Memory link compression to speedup scientific workloads

Chloe Alverti[1], Georgios Goumas[2], Konstantinos Nikas[2], Angelos Arelakis[1], Nectarios Koziris[2], and Per Stenström[1]

[1] Chalmers University of Technology, Sweden
[2] National Technical University of Athens, Greece

Abstract. Limited off-chip memory bandwidth poses a significant challenge for today's multicore systems. Link compression provides an effective solution, however its efficacy varies due to the diversity of transferred data. This work focuses on scientific applications that typically transfer huge amounts of floating-point data, which are notoriously difficult to compress. We explore the potential of BFPC, a recently prosposed software compression algorithm, but by modeling it for hardware compression and comparing it to three state-of-the-art schemes. We find that it can reduce floating-point data traffic by up to 40%, which for memory bound executions translates to significant performance improvement.

1 Introduction

Chip Multi-Processor systems (CMPs) have become the dominant design paradigm, utilizing the still increasing number of available transistors on-chip. However, delivering the peak performance of multiple cores to the application level is by no means a straightforward task. One of the main problems is the unsustainable off-chip memory bandwidth. Memory technology is not able to keep pace with the advancements in CPU technology and the increasing demand for memory access posed by multiple cores. Thus, memory bandwidth has emerged as a critical performance bottleneck.

This paper focuses on *hardware link compression*, an effective approach to save off-chip memory bandwidth by transferring data over the memory link in a compressed form. Research on the field has been rich and illuminating, highlighting important issues that need to be considered, such as the complexity in terms of compression/decompression latencies and hardware resources or their effectiveness (i.e. compression ratio) on diverse data types and application domains. Several novel lightweight compression algorithms that target a balance between compression efficiency and speed [1, 2, 3, 4] have been proposed. It has been shown that transferring compressed data over the link between on-chip cores and their last-level caches, and off-chip main memory (see Figure 1) has high potential especially for integer, multimedia and commercial workloads [1].

Performance improvement in state-of-the-art link compression schemes is sought mainly on single processor architectures. Alameldeen [4] has evaluated a

link compression scheme for CMP systems by using the *significance-based* Frequent Pattern Compression (FPC) algorithm as a complementary technique to the adaptive cache compression scheme [5]. Interestingly, most of the previous schemes focus on integer and commercial workloads, while they appear to have poor performance on scientific workloads and floating-point (FP) datasets.

The goal of this study is to evaluate the impact of link compression on the performance of scientific memory-bound applications running on modern CMP systems. We consider a simple compression scheme (similar to [1]), assuming uncompressed cache and main memory. Cache blocks are compressed before they are transferred over the link and decompressed after their reception, applying on the fly (de)compression. We initially evaluate three state-of-the-art algorithms proposed for cache [6] and link compression [2, 3]. Motivated by the poor performance of these schemes in floating-point datasets, we explore Martin Burtscher's FPC [7] algorithm (BFPC), a software compression scheme for double precision floating-point data which we adjust for hardware compression. To the best of our knowledge, BFPC has not been applied in a hardware compression scheme before.

Our experimental evaluation leads to several interesting observations regarding the compressibility of the workloads and the sensitivity of each scheme to the (de)compression overheads. The main contributions of this paper are:

- A detailed study of link compression as a technique to increase the effective bandwidth and improve the performance of modern CMP systems, particularly for multi-threaded scientific workloads. Our results indicate that effective compression can lead to significant speedup, especially when memory bandwidth bound applications are considered.
- An initial evaluation of the BFPC algorithm adapted to our hardware link compression scheme. Our results indicate that BFPC reduces off-chip memory traffic up to 40% on hard-to-compress scientific floating-point datasets.

The rest of the paper is organized as follows: Section 2 provides information on our motivation and background on the link compression scheme and state-of-the-art approaches. Section 3 presents our link compression approach, Section 4 presents our experimental evaluation and Section 5 concludes the paper.

2 Motivation and background

2.1 Motivation

Despite their characterization as "compute-intensive" or "number-crunching", large classes of scientific workloads are data-intensive and operate on large data sets, performing rather lightweight computations with limited data reuse. The performance of such applications is heavily dependent on the memory-link speed, i.e. on the capability of the memory subsystem to feed the fast processing cores with data. In that respect, memory-bound scientific workloads are ideal candidates for applying data compression. However, these workloads typically operate on floating-point data, which are very difficult to compress due to their binary representation in the IEEE 754 arithmetic format, which generates high entropy

to data representation. In this work we face the challenge of applying memory-link compression schemes on floating-point data in order to shed light on the opportunities of performance improvement inherent in this paradigm.

2.2 Compression at the memory link

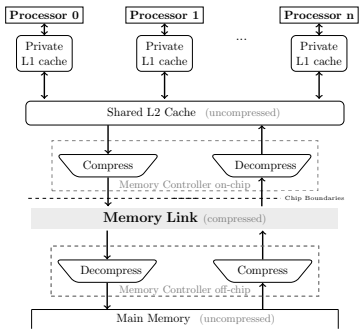


Fig. 1: Memory-Link compression

We examine memory-link compression as a technique to deal with the *bandwidth wall* in modern CMP systems. We model a simple scheme, as the one proposed in [1]. The main objective is to reduce the off-chip memory traffic and to improve performance by transferring compressed data over the link. Each cache line is compressed before it is being transferred over the link and decompressed right afterwards (as shown in Fig. 1). Compression and decompression units are required on both sides of the link.

2.3 State-of-the-art compression methods

A major challenge in memory-link compression is that the block of data to be compressed is very small (typically a cache block) and therefore difficult to contain compressible redundancies. This challenge becomes even harder for floating point datasets that have very small inherent redundancy. This section presents three previously proposed hardware compression algorithms. The first two (*bitwise* compressors) aim at eliminating redundancy by exploiting *nearby value locality* [8] at the cache-line granularity [2, 6], while the third extracts frequent value locality using a small memory table [3] monitoring the whole off-chip traffic.

BDI: The Base-Delta-Immediate compression algorithm (BDI) is a *delta encoding* [1] scheme previously applied for cache [6] and memory link [9] compression. BDI is motivated by the fact that the words residing in the same cache line most likely belong to the same dynamic range. Thus, their relative difference is small and a cache block can be encoded as a single base value with an array of delta (Δ) values. BDI's latency overheads are estimated to be: a) 1 cycle for decompression and b) 6 cycles for compression [6]. As BDI exploits mainly the potentially small variance in the representation of integer values stored in the same cache line, it is not expected to compress FP data effectively.

Diff3: Diff3 is a variation of the Differential algorithm [2], proposed for link compression and energy minimization in embedded systems. It is a bitwise compressor that works based on the observation that several cache line words have in common some of the most significant bits (MSBs). Thus, it eliminates the MSBs of each word that are common with the first block's word. The algorithm is designed for integer data, however it might compress FP data too, as the sign and

exponent fields (MSBs in IEEE-754 standard) of a cache block's FP values tend to be more regular. Diff3 has low latency overheads, as it uses basic binary operations and executes compression and decompression in parallel. Here we assume a pessimistic overhead of 5 cycles for both compression and decompression.

FVE: The Frequent Value Encoder has been applied in different scenarios including cache [10] and link compression [3] and NoC architectures [11]. It is based on the observation that a small set of frequent values tends to occupy a large portion of the off-chip traffic [12] for many applications. Thus, it stores this set in a small table, constructing the *frequent value (FV)* dictionary. Then compression can occur by encoding these values using their corresponding indexes. In link compression, dynamically updated (de)compression FV dictionaries are kept in both sides of the link. The algorithm's latency overheads depend on the size of the dictionary [3]. We assume a 512B dictionary and a pessimistic (de)compression overhead of 10 cycles (as in [13]).

3 A compression engine for floating-point data

As discussed in Section 2, state-of-the-art hardware compression schemes fail to compress FP datasets effectively. Therefore, we resort to a software compression algorithm designed explicitly for FP data. Specifically, we consider BFPC, a lossless, linear-time floating-point compression algorithm proposed by Burtscher et al. [7]. We chose BFPC for the following reasons: a) BFPC compresses scientific datasets exhibiting a compression performance competitive to that of known, complex FP compression algorithms (bzip2, gzip, e.a), while remaining simpler and faster (8 to 300 times) [7]. b) Its functionality and performance do not depend on prior knowledge of the specific data domain, contrary to other FP compression algorithms like geometric data [14], etc. c) It is a single-pass algorithm and thus (de)compression can occur on the fly. d) It interprets all doubles as 64-bit integers and uses mainly bitwise operations, simplifying a hardware implementation of the algorithm. BFPC has so far only been considered as a software algorithm. Next, we sketch the block diagram of BFPC.

3.1 BFPC compression logic

Compression: BFPC compresses streams of IEEE 754 double-precision FP data by comparing each value with a predicted one. Two different predictors are used for better performance, implemented as hash tables that store predicted values: *fcm*[15] and *dfcm* [16]. To compress a value, a bitwise XOR operation is applied between the most accurate prediction and the original value. Accurate predictions lead to small value XOR results and, hence, compression is achieved by truncating their leading zero bytes.
Prediction: The prediction logic is based on the observation that specific sequences of FP values appear multiple times in a dataset. Consequently, keeping a dynamic record of value sequences can result in more accurate value prediction. The *fcm* hash key, which is the "pointer" to the *fcm* prediction table, represents

the sequence of the most recently encountered FP values. The corresponding hash table cell stores the value that followed this sequence the previous time it was encountered [15]. This stored value forms the prediction. The *dfcm* predictor has the same logic but works with *deltas* instead of absolute values.
Decompression: Decompression is the exact reverse procedure of compression, and the value's lossless reconstruction is due to XOR's property of being completely reversible. Decompression algorithm reconstructs from scratch the prediction tables and no metadata is needed for consistency.

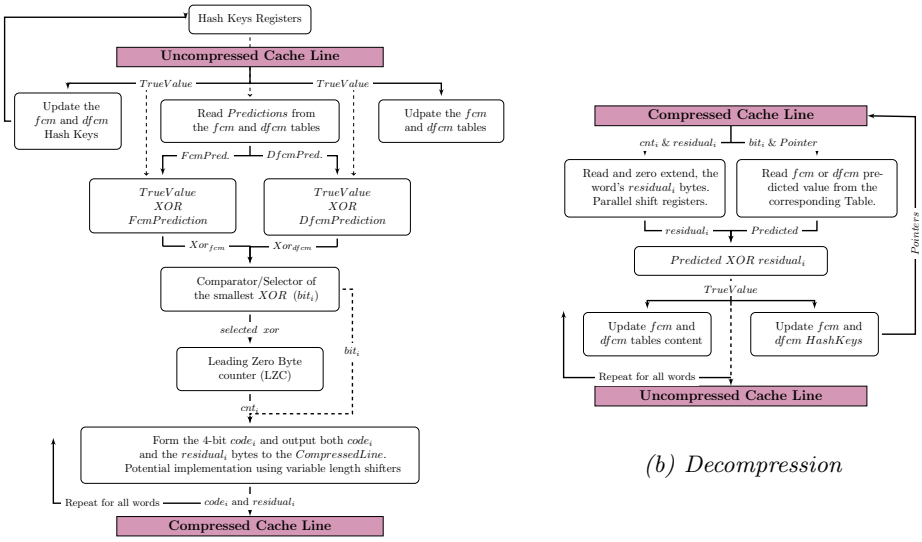
3.2 BFPC in hardware link compression

BFPC can be used with small modifications in a memory-link compression scheme. A small adjustment is required for prediction. BFPC is designed to process big streams of double-precision floating-point data and it constructs prediction tables per stream. The algorithm's efficiency is based on the gradual adaptation of the tables' content to the input value-stream through constant update, taking advantage of the stream size. In link compression, as the block (stream) of to-be-compressed data is typically small (a single cache block), the construction (reinitialization) of the tables for every cache block would be inefficient. Hence, we model these tables as small caches, built in both sides of the link (as part of the (de)compression circuitry). Every word transferred over the link dynamically updates the tables and no extra signals are required to maintain the compression and decompression tables consistent (compression and decompression occur on the fly, as in FVE [3]). Consistency prerequisites are: a) the common initialization of the tables during the system start up and b) the "one-to-one" relationship between the compression and decompression circuitry (the data must be decompressed in the same order as they are compressed).

3.2.1 Encoding a word in steps Fig. 2a depicts the required steps of encoding a cache block's word (W_i). Compression and decompression are inherently sequential due to the use of the prediction tables. However, they can be pipelined.

- *Step 1* reads the *fcm* and *dfcm* predicted values from the tables, indicated by the corresponding pointers ($HashKeys$). The content of the tables and pointers is updated in the same step, in parallel, to be set for the next prediction. The update of the tables consists of storing the word's original value ($TrueValue$) to the positions pointed by the current $HashKeys$. These $HashKeys$ form a type of "history register", holding sequentially the bits of the previously encountered double values (words). Thus, updating them includes loading (shifting) in their least significant bits (LSBs) the current word's $TrueValue$ [15] or a delta value [16]. However, this value is previously right shifted to eliminate a large part of the highly irregular mantissa [15].
- *Step 2* applies a XOR operation to the $TrueValue$ with the predicted values.
- *Step 3* selects the most accurate predicted value.
- *Step 4* counts the leading zero bytes of the XOR result (*cnt*) to be truncated.

– *Step 5* forms a 4-bit code (*code*), indicating the selected predictor and the *cnt* leading zero bytes. This *code* and the remaining bytes (the *residual*) of the XOR result form the compressed word. All *codes* are stored in the beginning of the compressed cache line and used as metadata in decompression.



(a) Compression

(b) Decompression

Fig. 2: BFPC (de)compression block diagrams.

3.2.2 Hardware implementation and overhead estimation The prediction tables are modeled as direct-mapped caches (DMC). The DMC size is an important design parameter, as it affects both compressibility and (de)compression latency. DMC is set to 512B in our experiments, as it gives high compression while keeping area and latency overheads low. $HashKeys$ are implemented as registers.

Fig. 2a also demonstrates the potential hardware implementation of the previously described steps that form a pipeline. *Step 1*'s critical path is reading and updating the DMC. Although reading/updating a 512B DMC requires 0.4ns based on CACTI [17], we assume that it operates at the bus speed (i.e., 1ns for 1-GHz memory-link speed), thus the total delay of reading and updating DMC is 2ns or 2 CPU cycles[3]. Updating the table pointers ($HashKeys$), which is also part of Step 1, is done using parallel shifters and XOR units. *Step 2* is a simple XOR operation (1 cycle). In *Step 3*, the selection of the most accurate prediction is implemented with a comparator and mux (1 cycle). Leading Zero byte compression of the selected XOR result in *Step 4* is done by an LZC [18] in 1 cycle delay. Finally, generating and storing the 4-bit *code* along with the *residual* in the compressed cache line buffer (*Step 5*) are done using parallel shifters (1 cycle).

Fig. 3 depicts a timing diagram of the compression pipeline. If the cache line size is 64 bytes (8 words), the overall compression delay can

[3] We assume cpu frequency equal to 1GHz

	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
...	...	...	...	...	...	...	...

Fig. 3: The overhead of the compression pipeline(in cycles)

be estimated to 14 cycles. However, in our experiment we assume a highly pessimistic overhead of 20 cycles.

Decompression: Decompression logic is very similar to compression. Fig. 2b depicts its basic steps. Note that the starting bit address of each word's *residual* part in the compressed block depends on the size of the previous stored residuals. Thus, these addresses can be computed in parallel using a multi-stage carry lookahead adder network having as input the compression metadata tag. This stage is not depicted in Figure 2b. Even if decompression is slightly faster, we will assume for simplicity that it has the same overhead as compression.

4 Experimental evaluation

4.1 Experimental setup

We evaluate the four compression schemes (BDI, Diff3, FVE and BFPC) using Sniper [19], an execution-driven application level simulator. We run multithreaded versions (8 threads) of three memory-bandwidth bound scientific applications: i) the LU decomposition algorithm, ii) the Floyd-Warshall (FW) algorithm and iii) the Sparse Matrix-Vector Multiplication (SpMV) algorithm. We experiment with both FP and integer input datasets. As the efficiency of the applied algorithms greatly depends on the input data set, we collected actual FP from three sparse matrices taken from the University of Florida Sparse Matrix Collection [20] and integer data from weighted graphs holding information for USA roads [21].

Baseline parameters	LU & FW	SpMV	
8 cores	OoO, 1GHz		
L1 cache size	4	32	KB
L2 cache size	256	4096	KB
L2 block size/assoc/hit time	64B/8-way/8 cycles		
shared cores (L2)	4		
Memory controllers	1		
Dram access penalty	100		cycles

Table 1: Baseline Configuration

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

Table 2: (De)compression overhead (cycles)

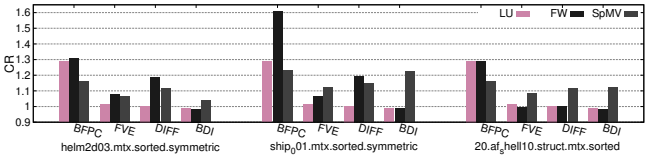
Table 1 presents the baseline simulation parameters and Table 2 summarizes the latency overheads of the compression algorithms. Due to simulation time constraints, we reduce the input data set of LU and FW and proportionally adapt the cache configuration. For SpMV we maintain a realistic CMP configuration. In all cases, the input dataset is at least 4 times larger than the LLC size. As we aim to evaluate the full potential of the studied schemes, for Diff3 and FVE we use a word size of 4 bytes in integer datasets and 8 bytes in FP datasets. On the other hand, the BDI algorithm examines itself different word granularities and selects the one that gives the best compression ratio.

We use 3 different configurations for the off-chip memory bandwidth: 2GB/s, 4GB/s and 8GB/s, as we aim to evaluate the efficiency of link compression in different traffic conditions. We assume a single memory controller, since this is essential to maintain BFPC and FVE encoding dictionaries consistent.

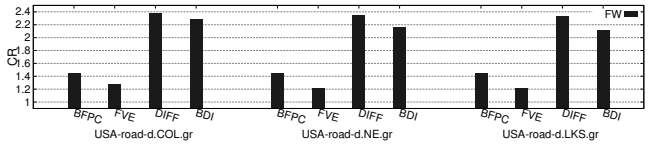
The efficiency of memory-link compression is evaluated using: i) the average main memory access latency (simulation statistic), ii) the data compressibility for each scheme: $Compression\ Ratio\ (CR) = \frac{Original\ Data\ Size_{(Total)}}{Compressed\ Data\ Size_{(Total)}}$ and iii) Speedup: $SpeedUp = \frac{Exec.\ Time\ without\ compression}{Exec.\ Time\ with\ compression}$. We verify our evaluation by running each simulation multiple times and average the results excluding outliers.

4.2 Experimental results

4.2.1 Compressibility Figs 4a and 4b present the compression ratio (CR) for the simulated applications and various input datasets for each compression scheme. CR can be less than 1 due to compression metadata overhead.



(a) Floating Point Datasets



(b) Integer Datasets

Fig. 4: Compression Ratio (CR)

Floating Point datasets. BFPC has the best performance across all applications and datasets. It reduces the off-chip memory traffic by 13% $\rightsquigarrow$ 37.8%, whereas the other schemes mostly suffer from poor compressibility. BFPC's lower efficiency for the SpMV kernel is attributed to its pure streaming nature, since BFPC's performance is highly boosted by data reuse of any degree. The SpMV kernel usually runs for multiple iterations in a scientific application (data reuse exists). Due to simulation limitations though, we gathered statistics only from single iterations of the kernel. Comparing the two bitwise compression schemes, Diff3 seems to perform better most of the times. Diff3, contrary to BDI, eliminates the common MSBs (i.e. typically sign and exponent) of the cache block values that tend to be more regular.

Interestingly, we notice that the compressibility for most schemes (except BDI) is better in FW than LU, two kernels with very similar access patterns. As application datasets are modified during execution, the compressibility of

the original input dataset does not characterize the compressibility of the workload over time. Therefore, the way the dataset changes constitutes an important compressibility parameter. We attribute the better compressibility of the FW kernel to the fact that it performs additions, while LU performs divisions – an operation that changes values fiercely, introducing higher entropy to the data. This affects mainly the efficiency of the BFPC and Diff3 compression schemes, since they both exploit the regularity of the FP MSBs.

Integer datasets. In Fig. 4b we observe that the bitwise compressors are more efficient. This is expected, as these schemes eliminate the redundancy found in binary representation due to their small variance. Diff3 has slightly better CR than BDI (57.5% vs. 54% respectively, on average), possibly because it operates on a finer granularity than BDI (bit vs. byte). On the other hand, BFPC has interestingly the same compressibility (i.e. $\sim 30\%$) as it had for floating-point workloads. This is important if overall consistency in compressibility is critical.

4.2.2 Effect on memory access latency Here we focus on the effect of link compression on the average Memory Access Latency (MAL). In Sniper, MAL consists of the DRAM access time, the over link transfer time and the queuing delays due to the competition of the cores for the shared memory resource. Thus, a link compression scheme affects only the transfer time and the queuing delays.

Fig. 5 depicts the effect of *ideal* link compression on the average MAL, i.e. when zero latency for (de)compression is assumed. The first bar corresponds to the baseline system without compression. Interestingly, the transfer time is only slightly affected and compression mainly affects the queuing delays. As the bandwidth increases, compression's effect is dramatically reduced. Hence, compression schemes are expected to be beneficial under high contention conditions.

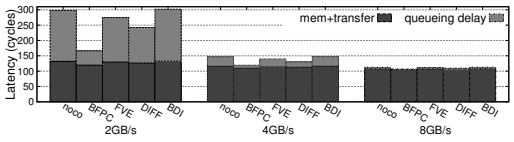


Fig. 5: Memory Access Latency breakdown

Fig. 6 illustrates the impact of each link compression scheme (with and w/o accounting for (de)compression latency) on total MAL for different bandwidth configurations for various applications and input datasets. When the application is bandwidth bound (2GB/s), memory access time is significantly reduced for all compression schemes. Moreover, under intense bandwidth conditions, the impact of (de)compression overhead is small. This is attributed to the fact that when an application is completely memory bound the rate at which it will be able to complete requests is fixed (bandwidth bottleneck domination). Therefore, the addition of an extra waiting component (de-compression overhead) just leads the requests to spend less time in the queue but hardly changes the total number of requests that can be completed per second (and hence access latency). However, as available bandwidth increases, the benefit is eliminated, while in several cases the memory access time is penalized quite significantly.

4.2.3 Effect on performance Fig. 7 demonstrates the impact of link compression on speedup for all applications and datasets. Each compression scheme

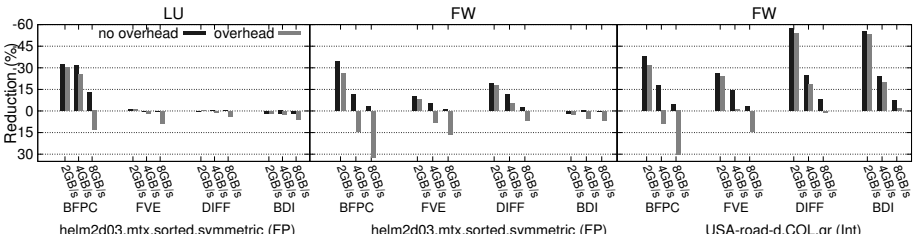
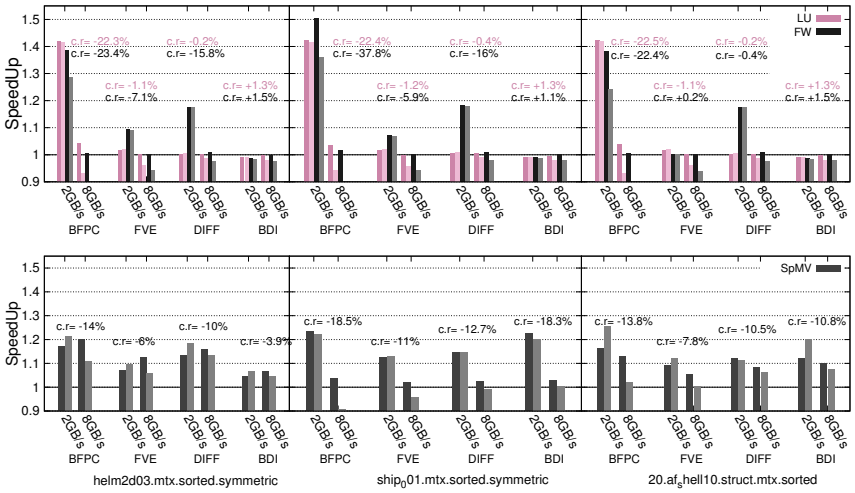
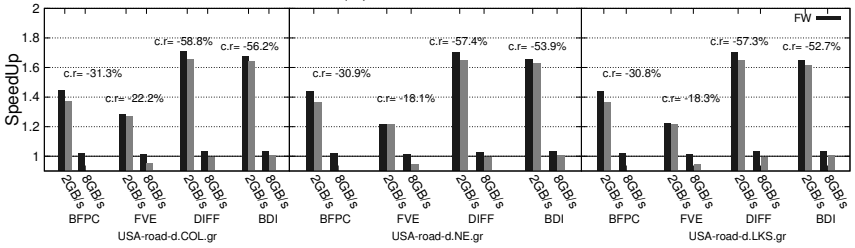


Fig. 6: Reduction of the average Memory Access Latency

is evaluated for two bandwidth conditions (2GB/s and 8 GB/s) and by accounting or not for the (de)compression overhead (light-color bars vs. dark-color ones, respectively). Furthermore, we annotate the CR on the top of each application bars. When an application is memory-bandwidth bound (2GB/s) and the compression scheme is successful, the execution time is significantly reduced. In FP datasets the higher speedup is observed for BFPC and is between ∼1.18 (CR=14%) to 1.41 (CR=29%). On the other hand, for the integer datasets, the highest speedup is provided by the bitwise compressors, i.e., ∼1.64 (CR=39%).



(a) Floating point



(b) Integer

Fig. 7: Average SpeedUp

When memory bandwidth is not a bottleneck (8GB/s bandwidth), the benefit of link compression on performance becomes negligible. In several cases, as expected, performance is actually penalized due to the introduced (de)compression time overheads. In particular, BFPC increases the execution time by 5%-10% for all applications and data sets and FVE by close to 5%. The overhead is higher for FW than LU because FW is characterized by smaller queuing delays. Therefore at 2GB/s bandwidth, the overhead of BFPC is nearly invisible for the LU kernel, whereas for the FW kernel it limits the performance by almost ∼10%. Regarding SpMV kernel though, the scheme's negative impact is limited even for 8GB/s and benefit can still be noticed, since the algorithm remains memory bound (especially for large input matrices as Helm Matrix). The system could benefit by using a mechanism to monitor the demands in memory traffic while measuring the efficiency of the compression scheme and decide to turn on/off compression.

Finally, the effect of link compression on an application's performance depends on the fraction of total execution time spent on memory accesses. Our kernels are memory bound at 2GB/s bandwidth, but not at the same degree. DRAM accesses constitute 90% of the total CPI for LU, while being much lower for FW (78%). This explains why slightly higher speedup is noticed in LU than FW, e.g when using BFPC with Helm input matrix, despite the similar compression ratio.

5 Conclusions

This work employs data compression on the off-chip link to boost the performance of scientific workloads, which manipulate large amounts of floating-point data that cannot be efficiently compressed by state-of-the-art compression schemes. We propose a hardware implementation of the FPC algorithm by Burtscher et al. (BFPC), that introduces small area overheads and has a compression ratio for FP data of 20% on average and up to 40%, while other schemes achieve less than 10% on average. This benefit off-chip traffic reduction can be translated, under bandwidth bound conditions, to performance improvement of up to 40% giving an important boost to scientific workloads.

For future work, we consider investigating an adaptive compression scheme, where compression is triggered by a monitoring mechanism only when bandwidth is constrained. We also plan to investigate the impact of Huffman coding on link compression, as a recent study has shown promising results for LLC compression [22].

Acknowledgments

Part of this work was carried out in National Technical University of Athens and was partially funded by project I-PARTS: "Integrating Parallel Run-Time Systems for Efficient Resource Allocation in Multicore Systems" (code 2504) of Action ARISTEIA, co-financed by the European Union (European Social Fund) and Hellenic national funds through the Operational Program "Education and Lifelong Learning" (NSRF 2007-2013).

Bibliography

[1] 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.

[2] 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.

[3] Jun Yang, Rajiv Gupta, and Chuanjun Zhang. Frequent value encoding for low power data buses. *ACM Trans. Des. Autom. Electron. Syst.*, 9(3):354–384, July 2004.

[4] Alaa R. Alameldeen and David A. Wood. Interactions between compression and prefetching in chip multiprocessors. In *13st International Conference on High-Performance Computer Architecture (HPCA-13 2007), 10-14 February 2007, Phoenix, Arizona, USA*, pages 228–239, 2007.

[5] Alaa R. Alameldeen and David A. Wood. Adaptive cache compression for high-performance processors. In *31st International Symposium on Computer Architecture (ISCA 2004), 19-23 June 2004, Munich, Germany*, pages 212–223, 2004.

[6] 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.

[7] 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.

[8] A. Arelakis and P. Stenstrom. A case for a value-aware cache. *Computer Architecture Letters*, PP(99):1–1, 2013.

[9] Gennady Pekhimenko, Vivek Seshadri, Yoongu Kim, Hongyi Xin, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. Linearly compressed pages: A low-complexity, low-latency main memory compression framework. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 172–184, New York, NY, USA, 2013. ACM.

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

[11] 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.

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

[13] 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.

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

[15] 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.

[16] 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.

[17] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. Cacti 6.0: A tool to model large caches. Technical report hpl-2009-85, HP Laboratories, 2009.

[18] 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.

[19] 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.

[20] T. Davis and Y. Hu. The university of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38:1–25, 2011.

[21] C. Demetrescu, A.V. Goldberg, and D. Johnson. 9th dimacs implementation challenge shortest paths, http://www.dis.uniroma1.it/challenge9/, 2005.

[22] Angelos Arelakis and Per Stenstrom. Sc2: A statistical compression cache scheme. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 145–156, Piscataway, NJ, USA, 2014. IEEE Press.