View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by DSpace@MIT

A Dynamically Partitionable Compressed Cache

David Chen*, Enoch Peserico and Larry Rudolph, *SMA Fellow*
MIT Laboratory for Computer Science, 200 Technology Square, Cambridge, MA 02139
*Now at Avici Systems

***Abstract*— The effective size of an L2 cache can be increased by using a dictionary-based compression scheme. Naive application of this idea performs poorly since the data values in a cache greatly vary in their "compressibility." The novelty of this paper is a scheme that dynamically partitions the cache into sections of different compressibilities. While compression is often researched in the context of a large stream, in this work it is applied repeatedly on smaller cache-line sized blocks so as to preserve the random access requirement of a cache. When a cache-line is brought into the L2 cache or the cache-line is to be modified, the line is compressed using a dynamic, LZW dictionary. Depending on the compression, it is placed into the relevant partition.**

The partitioning is dynamic in that the ratio of space allocated to compressed and uncompressed varies depending on the actual performance, Certain SPEC-2000 benchmarks using a compressed L2 cache show an 80reduction in L2 miss-rate when compared to using an uncompressed L2 cache of the same area, taking into account all area overhead associated with the compression circuitry. For other SPEC-2000 benchmarks, the compressed cache performs as well as a traditional cache that is 4.3 times as large as the compressed cache in terms of hit rate, The adaptivity ensures that, in terms of miss rates, the compressed cache never performs worse than a traditional cache.

I. Introduction

One obvious way to increase the effective on-chip cache size is data compression. Although compression and decompression do require extra computation, the savings in latency and energy that can be reaped often justify the additional effort. This is especially true in the case of L2/L3 caches, where avoiding a miss means avoiding a (very expensive) access to off-chip main memory, and where larger size offers greater opportunities for compression.

Cache data compression presents several challenges. One would like to involve only few words at a time in compression/decompression operations (avoiding excessive computational overhead) but at the same time to leverage the redundancy present in the contents of the *whole* cache (maximizing the compression opportunities). Also, the logic for accessing compressed and uncompressed data is different, and since the "compressibility" of data can change drastically depending on the application, choosing to devote two distinct areas of the chip respectively to compressed and uncompressed data would almost always result in underutilization of one of the two, negating the benefits of compression. Finally, the fact that compressed data at the same time requires less "space" but more "time" requires careful consideration when choosing which data to compress, and which data to evict from the cache when a miss occurs.

We solve these problems in our "Partitioned Compressed Cache" design, or PCC for brevity. PCC uses dictionary-based rather than sliding-window compression (unlike e.g. [2]), to allow selective, independent decompression of any cache line. Entries in the dictionary contain common strings of up to the size of a cache line; cache lines can thus be compressed and decompressed on a line by line basis. PCC uses a "clock-scheme" that cycles over the compressed entries in the cache marking the corresponding dictionary entries as active while another "clock-scheme" cycles over the dictionary entries clearing out any inactive entries.

Cache entries can be compressed or not, and assuming that a cache line could be compressed down to $1/s$ of its uncompressed size, we allow each cache set to contain anywhere from n to sn entries. No matter what, sn address tags and LRU bits are maintained even though only n entries might be present in the set. When a new cache line is brought in or part of a cache line value is updated, PCC attempts to compress the line. A line is considered to be compressed if its compressed size is below a fixed threshold, e.g., a threshold of 16 bytes for a 32 byte cache line. Then, enough of the least recently used items are evicted from the cache so as to make room for the new item. In this way, the number of entries in a cache set varies depending on how compressible are the entries.

We develop a new metric to evaluate performance: how much larger a traditional cache which does not exploit data compression would have to be to match the performance of our PCC. For fairness, performance is measured not in terms of the abstract notion of "hit rate", but rather, in terms of the amount of computation that can be carried out in a given time interval - including the area and time overheads that the compression/decompression mechanism requires (our extimate is conservative in the sense that we do not take into account the fact that a smaller cache could probably sustain a faster clock cycle). Our PCC design can significantly increase the "apparent" size of the cache, in some cases up to 80%.

After reviewing related work in section 2, the rest of the paper is organized as follows. In section 3 we look at the PCC compression mechanism; in section 4 at how PCC manages the data present in the cache, deciding which data to keep in the cache and which data to compress; in section 5 we present an experimental evaluation of the PCC design; we conclude in section 6 summarizing our results and looking to avenues of future research.

II. Related Work

While compression has been used in a variety of applications, it has only recently started to be researched extensively in the area of processor caches. Previous research includes compressing bus traffic to use narrower buses, compressing

code for embedded systems to reduce memory requirements and power consumption, compressing file systems to save disk storage, and compressing virtual and main memory to reduce page faults.

Citron et al. [7] found an effective way to compact data and addresses to fit 32-bit values over a 16-bit bus, while Thumb [17], [12] and others [22], [15], [16] apply compression to instruction sets and binary executables. Douglis [9] proposed using a fast compression scheme [20] in a main memory system partition and shows several-fold speed improvement in some cases and substantial performance loss in others. Kjelso et. al [14] and Wilson et al. [21] consider an additional compressed level of memory hierarchy and found up to an order of magnitude speedup. IBM's MXT technology [18] uses the scheme developed by Benveniste et al. [3] with 256 byte sub-blocks, a 1KB compression granularity, combining of partially filled blocks, along with the LZ77-like parallel compression with shared dictionaries compression method.

Yang et al. [23], [24] explored compressing frequently occurring data values in focusing on direct-mapped L1 configurations and found that a large portion of cache data is made of only a few values, which they name Frequent Values. By storing data as small pointers to Frequent Values plus the remaining data, compression can be achieved. They propose a scheme where a cache line is compressed if at least half of its values are frequent values.

PCC is similar to Douglis's Compression Cache in its use of partitions to separate compressed and uncompressed data. A major difference is that Douglis's Compression Cache serves data to the higher level in the hierarchy only from the uncompressed partition, and so if the data requested is in the compressed partition, it is first moved to the uncompressed partition. The scheme developed by Benveniste et al. and the Frequent Value cache developed by Yang et al. serve data from both compressed and uncompressed representations as the PCC does, but both lack dynamic, adaptive partitioning.

III. The PCC Compression Algorithm

The dictionary compression scheme of PCC is based on the common Lempel-Ziv-Welch (LZW) compression technique [19]. When an entry is first placed in the cache or when an entry is modified, the dictionary is used to compress the cache line. The dictionary values are purged of useless entries by using a "clock-like" scheme over the compressed cache to mark all useful dictionary entries.

With LZW compression, the raw input stream data is compressed into another, shorter output stream of compressed symbols. Usually, the size of each uncompressed symbol, say of d bits, is smaller than the size of each compressed symbol, say of c bits. The dictionary initially consists of one entry for each uncompressed symbol. See Figure 1 for an example.

Compression works as follows. Find the longest prefix of the input stream that is in the dictionary and output the compressed symbol that corresponds to this dictionary entry. Extend the prefix string by the next input symbol and add it to the dictionary. The dictionary may either stop changing or it may be cleared of all entries when it becomes full. The prefix is removed from the input stream and the process continues.

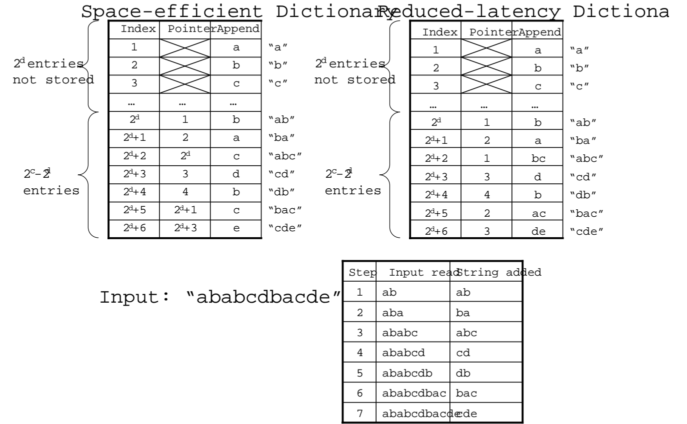

Fig. 1. The space-efficient dictionary stores only one uncompressed symbol per entry, while the reduced-latency dictionary stores the entire string. The table at the lower half of the figure shows the order in which entries are added to the initially empty dictionaries.

Unlike LZW, PCC compresses only a cache line's worth of data at a time. A space-efficient dictionary representation maintains a table of 2^c entries, each of which contains two values: a compressed symbol that points to some other dictionary entry and an uncompressed symbol, for a total of $c + d$ bits per entry. The uncompressed symbols need not be explicitly stored in the dictionary as the first 2^d values represent themselves. Given a table entry, the corresponding string is the concatenation of the second value to the end of the string pointed to by the first value.

To compress a cache line, find the longest matching string, then output its dictionary symbol. Repeat until the entire line has been compressed. Decompression is much faster than compression. Each compressed symbol indexes into the dictionary to provide an uncompressed string. For a line containing n_c compressed symbols, $s_l/d - n_c$ table lookups are needed for decompression. The decompression latency can be improved by increasing the dictionary size and by parallelizing the table lookups. Naturally, increasing the compressed symbol size c while keeping the uncompressed symbol size d constant will increase the size of the associated table and enable more strings to be stored.

One way to purge dictionary entries is by maintaining reference counts for each entry updated whenever a compressed cache line is evicted or replaced. PCC uses a more efficient method to purge entries sweeping through the contents of the cache slowly, using a clock scheme with two sets of flags. Each of the two sets has one flag per dictionary entry, and the status of the flag corresponds to whether or not the dictionary entry is used in the cache. If there is a flag in either set, it is assumed that the entry is being referenced. Compression or decompression also cause the appropriate dictionary entries flag to be set. A second process sweeps through the dictionary purging entries. Alternatively, one could keep track of the most frequently used symbols in a special "cache" - whenever one of these symbols appears to enjoy a sufficiently high utilization

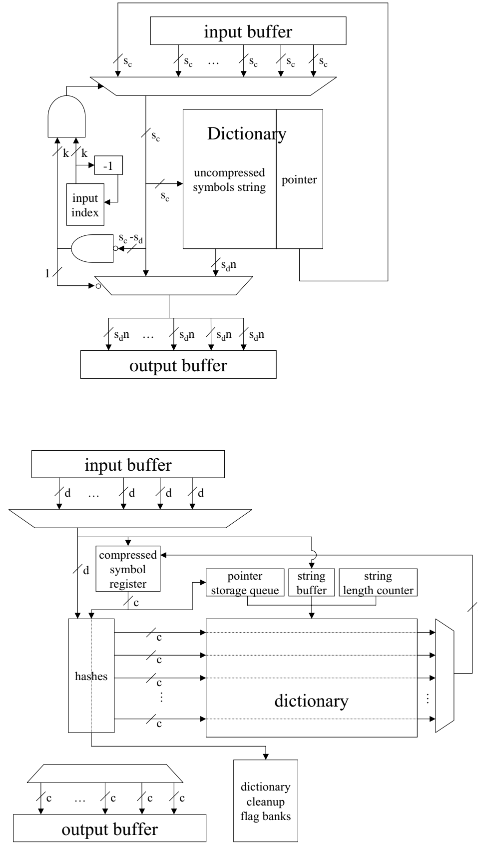

Fig. 2. Compression and Decompression Logic: note that the design shown here does not include parallel compression or decompression, and therefore exhibits longer latencies for larger compression partition line sizes. n represents the number of uncompressed symbols stored in each dictionary entry.

compared to the least utilized symbol in the dictionary, one could replace the latter with the former in the dictionary, while at the same either decompressing or simply invalidating all the correspoding entries in the cache. While we are currently investigating this second approach, the additional complexity involved does not seem to be balanced by a sufficiently large increase in performance, especially when larger dictionaries are involved.

Compression time can be reduced dramatically by searching through only a strict subset of the entire dictionary for each uncompressed symbol of the input. A hash of the input is used to determine which entries to examine. If the dictionary is stored in multiple banks of memory, choosing hash functions such that entries are picked to be in separate banks allows these lookups to be done in parallel. Alternatively, content addressable memory (CAM) can be used to search all entries at the same time, reducing the number of dictionary accesses to the number of repetitions needed, or s_l/d accesses.

Decompression and compression logic is illustrated in Figure 2.

Decompression and compression can each be done in parallel to reduce their latency. To do so effectively, a method of performing multiple dictionary lookups in parallel is needed. One solution is to increase the number of ports to the dictionary. Another possibility is to keep several dictionaries, each with the same information. This provides a reduction in latency at the expense of the increased area needed for each additional dictionary. While decompressing, there are multiple compressed symbols which need to be decompressed. Since these symbols are independent of one another, they can be decompressed in parallel.

In practice, parallelizing the decompression process may not actually reduce latency significantly. The experiments in this work show that performance is best when dictionary sizes are such that only one or two lookups are needed per compressed symbol. This is largely due to the low cost of increasing dictionary size in comparison to the benefits of decreasing the number of lookups.

IV. PCC Management

Since not all data values are compressible nor are they all accessed at the same rate, an adaptive scheme can make the best use of the limited resources. An examination of the data values occurring in six benchmarks clearly show this variability, see Figure 3. For all unique cache lines accessed during the execution, the figure shows a histogram of the number of data values as a function of their compressibility. What is important to note is the variability between the benchmarks. Moreover, the overlaying curves show the usefulness of data; that is how often the data is referenced.

It is possible to statically partition the cache into compressed and uncompressed sections. Experiments reported elsewhere show that each benchmark requires a different partitioning to get performance improvements. Not only do data values differ in how much they can be compressed and how often they are accessed, cache sets have different access patterns. Some sets are “hot” experiencing a large number of accesses and some sets experience “thrashing” indicating they do not have enough capacity.

PCC uses the same basic storage allocation scheme of a traditional cache, i.e., a set of address tags and a set of data values. The number of address tags (and comparators) is fixed, but the number of cache lines in the set varies. For purposes of exposition, assume that two compressed entries require about the same number of bits as a single uncompressed entry; modifications to other ratios are straightforward but require more complicated circuitry. Let n be the number of normal entries, that is the data store for a set is $32n$ bytes. This space could store $2n$ compressed entries. In general $2i + j = 2n$ entries can be stored where j entries are compressed. Associated with each 32 byte field is one bit indicating whether there is one uncompressed or two compressed items present. The cache has $2n$ tags and maintains LRU information on all $2n$ entries, even though there may only be n entries actually in the cache.

On a cache hit, the cache line is either fetched directly from the data table or it must first be decompressed. Decompression is done via the dictionary as outlined in the previous section. Since compressed items enable a larger number of items to be

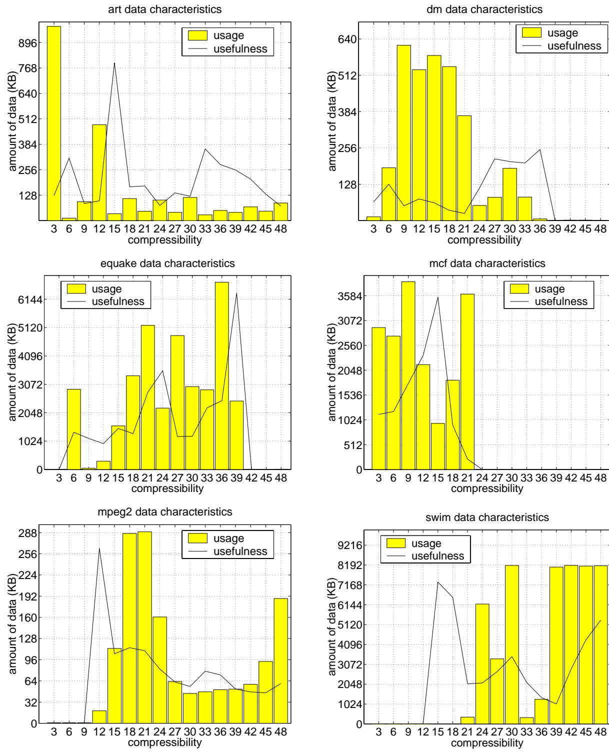

Fig. 3. The histograms indicate the amount of data available at different levels of compressibility. The x-axis gives the size of the compressed line in bytes. The y-axis gives the amount of data in kilobytes, covering all unique memory addresses accessed during the simulation (an infinite sized cache). The top two histograms show that most data values are highly compressible, while the bottom-most right histogram shows that many data values would require more than 39 bytes to store a 32 byte cache line if compressed. The overlaying curves show the usefulness of data at different levels of compressibility. The y-axis gives the probability that a hit is on a particular cache line in the corresponding partition. This y value is equivalent to taking the total number of hits to the corresponding partition and dividing by the number of cache lines in that partition as given by the bars.

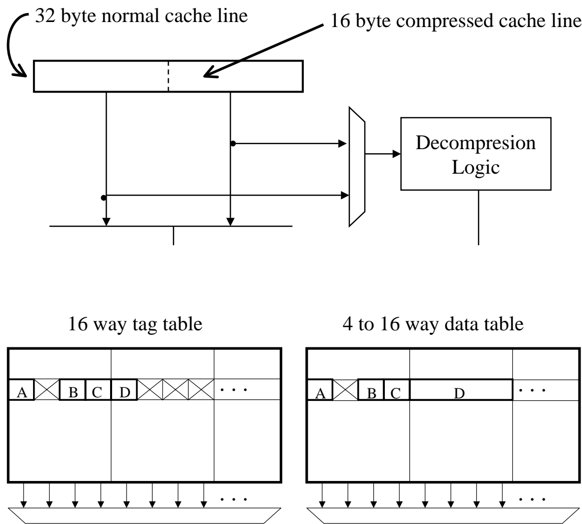

Fig. 4. A sample configuration in which cache lines are assumed to be compressible to 1/2 their size, e.g. a 32 byte line compressed requires only 16 bytes. A cache set has tags and LRU bits for 16 ways, but as few as 8 or as many as 16 actually entries.

	LRU Entry: Compressed		Normal
Compressed	Replace		expand
Normal	If LRU_2 cmprssed merge & replace	If LRU_2 normal replace LRU_2	Replace

TABLE I
REPLACEMENT ALGORITHM

stored in the cache, the decompression overhead is offset by the benefit of a smaller number of main memory accesses.

A. The replacement strategy

The most interesting situation happens during a cache miss - which is a comparatively rare event for L2/L3 caches. When a cache miss occurs, the newly retrieved item will replace either the LRU item except sometimes it may replace two items. When the new cache line is fetched from main memory, an attempt is made to compress it. If the size of the compressed line is below a threshold, e.g., 16 bytes, then it is considered *compressible* otherwise it is *incompressible*. It must then be stored into the data table.

PPC Replacement Scheme: There are several cases to consider when inserting the cache line into the data table, see Table I. The easiest case is when the new item and the LRU item have the same status – either both compressible or both incompressible – then the new item simply replaces the evicted LRU item. When the new item is compressible and the LRU item isn't, then the LRU item is evicted and its storage is converted to two compressible entries, one that is left empty (and marked as LRU). The case when the new item is incompressible has two subcases. If the second most LRU item is incompressible, then it is the only one that is evicted and replaced; the LRU item remains in the cache.If the two most LRU items are compressible, then they are both evicted and their space is converted to a normal entry.

Note that in this last case there is no need for the two "halves" of what will become the new uncompressed entry to be adjacent. Since we have tag bits for each half - which will be scanned in parallel anyway - we can simply break up the line in cache, and reassemble it when it is read with the same circuitry that extracts the line from the line set where it is stored. In fact, through this mechanism we could even save the additional "compressed/uncompressed bit": the fact that a line is compressed or uncompressed is indicated by the presence of respectively one or two tags corresponding to it.

Updating the value of a part of a cache line can obviously change its compressibility, potentially requiring allowing compression or requiring an "expansion" of the line. However, compressibility of a line can also change as a result of *another* line being modified, since this might cause an update in the

dictionary. It is important to note, though, that such an update can only increase, and not decrease the compressibility, since an entry in use will never be purged from the dictionary. The worst case scenario is then that a line will remain uncompressed in cache even when it could be compressed.

In terms of cache hits, PCC always performs at least as well as a comparable standard cache. This is easy to see when one realizes that the LRU element in any set of a PCC cache was used no more recently than the LRU element in the corresponding set of a standard cache. The contents of a PCC cache are then a superset of those of a traditional cache of the same size, guaranteeing that, in the worst possible case, the only additional cost of employing a PCC will be the (small) cost, in terms of area and latency, of the compression/decompression mechanism.

B. A latency-sensitive replacement strategy

While the above replacement strategy works well in improving hit rates, it does not account for the fact that a cache hit to a compressed item has longer latency than a hit to an uncompressed item.

It is possible to modify the strategy in a way that may slightly decrease the hit ratio (but never below that of a traditional cache) while also significantly decreasing the number of times an item is decompressed.

In a level one (L1) cache, the most recently used item in a cache set usually experiences the most accesses. The second most recently used item in the set experiences the second most accesses, and so on. When the L1 cache is too small for the application, the level two (L2) cache behaves in a similar fashion. In other cases, accesses to the L2 cache are fairly random as to which element is accessed in a set. In such situations, it is helpful if the number of items in the set is large. In other words, the behavior of an L2 cache differs among applications and the behavior of each set in the L2 cache differs.

When the L2 behaves like an L1 cache, the MRU item will be frequently accessed. In such a case, it would be better if that item was stored in its uncompressed form. On the other hand, when items are accessed uniformly, it is better for the items to be compressed.

We modify the replacement scheme as follows.

Latency-sensitive Replacement Scheme: Whenever an MRU item is accessed and it is compressed, then it is replaced in the cache in its uncompressed form. This may cause the LRU item to be evicted. Whenever an LRU item is to be evicted from the cache and it is not compressed, PCC attempts to keep the item in the cache by compressing two other items in the set. The items of set are scanned from LRU to MRU order and for each item that is not compressed, an attempt is made to compress it. If two items are found, then they are compressed and the LRU item remains in the cache.

This approach can be generalized as follows: whenever a compressed item is accessed for the mth time after the last access to the LRU item, uncompress that (frequently used) item. High values of m will lead to a cache which will aggressively compress everything, and are more reasonable in the case of a large gap between the additional cost of decompression and that of a miss, while small values of m (for example 2, as in the case described above) will lead to a cache which behaves more like a traditional cache, and compresses only items that are not used that frequently, an approach that makes more sense when the gap between the additional cost of decompression and that of a miss is not as large. A reasonable compromise seem to set m equal to the ratio between the two costs. Note that, for all values of m, a PCC will not suffer more misses than a traditional cache.

V. The Performance of PCC

We use simulation to evaluate the effectiveness of the PCC. Simulation is done using a hand-written cache simulator whose input consists of a trace of memory accesses. A trace of memory accesses is generated by the Simplescalar simulator[4], which has been modified to dump a trace of memory accesses in a PDATS[11] formatted file. Applications are compiled with gcc or F90 with full optimization for the Alpha instruction set and then simulated with Simplescalar. The benchmark applications are from the SPEC2000 benchmark suite and simulated for 30 to 50 million memory references.

The L1 cache is 16KB, 4 way set associative, with a 32 byte line size, and uses write-back. The L2 cache is simulated with varying size and associativity, with a 32 byte line size, and write-allocate (also known as fetch on write). We assume an uncompressed input symbol size d of 8 bits, and a compressed output symbol size c of 12 bits. The dictionary stores 16 uncompressed symbols per entry, making the size of the dictionary $(2^c - 2^d)(d * 16 + c)$, which evaluates to 537,600 bits, or 67,200 bytes.

A. Metrics

The common metric for the performance of a compression algorithm is to compare the sizes of the compressed and uncompressed data, i.e., the compression ratio [2], and for a cache is the miss rate reduction metric. However, the two configurations are not easily comparable as the partitioned cache uses more tags and comparators per area while at the same time using much less space to store data than the traditional cache.

We introduce the interpolated miss rate equivalent caches (IMRECs) metric that indicates the effective size of the cache. That is, how large must a standard cache be to have the same performance of a PCC cache (including the additional cost of tags and dictionary). We wish to maximize the IMREC value; a value above 1 means that the PCC cache behaves like a larger-sized standard cache. For a given PCC configuration and miss rate, there is usually no naturally corresponding cache size with the same miss rate. Consequently, we interpolate linearly to calculate a fractional cache size. Our sample points are chosen by picking the size of a cache way, and then increasing the number of ways.

The size of a standard cache is the total number of cache lines multiplied by the cache line size. The size of a PCC cache includes the size of the dictionary, the additional address tag bits and additional status bits.

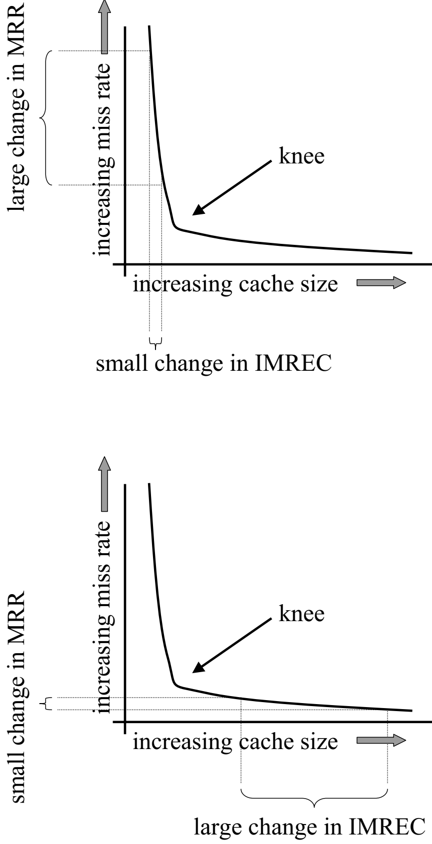

Fig. 5. To the left of the knee, small increases in IMREC ratio correspond to large increases in MRR. To the right of the knee, small increases in MRR correspond to large increases in IMREC ratio.

Let $M(C_j)$ be the miss rate of a j-way standard cache and $S(C)$ the size of cache C, the IMREC ratio is as follows:

$$\text{IMREC ratio} = S(C_i) + \frac{(S(C_{i+1})-S(C_i))(M(C_i)-M(PCC))}{M(C_i)-M(C_{i+1})}$$
$$\textit{when } M(C_i) >= M(PCC) \textit{ and } M(C_{i+1}) < M(PCC)$$

Another metric is the miss rate reduction (MRR), or the percent reduction in miss rate. But once again, we linearly interpolate to get the miss rate for an equivalently sized standard cache.

$$\text{Percent Miss Rate Reduction}$$
$$= \left(MR(C_i) - \frac{(MR(C_i)-MR(C_{i+1}))(S(PCC)-S(C_i))}{S(C_{i+1})-S(C_i)}\right) \times 100\%$$
$$\textit{when } S(C_i) <= S(PCC) \textit{ and } S(C_{i+1}) > S(PCC)$$

It is important to understand what can causes large swings in IMREC ratio and MRR. Figure 5 shows typical curves of miss rate versus cache size. Miss rate curves typically have a prominent knee where miss rate decreases rapidly until the knee and then very slowly afterwards. The graph to the right shows that to the right of the knee, a small increase in MRR corresponds to a large increase in IMREC ratio. The graph to the left shows that to the left of the knee, a small increase in IMREC ratio corresponds to a large increase in MRR. While it may seem that the small miss rate improvements gained when to the right of the knee are unimportant, applications operating to the left of the knee are likely to be performing so badly that the issue of whether to use a PCC is not a primary concern. Thus most situations of interest occur to the right of the knee, where large IMREC ratios indicate that a PCC provides the same performance gains as a large cache but with much less hardware.

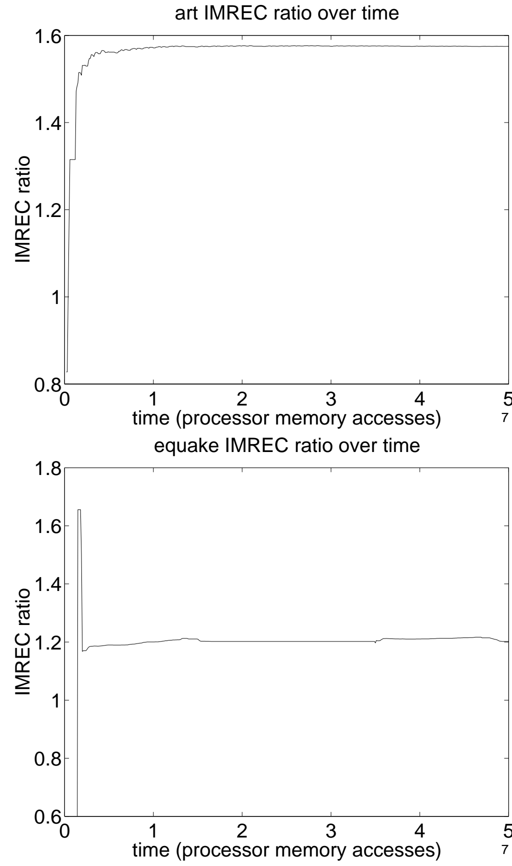

Fig. 6. We plot the `art` and `equake` IMREC ratios over time so as to know how long to simulate before recording results.

In our simulations, we account for the latency incured by the decompression.

Let L_1 be the number of hits to L1, H_u be the number of hits in PCC to items that are not compressed, H_c be the number of hits to items that are compressed, and M_{pcc} be the number of L2 misses. Then, the time to access memory with a PCC is

$MA_{pcc} = C_1L_1 + C_2H_u + C_3H_c + C_4M_{pcc}$

For a standard cache, the number of L1 hits are the same, but the number of L2 hits and L2 misses differ. The time to access memory with a standard cache is:

$MA_{sc} = C_1L_1 + C_2L_2 + C_4M_{sc}$

where $C_1, C_2, C_3, and C_4$ are the times to access L1, L2, L2 compressed, and DRAM.

We define the *average L2 access time quotient* as,

$ATQ_{PCC} = \frac{MA_{sc}}{MA_{PCC}}$

Finally we must make sure that our simulations have run

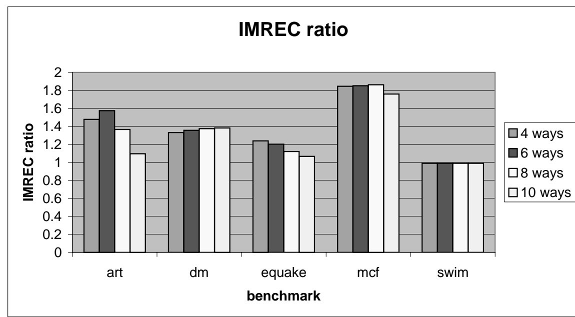

Fig. 7. The IMREC values for five benchmarks show that PCC clearly improves the performance. PCC can sometimes perform like a cache 80 percent larger. It never performs worse that the same sized standard cache.

for long enough that the values presented are representative of the benchmark. We do this by plotting IMREC ratios over time. While some benchmarks like `mcf` clearly reach steady state quickly, others, like `equake`, have more varied behavior and take longer, as shown in Figure 6.

B. Results of the simulations

Although we ran many simulations over the large space of configurations, we present only one slice. Varying the dictionary size, the compressibility threshold (e.g. requiring a line to be 8 bytes or less before we consider it to be compressible), and many others result in too many graphs. We present what we believe to be a reasonable configuration.

The IMREC values for five benchmarks, Figure 7, show a performance improvment. PCC can sometimes perform like a cache 80 percent larger. It never performs worse that the same sized standard cache. Note that since IMREC takes into account the extra storage allocated to the dictionary, when PCC contains no compressed values, it will be considered inferior to an equally sized cache.

When we take into account the latency to decompress a cache line, the results are less impressive, sometimes showing that PCC is slower than a standard cache, Figure 8.

The latency-aware replacement scheme indeed reduces the overhead of decompressing without significantly reducing the number of hits. Figure 9 shows that the performance in terms of hits does not change very much, but the latencies numbers are much better, Figure 10. Note that access time quotients just compare the times and since the standard cache may access memory more frequently, PCC can do much better.

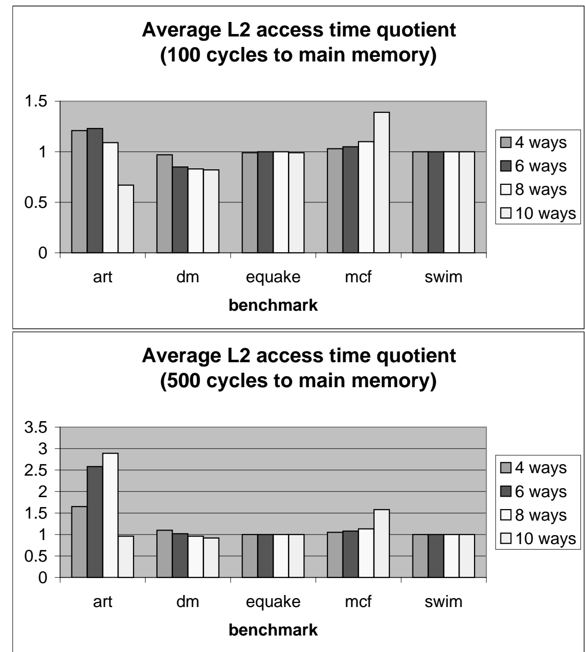

Fig. 8. The ATQ values for five benchmarks show that even when the latency of decompressing cache items, PCC still can improve the performance although not for all applications. The graphs assume 100 or 500 cycle time to main memory.

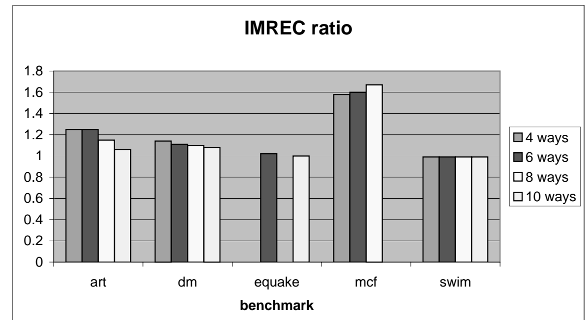

Fig. 9. The IMREC values for the modified PCC replacement strategy that tries to keep the MRU item decompressed.

VI. Conclusion

Compression can be added to caches to improve capacity - although one has to address several interesting issues for it to be effective. A dictionary-based compression scheme allows for reasonable compression and decompression latencies and good compression ratios. Keeping the data in the dictionary from becoming stale can be avoided with a clock scheme.

Various techniques can be used to reduce the latency involved in the compression and decompression process. Searching only part of the dictionary during compression, using multiple banks or CAMs to examine multiple dictionary entries simultaneously, and compressing a cache line starting at different points in parallel can reduce compression latency. Decompression latency can be reduced by storing more symbols per dictionary entry and decompressing multiple symbols in parallel. There are many different compression schemes some of which may perform better or be easier to implement in hardware.

The benefits of having a cache design like our PCC have not yet been fully explored. For example, CRCs of the cache data can be done for only a small incremental cost, an idea which is proposed also in [18]. The partitioning based on compressibility may also naturally improve the performance of a processor running multiple jobs, some of which are streaming applications. The streaming data is likely to be hard to compress, and can therefore automatically be placed into its own partition separate from non-streaming data.

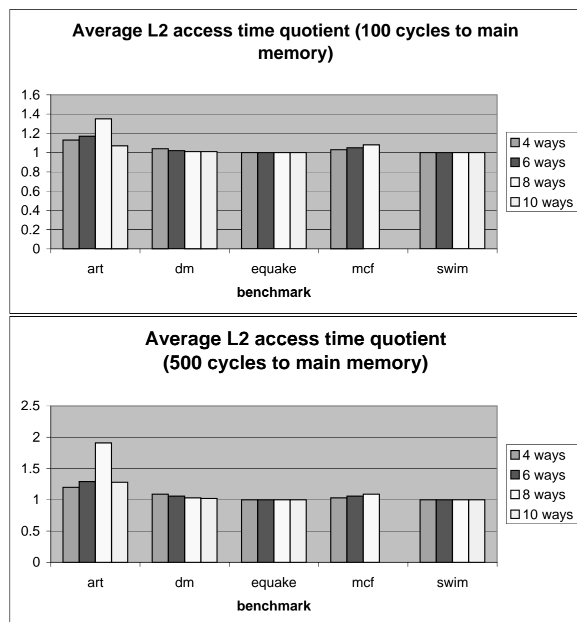

Fig. 10. The ATQ values corresponding to Figure 9.

ACKNOWLEDGMENTS*

The authors would like to thank the Malleable Cache team: Derek Chiou, Josh Jacobs Prabhat Jain,Peter Portante and Ed Shu, for many helpful discussions. We would like to thank Srini Devadas for in depth technical help. A part of this work was funded by Defense Advanced Research Projects Agency under the Air Force Research Lab contract F30602-99-2-0511, titled “Malleable Caches for Data-Intensive Computing”.

REFERENCES

[1] Data Compression Conference.

[2] B. Abali, H. Franke, X. Shen, D. Poff, and T. B. Smith. Performance of hardware compressed main memory, 2001.

[3] C. Benveniste, P. Franaszek, and J. Robinson. Cache-memory interfaces in compressed memory systems. In *IEEE Transactions on Computers, Volume #50 number 11*, November 2001.

[4] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. In *Technical report, University of Wisconsin-Madison Computer Science Department*, 1997.

[5] Miachael Burrows, Charles Jerian, Butler Lampson, and Timothy Mann. On-line data compression in a log-structured file system. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, pages 2–9, October 1992.

[6] D. T. Chiou. *Extending the Reach of Microprocessors: Column and Curious Caching*. PhD thesis, Massachusetts Institute of Technology, 1999.

[7] Daniel Citron and Larry Rudolph. Creating a wider bus using caching techniques. In *Proceedings of the First International Symposium on High-Performance Computer Architecture*, pages 90–99, Raleigh, North Carolina, 1995.

[8] M. Clark and S. Rago. The Desktop File System. In *Proceedings of the USENIX Summer 1994 Technical Conference*, pages 113–124, Boston, Massachusetts, 6–10 1994.

[9] Fred Douglis. The compression cache: Using on-line compression to extend physical memory. In *Proceedings of 1993 Winter USENIX Conference*, pages 519–529, San Diego, California, 1993.

[10] Peter A. Franaszek, John T. Robinson, and Joy Thomas. Parallel compression with cooperative dictionary construction. In *Data Compression Conference*, pages 200–209, 1996.

[11] E. E. Johnson and J. Ha. PDATS: Lossless addresss trace compression for reducing file size and access time. In *IEEE International Phoenix Conference on Computers and Communications*, 1994.

[12] Kevin D. Kissell. MIPS16: High-density MIPS for the embedded market. In *Proceedings of Real Time Systems ’97 (RTS97)*, 1997.

[13] M. Kjelso, M. Gooch, and S. Jones. Design and performance of a main memory hardware data compressor. In *Proceedings of the 22nd Euromicro Conference*, pages 423–430, September 1996.

[14] M. Kjelso, M. Gooch, and S. Jones. Performance evaluation of computer architectures with main memory data compression, 1999.

[15] Charles Lefurgy, Peter Bird, I-Cheng Chen, and Trevor Mudge. Improving code density using compression techniques. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 194–203, Research Triangle Park, North Carolina, December 1997.

[16] S. Liao. *Code Generation and Optimization for Embedded Digital Signal Processors*. PhD thesis, Massachusetts Institute of Technology, June 1996.

[17] Simon Segars, Keith Clarke, and Liam Goudge. Embedded control problems, Thumb, and the ARM7TDMI. *IEEE Micro*, 15(5):22–30, 1995.

[18] R. B. Tremaine, P. A. Franaszek, J. T. Robinson, C. O. Schulz, T. B. Smith, M. Wazlowski, and P. M. Bland. IBM Memory Expansion Technology (MXT). In *IBM Journal of Research and Development vol. 45, No. 2*, pages 271–285, March 2001.

[19] T. Welch. High speed data compression and decompression apparatus and method, US Patent 4,558,302, December 1985.

[20] Ross N. Williams. An extremely fast Ziv-Lempel compression algorithm. In *Data Compression Conference*, pages 362–371, April 1991.

[21] Paul R. Wilson, Scott F. Kaplan, and Yannis Smaragdakis. The case for compressed caching in virtual memory systems. In *Proceedings of 1999 Summer USENIX Conference*, pages 101–116, Monterey, California, 1999.

[22] A. Wolfe and A. Chanin. Executing compressed programs on an embedded RISC architecture. In *Proceedings of the 25th International Symposium on Microarchitecture*, Portland, Oregon, December 1992.

[23] J. Yang, Y. Zhang, and R. Gupta. Frequent value compression in data caches. In *33rd International Symposium on Microarchitecture*, Monterey, CA, December 2000.

[24] Y. Zhang, J. Yang, and R. Gupta. Frequent value locality and value-centric data cache design. In *The 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, November 2000.