Temporal Frequent Value Locality

Lois Orosa and Rodolfo Azevedo
Institute of Computing, University of Campinas (UNICAMP), Brazil
E-mail: {lois.orosa, rodolfo}@ic.unicamp.br

***Abstract*—Frequent value locality is a type of locality based on the observation that a small set of values is accessed very frequently. Several works have exploited it to construct different architectural schemes, such as memory and cache designs or bus and network optimizations. Although these previous works consider different criteria to establish what is a frequent value and what is not, they assume that these frequent values are constant for the whole program, or they analyze the dynamic properties at a granularity too high to take advantage of them. In this paper, we observe that the frequent values change dynamically during the program execution, presenting what we call temporal frequent value locality, and its potential depends on the granularity of the observation. Furthermore, we instrument and analyze the dynamic patterns of the SPEC2006 benchmarks using two realistic schemes, and we compare them with other classical frequent value locality proposals. The results show that temporal frequent value locality has the potential to be used successfully in architectural optimizations. We also simulate our scheme for main memory bandwidth compression, achieving an increase of 63% (up to 285%) in the effective bandwidth, the double of the best tested state-of-the-art compression algorithm.**

***Index Terms*—Frequent Value Locality, Compression**

I. Introduction

Frequent Value Locality [1] (FVL) is a type of value locality that refers to the fact that a small number of values are highly repeated in memory accesses. This observation can be leveraged to optimize the cache energy efficiency [2], to design compressed data caches [3] and value-centric caches [1], or for bus encoding techniques [4]. In this work, we will focus on the analysis of the dynamic characteristics of FVL (Temporal Frequent Value Locality - TFVL) for memory operations to open new ways for designing optimizations based on this type of locality.

The basis to leveraging FVL for architectural optimizations is the fact that some of the values are much more common than others, and therefore, encoding them with a small number of bits improves the overall efficiency. As a consequence, by applying these techniques, we can save space to encode values, reduce energy consumption, reduce bus traffic, etc.

A program has high frequent value locality if there are some few values that are very common in memory accesses. In case the most accessed values do not represent an important part of the memory accesses, we say that the program has low frequent value locality. In this paper, we call frequent values to the most accessed values in the program, independently if the program present high or low frequent value locality. For example, the 8 most frequent values may represent the most of the memory accesses in a benchmark with high frequent value locality (e.g. 80%), but they may be a small portion (e.g. 15%) in a benchmark with low frequent value locality. For analyzing temporal properties, we also consider frequent values in the scope of small chunks within the same benchmark.

Several works use FVL to optimize cache, memory or bus encoding, but they have different criteria to estimate the frequent values. Possible criteria are to consider only one fixed value (the zero value) as the most frequent value [5], to consider only small values (up to 8 bits) [6] or to consider the most common values in the initial states of program execution as a representation of the frequent values of the full program [2]. All these alternatives assume that frequent values are roughly constant in the whole execution of the programs.

In this work, we show that frequent values are not constant during the full program execution: for one particular time interval they may be different from the ones of another interval, and also different from the global frequent values. To evaluate that, we divide the execution in chunks, we experiment with different sizes, and collect the frequent values on those chunks. We observe that the frequent values are different among chunks, and they are also distinct from the global frequent values. We instrument the SPEC2006 benchmarks applying various criteria to obtain the frequent values.

In the same way as our work, Yang et al. [4] suggested to dynamically change the frequent values, but their study was limited to fixed intervals of 10 million instructions. Their results did not show a great advantage in dynamically changing the frequent values. In contrast, our work explores the temporal frequent value locality at a smaller granularity, and we demonstrate that this makes the difference for most of the tested benchmarks.

Overall, the contributions of this work are twofold: First, we introduce the concept of Temporal Frequent Value Locality, and we study its potential for future architecture optimizations (Section II). And second, we propose two simple implementable hardware approaches to leverage Temporal Frequent Value Locality (Section III). Furthermore, we evaluate our proposals (Section IV) and we show its potential in a practical case for increasing the effective bus memory bandwidth (Section V).

II. Temporal Frequent Value Locality (TFVL)

We define Temporal Frequent Value Locality (TFVL) as the temporal variations in the frequent values along the execution of the programs. This means that the frequent values of a program could not be the same if we observe the execution of a program as a whole, or if we observe at some delta (a chunk with a predetermined small size) at any point of the program execution.

In this section, we measure the potential of TFVL by analyzing the frequent values in fixed-size chunks along the execution of several benchmarks. These techniques are not profitable for any real implementation because the statistics presented are the results of monitoring the whole benchmarks. However, this analysis establishes an upper bound of the benefits that we could obtain leveraging TFVL. We will provide implementable approaches in Section III.

A. Evaluation of the Potential of TFVL

We designed a pintool [7] to dynamically analyze the benchmarks from SPEC2006 suite, and collect frequent value patterns. We use the reference inputs, and gather information from all load memory access by monitoring the 8 values of the corresponding requested cache block (64 byte blocks). We use the pinpoints [8] methodology to find representative simulation points, and pinplay [9] to log these regions and deterministically replay them.

To quantify the potential of TFVL, we divided the program execution into chunks of consecutive instructions containing a fixed number of memory accesses, and we monitored the frequent values in each chunk. The overall frequent value stats of an execution are obtained by accumulating all the frequent value counts of each chunk. We reference this experiment as **Ideal Chunk (IC)** in the remaining of this paper. Notice that as the chunk decreases, the percentage of frequent values increases, up to 100% when the chunk is the size of the number of frequent values. Notice also that, despite it is well known that the programs go through phases [8], our scheme is agnostic of these phases, as they are orders of magnitude bigger (in the order of more than 100 million instructions) than our chunk size, and therefore, the set of frequent values can change a lot among the many chunks contained in a program phase.

In the IC experiment, when the chunks are very small, the results are close to the ideal (the frequent values are updated very frequently). The extreme case is when the number of frequent values match the size of the interval, implying that all the values of the interval are frequent values. On the other hand, when the chunk gets bigger, the frequent values are renewed with less frequency, and the presence of TFVL decreases with the size of the chunk. The extreme case is when there is only one chunk in the whole execution, and TFVL is not leveraged at all (equivalent to the IG experiment).

We compare TFVL against the classical FVL, which we quantify by obtaining the most frequent values in the entire execution of the program (we reference this experiment as **Ideal Global (IG)** in the remaining of this paper).

Figure 1 shows the percentage of loads that are frequent values in the IG and IC experiments (considering 8, 16, 32 and 64 frequent values). The IC experiments use chunks of 2000 load instructions. From this graph we can observe two phenomena. The first is that in some of the benchmarks the difference between the global and the chunk approaches is huge: in *h264ref* or *wrf* is approximately 2× and, in other cases, even more (*namd calculix* or *omnetpp*). The second phenomenon is that some benchmarks have a narrow frequent values range, whereas other benchmarks have a wider range of frequent values. For example, in *zeusmp* the difference between using 8 or 64 frequent values is small, however, in other benchmarks like *calculix* the difference is much bigger.

Figure 2 shows the percentage of frequent values depending on the monitored chunk's size for the IC experiment. We also included the IG experiment for reference. This graph clearly reflects that the percentage of frequent values decrease with bigger chunks. In contrast, early experiments by Yang et al. [4] use chunks of 10 million instructions, which is roughly equivalent to our experiment with chunks of 2304000 load instructions, and does not allow to experiment the advantages of changing the frequent values dynamically. Their results are very close to the global frequent value solution in most of the cases, precisely because of the chosen granularity.

We can make several observations about the Figure 2. First, some benchmarks present more TFVL than others. In *omnetpp* we can observe a big TFVL, as the difference in the percentage of frequent values decrease a lot when the chunk size increases. On the other hand, *zeusmp* or *lbm* presents low TFVL. The cases of *povray* and *milc* are particular: it presents TFVL until size 18000 (approximately), and then the frequent values in the chunks are roughly the same that the global frequent values.

III. Hardware Approaches

The results showed in the previous section are a good indicative of the potential of TFVL. However, it is not possible to take advantage of these schemes in a realistic scenario. In this section we briefly review some of the state-of-the-art proposals for leveraging frequent value locality (Section III-A), and we propose two realistic implementable techniques to take advantage of TFVL (Section III-B), one with fixed chunk sizes, and another with variable chunk sizes.

A. Approaches for Leveraging FVL

There are some previous works doing practical implementations leveraging FVL for optimizing cache designs. We describe two representative implementations that take entirely different approaches to defining frequent values.

Small Values (SV)*:* Small-value locality [6], [10] can be considered as a type of frequent value locality, as small values are usually more common than big values, and they usually are a good estimation of the actual frequent values. Small values are fixed and known during the full execution of the program, and, therefore, the hardware approaches for leveraging them can be simpler. The most typical small values are from a unique value (usually the most common is the zero) [5] to 255 values (the lowest values, represented by 8bits) [10]. However, as we will see in Section IV, the results are not as good as our chunk-based solutions.

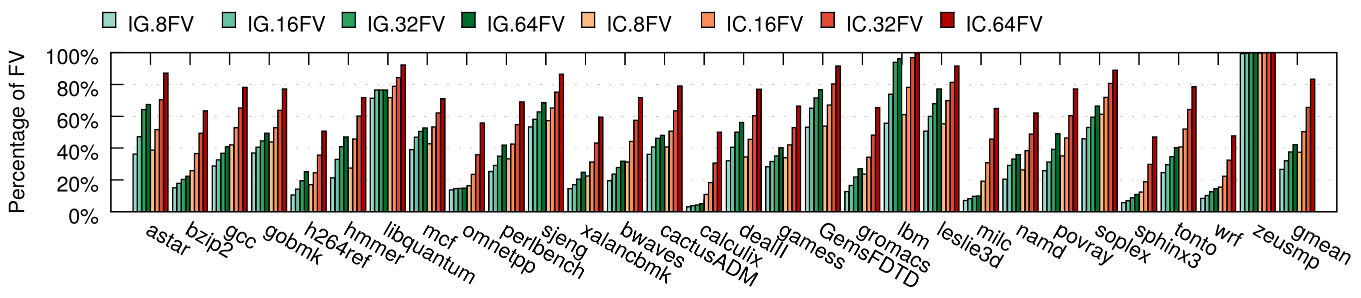

Fig. 1. Percentage of frequent values for the Ideal Global (IG) and Ideal Chunk (IC) experiments considering a different number of frequent values (FV). The IC results are obtained with chunks of 2000 load instructions.

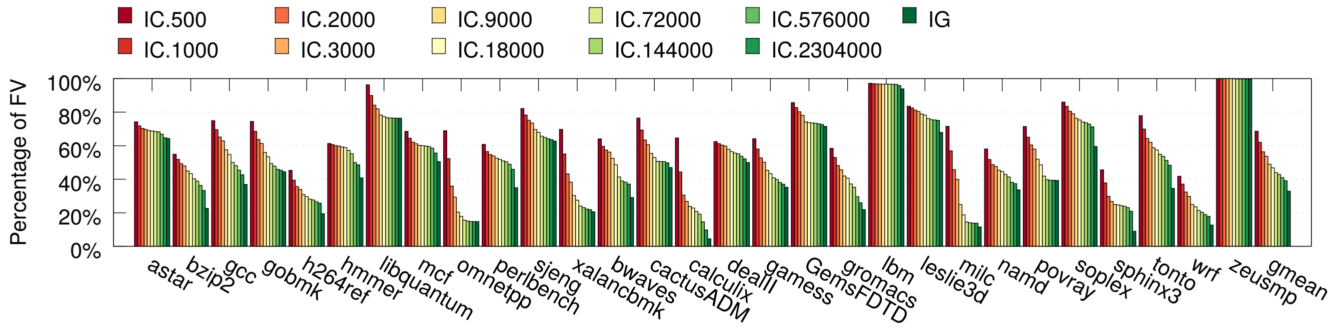

Fig. 2. Percentage of frequent values for the Ideal Chunk (IC) experiment considering chunks from sizes 500 to 2304000 load instructions. We are considering 32 frequent values.

Frequent Values at Initial States (FVIS)*:* The first paper proposing frequent value locality [1] also proposed a cache scheme that compresses the frequent values in only a few bits. The technique to estimate the program's frequent values consists of monitoring the initial states of the execution to find them out. These values discovered in the initial states will be considered in the remainder of the execution as the most frequent values, and encoded with only a few bits (compression).

This approach to find frequent values has two problems. First, the frequent values are only monitored during the initial states of the program (5% in previous works [2]) and, therefore, in most cases, they are not very representative of the whole program execution. And second, during the monitoring phase, the frequent values can not be used for compression. Therefore, with long monitoring phases, we will miss optimization opportunities, and with short monitoring phases, the estimation of the frequent values could be very imprecise.

Our approaches to leverage TFVL described in Section III-B use a model inspired by a FVL table proposed by Yang et al. [2]. They develop an energy efficient frequent value cache that uses a hardware table to calculate the frequent values. This table has $2n$ entries that are necessary to find the first n most frequent values. Each entry of this table contains two fields, one for the value, and another for a c bit saturating counter. At each memory access, this table is updated as follows: if the value is present in the table, its counter is increased. If it is not present, the value is added to the bottom half of the table

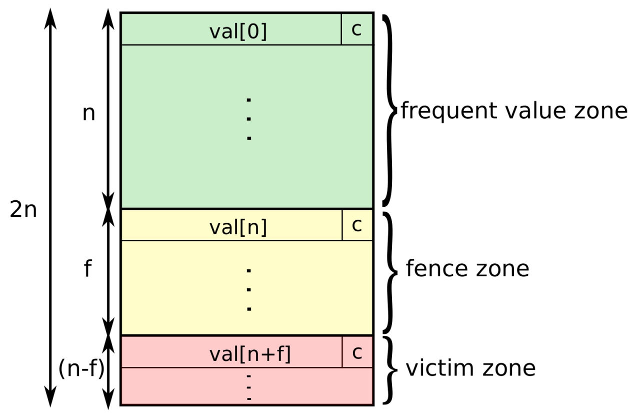

Fig. 3. Hardware table to discover frequent values, showing the frequent value zone, the fence zone and the victim zone.

with the counter set to zero (the victim is the entry with the lowest counter - the less recently seen value). If the counter saturates, the value is promoted one position up in the table (it is swapped with the next position), and the counter is reset. The top n values of the table are considered the most frequent values.

B. Approaches for Leveraging TFVL

We propose two schemes to discover frequent values dynamically and leverage TFVL, both based in a modified version of the hardware table defined previously in Section III-A.

We introduce some optimizations that work better for quickly discovering frequent values in small chunks:

- As showed in Figure 3, our hardware table has $2n$ entries, the first n represent the current frequent values (frequent value zone), the next f values are in the fence zone, and the last $n - f$ values (victim zone) are the candidates for replacement when a new entry has to be allocated. The addition of the fence zone improve the results in our scenario of small chunks by avoiding to discard some potential future frequent values.
- In our approach, we use a c field with only one bit, because we want to discover the frequent values quickly. Therefore, if we replace the values based on the smallest counter, we could have many values with the c counter set to zero. For this reason, a random replacement policy works better in our victim zone.

Next, we will describe particular changes for each proposal.

Fixed Chunks (FC)*:* The Fixed Chunks (FC) scheme divides the program execution into chunks of the same size, in the same way than IC experiment. However, to make it implementable, the frequent values considered in one chunk are calculated in the previous chunk. This model works well when there is temporal locality between consecutive chunks, which is the common case, as we will see in Section IV.

We use the hardware table previously described (Table 3) to calculate the frequent values. When a chunk finishes, we get the most frequent values in that chunk (the first n positions in the table are copied to a translation table), which are assumed to be the frequent values for the next chunk. At the end of each chunk, the table resets the victim zone, keeping the two other zones (frequent value zone and fence zone) untouched.

Variable Chunks (VC)*:* The Variable Chunks (VC) scheme does not have a fixed chunk size. The frequent values change at some point of the execution when there is a change on the first n values in the hardware table (Table 3). In practice, this happens when the entry n saturates its counter (c) and moves up a position in the table (the positions n and $n - 1$ are swapped).

Furthermore, we propose two optimizations for our approach. The first optimization, for relaxing the number of changes in the frequent values, is to change the frequent values when the n entry saturates its counter more than one time (allowing several changes in the upper n values of the table before changing the frequent values). The second optimization is to establish a minimum interval size, trying to avoid a lot of unnecessary changes mainly at the beginning of the execution.

Unlike the FC proposal, we could use the same hardware table for collecting frequent values and for encoding frequent values (it does not require an extra translation table), as the upper values of this table are always the actual estimated frequent values, which allows reducing the hardware cost.

IV. Evaluation of Hardware Approaches

We use the experimental setup described in Section II-A to evaluate our hardware proposals by comparing them with the state-of-the-art schemes of Section III and with the ideal experiments of Section II.

We track 32 frequent values for all the experiments and chunks of 2000 load instructions in Ideal Chunk (IC) and Fixed Chunks (FC). We chose these values because they achieve good results with moderate complexity (see Figure 1 and Figure 2). The small value scheme (SV) considers small values up to 255 (8bits), and the frequent value at initial states (FVIS) monitors the first 5% of the instructions to estimate the 32 frequent values (we do not use pinpoints methodology for this initial states). Our VC and FC approaches use a table (as described in Section III-B) with 64 entries, of which 32 are used for the frequent value zone, 8 for the fence zone and 24 for the victim zone. Each entry contains the value and a 1 bit counter (implemented as a simple flag). We set to 500 the minimum chunk size to change frequent values, and we establish to 1 the number of changes in the frequent value zone to update the frequent values (as we found 1 value enough in combination with the minimum interval).

Figure 4 shows the percentage of frequent values considering all the schemes. Our two approaches based in chunks (FC and VC) overcome the ideal global (IG) in 23 of 29 benchmarks, and by far in some of them (*perlbench*, *calculix* or *sphinx3*). The FVIS scheme demonstrates poor performance in general, which shows that it is not a good strategy when running reference inputs and complete benchmarks (the initial values, usually corresponding to the initialization phase of the benchmarks, do not represent the whole execution). The SV approach works pretty well for some benchmarks (e.g. $sjeng$), but in average is still very far from our approaches. Notice also that the SV and the FVIS are realistic ways to achieve the best global frequent values, and therefore their results are always below the ideal global frequent values (IG).

Furthermore, the results for our FC and VC are very close to the ideal chunk scheme (IC), which demonstrates that our realistic schemes perform pretty well. Finally, we can see in the figure that VC outperform the FC scheme in almost all the cases, mainly for its runtime adaptability characteristics.

Figure 5 shows the average interval size of our variable chunk (VC) approach; 12 of 29 benchmarks have an interval average size superior to the FC fixed 2000 chunk size, and the remainder have a size close to 2000 (only *lbm* has a size inferior to 1000). Despite the difference in the number of frequent values between FC and VC (see Figure 4) is not big in any case (although superior in VC in most of the cases), bigger chunks allow to reduce complexity and save energy.

Figure 5 also shows the approximate energy consumption of the VC scheme, calculated using a customized mcpat [11] version. We modeled the VC table as a CAM, and we compare its energy consumption with the energy consumed by a 32KB data cache with 8 ways and 4 cycle latency. The figure shows that the energy consumption is contended for the most of the benchmarks (between 4,8% and 10,8%), but still with some space for optimizations.

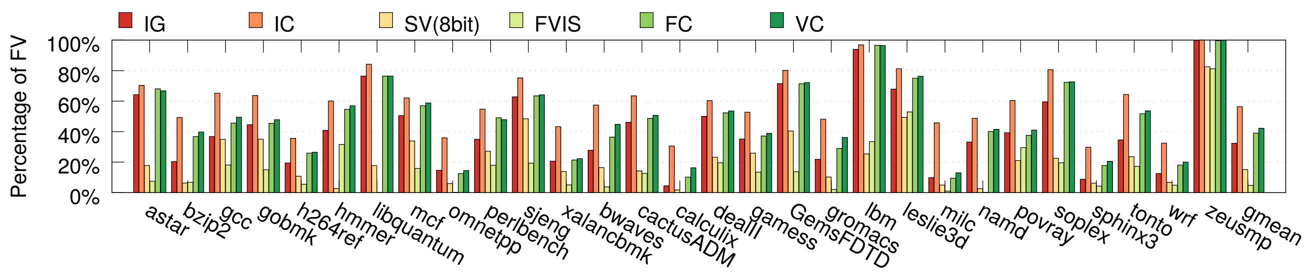

Fig. 4. Percentage of frequent values in all approaches considered in this work. We track 32 frequent values and chunks of size 2000.

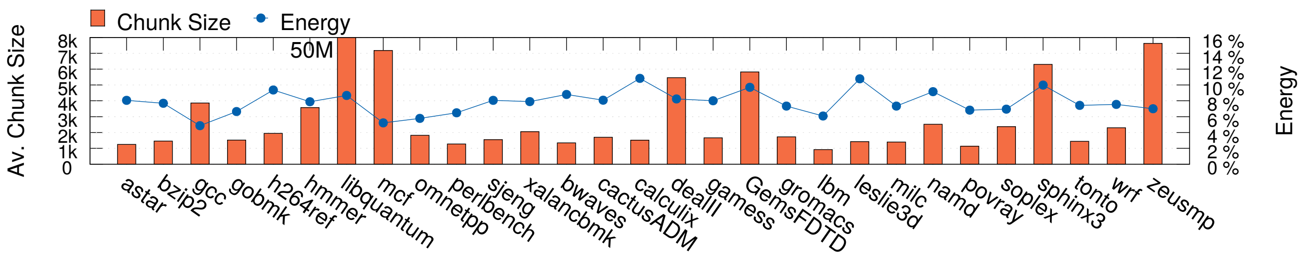

Fig. 5. Average interval size in the VC approach, and the energy consumption compared with a 32KB data cache.

A. Discussion

The most challenging problem to take advantage of chunk-based scheme is the fact that the frequent values could potentially change at every chunk. When designing a memory device, this could introduce a source of complexity and energy consumption, and it requires taking the design carefully. The size of the chunks in the FC scheme is also a key factor to reduce complexity and energy.

However, leveraging TFVL for compression of data in bus or communication networks is more straightforward, as the compressed data do not remain in the network, and therefore, we do not need to change them when the frequent values change. Notice also that, depending on the optimization for which TFVL is been used, the frequent values should be accumulated in different places (cache, memory, etc.).

V. A Practical Case: DRAM Bandwidth Compression

In this section, we show a practical case for leveraging TFVL to increase the main memory bandwidth. To show that, we used zsim [12] for simulating an OoO core and the main memory system. Our baseline configuration, described in Table I, is composed of one Westmere OoO core, separate 32kB L1 caches for instructions and data, a shared 256KB L2 cache, and a DLP L2 prefetcher. The memory system has 2 channels and a DDR3-1333-CL10 module per channel, with a theoretical maximum bandwidth of 20,8 GB/s. Notice that, unlike previous section, the only compressed memory requests are the ones that miss the last level cache.

We implemented compression techniques to the data read from memory with the aim of increasing the effective bandwidth. We compare our VC scheme with three state-of-the-art compression algorithms: Small Values (SV) [5], Frequent Pattern Compression (FPC) [13] and Base-Delta-Immediate compression (BDI) [14]. In the SV scheme, we consider 256 frequent values (from 0 to 255). Our implementation of the FPC scheme has 8 different patterns (3 bits), and it compresses 32 bit words. BDI is configured with two bases, 4-byte offset, four bits to encode the compression scheme, and a bit mask to differentiate between two bases.

We test all the SPEC2006 benchmarks, using the same pinpoints methodology [8] of Section II-A. Figure 6 shows the effective bandwidth increase when using SV, FPC, BDI and VC schemes, compared with a baseline system without any compression mechanism. In 10 of 26 benchmarks, our VC scheme achieves more than 100% of bandwidth increase, and 20 of 26 benchmarks reach more than 50% of bandwidth increase. Compared with the other schemes, our VC only has worst performance in 5 of 26 benchmarks (and for a low margin). Overall, the SV scheme shows an average 16.6% bandwidth increase (geometric mean), the BDI achieves 16.3%, the FPC schemes reach 31.7% and our VC scheme goes up to 62.8%, which demonstrates the advantages of leveraging TFVL. Notice also that the energy consumption of our VC scheme in this scenario is much lower than in Figure 5, as we are compressing only on Last Level Cache misses.

In general, the benchmarks that exhibit a good frequent value locality, have a very good temporal frequent value locality (the most of the SPEC2006 benchmarks). Furthermore, some benchmarks with low frequent value locality have a good temporal frequent value locality (the cases of $bzip2$ or $hmmer$).

VI. Conclusions

In this paper we propose temporal frequent value locality as a new opportunity to implement architecture level optimiza-

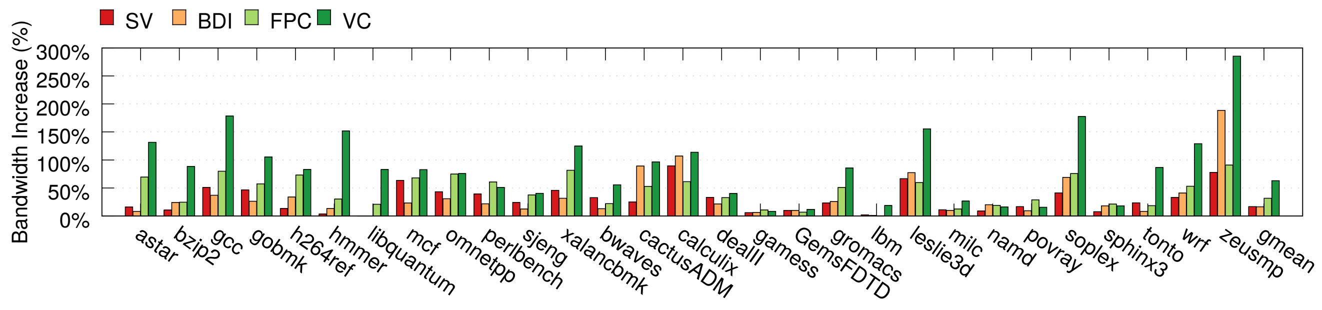

Fig. 6. Effective bandwidth increase of the SV, BDI, FPC and our VC scheme, compared with the uncompressed baseline.

TABLE I
PROCESSOR AND MEMORY CONFIGURATION OF THE BASELINE SIMULATOR FOR EVALUATING DRAM BANDWIDTH COMPRESSION.

core	Westmere OOO x86-64bit, 4-issue 128-entry ROB, 32-entry load queue 32-entry store queue
L1I Cache	32KB 4-way, LRU, 3-cycle latency
L1D Cache	32KB 8-way, LRU, 4-cycle latency
L2 Cache	256KB, 16-way, LRU, 12-cycle latency 16 MSHRs
Prefetcher	DLP L2, 16 stream buffers, 64-line buffers
Memory	DDR3-1333-CL10, 2 channels, 4 ranks per channel, 8 banks per rank (max. 20.8 GB/s)

tions (memory hierarchy and communication networks). First, we analyze the potential of the temporal frequent value locality (TFVL) and we observe that for some benchmarks the benefit is very big compared with the classical frequent value locality (FVL). Second, we propose two realistic hardware schemes that leverages TFVL, achieving better results (in average, 49% of the memory accesses in SPEC2006 get one of the 32 most frequent values) than the ideal scheme for FVL in all the cases, and getting close to the ideal chunk based scheme. Finally, for proving the potential of leveraging TFVL properties in the memory subsystem, we evaluate one of our TFVL schemes for bandwidth compression in an OoO core with ordinary DRAM memory. The results show that our scheme increases the average bandwidth in 63% (up to 285%), the double than the best state-of-the-art compression algorithm we tested.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments. This work was supported by FAPESP (grants 2014/03840-2 and 2013/08293-7), CNPq and CAPES.

REFERENCES

[1] Y. Zhang, J. Yang, and R. Gupta, “Frequent value locality and value-centric data cache design,” *SIGPLAN Not.*, vol. 35, no. 11, pp. 150–159, Nov. 2000.

[2] J. Yang and R. Gupta, “Energy efficient frequent value data cache design,” in *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 35. Los Alamitos, CA, USA: IEEE Computer Society Press, 2002, pp. 197–207.

[3] J. Yang, Y. Zhang, and R. Gupta, “Frequent value compression in data caches,” in *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 33. New York, NY, USA: ACM, 2000, pp. 258–265.

[4] J. Yang and R. Gupta, “Frequent value locality and its applications,” *ACM Trans. Embed. Comput. Syst.*, vol. 1, no. 1, pp. 79–105, Nov. 2002.

[5] M. M. Islam and P. Stenstrom, “Zero-value caches: Cancelling loads that return zero,” in *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 237–245.

[6] ——, “Characterizing and exploiting small-value memory instructions,” *IEEE Transactions on Computers*, vol. 63, no. 7, pp. 1640–1655, 2014.

[7] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: Building customized program analysis tools with dynamic instrumentation,” in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’05. New York, NY, USA: ACM, 2005, pp. 190–200.

[8] E. Perelman, G. Hamerly, and B. Calder, “Picking statistically valid and early simulation points,” in *Parallel Architectures and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on*. IEEE, 2003, pp. 244–255.

[9] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, “Pinplay: A framework for deterministic replay and reproducible analysis of parallel programs,” in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO ’10. New York, NY, USA: ACM, 2010, pp. 2–11.

[10] M. M. Islam and P. Stenstrom, “Characterization and exploitation of narrow-width loads: The narrow-width cache approach,” in *Proceedings of the 2010 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES ’10. New York, NY, USA: ACM, 2010, pp. 227–236.

[11] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, “Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures,” in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: ACM, 2009, pp. 469–480.

[12] D. Sanchez and C. Kozyrakis, “Zsim: Fast and accurate microarchitectural simulation of thousand-core systems,” in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA ’13. New York, NY, USA: ACM, 2013, pp. 475–486.

[13] A. R. Alameldeen and D. A. Wood, “Adaptive cache compression for high-performance processors,” in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ser. ISCA ’04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 212–.

[14] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, “Base-delta-immediate compression: Practical data compression for on-chip caches,” in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT ’12. New York, NY, USA: ACM, 2012, pp. 377–388.