-bits wide while nine possibilities can be found between zero and eight leading zero bytes. For performance reasons, the leading zero count of four is not supported since it occurs only rarely. Finally, step 5 includes the formation of the 4-bit $code_i$ (cnt_i bit_i) and its storage, together with the word's $resiual_i$ non-zero bytes, to the compressed cache line buffer. Parallel shifters could be used for this step's implementation and its delay is assumed to be 1 cycle. These steps can form the stages of a compression pipeline.

Figure 18 depicts a timing diagram of the compression pipeline. If the cache line size is 64 bytes -8 words wide- the overall compression delay can be estimated as 14 cycles. Apart from the first word's encoding (7 cycles), the visible time overhead of each of the rest of the words' compression is a single cycle, due to the pipelined procedure. A possible optimization could be the immediate transmission of every newly compressed word to the bus after the completion of its compression. This way, and if the bus cycle is bigger or equal to the cpu cycle, the overhead of a word's compression (1 cycle) can be hidden by the previous word's transmission. This would leave the overhead of the first word's compression as the only "visible" one

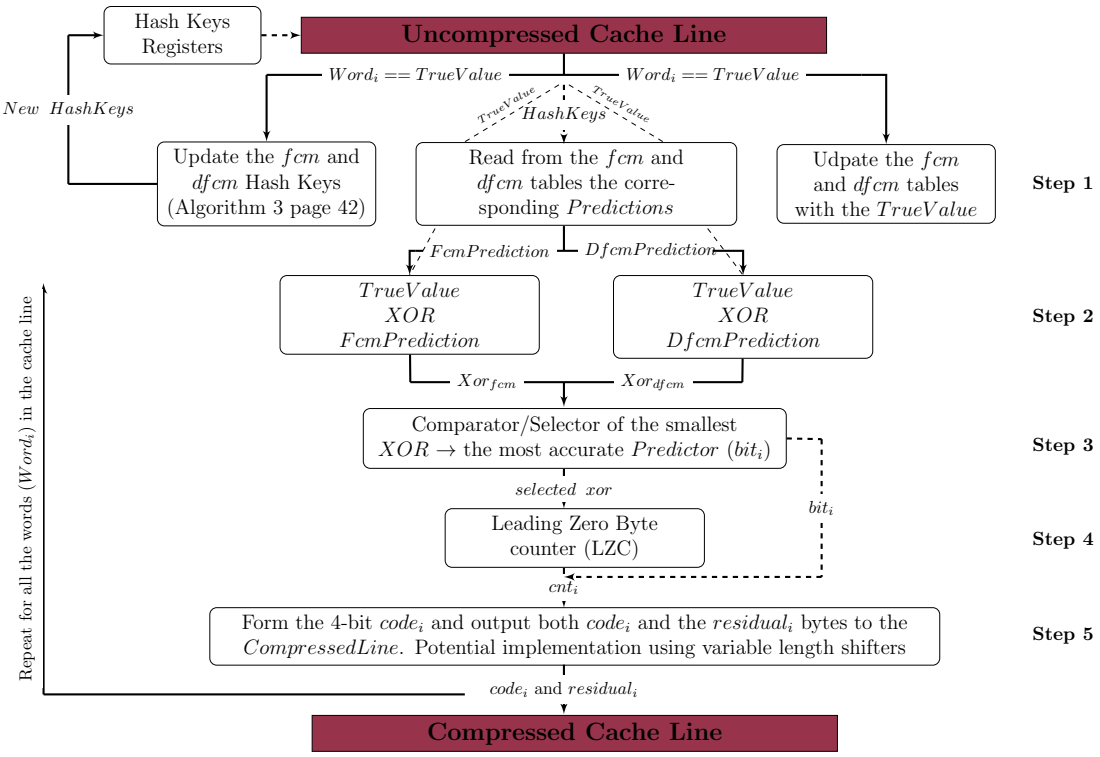


Figure 17: FPC Compression Logic

	Read the word's $TrueValue$ from the cache line (W_i)	Update the $Hash\ Keys$	Read the $Predictions$ from the fcm and $dfcm$ tables / Update the tables with the $TrueValue$	Xor the $TrueValue$ with the $Predictions$	Compare and Select the most accurate $Prediction$	Leading Zero byte counter of the Selected Xor	Encode and output the compr. $code_i$ and the $residual_i$ bytes
W_1	1	2	2-3	4	5	6	7
W_2	2	3	3-4	5	6	7	8
W_3	3	4	4-5	6	7	8	9
...	...	...	...	...	...	...	...

Figure 18: The overhead of the compression pipeline(in cycles)

and it would reduce the scheme's compression latency to 7 cycles. This technique was firstly proposed for the implementation of the FVE [49].

Decompression. The decompression logic is very similar to compression. For the shake of brevity and due to this great similarity, we will not describe thoroughly the decompression pipeline. Figure 19 depicts the basic steps of decompression logic. At this point we must notice that the starting bit addresses of each word's $residual_i$ bytes in the compressed cache line depend on the size of the previous stored $residuals$. Thus, as in the Differential algorithm (3.2.2.3), these addresses can be computed in parallel using a multi-stage carry lookahead adder network having as input the compression tag. This stage is not depicted in Figure 19. Even if decompression might be slightly faster, for simplicity we will assume that it has the same overhead with compression.

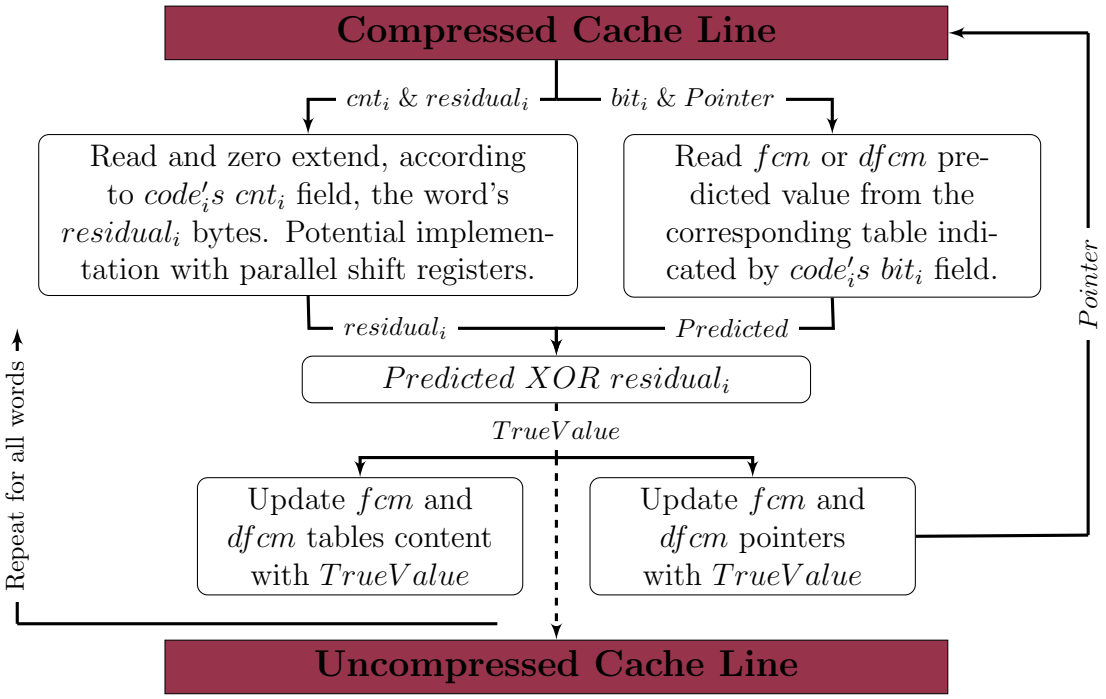


Figure 19: FPC decompression logic

Keeping compression and decompression prediction tables consistent. It is extremely important to keep the sender side encoder and the receiver side decoder consistent all the time. A major advantage of the FPC scheme is that no extra signals or data transfers are required for the maintenance of this consistency since the algorithm ensures it itself. The content of the corresponding prediction tables remains consistent as long as the compressors' and decompressors' tables are initialized with the same values and compression and decompression occurs on the fly (as it happens in a link compression scheme). This is due to the mutual "warm-up" misprediction period of the tables when they are simply loaded with values transferred over the link (mentioned on page 42).

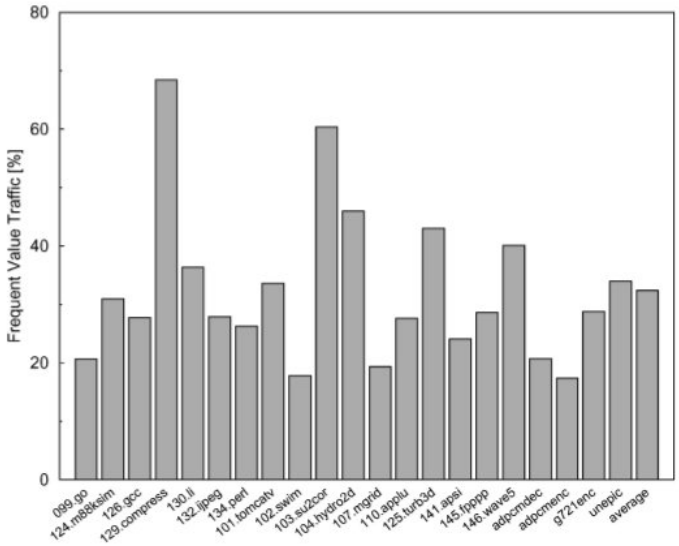
3.2.4 Frequent Value Encoding (FVE)

Frequent Value Encoding (FVE), as previously mentioned in chapter 2, is based on the observation that a small number of frequently repeated values (FV) account for a large percentage of on-chip and off-chip data traffic for many applications (Figure 20 [18]). So, the key notion of *frequent value* compression schemes, is to store these values in small value caches and encode them using the corresponding indexes. A great performance challenge, as expected, is the identification of the frequent value set.

Yang et al, who first discovered the *frequent value* characteristic, propose and describe in detail a Frequent Value Encoding scheme for data encoding on off-chip buses between the chip and memory [49]. The scheme is evaluated for energy saving, aiming to the significant reduction of the bus energy through the reduction of its switching activities. In our link compression scheme we used a simple version of this scheme, similar to the version used in [51] for compression in packet-based NoC architectures. In this subsection we describe it.

3.2.4.1 Compression algorithm

FVE compression and decompression logic is based on the use of *frequent value* (FV) dictionaries. The set of frequent values is kept in a table both by the compressor and the decompressor unit. Before the transmission of a value over the bus, the sender matches the value with its compression FV table. If a hit occurs, the value is regarded as frequent and is substituted with its index into the FV table. Otherwise the original value is passed over the data bus. Thus, the frequent values are transmitted over the bus in an encoded form while the nonfrequent values are transmitted in their original unencoded form. To distinguish between compressed (hit) and uncompressed (missed) values, an extra flag bit is attached to each value. This bit is used to indicate whether the corresponding value is compressed. In our scheme,



The diagram shows the percentage of total data bus traffic that is the result of transferring top 32 frequent values for SPEC95 and mediabench programs. The statistics are obtained by measuring the switching activity on the data bus connecting the CPU and the off-chip memory. On average, over 32% of values transmitted are frequent values and this number reaches 68% for compress.

Figure 20: Data bus traffic due to 32 frequent values.

since an entire cache line has to be compressed at a time, the described procedure is repeated for all the words of the line.

Identifying the frequent value set. The identification of an application's frequent value set is critical for the performance of FVE. Using fixed values to preset both the encoder and the decoder is a possible solution. In this case the values must be known beforehand and a profiling run of the program is required. This technique, though, is not expected to give the best performance since it is not adaptive to the application's behavior over time. It is common that a value with high frequency in one span of time doesn't occur as frequently in another span of time during a program run.

To capture dynamically appeared frequent values, the FV table must be updated periodically at runtime. It is crucial, for consistency reasons, that the update is mutual and synchronized both for the sender (encoder) and the receiver (decoder) side. The fact that data blocks are compressed and decompressed on the fly in a link compression scheme gives the opportunity to use dynamically updated FV dictionaries and maintain the desired consistency without extra overhead. We will come back to this later on the same subsection.

When replacing new values into the FV table, two factors must be considered. On one hand, replacing new values more aggressively leads to faster adaptivity

to workloads dynamic behavior. On the other hand, however, it is desirable to give old values enough time to take effect before they are evicted. Different FV replacement policies exhibit different trade-offs. We chose to use a counter-based replacement policy described in [51]. With this replacement policy, each entry in the FV table contains, apart from a frequent value, an additional 8-bit counter. The update of the table occurs when the compression of the entire cache line is completed. During compression, the controller tries to match each line's value in the FV table. A hit increases the corresponding entry's counter by two while multiple appearances of the same value update the same hit-entry multiple times. At the end of processing, the counters of all the table's entries, whose values weren't found in the line, are decreased by one. The counter value ranges from 0 to 255 and it does not overflow or underflow i.e. increasing a counter with value 255 gets 255 while decreasing a counter with value 0 still gets 0. Finally, the controller tries to update the table based on the counter values. If there are cache line values that missed the FV table and there are also table entries whose counter is zero (miss-entries), the controller does the corresponding replacements until no distinct miss values or zero-counter entries can be found.

Algorithm 5's pseudocode describes the logic of a cache line's Frequent Value Encoding, irrespective to the replacement policy used and assuming that the update of FV table occurs, if necessary (miss event), after every word's encoding. We must notice that in Algorithm 5 all the encoded words are packed in a buffer (the compressed cache line) before they are passed over the bus. In this occasion, all the *encoding_bits*, indicating whether a word is encoded or not, should be stored together forming a compression tag to facilitate decompression. FVE algorithm, though, as we will discuss in 3.2.4.3 can be implemented in a way that words are transmitted individually over the bus as soon as they are compressed.

In the FVE scheme the size of the compressed words is predefined and depends on the original word size (k) and the size of the FV table. An m-entries table requires $log_2 k$ bits to be indexed and thus a compressed word needs m+1(*encoding_bit*) bits to be represented. An uncompressed word, respectively, needs an additional bit (*encoding_bit*) to its original k bytes.

3.2.4.2 Decompression

FVE's decompression process is the reverse of compression. When a word arrives at the decoder, the *encoding_bit* is read. If the word is in a compressed form the stored *index* is used to access the corresponding entry of the decoder's FV table and the $TrueValue$ is read. Otherwise, the word is uncompressed and remains the same. In both cases the FV table must be updated.

Algorithm 5 FVE cache line compression

```
//"view" the cache line with k-bytes wide words
CacheLine ← (k*)CacheLine
CompressedCacheLine ← null

for each Word_i in CacheLine do
    {hit, index}← Search the FV table trying to match Word_i
    if (hit) then
        encoding_bit ← 1
        encoded_word_i ← encoding_bit ∪ index
    else
        encoding_bit ← 0
        encoded_word_i ← Word_i
        Update the FV table with Word_i
    end if
    Update the FV Replacement Policy
    CompressedCacheLine ← CompressedCacheLine ∪ encoded_word_i
end for

return CompressedCacheLine
```

3.2.4.3 Hardware implementation and latency restraints (overhead)

In this section we will shortly describe a possible hardware implementation of the Frequent Value compression scheme based on the detailed descriptions found in [49] and [51]. We will also estimate the overhead of compression and decompression (the delay they introduce) is terms of CPU cycles. We assume 1GHz CPU, thus 1 cycle $\rightarrow$ 1 ns.

The FVE decoder has a symmetric structure as the encoder, therefore we will not describe them separately. The basic modules of the (de)compression unit are the FV table and its controller. The table can be implemented as a content-addressable memory (CAM). A critical performance factor is the table's size. A bigger FV dictionary leads possibly to better compression performance but introduces higher latency too. We decided to use a 64-entries FV table, since this is the table size we used in our implementation of FPC compression scheme (3.2.3) and it is also the table size used by Stenstrom et al in [50]. Thus, the size of the CAM used in our experiment is 64x64=512 bytes.

The critical path of a word's FV encoding consists of a CAM look-up followed by a CAM update (in the case of a miss-event). According to [49] this path's delay can be 1.5 ns using a 32x32 CAM for a slightly more complex FV scheme. Moreover, according to [62] and as mentioned in 3.2.3.3, a 2KB (128x128) CAM can

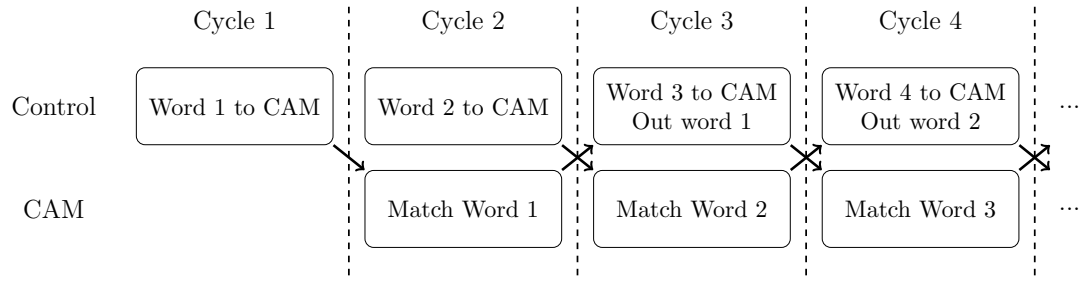


Figure 21: Pipelined operation of CAM and controller [51]

have a look-up delay of 1.07ns. Therefore, even if we can not describe in detail our scheme's CAM update unit -part of the controller- we assume that it is possible for every word's encoding to last 2 cycles. The CAM and control circuits can operate in a pipelined way, as shown in Figure 21, so that the total cycles needed to process N values (cache line) are N +2. Furthermore, according to [49], it is possible the transmission of a compressed word over the bus to start immediately after the end of its encoding. This transmission, using a fast system bus, can overlap completely with the next word's encoding. This way, the only visible latency of multiple words' compression is the delay of the first word's encoding and the overhead of the compression overhead of an entire cache line is reduced to 2 cycles. Regarding decompression, we assume the exact same overhead with compression, even if it might be a slightly faster procedure since CAM is not "looked-up" but accessed with the use of an index.

The consistency of the decoder/encoder FV tables. The proper, lossless decompression of a word requires that both the encoder and decoder FV tables contain the same values and the same indices for every value. Thus, the consistency of these tables is crucial. The FV scheme guarantees this consistency itself, requiring no extra signals on the bus, because compression and decompression occur on the fly and thus the update of both tables is synchronized. It is essential, though, that the same replacement policy is used on both sides.

Chapter 4

Experimental evaluation

4.1 Simulation Tool

We modeled and evaluated our link compression scheme using the Sniper multicore simulator [63]. Sniper is an execution driven, application-level simulator for multicore architectures based on the Graphite simulation infrastructure [64]. This 'multi-core on multi-core' simulator is completely parallel and its major improvement compared to Graphite is the addition of the interval core model [65], a high abstraction simulation approach based on mechanistic analytical modeling.

A simulation in Sniper consists of executing a multi-threaded application and modeling its behavior on a target multicore architecture defined by the simulator's models and runtime configuration parameters. The simulation runs on a single host multicore machine. In this section we use the term *target* to refer to the simulated architecture and the term *host* to refer to the physical machine where the simulation executes.

Sniper, as Graphite, is a simulation tool based on dynamic binary translation. Particularly, it is building upon Pin, a dynamic instrumentation tool.

4.1.1 Pin: a dynamic binary instrumentation tool

Pin [66] is a free tool for dynamic instrumentation of program binaries provided by Intel. Code running under Pin can be dynamically instrumented to insert arbitrary C/C++ code in arbitrary places. The instrumented code is cached and reused, so that one only has to pay the cost of instrumenting the code once. Pin defines many logical entities within a binary from a single instruction to more complex blocks of code such as sequences of instructions with a single entry and a single exit point (conditional or unconditional branch) or routines etc. It allows the code to be instrumented at various granularity levels and one may specify weather the inserted code executes before or after the corresponding application code.

The code to be inserted into the instrumented application, as well as the places where it should be inserted, is specified in a *Pintool*. The Pintool registers *instrumentation routines* with Pin that are called whenever Pin generates new code. The instrumentation routines inspect the new code and decide where to inject calls to analysis routines, also defined in the Pintool, that are called at run time. Various static and dynamic information, such as the thread id of the thread executing the instruction, the values of various registers, various properties of the code block being instrumented (e.g. the addresses of memory references for a memory access instruction) etc. may be passed to the analysis routine. Pin as well as the Pintool reside in the application's address space.

Sniper's back end, including the various performance models as well as the functional features, reside in a Pintool. The simulator uses the DBT front-end to modify the application to insert simulator callbacks. It extracts dynamic information (memory addresses etc) about the program execution and use it to simulate the behavior of the running application on the target architecture. Thus, many classes of instructions, such as arithmetic and logical operations or memory references do not need to be emulated (functionally implemented) and run natively on the host machine, providing significant speedup. Sniper may also use Pin to control or change the execution of the program at special events (a thread spawn etc.). In general, Sniper relies on the host system for correctness, since it is running in a single host machine and is residing in the same address space with the application.

4.1.2 System architecture

Sniper is a high-speed parallel multi-threaded simulator running on a single multicore host machine. Every thread of the executed application is mapped to a simulated, target "core" (Figure 22). Each target core can only execute a single application thread at a time, and the number of threads in the application at any time cannot exceed the total number of cores in the target architecture as specified in the runtime configuration parameters. Simulation results are unaffected by the host configuration and the simulator scales with the number of simulated cores.

Figure 23 depicts the components of a target architecture in Graphite. The architecture contains a set of tiles interconnected by an on-chip network. Each tile is composed of a compute core, a network switch and a part of the memory subsystem. Tiles may be homogeneous or heterogeneous. The memory subsystem is composed of several modules as instruction-data caches and Dram controllers, each associated to one of the simulated tiles and connected using the network layer. The memory system is responsible for simulating the cache hierarchies, memory controllers and cache coherence engines of the target architecture. Sniper has the improvement of the addition of a shared multi-level cache hierarchy supporting write-back first-level caches and an MSI snooping coherency protocol.

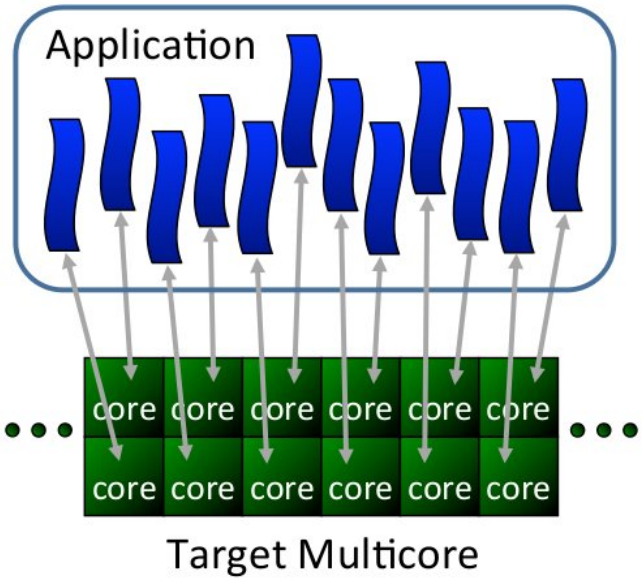


Figure 22: Every application thread is pinned to a simulated core

We integrated (de)compression in the Dram controller module (the controller for the off-chip memory). Pseudocode of Algorithm 6 depicts the "load" routine of the controller and the "store" routine is almost the same.

Algorithm 6 Get Data from Dram (address)

$AccessMemory\,(READ, address, data\ block)$
$data\ length\ = CacheBlockSize$
if $(Compression\ Enabled)$ **then**
    $data\ length\ = compressor \rightarrow Compress\,(data\ block, compressed\ block)$
end if

// Run Dram's performance model including default penalties,
// queuing delays and transfer delays
$DramAccessLatency = RunDramPerformanceModel(address, data\ length)$

if $(Compression\ Enabled)$ **then**
    $compressor \rightarrow Decompress(compressed\ block, data\ block)$
end if

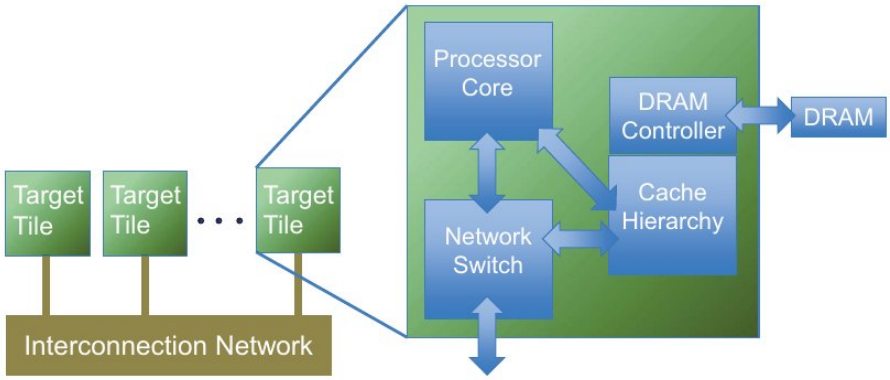


Figure 23: Target architecture modules in Graphite

4.1.3 Simulation Accuracy

A key problem in parallel simulation is to accurately model timing at high speed. Cycle-by-cycle simulation advances one cycle at a time, and thus the simulator threads simulating the target threads need to synchronize every cycle. Whereas this is a very accurate approach, its performance may be reduced because it requires barrier synchronization between all simulation threads at every simulated cycle. If the number of simulator instructions per simulated cycle is low, parallel cycle-by-cycle simulation is not going to yield substantial simulation speed benefits and scalability will be poor. Relaxing timing synchronization among the simulated cores improves simulation speed at the cost of introducing modeling inaccuracies.

Sniper by design is not a cycle-accurate simulator, it trades some accuracy for simulation speed by allowing simulated cores to run independently with relaxed synchronization. There is no global system clock and each core maintains and updates its own local clock according to the operations it performs. These local clocks are allowed to go out of synch with the clocks of other cores. A number of different synchronization strategies have been proposed. When taken to the extreme, no synchronization is performed at all, and all simulated cores progress at a rate determined by their relative simulation speed (*Lax synchronization*). To keep the simulated clocks in a reasonable agreement, application events are used to synchronize simulated threads, but otherwise they run freely. Another more popular and effective approach is based on barrier synchronization. The entire simulation is divided into quanta, and each quantum comprises multiple simulated cycles. Quanta are separated through barrier synchronization. Simulation threads can advance independently from each other between barriers, and simulated events become visible to all threads at each barrier. This could be

used for validation of lax synchronization, as very frequent barriers closely approximate cycle-accurate simulation. Sniper's default synchronization strategy is *Lax Barrier* and the quantum size is 100 cycles.

Lax synchronization is best viewed from the perspective of a single tile. All interaction with the rest of the simulation takes place via network messages, each of which carries a time-stamp that is initially set to the clock of the sender. These timestamps are used to update clocks during synchronization events. A tile's clock is updated primarily when instructions executed on that tile's core are retired. With the exception of memory operations, these events are independent of the rest of the simulation. However, memory operations use message round-trip time to determine latency, so they do not force synchronization with other tiles. True synchronization only occurs in the following events: application synchronization such as locks, barriers, etc., receiving a message via the message-passing API, and spawning or joining a thread. In all cases, the clock of the tile is forwarded to the time that the event occurred. If the event occurred earlier in simulated time, then no updates take place.

Lax synchronization, even when *barriers* are used, means that the cores' local clocks do not always agree, and events may be seen and processed out-of-order in simulated time. This leads to many challenges in modeling certain aspects of system behavior, such as network contention and DRAM access latencies. We will focus on some of the difficulties occurring in the memory subsystem modeling, since this primarily affects our experiment.

The general strategy to handle out-of-order events in Sniper is to ignore simulated time and process events in the order they are received. An alternative is to re-order events so they are handled in simulated-time order, but this has some fundamental problems. Buffering and re-ordering events leads to deadlock in the memory system, and is difficult to implement anyway because there is no global cycle count. Alternatively, one could optimistically process events in the order they are received and roll them back when an earlier event arrives, as done in BigSim [67]. However, this requires state to be maintained throughout the simulation and hurts performance. This complicates models, however, as events are processed out-of-order. Queue modeling, e.g. at memory controllers and network switches, illustrates many of the difficulties. In a cycle-accurate simulation, a packet arriving at a queue is buffered. At each cycle, the buffer head is dequeued and processed. This matches the actual operation of the queue and is the natural way to implement such a model. In Sniper, however, the packet is processed immediately and potentially carries a time-stamp in the past or far future, so this strategy does not work. Instead, queuing latency is modeled by keeping a window of the most recently-seen packet timestamps. Because messages are generated frequently (e.g., on every cache miss), this window gives an up-to-date representation of queue's

global progress even with a large window size while mitigating the effect of outliers. Error is definitely introduced because packets are modeled out-of-order in simulated time, but the aggregate queuing delay is a quite valid approximation.

Apart from queuing delays, another factor that introduces inaccuracies regarding memory operations is that requests are performed to completion as soon as they are received. Thus, even if a single cycle quantum is used for *barrier synchronization*, memory operations simulation is still non cycle-accurate. Moreover, Sniper has a very simple Dram model assuming no row buffer locality and infinite number of banks.

4.2 Experimental Methodology

4.2.1 Applications

For the evaluation of our link compression scheme statistics where gathered running simple parallel versions of memory bandwidth bound scientific applications:

- the LU decomposition algorithm, a matrix version of the Gaussian elimination usually used to solve square systems of linear equations (Algorithm 7)
- the Floyd-Warshall algorithm, a graph analysis algorithm for finding shortest paths in a weighted graph (Algorithm 8)
- the Sparse Matrix-Vector multiplication algorithm, using the CSR (Compressed sparse row) format for the storage of the sparse matrices (Algorithm 9)

We experimented with both floating point and integer input datasets:

- Floating Point: We run the SpMV algorithm on a set of sparse matrices that arise in real applications. For the LU and FW algorithms we used input double matrices initialized with values parsed from these sparse matrices.
- Integer: We run the FW algorithm having as input weighted graphs holding information for USA roads.

Algorithm 7 LU algorithm

```
for (k = 0; k < |V|−1 ;k++) do
  parfor (i = k+1; k <|V|; i++) do
    l = V_{i,k}/V_{k,k}
    for (j = k + 1; k <|V|; j++) do
      V_{i,j} = V_{i,j} − l ∗ V_{k,j}
    end for
  end parfor
end for
```

Algorithm 8 FW algorithm

```
for (k = 0; k < |V|; k++) do
  parfor (i = 0; k <|V|; i++) do
    for (j = 0; k <|V|; j++) do
      V_{i,j} = min{V_{i,j}, V_{i,k} + V_{k,j}}
    end for
  end parfor
end for
```

Algorithm 9 SpMV algorithm

parfor $(i = 0;\ i < |V|;\ i{+}{+})$ **do**
  for $(j = row\_ptr[i];\ j < j < row\_ptr[i+1];\ j{+}{+})$ **do**
    $y[i] += values[j] * x[col_ind[j]]$
  end for
end parfor

4.2.2 Simulated CMP - Default Parameters

Because of simulation constraints and the limited time of this experiment, we used a real configuration of the simulated CMP only for the SpMV algorithm. For the LU and FW algorithms we used a proportionally smaller configuration (regarding the memory subsystem). In both cases we have opted for an out-of-order processor model. Table 3 depicts the baseline parameters for the different applications and Table 4 resumes the overheads of the used compression algorithms.

Table 3: 8-core CMP with private L1 and shared L2 data caches,1GHz CPU

Baseline parameters	Lu & FW	SpMV	
no. cores	8	8	
L1 cache size	4	32	Kb
L2 cache size	256	4096	Kb
L2 associativity	8	8	
L2 block size	64	64	b
L2 cache hit time	8	8	cycles
shared cores (L2)	4	4	
Memory controllers	1	1	
Dram access penalty	100	100	cycles
Prefetchers	none	none	

Table 4: (De)compression overheads in cycles

	comp. o.h	decomp o.h
Diff3	5	5
BDI	6	1
FVE	10	10
FPC	20	20

Regarding the memory bus bandwidth parameter, we collected statistics for 2GB/s, 4GB/s and 8GB/s. Our aim was to create different "traffic conditions" on the bus and evaluate the link compression scheme's effect accordingly.

4.2.3 Methodological approach - Metrics

The metrics we used to study and evaluate the link compression scheme are: i) the Dram average access latency, ii) the compressibility of the scheme and iii) the system speedup (overall performance). The Dram average access latency is a statistic collected straightforward from the simulations. The other two metrics definitions are:

- $$Compression\ Ratio = \frac{Original\ Data\ Size_{(Total)}}{Compressed\ Data\ Size_{(Total)}}$$
- $$System\ SpeedUp = \frac{Exec.\ Time\ without\ compression}{Exec.\ Time\ with\ compression}$$

We must note that we've collected statistics from multiple runs of each experiment (different compressors) with and without compression. The final results presented in the next section (4.3) are an average of the collected statistics excluding outliers.

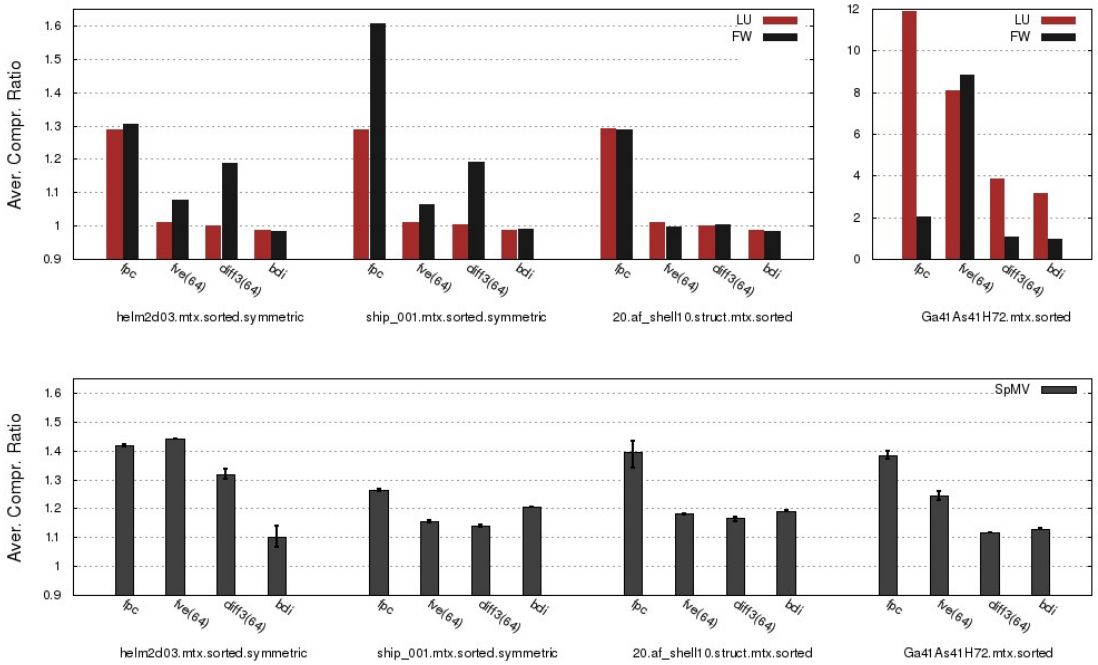
4.3 Experimental Results

This section focuses on the evaluation of the link compression scheme. The compressibility of the scheme, its impact on the DRAM access latency and the system's overall performance are examined in sections 4.3.1,4.3.2 and 4.3.3 respectively. Some important notices for the evaluation of the link compression scheme:

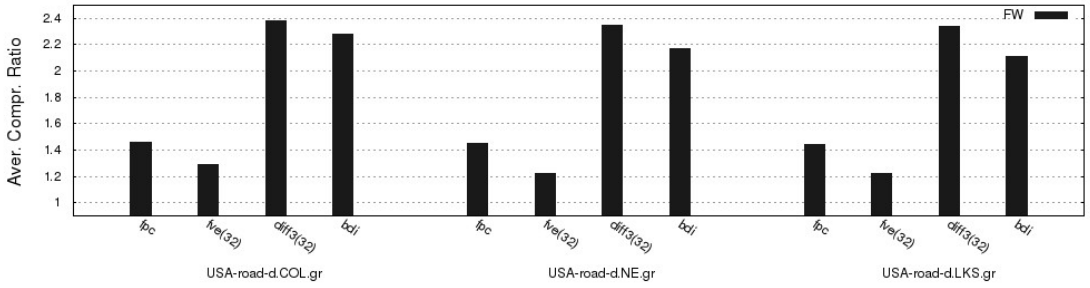
- For the SpMV application a realistic CMP configuration has been used while for the rest of the applications a proportionally smaller. In both cases the size of the input datasets has been selected to be much bigger than the system's last-level cache. Therefore, it is not right to make a straight comparison of the SpMV results with the results of the FW and LU algorithms. Thus, we decided to present their corresponding results separately. In general, though, we believe that the qualitative results of the FW and the LU algorithms would not differ much for a realistic configuration, and some random corresponding simulation results enhance our assumption.
- Regarding SpMV we must also notice that the combination of Sniper's inaccurate nature and SpMV's lack of explicit thread synchronization leads to a variation between the results of identical simulation runs. This makes SpMV results less accurate. We're using error bars to the corresponding plots to highlight this fact and indicate the extend of the phenomenon.

4.3.1 Compressibility

In this section we examine the performance of the four different compression algorithms used in our link compression scheme. Figure 24a depicts the average compression ratio achieved for the FW, LU and SpMV applications using floating point datasets, while Figure 24b depicts the average cr for the FW application using integer datasets. Compression ratios (CR) < 1 are due to compression tags overhead.



(a) floating point



(b) integer

Figure 24: Average CR = (Original Data Size)/(Compressed Data Size)

Floating Point datasets. We present the FW and LU results together, since these two algorithms have similar behavior and they are both basically streaming applications having access patterns unsuitable for data reuse. SpMV is a pure streaming application but for the reasons mentioned in 4.3 we present its results separately.

In general terms, FPC has the best performance. It achieves 20% $\rightsquigarrow$ 37.8% reduction of the off-chip memory traffic. This is quite a successful result due to the previously mentioned (3.2.3) difficulty of compressing floating point data.

For the FW and LU applications, the rest of the algorithms do not achieve any compression for most of the input matrices. Exceptions are: the bitwise differential algorithm achieving a ∼16% compression for two of FW input matrices and the Frequent Value Encoding scheme achieving ∼5% compression for the same input. There are also some special results presented for the last input matrix (Ga41As41H72.mtx). This matrix has occasionally repeated sequences of values offering advanced compression opportunities especially for the FVE and FPC schemes. Moreover, in the LU algorithm, due to the constant divisions, it is possible that an important amount of zero or one values is produced increasing the compression efficiency of the bitwise algorithms too.

It is interesting to notice that compressors tend to perform better in the FW application. We assume that an important factor is the arithmetic operations that these algorithms perform. FW performs additions while LU divisions which tend to change values fiercely, creating high entropy in the data. Another reason could be a more beneficial (regarding compression) data access pattern.

For the SpMV algorithm, FPC still has the best performance but none of the compressors has a negative impact as it happened for the FW and LU algorithms. Bitwise compressors (Diff3 and BDI) achieve $\sim$ 10% compression on average and BDI even manages $\sim$ 17% comrpession for one of the input matrices. The FVE appears to achieve ∼13,4% $\rightsquigarrow$ 19,6% and even $\sim$ 30% compression for the first matrix (helm2d03.mtx).

Integer datasets. For the integer datasets the results are quite different. The bitwise compressors have the best performance. This is not unexpected, though, since these algorithms stem their efficiency from redundancy found in stored values representation and variance. This type of redundancy can mainly be found in integer data types. Differential algorithm achieves 57.3% $\rightsquigarrow$ 58% compression and BDI 52.7 $\rightsquigarrow$ 56.2%. One possible reason for differential algorithm's better performance is the fact that it works on a bit granularity compared to BDI that works on a byte granularity. Therefore, Diff takes full advantage of the existing redundancy. Regarding FPC we must notice that it may not have such a good performance but it still manages to reduce off-chip traffic at $\sim$ 31%. FVE achieves ∼17% $\rightsquigarrow$ ∼22% compression.

We must highlight the importance of the selection of data-type granularity (1,2,4 or 8 bytes) for the FVE and Diff algorithms. These algorithms use a statically defined "word size" to process and compress a cache line. In the results of Figure 4.3 an 8-byte "word size" is used for the fp datasets, while a 4-byte for the integer datasets. The sensitivity of the compressors performance to this factor is shown in Figure 25 for an integer matrix. A major advantage of the BDI algorithm compared to the differential is that BDI processes a cache line for multiple "word sizes" and finally selects the best.

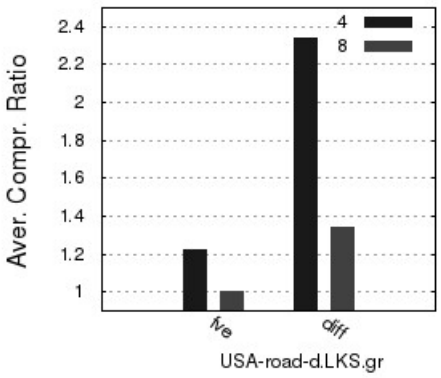
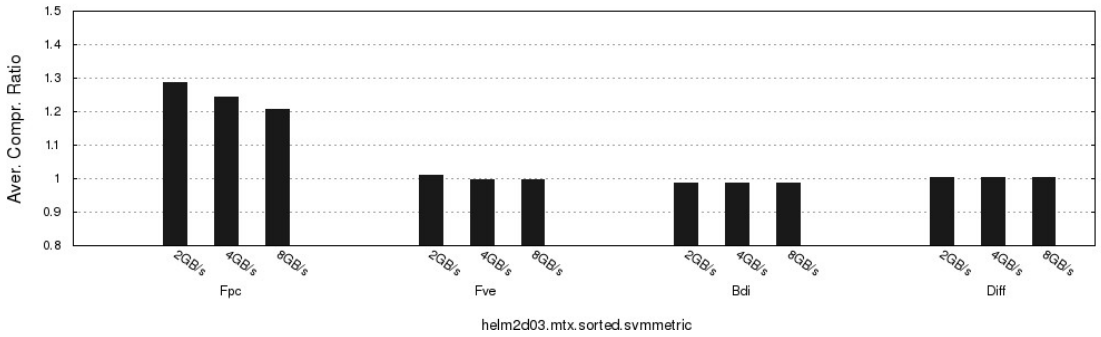


Figure 25: Sensitivity of FVE and Diff to the data-type granularity

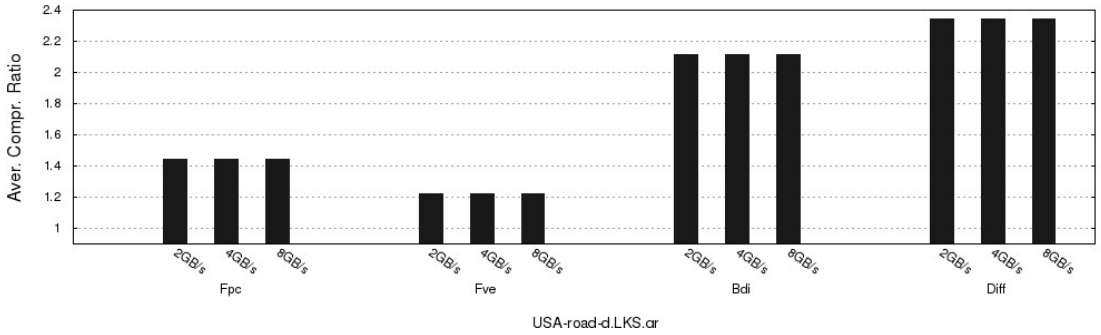
Factors not examined. We must notice two factors that we did not examine/study or take into consideration while they may affect to some extend the evaluation of the results, especially in the following sections:

- We did not examine the variance of the compression ratio over time. Particularly, we did not collect statistics from multiple execution points of an application. We calculated compression ratio by dividing the total bytes transferred over the link without and with compression.
- We did not take into consideration the fact that the compression ratio might slightly change for different bus bandwidths. Figure 26 shows that this might affect FPC algorithm for floating point datasets. This is due to the fact that the access pattern might change for different BBs. The ratio we've presented is the ratio achieved for BB = 2GB/s.

However, we believe that not taking into consideration these factors may slightly affect our interpretation of the following results but it does not change the "big picture" nor our conclusions.



(a) LU - floating point



(b) FW - integer

Figure 26: Compression Ratio for different bus bandwidths

4.3.2 Effect on Dram Access Latency

In this section we focus on the effect of the link compression scheme on the average Dram access penalty (latency). In general terms, the latency associated with a memory access in Sniper is composed of the Dram access penalty (fixed 100 cycles), the transfer over the link and the queuing delays due to the competition of the cores for the shared resource. The link compression scheme can affect only the transfer time and the queuing delays.

Figure 27 depicts the effect of an ideal compression scheme on the average Dram Access Latency for the FW algorithm using two different fp matrices as input. With the term ideal we refer to a scheme that introduces no delays (has zero time (de)compression overheads). We notice that the main impact of compression is on the queuing delays. It slightly affects the transfer time over the bus, but mainly if no contention exists then the scheme doesn't offer any significant improvement. Therefore, as the bandwidth increases from 2GB/s to 8GB/s the scheme's ideal effect is dramatically reduced until it is nearly eliminated (8GB/s).

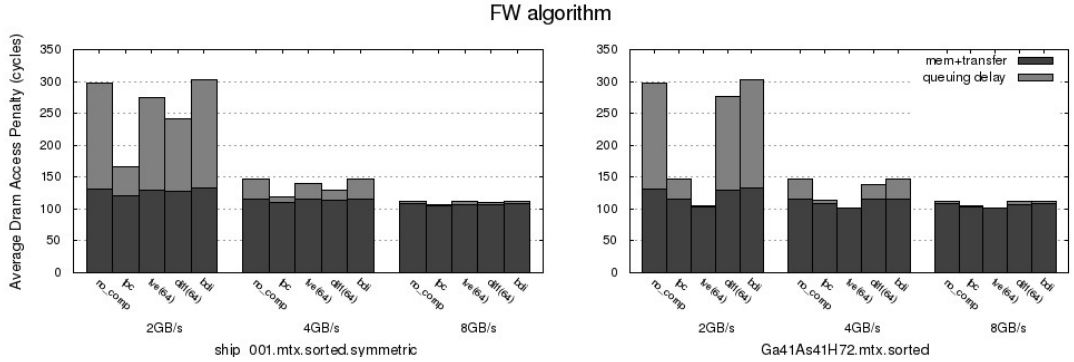
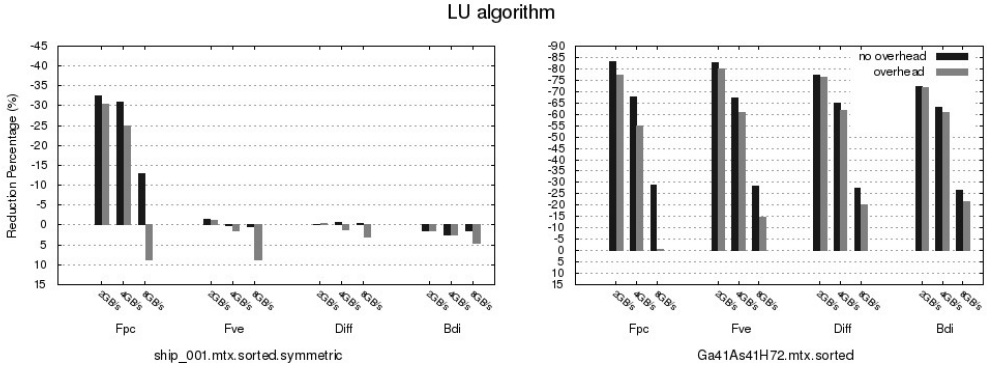


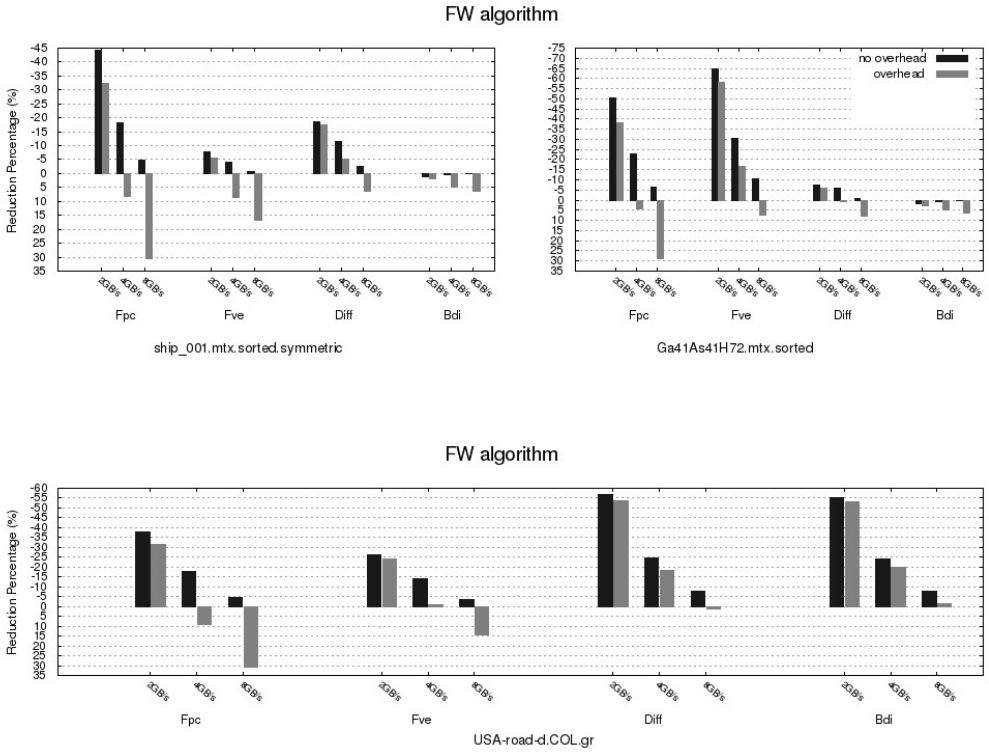
Figure 27: Average Dram Access Latency components

Figure 28's black histogram bars show the average reduction (%) of the Dram Access latency achieved by ideal compressors for some of the applications and the corresponding inputs. We notice that successful compression can ideally reduce to a significant extend the access latency if the algorithm is bandwidth bound (for 2GB/s and less for 4GB/s bandwidth). $\sim$30% $\rightsquigarrow$ $\sim$45% reduction is noticed for the floating point datasets while for the integer (easier to compress datasets) over 50%. We also notice again that as bandwidth grows the impact of the scheme is dramatically reduced.

4.3.2.1 (De)compression overhead.

In reality though, the (de)compressor units introduce latencies that affect and may significantly reduce the scheme's performance. In our link compression scheme (where main memory is assumed to be unable of storing compressed data) both compression and decompression delays are added to the main memory access latency. Figure 28 depicts the reduction of the memory access latency achieved also when the compressor's overhead is taken into consideration (gray bars).





(a) Integer

Figure 28: Reduction of the Average Dram Access Latency

Surprisingly we observe that the impact of the (de)compression overhead isn't always the same and changes significantly for different bus bandwidths. Particularly, as the bandwidth increases the impact increases too. To examine the scheme's sensitivity to overhead under different "traffic" conditions we've collected some statistics for the FPC algorithm using various overheads. Figure 29 shows the results regarding the average Dram Access latency.

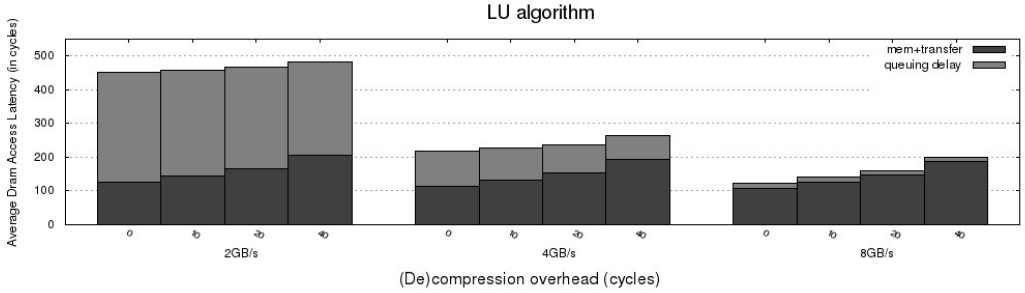


Figure 29: Sensitivity to Overhead (1)

We notice that increasing the overhead leads to a reduction of the queuing delays. If an application is completely memory bandwidth bound the rate at which it will be able to complete requests is fixed (the bandwidth bottleneck dominates). The addition of an extra waiting component for (de)compression just leads the requests to spend less time in the queue but the total number of requests that can be completed per second (and hence access latency) remains approximately the same (as we notice for bb=2GB/s - LU algorithm). Therefore, (de)compression overhead becomes generally "visible" when it exceeds the queuing delays. Hence, as the bandwidth increases from 2GB/s to 8GB/s and the queuing delays decrease, the sensitivity to the overhead becomes greater (the bandwidth is no longer the bottleneck). This is better shown in Figure 30.

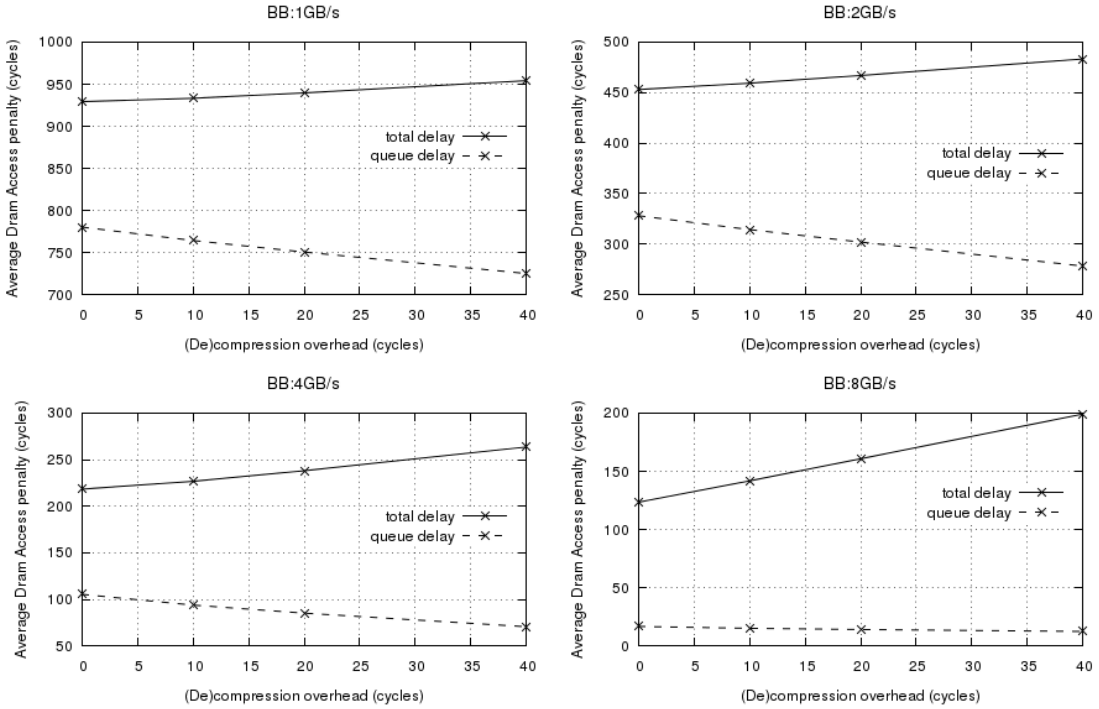


Figure 30: Sensitivity to Overhead (2)

To explain further this observation we will use an example. If there are multiple concurrent DRAM request streams, the DRAM bus will only be able to satisfy a request from one of these streams every X ns. Using a 64-byte cache line width and 8 GB/s bandwidth, a bus transfer takes 8 ns. With 8 parallel streams this means one request can be completed (per stream) only once every 64 ns. If the application makes the requests back-to-back, and with a DRAM base latency of 45 ns, if a request completes at time 0, a new request will be launched immediately incurring 45 ns of base latency, and then spending another 64-45 = 19 ns in the queue

before it can be sent over the DRAM bus. If the (de)compression latency of say 10 ns is added, the request will spend 55 ns doing other things (base + (de)comp. latency) and will have to wait an additional 9 ns in the queue waiting for its slot on the bus to come up. Total DRAM latency will remain constant at 64 ns. But if the extra latency is 20 ns the requests will be spaced 65 ns apart making DRAM bandwidth no longer the bottleneck, avoiding all queuing delay. In this case the extra (de)compression latency will increase the total DRAM latency (now to 65 ns).

A real application is DRAM bandwidth bound in some phases but not in others, so the observed average latency is a mix of both cases above, leading to an average DRAM latency that goes up by some fraction of the (de)compression latency.

From Figure 28 we also notice that even for the same bandwidth the overhead's impact may differ for the various applications. This is due to the same reason described above and can be justified by the facts that: i) Very successful compression can reduce queuing delays to a great extend thus make (de)compression overheads more "visible" (basically referring to small bandwidths) ii) The applications are not equally "bound" to bandwidth. Particularly, SpMV has the highest queuing delays that remain extremely high for BB=4GB/s too, while the FW algorithm has the smaller ones compared to the other applications.

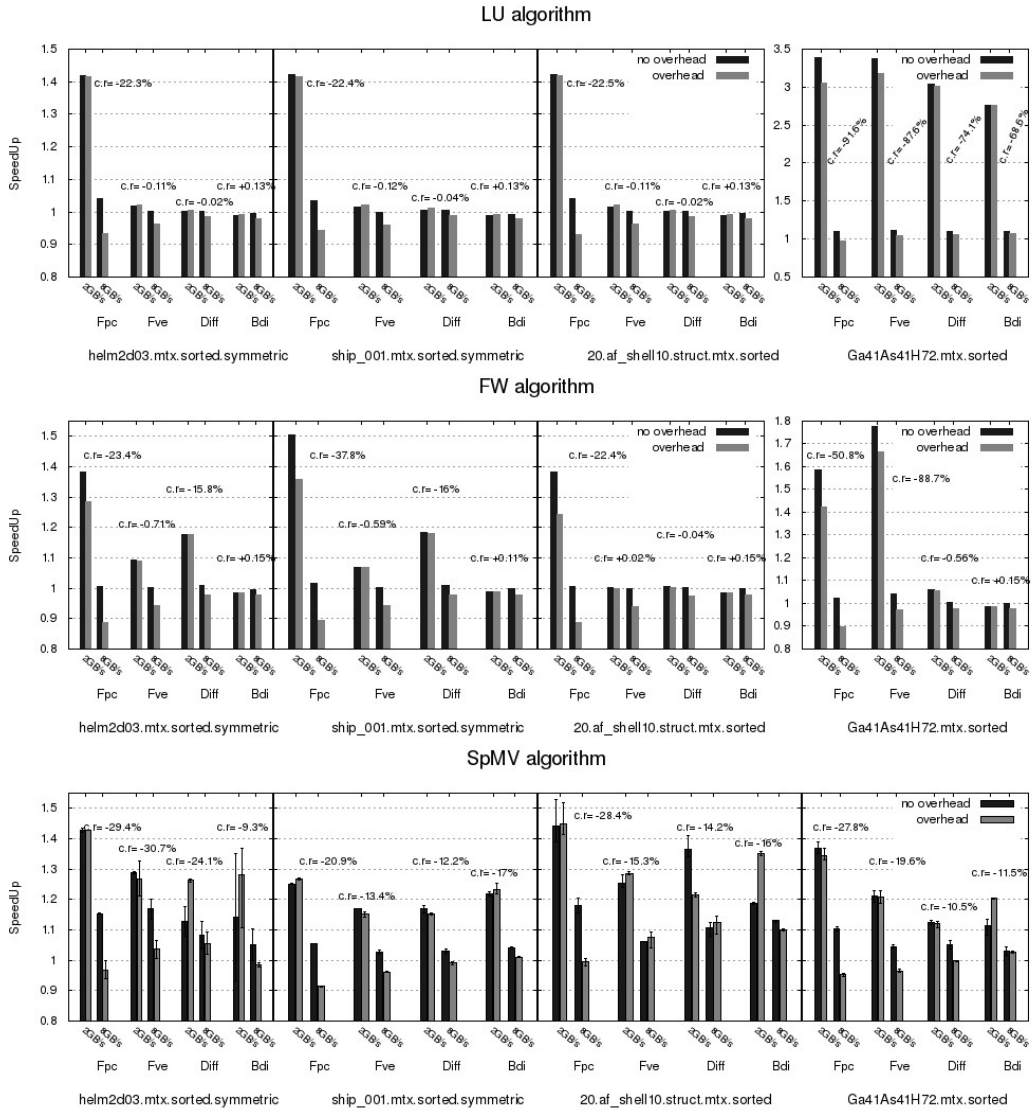
4.3.3 Effect on Performance

In this section we evaluate the performance impact of the proposed compression scheme using the metric of SpeedUp. Figure 31 depicts the SpeedUp observed for all applications and input datasets. For every case the corresponding off-chip memory traffic reduction (compression ratio -cr-) is also indicated in terms of percentage. For better sharpness of Figures we decided not to present the results for 4GB/s bus bandwidth.

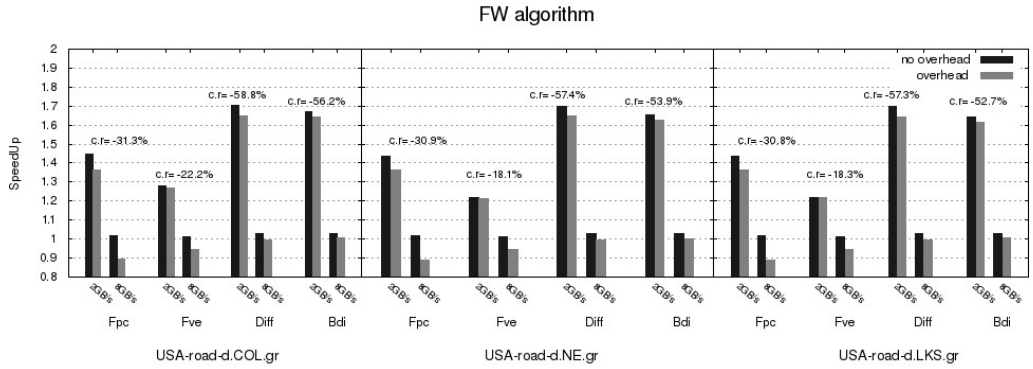
Figure 31 indicates that successful link compression can actually lead to a performance boost when the application is memory bandwidth bound. For 2GB/s bandwidth, the compression schemes that manage to successfully compress the transferred data (4.3.1) achieve also to significantly reduce the execution time of the applications. For 8GB/s, though, when memory bandwidth isn't a bottleneck, the scheme's effect on performance is negligible and most of the times even negative due to time (de)compression overheads.

Excluding the SpMV results that exhibit some peculiarities and will be discussed below, the rest of the graphs reveal a steady behavior where SpeedUps "follow" the corresponding compression ratios (higher cr $\rightarrow$ higher SpeedUp). Therefore, regarding floating point datasets and excluding the special Ga41As41H72.mtx matrix, the higher SpeedUps are achieved by FPC. They range from $\sim$1.29(22.5%) to 1.41(29%). For the special Ga41As41H72.mtx matrix speedups overcome 68%. Regarding the integer datasets, the bitwise compressors (Diff and BDI) achieve SpeedUps $\sim$1.64(39%).

Figure 31 depicts also the impact of the (de)compression overhead on SpeedUp. For the same reasons discussed in the previous section 4.3.2.1, the impact of the overhead is greater for 8GB/s bandwidth. The extra (de)compression latencies usually lead even to performance degradation (SpeedUp < 1), especially for FPC whose overhead is high (20 + 20 cycles). The impact of the overhead is noticed also to be greater for the FW algorithm compared to LU because it originally has smaller queuing delays. Therefore at 2GB/s bandwidth, FPC's overhead is nearly invisible for the LU algorithm while it causes at least ∼10% reduction of the scheme's performance for FW.



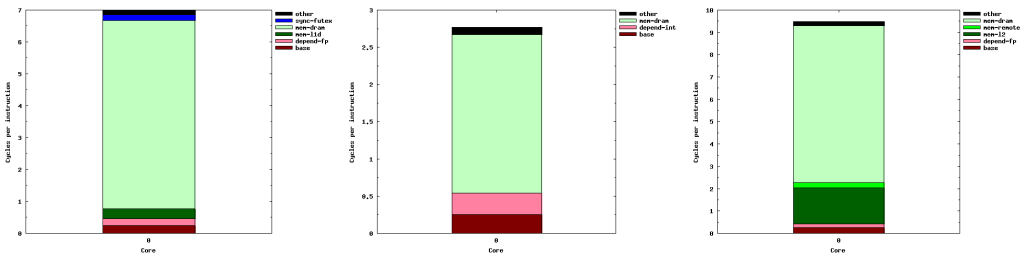
(a) Floating point



(b) Integer

Figure 31: Average SpeedUp = (Original Execution Time)/(Execution Time with compression enabled)

At this point we must notice that the effect of link compression on an application's performance (the expected SpeedUp) depends also on the portion of the total execution time that is spent on memory references. Figure 32 depicts the simple aggregated CPI stacks of the applications for 2GB/s bus bandwidth. All three applications are DRAM "bound" but not exactly equally. For example, the DRAM CPI component constitutes a greater percentage of LU's CPI stack compared to FW's. This contributes on seeing slightly higher speedups for similar compression ratios in LU (e.g FPC - Helm input matrix).



(a) LU algorithm *(b) FW algorithm* *(c) SpMV algorithm*

Figure 32: CPI stacks - Bus Bandwidth 2GB/s

SpMV results. The statistics collected from SpMV runs have various peculiarities and contradictions. Two visible in Figure 31 are i) the higher speedups achieved by smaller compression ratios (occasionally observed - e.g FPC vs FVE for helm2d03.mtx input matrix) ii) the higher speedups occasionally achieved when (de)compression overhead is taken into account (usually when bb=2GB/s e.g BDI helm2d03.mtx)

At this point we must resume that identical runs of SpMV give results with a significant variation, as was mentioned in the beginning of 4.3. We present the average of these results excluding some outliers. This contributes to the emergence of the previous contradictions but doesn't completely explain them. Another important parameter that affects results, and especially SpeedUp, is the number of the performed Dram Accesses in every examined case. To be more specific, if the number of Dram Accesses in a compressor case differs a lot from the corresponding number in the original case (without compression) then this will affect decisively the visible SpeedUp. While in FW and LU algorithms this number is approximately the same for all cases, in SpMV results a significant variation is occasionally observed. Figure 33 depicts this variation of Dram Accesses compared to the original case in terms of percentage. We notice, for example, that for the helm2dO3 input matrix and the FVE case, ∼10% more dram accesses are on average counted compared to the original case and even more compared to the FPC case. This contributes on finally seeing a higher speedup by FPC while FVE compresses better. The same reason contributes on seeing occasionally higher ratios when (de)compression overhead is added to the system. This observed variation in Dram accesses is due to:

- runtime scheduling (different work sharing between threads from run to run → different memory access pattern)
- simulation inaccuracy/error

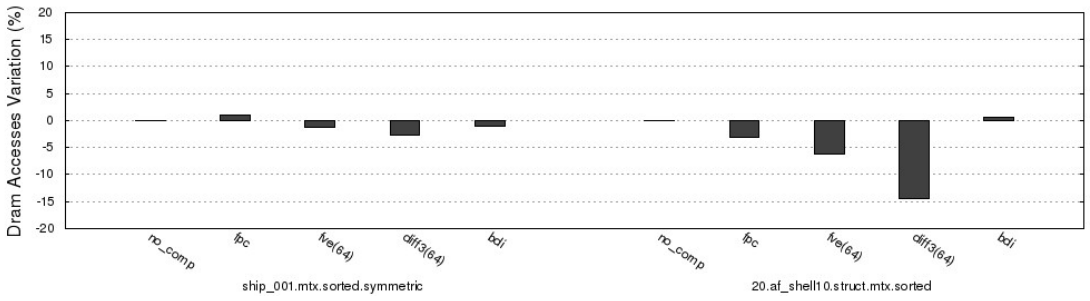


Figure 33: The variation of the number of the Dram Accesses of each compression case compared to the original - SpMV

- possibly the fact that compression also changes the memory access pattern

Finally, we must also notice that the speedup measured in our experiment is sometimes higher than the one expected intuitively. There are multiple reasons that could attribute to this:

- dependencies between threads → speeding up one thread has an effect twice because it also reduces the waiting time on other threads
- instantaneous compression versus average
- higher compressibility of the lines on the critical path (loads more critical than stores)
- simulation inaccuracy

Chapter 5

Conclusions and Future Work

In this Diploma thesis, we studied hardware link compression as a technique to deal with the *Bandwidth Wall* in multicore architectures. Motivated by the unsustainable bandwidth in modern CMP systems and the performance limitations it imposes, we examined and evaluated the effect of transferring data in a compressed form over the link. In our limited research work we experimented with memory bandwidth bound scientific applications and both floating point and integer datasets.

We modeled and evaluated a simple link compression scheme assuming that both caches and main memory store data only in their original form (uncompressed). Therefore, in our scheme data are compressed before their transfer over the link and always decompressed afterwards. For our study we used and evaluated various compression techniques that have been previously proposed for hardware compression. Motivated by the fact that most schemes fail to compress floating point data we've also applied Martin Burtscher's FPC [57] compression algorithm for double-precision floating point data. Our experimental evaluation of the scheme (simulation results for an 8-core CMP) show that:

- FPC manages to successfully reduce the amount of data transferred over the link when applications with hard-to-compress scientific datasets (floating point) are running on the system. It achieves a $\sim$ 22-30% average reduction of the off-chip memory traffic while most of the other compression algorithms fully fail. When integer datasets are used, a different family of compression algorithms (bitwise compressors) "wins", achieving a $\sim$ 50-58% compression.
- The main effect of this compression scheme is on the queuing delays caused by the cores competition for the bandwidth shared resource. Therefore, the scheme has a positive impact when the running application is memory bandwidth bound and memory requests suffer high queuing delays. When no contention over the bus exists and the bandwidth is not a bottleneck, the scheme can't offer

any significant improvement and may even cause a negative impact due to its overheads - the (de)compression delays it introduces. In such conditions, FPC can cause significant performance degradation, due to its high overheads (pessimistic baseline of 20 cycles).

- If Dram accesses dominate the execution time of an application and the application is "bandwidth" bound then successful compression can lead to a significant performance boost, speedup. Particularly in our experiment up to 28% speedup was observed for floating point datasets and up to 39% for integer.
- The sensitivity to the (de)compression overhead is not always constant. In particular, the impact of the compression/decompression latency on the speedup is smaller or even negligible when high queuing delays exist (bandwidth bottleneck). This is due to the fact that (de)compression and queuing delays may partially overlap.

Based on these results we believe that link compression is a technique capable to offer significant performance improvement under certain conditions and worths to be studied and evolved. Particularly, as a continuation of the present work we would find it interesting and challenging to:

- Study and evaluate in detail (time, area and power overheads) a potential hardware implementation of FPC.
- Study a case that compression mechanism is "triggered" only when it is necessary, when the available bandwidth is constrained to such an extend that compression can offer performance improvement. An adaptive scheme to the various "traffic" conditions on the bus could avoid compression's negative impact when bandwidth is not the bottleneck. This mainly concerns compression schemes that aren't light-weighted and have overheads that can cause performance degradation.
- Study the interaction of the link compression scheme with hardware prefetching.
- Study the effect that the scheme could have on power consumption. Nowadays, power efficiency is a first-order concern for modern systems and a significant portion of power is spent in the memory hierarchy. Link compression has been proposed in the past as a technique that can significantly reduce power consumption by reducing the switching activity on the bus.
- Study the combination of link with main memory compression. In this case only compression or decompression latency would be in the critical path and not both of them, as it happens in the present scheme. Studying such a scheme would give also the chance to explore the possible advantages of a compressed main memory not only regarding performance (usage of the freed-up space) but power consumption too.

Bibliography

[1] G.Moore. "Gramming more components onto integrated circuits". *Electronics Magazine*, April 1965.

[2] Wum. A. Wulf and Sally A. McKee. "Hitting the Memory Wall: Implications of the Obvious". *ACM Computer Architecture News*, March 1995.

[3] Richard Sites. "It's the Memory, Stupid!". *Microprocessor Report*, March 1996.

[4] Russell Fish. "The future of computers-Part 1: Multicore and the Memory Wall". http://www.edn.com/design/systems-design/4368705/The-future-of-computers–Part-1-Multicore-and-the-Memory-Wall, November 2011.

[5] "More chip cores can mean slower supercomputing, Sandia simulation shows". https://share.sandia.gov, January 2009.

[6] "International Technology Roadmap for Semiconductors: ITRS update". http://www.itrs.net/Common/2004Update/2004Update.html, 2004.

[7] Doug Burger, James R. Goodman, and Alain Kagi. "Memory Bandwidth Limitations of Future Microprocessors". *ISCA '96 Proceedings of the 23rd annual international symposium on Computer architecture*, May 1996.

[8] "The Berkeley Intelligent RAM (IRAM) Project". http://iram.cs.berkeley.edu.

[9] Ashley Saulsbury, Fong Pong, and Andreas Nowatzyk. "Missing the Memory Wall:The Case for Processor/Memory Integration". *ISCA '96 Proceedings of the 23rd annual international symposium on Computer architecture*, May 1996.

[10] Doug Burger. System-level implications of processor-memory integration, 1997.

[11] Irina Chihaia Tuduce and Thomas Gross. Adaptive main memory compression. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 29–29, Berkeley, CA, USA, 2005. USENIX Association.

[12] Vicenç Beltran, Jordi Torres, and Eduard Ayguadé. Improving disk bandwidth-bound applications through main memory compression. In *Proceedings of the 2007 Workshop on MEmory Performance: DEaling with Applications, Systems and Architecture*, MEDEA '07, pages 57–63, New York, NY, USA, 2007. ACM.

[13] Lei Yang, Robert P. Dick, Haris Lekatsas, and Srimat Chakradhar. Online memory compression for embedded systems. *ACM Trans. Embed. Comput. Syst.*, 9(3):27:1–27:30, March 2010.

[14] R. B. Tremaine, P. A. Franaszek, J. T. Robinson, C. O. Schulz, T. B. Smith, M. E. Wazlowski, and P. M. Bland. Ibm memory expansion technology (mxt). *IBM J. of Research and Development*, 45:271–285, 2001.

[15] Bulent Abali, Hubertus Franke, Xiaowei Shen, Dan E. Poff, and T. Basil Smith. Performance of hardware compressed main memory. In *The Seventh International Symposium on High-Performance Computer Architecture*, pages 73–81, 2000.

[16] P. Franaszek, J. Robinson, and J. Thomas. Parallel compression with cooperative dictionary construction. In *Proceedings of the Conference on Data Compression*, DCC '96, pages 200–, Washington, DC, USA, 1996. IEEE Computer Society.

[17] Magnus Ekman and Per Stenstrom. A robust main-memory compression scheme. In *In Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 74–85, 2005.

[18] Jun Yang and Rajiv Gupta. Frequent value locality and its applications. *ACM Trans. Embed. Comput. Syst.*, 1(1):79–105, November 2002.

[19] Alaa R. Alameldeen and David A. Wood. Frequent pattern compression: A significance-based compression scheme for l2 caches. Technical report, 2004.

[20] N.R. Mahapatra, Jiangjiang Liu, K. Sundaresan, S. Dangeti, and B.V. Venkatrao. The potential of compression to improve memory system performance, power consumption, and cost. In *Performance, Computing, and Communications Conference, 2003. Conference Proceedings of the 2003 IEEE International*, pages 343–350, April 2003.

[21] Charles Lefurgy, Eva Piccininni, and Trevor Mudge. Evaluation of a high performance code compression method. In *Proceedings of the 32Nd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 32, pages 93–102, Washington, DC, USA, 1999. IEEE Computer Society.

[22] Haris Lekatsas, Jörg Henkel, and Wayne Wolf. Code compression for low power embedded system design. In *Proceedings of the 37th Annual Design Automation Conference*, DAC '00, pages 294–299, New York, NY, USA, 2000. ACM.

[23] I-Cheng K. Chen. The impact of instruction compression on i-cache performance, 1997.

[24] Erik G. Hallnor and Steven K. Reinhardt. A compressed memory hierarchy using an indirect index cache. In *Proceedings of the 3rd Workshop on Memory Performance Issues: In Conjunction with the 31st International Symposium on Computer Architecture*, WMPI '04, pages 9–15, New York, NY, USA, 2004. ACM.

[25] Erik G. Hallnor and Steven K. Reinhardt. A fully associative softwaremanaged cache design. In *In Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.

[26] 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*, MICRO-46, pages 62–73, New York, NY, USA, 2013. ACM.

[27] Youtao Zhang, Jun Yang, and Rajiv Gupta. Frequent value locality and value-centric data cache design. *SIGPLAN Not.*, 35(11):150–159, November 2000.

[28] Jun Yang and Rajiv Gupta. Frequent value locality and its applications. *ACM Trans. Embed. Comput. Syst.*, 1(1):79–105, November 2002.

[29] Jun Yang, Youtao Zhang, and Rajiv Gupta. Frequent value compression in data caches. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 33, pages 258–265, New York, NY, USA, 2000. ACM.

[30] Georgios Keramidas, Konstantinos Aisopos, and Stefanos Kaxiras. Dynamic dictionary-based data compression for level-1 caches. In *Proceedings of the*

19th International Conference on Architecture of Computing Systems, ARCS'06, pages 114–129, Berlin, Heidelberg, 2006. Springer-Verlag.

[31] Julien Dusser, Thomas Piquet, and André Seznec. Zero-content augmented caches. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 46–55, New York, NY, USA, 2009. ACM.

[32] Trevor Mudge Nam Sung Kim, Todd Austin. Low-energy data cache using sign compression and cache line bisection. *2nd Annual Workshop on Memory Performance Issues*, May 2002.

[33] Prateek Pujara and Aneesh Aggarwal. Restrictive compression techniques to increase level 1 cache capacity. In *ICCD*, pages 327–333, 2005.

[34] Alaa R. Alameldeen and David A. Wood. Frequent pattern compression: A significance-based compression scheme for l2 caches. Technical report, 2004.

[35] Alaa R. Alameldeen and David A. Wood. Adaptive cache compression for high-performance processors. *SIGARCH Comput. Archit. News*, 32(2):212–, March 2004.

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

[37] 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*, PACT '12, pages 377–388, New York, NY, USA, 2012. ACM.

[38] Youtao Zhang and R. Gupta. Enabling partial cache line prefetching through data compression. In *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, pages 277–285, Oct 2003.

[39] A.R. Alameldeen and D.A. Wood. Interactions between compression and prefetching in chip multiprocessors. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 228–239, Feb 2007.

[40] Michel Dubois, Murali Annavaram, and Per Stenstrm. *Parallel Computer Organization and Design*. Cambridge University Press, New York, NY, USA, 2012.

[41] Alaa R. Alameldeen. Using compression to improve chip multiprocessor performance. Technical report, 2006.

[42] Enric Musoll, Tomás Lang, and Jordi Cortadella. Exploiting the locality of memory references to reduce the address bus energy. In *Proceedings of the 1997 International Symposium on Low Power Electronics and Design*, ISLPED '97, pages 202–207, New York, NY, USA, 1997. ACM.

[43] L. Benini, G. De Micheli, E. Macii, D. Sciuto, and C. Silvano. Address bus encoding techniques for system-level power optimization. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '98, pages 861–867, Washington, DC, USA, 1998. IEEE Computer Society.

[44] Y. Aghaghiri, F. Fallah, and M. Pedram. Irredundant address bus encoding for low power. In *Low Power Electronics and Design, International Symposium on, 2001.*, pages 182–187, 2001.

[45] L. Benini, D. Bruni, A. Macii, and E. Macii. Hardware-assisted data compression for energy minimization in systems with embedded processors. In *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, pages 449–453, 2002.

[46] M. Thuresson, L. Spracklen, and P. Stenstrom. Memory-link compression schemes: A value locality perspective. *Computers, IEEE Transactions on*, 57(7):916–927, July 2008.

[47] D. W. Hammerstrom and E. S. Davidson. Information content of cpu memory referencing behavior. In *Proceedings of the 4th Annual Symposium on Computer Architecture*, ISCA '77, pages 184–192, New York, NY, USA, 1977. ACM.

[48] D. Citron and L. Rudolph. Creating a wider bus using caching techniques. In *High-Performance Computer Architecture, 1995. Proceedings., First IEEE Symposium on*, pages 90–99, 1995.

[49] Jun Yang and Chuanjun Zhang. Frequent value encoding for low power data buses. *ACM Transactions on Design Automation of Electronic Systems*, 9:354–384, 2004.

[50] M. Thuresson and P. Stenstrom. Accommodation of the bandwidth of large cache blocks using cache/memory link compression. In *Parallel Processing, 2008. ICPP '08. 37th International Conference on*, pages 478–486, Sept 2008.

[51] Ping Zhou, Bo Zhao, Yu Du, Yi Xu, Youtao Zhang, Jun Yang, and Li Zhao. Frequent value compression in packet-based noc architectures. In *Design Automation Conference, 2009. ASP-DAC 2009. Asia and South Pacific*, pages 13–18, Jan 2009.

[52] 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. *SIGARCH Comput. Archit. News*, 37(3):371–382, June 2009.

[53] Lei Fan and Martyn Romanko. Implementation and energy analysis of base-delta-immediate compression.

[54] G. Dimitrakopoulos, K. Galanopoulos, Christos Mavrokefalidis, and D. Nikolos. Low-power leading-zero counting and anticipation logic for high-speed floating point units. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(7):837–850, July 2008.

[55] V.G. Oklobdzija. An algorithmic and novel design of a leading zero detector circuit: comparison with logic synthesis. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2(1):124–128, March 1994.

[56] P.I.-J. Chuang, M. Sachdev, and V.C. Gaudet. A 167-ps 2.34-mw single-cycle 64-bit binary tree comparator with constant-delay logic in 65-nm cmos. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 61(1):160–171, Jan 2014.

[57] M. Burtscher and P. Ratanaworabhan. Fpc: A high-speed compressor for double-precision floating-point data. *Computers, IEEE Transactions on*, 58(1):18–31, Jan 2009.

[58] Martin Isenburg, Peter Lindstrom, and Jack Snoeyink. Lossless compression of predicted floating-point geometry. *Comput. Aided Des.*, 37(8):869–877, July 2005.

[59] Bryan E. Usevitch. Jpeg2000 compatible lossless coding of floating-point data. *J. Image Video Process.*, 2007(1):22–22, January 2007.

[60] Yiannakis Sazeides and James E. Smith. The predictability of data values. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 30, pages 248–258, Washington, DC, USA, 1997. IEEE Computer Society.

[61] Bart Goeman, Hans V, and Koen De Bosschere. Differential fcm: Increasing value prediction accuracy by improving table usage efficiency. In *Seventh*

International Symposium on High Performance Computer Architecture, pages 207–216, 2001.

[62] A.T. Do, C. Yin, K. Velayudhan, Z.C. Lee, K.S. Yeo, and T.T.-H. Kim. 0.77 fj/bit/search content addressable memory using small match line swing and automated background checking scheme for variation tolerance. volume PP, pages 1–12, 2014.

[63] T.E. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–12, Nov 2011.

[64] J.E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12, Jan 2010.

[65] D. Genbrugge, S. Eyerman, and L. Eeckhout. Interval simulation: Raising the level of abstraction in architectural simulation. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12, Jan 2010.

[66] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[67] Gengbin Zheng, Gunavardhan Kakulapati, and L.V. Kale. Bigsim: a parallel simulator for performance prediction of extremely large parallel machines. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 78–, April 2004.