Frequent Value Compression in Data Caches *

Jun Yang Youtao Zhang Rajiv Gupta
Department of Computer Science
The University of Arizona, Tucson, AZ 85721

Abstract

Since the area occupied by cache memories on processor chips continues to grow, an increasing percentage of power is consumed by memory. We present the design and evaluation of the **compression cache** *(CC) which is a first level cache that has been designed so that each cache line can either hold one uncompressed line or two cache lines which have been compressed to at least half their lengths. We use a novel data compression scheme based upon encoding of a small number of values that appear frequently during memory accesses. This compression scheme preserves the ability to randomly access individual data items. We observed that the contents of 40%, 52% and 51% of the memory blocks of size 4, 8, and 16 words respectively in SPECint95 benchmarks can be compressed to at least half their sizes by encoding the top 2, 4, and 8 frequent values respectively. Compression allows greater amounts of data to be stored leading to substantial reductions in miss rates (0-36.4%), off-chip traffic (3.9-48.1%), and energy consumed (1-27%). Traffic and energy reductions are in part derived by transferring data over external buses in compressed form.*

1 Introduction

The portable computing devices being designed today are typically battery powered. Thus in addition to meeting the performance goals, the designs for such devices must also be power efficient. One significant source of power consumption is the cache memory on processor chips which continue to occupy increasing amounts of chip area. Reducing the sizes of on-chip caches is not the answer because higher miss rates result in performance loss and an increase in power consumed by external buses.

By storing code and data in compressed form, smaller caches can provide lower miss rates and reduce power consumption. The power consumed by external buses can be further reduced by transferring code or data that is fetched across the buses in compressed form. Code compression is being widely studied today by researchers for the purpose of reducing power consumption [9, 11]. Recently techniques for packing of narrow width data operands in multimedia applications have been explored [4, 15, 16]. However, little work has been done on compression of data in caches for general purpose applications.

The memory compression techniques proposed in [2, 3] are applicable to data in main memory – the data is uncompressed when it is brought into any of cache levels in the memory hierarchy. In [8] authors proposed the use of the X-RL [7] data compression algorithm to achieve compression of data. The X-RL algorithm does not preserve random access of individual data elements. Therefore in [8] compressed data cannot be stored at the top level L1 cache; only at the lower levels of the memory system (L2 cache) can it be kept in compressed form. We present a compression scheme applicable to L1 caches in this paper. The only other technique which meets this requirement has been developed independently by Larin and Conte [12].

In order to achieve higher performance and lower power consumption, data compression should be applied at the first level cache. The higher performance results because by storing data in compressed form we can store greater amounts of it and therefore maximize the hits to this *fast* cache. Optimizing the performance of L1 cache has the most reduction in power consumption because the power consumed by L1 cache is over three times of that consumed by the L2 cache [5]. This is because L1 cache services far more memory references than the L2 cache.

In this paper we present the design and evaluation of the *compression cache* (CC) which not only stores data in compressed form, but it can also be used as the top level cache. This is because CC employs a

*Supported by DARPA award no. F29601-00-1-0183 and NSF grants CCR-0096122 and EIA-9806525 to the Univ. of Arizona.

novel compression scheme which allows random access of data elements in the cache. The CC has been designed to improve the performance of on-chip Direct-Mapped write-back data Caches (DMCs). We only consider write-back caches because write-through caches generate much greater degrees of off-chip traffic and are therefore not power efficient. In a direct-mapped CC each line of $2l$ words is also capable of storing two compressed lines as long as each of the lines can be compressed to size l. Since two compressed lines can potentially reside in a cache line simultaneously, more data can be held by the cache and reduced miss rates are observed in comparison to a conventional DMC. The compression scheme we have developed exploits *frequent value locality* [18] observed in programs. It preserves random access of data, it is applicable to all data (not just narrow width data), and is therefore useful in context of general purpose applications (not just multimedia applications).

The CC provides substantial reductions in off-chip traffic. The reduction in off-chip traffic is only in part due to reduced miss rates. There is an additional source of traffic reduction. Since cache lines can be stored into the CC in compressed form, they are compressed off-chip before they are brought into the on-chip cache. Also evicted cache lines are transmitted off-chip in compressed form where they are decompressed before being stored into memory or another off-chip cache. CC provides substantial reductions in miss rates (0-36.4%), off-chip traffic (3.9-48.1%), and energy consumed (1-27%) over a DMC.

Section 2 evaluates the potential for compressing cache lines to half their sizes by encoding frequently accessed values for SPECint95 suite. In section 3 we describe the design of CC. Section 4 presents an evaluation of CC. Related work is discussed in section 5.

2 Frequent Value Compression

In our prior work we studied the behavior of programs in the SPECint95 suite and found that six out of eight benchmarks exhibit *frequent value locality* [18]. In these six programs ten distinct values occupy over 50% of all memory locations and on an average account for nearly 50% of all memory accesses during program execution. The two benchmarks that do not exhibit high degree of frequently value locality are `129.compress` and `132.ijpeg`. In this paper we develop a data compression scheme for use in a first level cache which exploits frequently accessed values. Profiling techniques for identifying frequent values are described in [18].

Compression of the data in a cache line can be achieved by storing selected values in encoded form, as opposed to their original form which takes up a full word. The values that should be selected for encoding should be the frequently accessed values to maximize the compression that can be achieved. In particular, our goal is to exploit the instances in which $2l$ words of data can be compressed into l words. This would allow a cache line that holds $2l$ words of uncompressed data, to be able to hold two cache lines of compressed data.

In order to evaluate the potential of the above compression strategy, we studied the distribution of the top n frequently accessed values in individual cache lines. We pick n as power of 2 for efficient encoding. We ran the six benchmarks with frequent value locality and examined the memory contents midway through their executions. We divided the memory into blocks of 4, 8 and 16 words each to mimic cache line sizes of 4, 8 and 16 words. The number of frequent values chosen are 2, 4 and 8. In order to estimate the likelihood that a cache line could be compressed to half of its size we plotted the percentage of lines in which at least half of the values are frequently accessed values (see Figure 1). On an average nearly 40%, 52% and 51% of cache lines of sizes 4, 8 and 16 respectively can be compressed to at least half their size by exploiting top 2, 4, and 8 frequent values respectively.

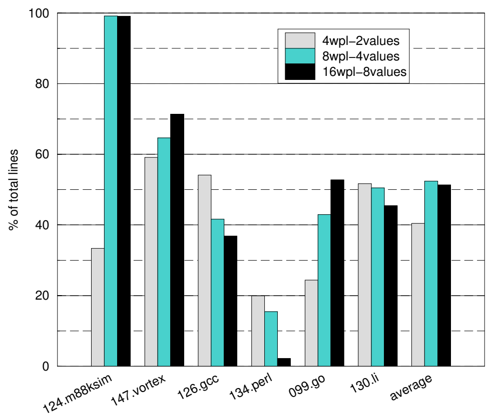

Figure 1. Lines that can be compressed to at least half of their sizes.

3 The Compression Cache

The basic idea behind the *compression cache* (CC) is to store cache lines in a compressed form so a greater number of cache lines can reside in the data cache at any given time and thus lower miss rates would result. Since we are also interested in reducing off-chip traffic, to reduce the power consumed by external buses, we compress the data in a cache line before it is brought into the on-chip cache. Also when a compressed cache line is evicted from the data cache, it is transmitted off-chip in compressed form and then uncompressed before being stored in off-chip memory (see Figure 2).

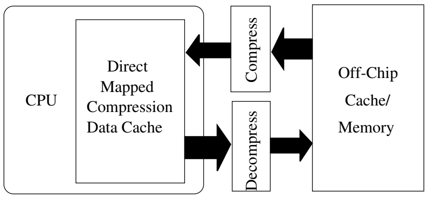

Figure 2. Compression cache.

We assume that each given cache line of $2l$ words can accommodate either one uncompressed cache line or two compressed cache lines. If the line cannot be compressed to l words we keep it in uncompressed form. However, if two lines, each of which has been compressed to l words, map to the same cache line, they can reside in that line simultaneously. It should be noted that by incorporating compression, we will always improve the hit rates of a direct-mapped cache. This is because if no compression opportunities exist, the behavior of the CC will simply be identical to that of DMC. However, if compression opportunities exist, some amount of cached data will be able to reside in the cache for a longer duration and thus potentially contribute to increased hit rate. It is possible to design more general strategies which can compact a cache line to any size less than $2l$ and thus we could allow varying number of cache lines to fit in $2l$ words. However, such complex compression strategies would make it more difficult to determine whether a hit or a miss has occurred and will therefore slow down all cache accesses.

Compression/decompression of already cached data. Compression techniques have been used effectively for instructions because typically code is not modified by a running program. However, data compression techniques have been hard to design because data values change as the program executes. As described above the data is transferred on-chip and off-chip in compressed form. However, when already cached data is modified, opportunities to compress currently uncompressed line may arise. Also if an infrequent value is written to a location in the compressed line where previously a frequent value was stored, the need to uncompress the line may arise.

We conducted experiments in which compression and decompression of already cached data was allowed to occur to see how often compression opportunities for cached data arose. We measured the percentage of total cache hits during which a compression opportunity arose. We found that such opportunities are quite infrequent for most benchmarks. As shown in Table 1, the only case in which compression opportunities are substantial is for `124.m88ksim` benchmark when a line size of 8 words is used. Therefore we decided not to exploit these opportunities. Carrying out compression of cached data is an expensive operation with associated hardware and execution time costs. By sacrificing a small number of compression opportunities we greatly simplify the cache design.

Assuming that no compression of already cached data will be carried out, we next determined how often a compressed cache line must be decompressed as a result of writing an infrequent value to a location in the compressed line. In Table 2 the percentage of total cache hits during which a need for decompression arose is given. Although decompression operations cannot be avoided, they are so infrequent (a maximum of 0.367% of the hits was observed) that they will not seriously effect cache performance. During decompression a compressed line can be read out of the cache in a buffer, decompressed, and then written back to the cache line. Alternatively we can also send it to memory thereby evicting the decompressed line from the cache. We opted for the former solution since it is faster to update a cache line on-chip than transferring its contents off-chip.

From the above experiments we can conclude that if data is brought on-chip in compressed form, it generally continues to stay in compressed form. The reverse is also true, that is, in most benchmarks if data is brought on-chip in uncompressed form it cannot often be compressed later on.

CC design details. As shown in Figure 3, the cache entries must be modified to indicate whether or not they contain compressed lines. The C bit is used for this purpose. We must also modify the entries so that they can hold the relevant information for the two compressed cache lines. Each of the lines has its own tag (Tag1, Tag2) as well as valid (V1, V2) and dirty (D1, D2) bits. In addition, the mask fields (mask1, mask2) provide useful information for compressed lines. The determination of a cache hit is straightforward. If there is a match with any of the valid tags we have a hit. The retrieval of the value requires examining the mask. If the mask indicates that the value is one of the frequent value, then the mask can provide the value as it stores the value in encoded form. On the other hand if the mask indicates that the value is not one of the frequent values, then it identifies the word in the cache line where it is stored. When compressed cache lines

benchmark	line size = 4 words = 16 bytes				line size = 8 words = 32 bytes				line size = 16 words = 64 bytes			
	4Kb	8Kb	16Kb	32Kb	4Kb	8Kb	16Kb	32Kb	4Kb	8Kb	16Kb	32Kb
124.m88ksim	0.897	0.901	0.965	0.983	8.249	8.194	16.060	16.077	0.341	0.379	8.243	8.297
147.vortex	5.746	5.974	6.358	6.591	2.566	2.650	2.833	2.923	1.147	1.241	1.336	1.410
126.gcc	2.630	2.859	3.095	3.340	1.656	1.787	1.944	2.072	0.890	0.970	1.051	1.091
134.perl	7.152	8.423	7.039	7.048	4.303	2.283	4.963	4.964	2.112	1.219	2.907	2.907
099.go	2.042	2.323	2.718	2.778	0.941	1.093	1.261	1.319	0.816	0.890	1.147	1.798
130.li	4.031	4.083	4.472	5.123	2.523	2.514	2.591	2.868	0.384	0.381	0.393	0.536

Table 1. % of cache hits creating compression opportunities.

benchmark	line size = 4 words = 16 bytes				line size = 8 words = 32 bytes				line size = 16 words = 64 bytes			
	4Kb	8Kb	16Kb	32Kb	4Kb	8Kb	16Kb	32Kb	4Kb	8Kb	16Kb	32Kb
124.m88ksim	0.284	0.282	0.277	0.277	0.207	0.208	0.204	0.205	0.015	0.006	0.003	0.003
147.vortex	0.304	0.232	0.123	0.087	0.143	0.110	0.063	0.044	0.075	0.058	0.037	0.023
126.gcc	0.242	0.197	0.167	0.153	0.151	0.132	0.109	0.102	0.071	0.058	0.051	0.048
134.perl	0.183	0.009	0.179	0.178	0.093	0.005	0.005	0.004	0.266	0.003	0.003	0.003
099.go	0.203	0.136	0.081	0.058	0.136	0.093	0.056	0.039	0.154	0.110	0.062	0.038
130.li	0.404	0.367	0.284	0.118	0.136	0.128	0.112	0.045	0.064	0.063	0.059	0.022

Table 2. % of cache hits requiring decompression.

are transmitted across the chip boundary, the contents of the masks must also be transmitted along with the frequent values. Figure 3 depicts the logic for retrieving a value from a cache line size of four words.

Next let us consider the encoding scheme in greater detail. When a cache line has been compressed to at least half its size, access to data requires consulting the mask corresponding to the compressed line. The mask contains as many fields as the original line size. Each field provides us with the necessary information regarding the data value at the corresponding location in the cache line. Each field is $log_2 l$ bits long where l is the line size. The first bit in a field is 0 if the corresponding location contains a frequent value; otherwise it is 1. The remainder of the bits serve a dual purpose. If the value at the corresponding location is a frequent value, the remaining bits of the field provide an encoding of this frequent value. Since $log_2(l/2)$ bits are available for encoding frequent values, at most $l/2$ frequent values can be exploited by the above design. On the other hand if the corresponding value is an infrequent value, then the remaining bits in the field indicate the position in the compressed data line at which that value is stored. Notice that the increase in cache line lengths due to additional information we store is quite modest. For cache line size of 32 bytes or 8 words, the increase is 6 bytes.

The example in Figure 4 illustrates the above encoding scheme. We assume that the line size is eight words and therefore four frequent values can be exploited. The frequent values and their encodings are given in the figure. Note that the leading bit is zero for all four frequent value codes. The size of an uncompressed line is 256 bits ($= 8 \times 32$) while the size of a compressed line is 152 bits($= 8 \times 3 + 4 \times 32$). We show the contents of corresponding uncompressed and compressed cache lines and describe how the contents to the latter change with changes to the former.

In its initial state the uncompressed line contains four frequent and four infrequent values and therefore it can be compressed as shown in the figure. The leading bit is zero for the fields at positions 1, 3, 5, and 8 because they correspond to frequent values, 0 and -1, which are appropriately encoded by the remaining two bits. The fields at positions 2, 4, 6, and 7 have their leading bits as one to indicate the presence of four infrequent values which are stored in uncompressed form in the first half of the cache line. The last two bits of the mask encodes the positions in the first half of the cache line where the infrequent values are stored.

Next we illustrate how the changes in the contents of an uncompressed line are reflected by changes in the corresponding compressed line. The example illustrates the following cases: (a) the overwriting of a frequent value (0) by a different frequent value (1) results only in a change in the appropriate field of the mask; (b) the overwriting of an infrequent value (1000) by a different infrequent value (2000) results only in a change in the location in the cache line indicated by the appropriate field of the mask; (c) increase in the number of frequent values, due to overwriting of 99999 by -1, changes the mask; (d) decrease in the number of frequent values, due to overwriting of -1 by 6a8d, changes the mask and the cache line; and (e) finally the over-

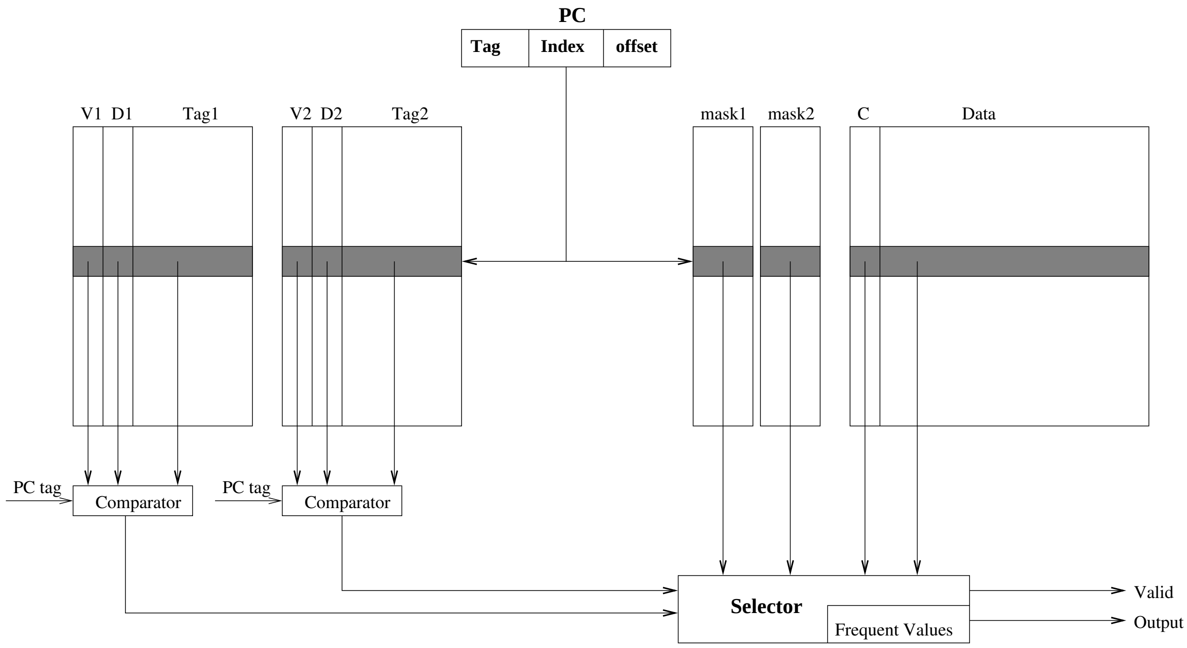

Figure 3. Compression cache design details.

Frequent Value(32bits)	0	-1	1	2
Encoding(3bit)	000	001	010	011

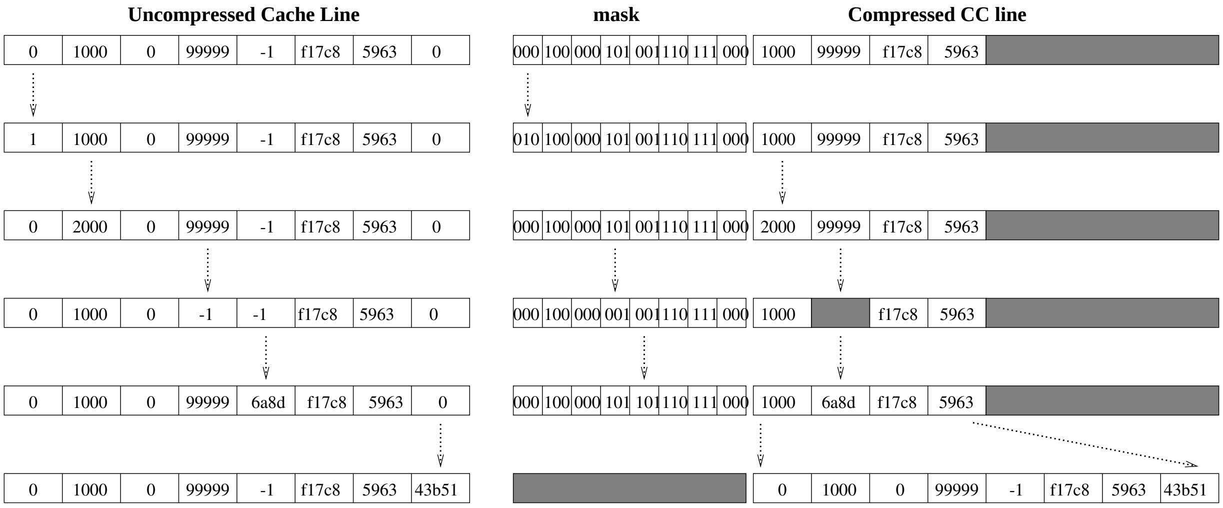

Figure 4. Compressed encoded data in CC.

writing of the frequent value (0) by an infrequent value (43b51) causes the line to be decompressed.

4 Experimental Evaluation

The goal of the experimentation was to determine by how much can the CC enhance the performance of DMCs of varying configurations. We considered DMCs of sizes 4, 8, 16, and 32 Kbytes for line sizes of 4, 8, and 16 words in our experiments. The corresponding CC therefore could potentially hold two lines each compressed down to a size of 2, 4, and 8 words respectively. Moreover these caches could exploit 2, 4, and 8 frequent values respectively. All programs were compiled using the `gcc 2.7.2` compiler with the `-O3` level of optimization, the instruction set used was MIPS-I, and the programs were executed using simulators generated using the FAST system [13].

Miss rate reductions. The cache miss rates of the DMC and CC data caches for the six integer benchmarks are shown in Table 3. In every single case the miss rate of CC is lower than that of DMC which shows that CC only improves performance. The percentage reduction in miss rate given by the column IMP varies from 0% to 36.4%.

In the above comparisons due to the additional information in the CC's cache lines, actually the sizes of the CC's are somewhat larger than that of corresponding DMC's. In fact the CC cache lines are longer by 1 to 2 words depending upon the configuration. However, this does not mean that the same improvements could have been obtained by simply constructing a larger DMC. For example, if we look at the miss rates shown in Table 3, we notice that for the `124.m88ksim` benchmark the performance of the 8Kb CC is very close to the performance of a 16Kb DMC. In contrast the increase in size of the CC due to additional information stored in each cache line is quite modest (15% - 36%).

Traffic reduction. The reductions in traffic are substantial for most benchmarks and cache sizes as shown in Figure 5. They vary from 3.9% to 48.1%. The percentage reductions in the traffic are higher than the miss rate reductions because only part of the traffic reductions are derived from miss rate reductions. The remaining traffic reduction is due to transmission of data across the processor chip boundary in compressed form.

The relative improvements provided by CC do not necessarily correspond to the relative degree of compressibility observed in the data presented in Figure 1. For example, in Figure 1 we observed that `134.perl` demonstrated much lower levels of compressibility than `099.go`. However, CC provides greater performance improvements for `134.perl` than those achieved for `099.go`. This is due to a number of reasons. First, in the study the compressibility was measured at one program point during program's execution while the performance of CC depends upon the compressibility of cache lines over the entire execution of the program. Second, in the study all memory lines are treated equally while CC will benefit most from the compressibility of the most frequently referenced cache lines. Finally, compressibility of a cache line is only beneficial to CC if other conflicting cache lines are also compressible.

Energy savings. The energy savings were computed using an energy model for 0.8 micron technology used in the *Simplepower* tool [17]. This model computes the energy consumed by cache memory cells, address and data buses, address and data pads, and main memory. The savings in energy consumed by using CC instead of DMC are shown in Figure 6. The energy savings are also significant and vary from 1% to 27%.

Access times. The additional logic in CC should result in greater access time than the same sized DMC. However, since the CC provides a more cost effective solution for improving miss rates, than simply increasing the size of the DMC, we expect that the access time of a CC would be close to the access time of a DMC which provides comparable miss rates. This is because the access time of a cache typically depends upon the size of the cache and larger caches have longer word line lengths and bit line lengths which leads to greater word line and bit line capacitances. Therefore a CC should provide energy savings over a DMC with similar access times and miss rates.

5 Related Work

Data Compression. Recently Larin and Conte [12] have also proposed data compression schemes which share an important characteristic with our compression scheme. Both preserve random access to data since they encode individual values to achieve compression. They explore three different encoding schemes (Rigid Huffman, Flexible Huffman, and Flexible Huffman with Long Memory) in their work. There are two key differences between our approaches. First, while we encode a very small number of frequently accessed values, they encode a larger number of values. Therefore while a simple encoding scheme suffices in our

benchmark	4Kb			8Kb			16Kb			32Kb		
	DMC	CC	IMP	DMC	CC	IMP	DMC	CC	IMP	DMC	CC	IMP
	line size = 4 words = 16 bytes											
124.m88ksim	2.1	1.4	33.3	1.4	0.9	35.7	1.1	0.7	36.4	1.0	0.7	30.0
147.vortex	5.2	4.6	11.5	3.7	3.1	16.2	1.7	1.6	5.9	1.1	1.0	9.1
126.gcc	6.0	5.5	8.3	4.1	3.8	7.3	2.9	2.7	6.9	2.2	2.0	9.1
134.perl	5.8	4.4	24.1	4.2	3.4	19.0	3.5	2.7	22.9	3.5	2.7	22.9
099.go	14.4	14.1	2.1	9.4	9.3	1.1	5.7	5.6	1.8	3.3	3.2	3.0
130.li	4.8	4.5	6.2	3.2	3.1	3.1	2.2	2.2	0.0	1.3	1.3	0.0
	line size = 8 words = 32 bytes											
124.m88ksim	1.9	1.6	15.8	1.3	1.0	23.1	0.9	0.7	22.2	0.9	0.6	33.3
147.vortex	5.5	4.6	16.4	3.9	3.2	17.9	1.8	1.5	16.7	1.2	0.9	25.0
126.gcc	5.9	5.5	6.8	3.9	3.7	5.1	2.6	2.4	7.7	1.9	1.7	10.5
134.perl	6.5	5.8	10.8	4.5	4.0	11.1	3.9	3.3	15.4	3.8	3.3	13.2
099.go	17.3	16.6	4.0	11.2	10.7	4.5	6.4	6.1	4.7	3.5	3.2	8.6
130.li	3.7	3.5	5.4	2.4	2.3	4.2	1.4	1.4	0.0	0.8	0.8	0.0
	line size = 16 words = 64 bytes											
124.m88ksim	2.4	1.7	29.2	1.2	0.9	25.0	0.8	0.6	25.0	0.8	0.5	37.5
147.vortex	6.0	5.0	16.7	4.3	3.7	14.0	2.0	1.5	25.0	1.4	0.9	35.7
126.gcc	6.2	5.8	6.5	4.0	3.7	7.5	2.6	2.3	11.5	1.8	1.6	11.1
134.perl	6.3	5.3	15.9	3.7	3.0	18.9	2.9	2.5	13.8	2.8	2.5	10.7
099.go	21.9	20.4	6.8	14.4	13.5	6.3	8.2	7.5	8.5	4.4	3.9	11.4
130.li	3.4	3.2	5.9	2.0	2.0	0.0	1.0	1.0	0.0	0.5	0.5	0.0

Table 3. DMC vs CC : % miss rates (DMC,CC) and % miss rate reduction (IMP).

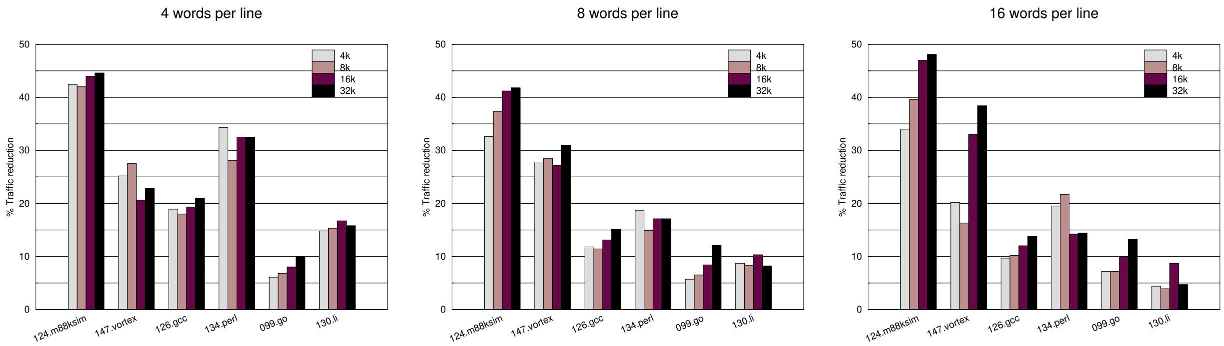

Figure 5. % Traffic reduction.

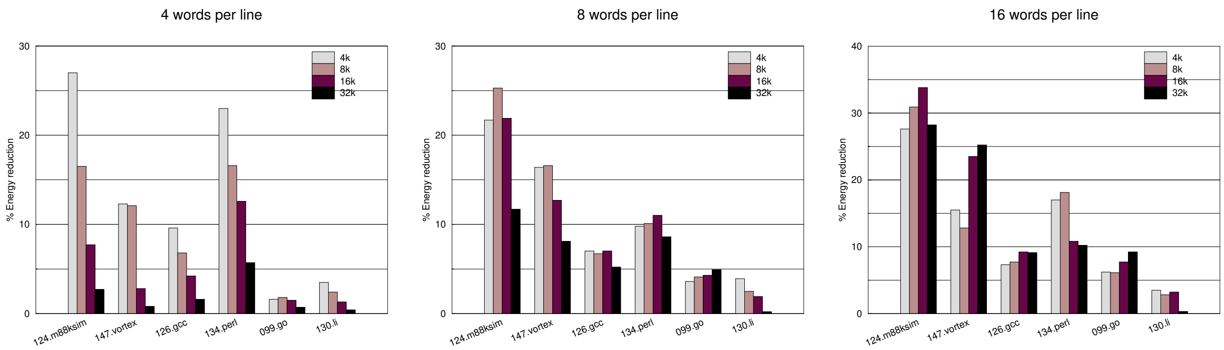

Figure 6. % Energy savings.

case they require more sophisticated compression encodings. Second, while the list of frequent values is kept fixed for the entire execution of the program in our approach, they dynamically adapt the encodings according to changing frequency distribution of data values. Therefore the two designs of data compression schemes represent two points in a spectrum of techniques that are possible by trading off generality with hardware and runtime costs.

Data cache. In [18] we proposed the *frequent value cache* (FVC) which is a type of a victim cache designed to exploit frequent value locality. FVC is a small direct-mapped cache which is dedicated to holding only frequent values. We conducted an experiment in which we measured the improvements in traffic and miss rates that can be obtained by a combination of CC and FVC. We considered line size of 8 words with CC and FVC configurations that exploit top four frequent values. The results show that the combination of CC and FVC can result in traffic reductions of 2-72% over a DMC and miss reductions of 3-69.5%. Therefore a combination of CC and FVC is highly effective. We are currently exploring the impact of compression on other power efficient cache organizations [1, 15, 14, 10, 5].

References

[1] D.H. Albonesi, "Selective Cache Ways: On Demand Cache Resource Allocation," *32nd Annual International Conference on Microarchitecture*, pages 248-259, 1999.

[2] B. Abali and H. Franke, "Operating System Support for Fast Hardware Compression of Main Memory Contents," *Workshop on Solving the Memory Wall Problem*, June 2000.

[3] C.D. Benveniste, P.A. Franaszek, and J.T. Robinson, "Cache-Memory Interfaces in Compressed Memory Systems," *Workshop on Solving the Memory Wall Problem*, June 2000.

[4] D. Brooks and M. Martonosi, "Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance," *Fifth International Symposium on High-Performance Computer Architecture*, Orlando, Florida, January 1999.

[5] K. Ghose, "Reducing Power in Superscalar Processor Caches using Subbanking, Multiple Line Buffers, and Bit Line Segmentation," *International Symposium on Low Power Electronics and Design*, pages 70-75, 1999.

[6] N.P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," *17th Annual International Symposium on Computer Architecture*, Seattle, pages 364-373, 1990.

[7] M. Kjelso, M. Gooch, and S. Jones, "Empirical Study of Memory-data: Characteristics and Compressibility," *IEE Computers and Digital Techniques*, Vol. 145, No. 1, pages 63-67, January 1998.

[8] J-S. Lee, W-K. Hong, and S-D. Kim, "Design and Evaluation of a Selective Compressed Memory System," *IEEE International Conference on Computer Design*, Austin, TX, pages 184-191, October 1999.

[9] C. Lefurgy, P. Bird, I.-C. Chen, and T. Mudge, "Improving Code Density Using Compression Techniques," *30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 194-203, 1997.

[10] J. Kin, M. Gupta, and W.H. Mangione-Smith, "The Filter Cache: An Energy Efficient Memory Structure," *30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 184-193, 1997.

[11] D. Kirovski, J. Kin, and W. H. Mangione-Smith, "Procedure Based Program Compression," *30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 204-217, 1997.

[12] S. Y. Larin, "Exploiting Program Redundancy to Improve Performance, Cost and Power Consumption in Embedded Systems," Ph.D. thesis, ECE Dept., North Carolina State Univ., Raleigh, North Carolina, August 2000.

[13] S. Onder and R. Gupta, "Automatic Generation of Microarchitecture Simulators," *IEEE International Conference on Computer Languages*, pages 80-89, Chicago, Illinois, May 1998.

[14] M.D. Powell, S-H. Yang, B. Falsafi, K. Roy, T.N. Vijaykumar, "Gated Vdd: A Circuit Technique to Reduce Leakage in Deep-submicron Cache Memories," *ACM/IEEE International Symposium on Low Power Electronics and Design*, 2000.

[15] P. Ranganathan, S. Adve, and N. Jouppi, "Reconfigurable Caches and their Application to Media Processing," *27th Annual International Symposium on Computer Architecture*, Vancouver, British Columbia, Canada, June 2000.

[16] M. Stephenson, J. Babb, and S. Amarasinghe, "Bitwidth Analysis with Application to Silicon Compilation," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Vancouver, British Columbia, Canada, June 2000.

[17] W. Ye, N. Vijaykrishnan, M. Kandemir, and M.J. Irwin, "The Design and Use of Simplepower: A Cycle-accurate Energy Estimation Tool," *37th Design Automation Conference*, Los Angeles, CA, June 2000.

[18] Y. Zhang, J. Yang, and R. Gupta, "Frequent Value Locality and Value-centric Data Cache Design," *The Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, November 2000.