Dynamic Dictionary-Based Data Compression for Level-1 Caches

Georgios Keramidas, Konstantinos Aisopos, Stefanos Kaxiras

Department of Electrical and Computer Engineering, University of Patras,
26500 Patras, Greece
Tel.: +302610 996441, Fax.: + 302610997 333
{keramidas,aisopos,kaxiras}@ee.upatras.gr

Abstract. Data cache compression is actively studied as a venue to make better use of on-chip transistors, increase apparent capacity of caches, and hide the long memory latencies. While several techniques have been proposed for L2 compression, L1 compression is an elusive goal. This is due to L1's sensitivity to latency and the inability to create compression schemes that are both *fast* and *adaptable* to program behavior, i.e. dynamic. In this paper, we propose the first dynamic dictionary-based compression mechanism for L1 data caches. Our design solves the problem of keeping the compressed contents of the cache and the dictionary entries consistent, using a timekeeping decay technique. A dynamic compression dictionary adapts to program behavior without the need of profiling techniques and/or training phases. We compare our approach to previously proposed static dictionary techniques and we show that we surpass them in terms of power, hit ratio and energy delay product.

1 Introduction

Cache compression increases the apparent capacity of the cache and reduces the miss-rate at the expense of increased access latency associated with compression and decompression. As long as the cost of accessing a compressed datum in the cache does not exceed the cost of servicing a miss from the lower level of the memory hierarchy, cache compression is a wining proposition. This is easily attainable in the L2 or L3 caches where the compression/decompression costs compare favourably (i.e. are much lower) to the cost of going to main memory [1,11,14,15]. The same is true for main memory with respect to the long disc delays and this is why many proposals consider main memory compression [6].

But why would one consider compression in an age of excessively large transistor budgets? The reason is that it almost always pays to have a "bigger" cache: Alameldeen and Wood [1] show that in commercial applications where the miss rate is relatively high, compression of the L2, regardless of its size, is beneficial. Furthermore, we are fast moving towards multiple cores on a chip, which will exacerbate the problem of adequate cache capacity since the cache will have to be shared by many applications or threads. L2/L3 compression is therefore a useful technique for the foreseeable future and especially for future CMP architectures. Alternatively, compression can be used to free space in the cache which can then be translated to power savings [13,22,23,25].

Of course this reasoning can be extended to include the L1 if it were not for a serious issue. L1 is typically very fast —single-digit number of cycles when pipelined— and therefore latency-sensitive: a few additional cycles for compression/decompression can more than double its latency and hurt system performance. In addition, at the top level of the hierarchy, compression/decompression costs of the order of 10 to 15 cycles approach L2 access latencies, which means that accessing a compressed L1 datum (that would be a miss) has little difference from going directly to the L2. Thus, latency alone rules out all complex compression/decompression schemes such as those used in L2 compression [1,11,14,15]. Power consumption is also becoming a critical issue in L1. The L1 is accessed much more frequently than the L2, rendering complex and power-hungry L1 compression mechanisms undesirable.

This leaves at our disposal only the simplest mechanisms for compression. One such simple, yet effective, mechanism is the *dictionary* (or *directory*) for frequent values [20,23,25,26]. A frequent-value dictionary stores the program's frequent values (e.g., the 32-bit value "0") and replaces all their occurrences in the cache with the respective dictionary indices (e.g., 8-bit indices for a 256-entry dictionary). However, until now, no mechanism has been proposed to implement a dynamic dictionary for caches [23,24]; in other words, a dictionary whose contents can adapt to the requirements of the running program. The dictionaries proposed so far for caches are loaded statically via profiling or are "trained" for a small period of time to detect and store frequent values for the remainder of a program's run [23,25,26]. In practice, such a static approach is avoided by designers since it is cumbersome in real systems. There, we would like adaptivity under different workloads without needing to resort to profiling or training. Moreover, a static approach is incompatible with multiprogrammed/multithreaded environments since the contents of the dictionary are part of program state and need to be changed accordingly with context switches. The need for dynamic dictonaries was also reported by Yang and Gupta [24] in the context of bus compression.

On the other hand, a straightforward dynamic dictionary is impractical. The difficulty in building a dynamic dictionary lies in that we cannot delete an entry from the dictionary unless we are certain that no cache line is compressed with it —otherwise the line cannot be decompressed with the correct dictionary index leading to consistency problems. It is far too expensive, unfortunately, to keep track of the all the cache lines that are compressed with any particular entry in the dictionary.

Contribution. The contribution of our work is the first mechanism for a dynamic L1 dictionary resulting from coupling a decay cache [12] to a decaying dictionary. The principle of cache decay is the identification of cache lines which are unlikely to be accessed in the future (before their replacement). Such lines can be safely discarded with minimal impact on performance. We apply the same principal to the dictionary and discard entries which are unlikely to be used in the future. By decaying the cache and the dictionary in exactly the same way, we are guaranteeing that when a dictionary entry is decayed no live line in the cache can possibly refer to this entry. This allows us to replace dead entries in the dictionary with new frequent values, thus adapting the dictionary contents to the requirements of the running programs.

In this paper, we exploit this mechanism for two compressed L1 schemes: i) a low-power, and ii) a high-performance cache. We study the proposed mechanism and we present our comparisons with static dictionaries (using either profiling or training) and with uncompressed caches of larger capacity. Our evaluation shows that our cache compression design can improve the Energy-Delay Product by 10% (on average) compared to the static and the training approaches. When high performance is the issue, our proposal shows 45% reduction in miss ratio compared to a conventional cache of the same capacity and up to 27% improvement compared to the static case.

Structure of this paper. We begin by motivating the need for dynamic dictionaries dictated by the behavior of frequent values in Section 2. In Section 3 we show how cache decay leads to a solution for dynamic dictionaries and describe our proposal in detail while in Section 4 we delve into design issues for our approach. We continue in Section 5 by presenting the evaluation of our proposal. In Section 6 we survey related work and in Section 7 we offer our conclusions.

2 The Dynamic Behavior of Frequent Values

Many algorithms for compression of the memory subsystem have been proposed in the literature. Such techniques try to exploit different characteristics of the address/data streams to achieve high compression ratios [1,11,14,15]. One direction concentrates on exploiting the well known phenomenon of locality and especially value locality. Value locality has been initially utilized in the design of value reuse and value prediction mechanisms for superscalar processors [8,16].

The main motivation of this work is *frequent value locality* introduced by Zhang et al. [26]. Zhang et al. showed that such locality is quite prevalent in programs. They applied their observations in the design of L1 cache compression schemes and bus encoding schemes. These approaches are based on a small number of distinct values, that are very frequently accessed and are found by profiling the application or by training the dictionary during a small initial phase of the program. These approaches are limited by the static nature of the dictionaries: only a small fraction of the frequent values are accommodated and this is not optimal for all program execution. Excluding a small set of values (e.g., 0, 1, -1) that are universally useful, other values which are frequent in one part of the program may not occur as frequently in other parts and vice versa. A dictionary whose content changes dynamically, not only frees the designers from the burden of initializing it properly, but has the potential for better performance and lower power.

To show the need for a dynamic dictionary we conduct the following experiment. We choose two benchmarks from the SPEC2000 suite, one from the integer suite — *vpr*— and one from the floating point suite —*galgel*. For each program, we divide its execution into smaller time intervals and for each of these intervals we find its top N frequent values. We then examine the commonality between each interval's set of N frequent values and the fixed set of N frequent values of a static dictionary (i.e., the set of N for the whole program).

The results of this experiment are presented in Fig. 1. In both graphs, the horizontal axis represents the number of time intervals that fit in the execution of the program (200 to 2), and the vertical axis represents the overlap (as a percentage of values) between the dynamically created dictionary and a fixed dictionary. The four curves in each graph plot the overlap of the dynamically created dictionaries of various sizes (32, 64, 128, and 256) to a "static" dictionary of equal size for the whole program run. For both programs, the more frequent is the creation of dictionaries (smaller time interval), the smaller is the overlap with the "static" dictionary. This signifies the need to change the contents of the dictionary continuously. For *vpr* dictionary overlap is least with the smallest time intervals but reaches 100% when the dynamic directories are created less frequently (showing *vpr*'s highly dynamic nature at small time scales).

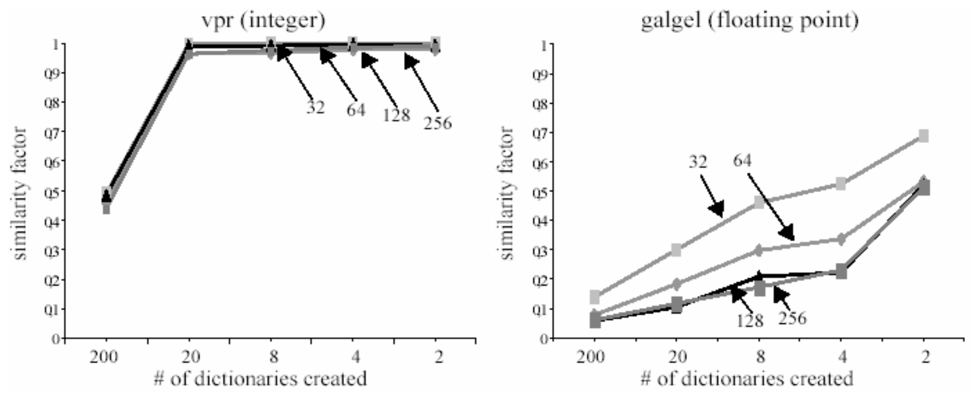

Fig. 1. Overlap of dynamically created dictionaries vs. static dictionary for various sizes

Galgel on the other hand shows a more gradual change in the overlap at various time scales and even in the far right case of only two dynamically created dictionaries, their overlap with a single dynamic barely reaches 68%. With respect to dictionary size, smaller dictionaries have more overlap (because the topmost frequent values do tend to be the same) while larger dictionaries have more room to accommodate a more diverse set of values. At the largest time scales the overlap for the large dictionaries converges to about 50% for *galgel*, meaning that a full half of the dynamic dictionaries is different than the corresponding static dictionary —of course, *the absolute number of values that differ from the static dictionary is a function of size.*

Having described the need to create a dictionary whose context must be able to change on the fly, let us now discuss our proposal for keeping the dictionary and the cache context consistent. The next section presents the first mechanism —to the best of our knowledge— for a dynamic dictionary for cache compression.

3 Dynamic Dictionary and Compressed Data Consistency

As of yet, no mechanism has been proposed to implement a dynamic dictionary for caches; instead the dictionaries proposed so far are loaded statically via profiling or are "trained" for a small period of time but remain static once they are loaded with values [1,14,23,25,26]. Dynamic (adaptive) dictonaries are reported in the context of *bus compression* [2,7,17,19,20,24]. What makes dynamic dictionaries possible for bus compression is that there is no need to keep them consistent with any other state. Data are compressed on the fly as they enter the bus and are decompressed as they are delivered at the other end —there is no storage of compressed state to worry about. But, this is the main impediment for dynamic dictionaries when it comes to cache compression.

The problem of dynamic dictionaries for caches is that the compressed cache state (data) needs to be kept consistent at all times with the dictionary contents. This makes it very hard to replace an entry in the dictionary because we must be sure that no cache line is compressed using the particular dictionary entry under eviction. Otherwise, the compressed data are going to be decompressed with the new (wrong) value that enters the dictionary, rather than with the old (correct) one.

A possible solution would be to keep track of all the cache lines compressed with any particular dictionary entry. Upon replacement of that entry the corresponding cache lines would be decompressed. Although this approach solves the consistency problem, it is extremely costly, invalidating the whole premise of efficient cache compression.

Our technique to attack this problem leverages on a leakage-saving proposal, namely cache decay, proposed by Kaxiras et al. [12]. Cache decay identifies cache lines which are unlikely to be accessed in the future (before their replacement). In [12] such cache lines (deemed to be "useless") are switched off in order to save leakage power. About 70% of the L1 can be discarded this way with minimal performance loss. The main idea of our work is to apply decay both in the cache and in the dictionary, discarding both unneeded compressed cache lines *and* their corresponding "frequent values" that are no longer needed by the remaining live cache lines. By decaying the cache and the dictionary in exactly the same way —in concert— we are guaranteeing that when a dictionary entry is decayed no live line in the cache can possibly refer to this entry.

Decay is implemented by measuring time since the last access to a cache line/dictionary entry. If a specified time interval (called the *decay interval*) passes without any access, the cache line/dictionary entry is discarded. We assume that as in [12], power to the cache line is switched off to save leakage power but the dictionary entry is simply marked as empty (available for replacement).

To measure the decay interval we use counters in each line/entry. The counters are reset with every access but advance when the line/entry is idle. When a counter reaches the decay interval the corresponding cache line/dictionary entry is decayed. Since reasonable decay intervals for the cache are in the range of a few thousand cycles [12], we use a hierarchical counter scheme where a global cycle counter advances every few hundred cycles small (e.g., 2-bit) local counters in each line/entry.

To show that decay keeps the compressed cache and the decaying dictionary consistent let us walk trough an example:

- Initially the dictionary and the cache are empty.
- When a cache line is brought into the cache, all its words are checked against the contents of the dictionary; if a word matches a value in the directory it is compressed; otherwise if there are empty slots in the dictionary the word is entered as a new frequent value. Thus, when a cache line is brought into the cache all of its frequent values in the dictionary are accessed and kept live.
- Similarly, when a compressed cache line is accessed (and therefore live) all its frequent values should be kept live too. Besides the requested word which is decompressed if needed (accessing the corresponding frequent value), all other compressed words in the cache line are used to reset the decay counters of the corresponding frequent values. This is a lightweight operation since we just reset decay counters —not access the frequent values.

Thus far we have established that any live line will keep its frequent values live in the dictionary for at least a decay interval after the line's last access. Consequently, when a frequent value decays in the dictionary, it means that *no cache line that uses this frequent value for its compression has been accessed for at least a full decay interval* (otherwise the frequent value would not have a chance to decay). But this last condition means that *all cache lines copressed with this frequent value have also decayed.* This allows us to replace decayed entries in the dictionary with new frequent values, thus adapting the contents of the dictionary to the set of

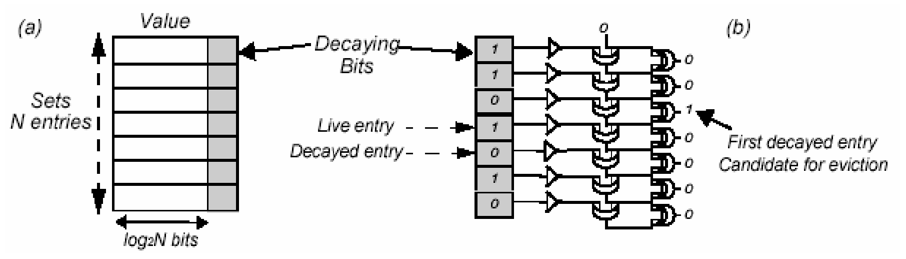

Fig. 2. (a) The dynamic decaying dictionary, (b) circuit to indetify the first decayed entry with an

frequent values that are most relevant during different phases of execution. An important characteristic of our proposal is that we do not replace entries on demand —as we would do with an LRU algorithm —but simply replace according to the availability of dead (decayed) entries.

4 Design Issues

L1 cache compression techniques must be designed in a very cautious manner since this level of hierarchy lies on the most critical path of the processor-memory model. In this Section, we use the XCACTI 2.0 [10] to estimate all the design issues of our *Dynamic Frequent Value Cache* (DFVC) in terms of access time and power.

4.1 Design Issues of Decaying Dictionaries

The decaying dictionary is a critical part of the design, because it must be accessed/ updated every time a read/write operation is performed in the DFVC. As we will see in the rest of this section, the decode/encode operation is in the critical path of the cache. Therefore, having an efficient dictionary design is very important. Our
solution, shown in Fig. 2.a, resembles a dual port register file design. In addition to the registers (holding the frequent values), there is an extra column that encapsulates the decaying functionality. This column contains the local decay counter and a decay status bit per entry showing its "liveliness" state. Collectively, the counter and the status bit are referred to as "decaying" bits since their overall functionality (and indeed their implementation) is captured by decaying 4-transistor (4T) memory cells.

We have modified XCACTI to estimate the access time required for a read/write operation in the dynamic dictionary. We adopted the register file model proposed in Wattch [3], using process parameters for a 130 nm technology. Our XCACTI estimates showed that the decode/encode operation of the register file is quite small varying from 0.39 ns for decoding 4 bits (16 entries) to 0.629 ns for decoding 7 bits (128 entries).

The decaying hardware (counter and status bit) comes at a negligible cost (in terms of time and power). We refer the reader to the work of Kaxiras et al. for this analysis [12]. From the other hand, to insert a new entry in the dictionary is not so trivial. A new frequent value must be inserted in the first decayed entry of the dictionary (considering a top-down ordering). Searching sequentially the dictionary for the first decayed entry (if any) is unacceptable since it will make the insertion of a new value extremely slow and costly. To alleviate this problem,

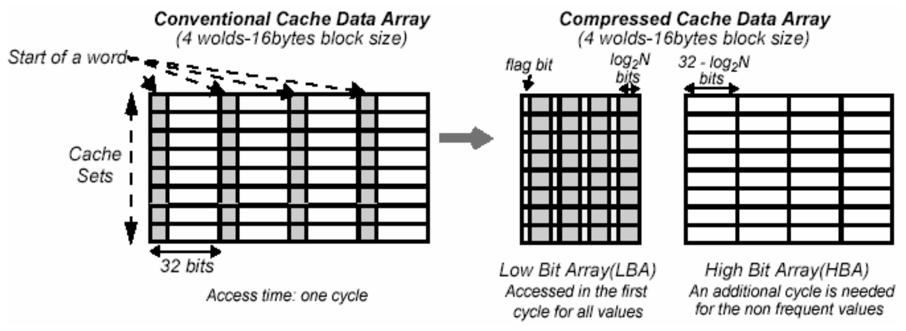
Conventional Cache Data Array
(4 wolds-16bytes block size)
Start of a word
Cache Sets
32 bits
Access time: one cycle
Compressed Cache Data Array
(4 wolds-16bytes block size)
flag bit
log_2N bits
$32 - log_2N$ bits
Low Bit Array(LBA)
Accessed in the first cycle for all values
High Bit Array(HBA)
An additional cycle is needed for the non frequent values

Fig. 3. PA-DFVC: data array partitioning

we use a simple combinatorial circuit, shown in Fig. 2.b, which identifies the first decayed dictionary entry at the cost of a few gates.

Having discussed the design issues of the decaying dictionary, let us now demonstrate our proposals for a *Power-Aware DFVC* (PA-DFVC) and a *High-Performance DFVC* (HP-DFVC). The hope is to create an efficient design where the time spent on encoding/decoding of the values has little impact —if any— in the cache access time.

4.2 Design Issues of Power-Aware DFVC (PA-DFVC)

In this section, we will show how the dynamic behavior of the frequent values, explained in Section 2, can be exploited in a power-aware compressed cache proposed by Yang and Gupta [23,26]. In contrast to their proposal, our dynamic compression scheme is able to adapt to changes of the frequent values for different parts of the execution during the execution of a program. Fig. 3 shows the partitioning of the data array of the PA-DFVC —no changes required in the tag array in this case.

In the PA-DVFC cache, the data values are divided in two categories: a small number of *N* frequent values (*N* reflects the number of the dictionary entries) and all the remaining values that are marked as nonfrequent values. The frequent values are stored in encoded form, and therefore can be stored in $\log_2 N$ number of bits, while the nonfrequent values are stored in unencoded form in 32-bit words.

As we can see from Fig. 3, the cache data array is partitioned so that one array contains $\log_2 N$ bits corresponding to each word (4 words in this example) and the other contains the remaining $32\text{-}\log_2 N$ bits. Frequent values are stored in encoded form in the Low Bit Array (LBA), while non-frequent values fill both data arrays. An additional bit (flag bit) corresponding to each word in a cache line is needed to indicate whether the word contains an encoded frequent value or an unencoded nonfrequent value.

The overall approach is as follows: when reading a word from the cache, initially we read from the LBA. Since the bits read out contain a flag bit, we examine it to determine what comes next. If the bit is set, which means the value was stored in encoded form, we do not need to read the HBA and must proceed to decode the value. In this case, the power consumption of the cache is reduced. However, if the value is stored in unencoded form, we proceed to access the remainder of the word from the HBA. Since the read from the LBA and the read from the HBA is serialized, it takes longer to read a non-frequent value than it would have taken to read the same value from a conventional cache.

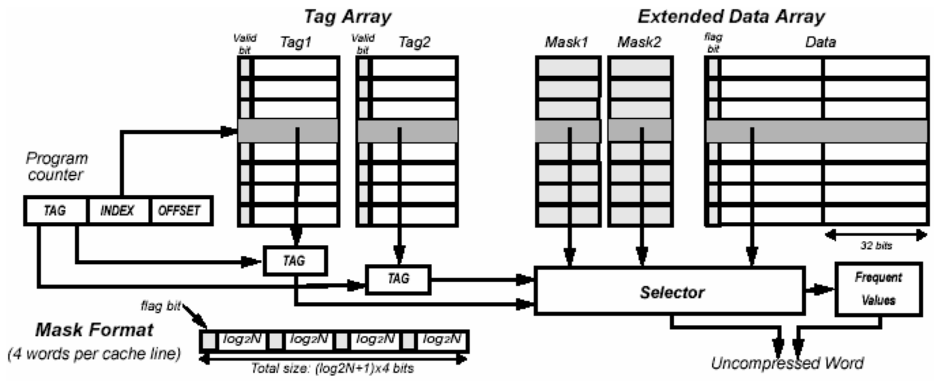

Fig. 4. HP-DFVC: detailed design

The hope is to reduce the energy consumed in the data array by accessing as much as possible the LBA. The reduction in energy comes at a cost of an additional cycle needed to access non-frequent values. Thus, the PA-DFVC design trades power for performance.

In this design, we assume that the $\log_2 N$ bits from the LBA are accessed in one cycle and we account for an additional cycle when the HBA is accessed. Recall that we account for one cycle for the conventional cache. In fact, this is true only if the time spent to encode/decode a value does not impact the overall latency. In order to meet this condition, we turn our attention to set-associative caches. In other words, the target for comparison are the cache architectures whose (pipelined) access time is defined by the tag-array and not the data array. As long as the time spent to perform tag matching is greater than the time spent to read the data plus the time to do the encoding/decoding, no additional overhead will be introduced in our PA-DFVC compared to a conventional cache.

Our XCACTI experiments show that the access times of a conventional cache and the PA-DFVC are the same for a $\log_2 N$ range up 7 bits. In our XCACTI experiments, we use the cache model adopted by Yang and Gupta [23,26]. This cache model, initally presented by Ghose and Kamble [9], is based in a subbanking scheme and has the advantage that each word (within a cache line) can be read independently without the need to read the whole cache line. The same model was used by Villa et al. [22] in their dynamic zero compression scheme. The results of the PA-DFVC, in terms of Energy-Delay Product and power reduction, are presented in Section 5.2.

4.3 Design Issues of High-Performance DFVC (HP-DFVC)

Our dynamic cache compression technique can be used to improve the behavior of the L1 cache by increasing its effective capacity. Cache/Memory compression has been proposed for better utilization of the available transistor budgets [1,13,22]. The idea behind this approach is to store cache lines in a compressed form so a greater number of cache lines can reside in the cache at any given time, lowering the miss rate.

Yang and Gupta [25] proposed a compressed L1 cache design where each set can store either one uncompressed line or two compressed lines. A static dictionary was used in their design. We solve the problem of keeping the cache and the dictionary contexts consistent, and we evaluate our approach using their framework. The overall scheme works as follows: we assume that each cache line of 2*L* words can store either one uncompressed 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. The architecture of the design is shown in Fig. 4.

As we can see, the cache entries must be accordingly modified to indicate whether or not they contain compressed lines. A flag 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 line has its own tag (Tag1, Tag2) and a valid bit. In addition, the mask fields (Mask1, Mask2) provide useful information for the compressed lines. The determination of a cache hit is as follows: if there is a tag match and the valid bit is set, we have a hit. The retrieval of a word requires examining the mask. If the mask indicates that the word is compressed, then the mask provides the index of the dictionary entry that holds the value in a compressed form. Conversely, if the mask indicates that the value is not compressed, it specifies the location of the word in the cache line where it is stored in uncompressed form. We refer the reader to the work of Yang and Gupta [25] for more details about this design. The results of our evaluation for the HP-DFVC are presented in Section 5.3. The target for comparison is an uncompressed cache of larger capacity.

4.4 Compression/Decompression of already Cached Data

Compression techniques have been initially used for instructions because code is not modified by a running program. Data compression techniques are harder to design because data values change as the program runs. This means that when cached data are modified, opportunities may arise to compress a previously uncompressed line. Our experiments show that compression opportunities for already cached data are rare for most benchmarks used in this paper. Thus, to simplify our designs we do not support compression of cached data —this can happen only when they are brought in the cache.

Furthermore, if an infrequent value is written on a compressed word, the need to uncompress the line may arise. In this case we immediately uncompress the whole cache line, possibly evicting its neighboring compressed line in the case of HP-DFVC.

5 Dynamic Frequent Value Cache Evaluation

5.1 Evaluation Methodology

To evaluate the effectiveness of our proposals, we perform simulations using Wattch [3], a detailed cycle level simulator which tracks dynamic power for each CPU structure. The processor model is based on the Alpha 21264. The execution core is 4-wide superscalar. The memory hierarchy includes a unified, 8-way set-associative, 1MB L2 cache. The latency of the main memory is 120 cycles. This configuration reflects prior work that examines the trade-off between power and performance using a static dictionary [23,25,26]. We use process parameters for a 130 nm technology and XCACTI 2.0 [10] to estimate all the modifications required by the proposed design. For the L1 data cache, we assume a decay interval of 8K cycles [12]. The same decay interval is used for the dictionary as explained in Section 3. We do not count

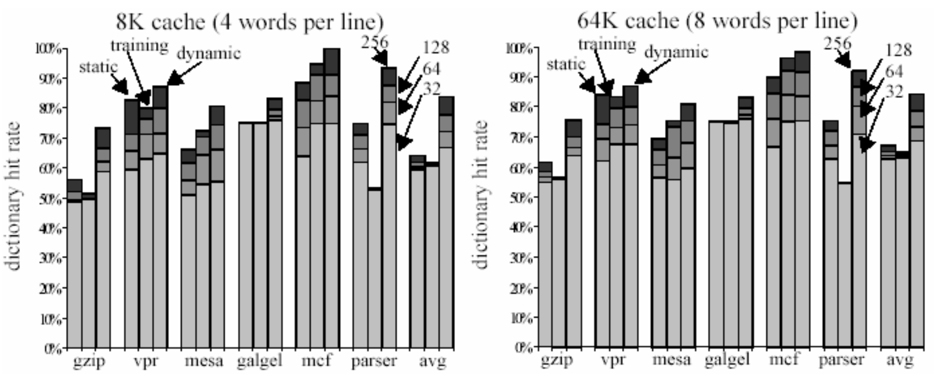

Fig. 5. Dictionary hit ratio using static, training and dynamic dictionary

leakage reduction from decay in our power consumption results —only dynamic power— since this would obscure the power benefit of compression.

The benchmarks suite for this study consists of a set of six SPEC2000 benchmarks (4 integer and 2 floating-point): *gzip*, *vpr*, *mesa*, *galgel*, *mcf* and *parser*, compiled for the Alpha ISA. For each program, we skip the first billion committed instructions to avoid unrepresentative startup behavior at the beginning of the program's execution, and then we simulate 200 million committed instructions using the reference input set.

5.2 Evaluation of the Power-Aware DFVC (PA-DFVC)

The main result of this Section is that cache compression using a dynamic dictionary leads to a more power efficient solution compared to the static/training dictionary approach. We conduct experiments using two cache configurations: an 8KB, 16-bytes-per-line, 4-way set associative cache and a 64KB, 32-bytes-per-line, 8-way set associative cache. Recall that the requirement for set-associativity is dictated by the need to hide the compression/decompression latency, as explained in Section 4.2.

Fig. 5 depicts the percent of hits in the frequent value dictionary (the left graph shows the results for the 8K cache configuration, while the right graph depicts the results for the 64K case). We compare our dynamic approach with static (referred as ideal in [24]) and training dictionaries of equal sizes. The static dictionary is created by profiling the benchmarks. The training dictionary is created by snooping at run time the values accessed during the first 10% of the program's execution, which are then used for the reminder of the run. The vertical bars in both graphs represent the static, training, and the dynamic techniques respectively. The light (bottom) bars stand for a dictionary size of 32 entries; every additional darker segment on top shows the increase in the dictionary hit ratio when a 64, 128 and 256 (darkest bar) entry dictionary is used.

As we can see from Fig. 5, the static and the training approaches are fairly close. The static approach yields better results for *gzip*, *vpr*, and *parser*, while the training dictionary seems a better solution for *mesa* and *mcf*. Both approaches have almost the same behavior in *galgel*. The dynamic dictionary technique outperforms the other two techniques in all benchmarks independently of the dictionary size. The improvement (average for both cache configurations) for a 256-entry dictionary is 18% and 21% compared to the static and to the training dictionary

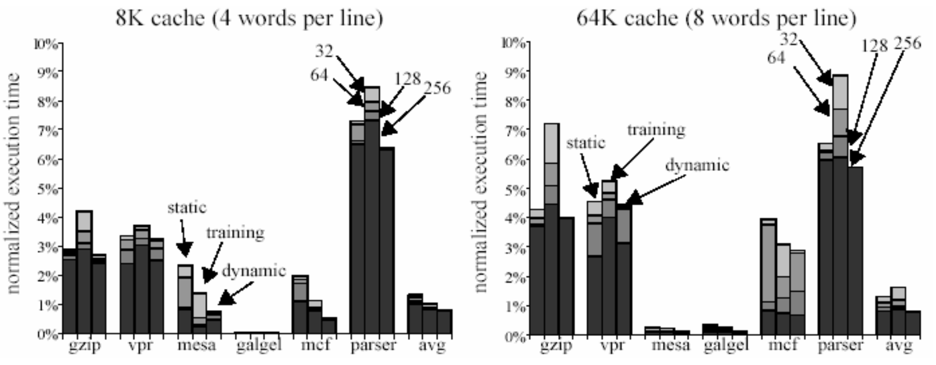

Fig. 6. Execution time for the static, training and the dynamic dictionary (normalized to non-decayed cache)

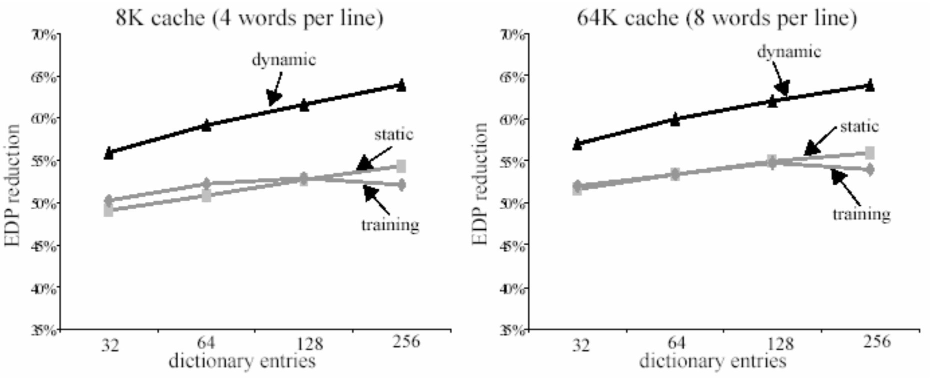

Fig. 7. Relative EDP reduction for the static, training and the dynamic dictionary

approaches respectively. In fact, the hit ratio in *mcf* of the 256-entry dynamic dictionary reaches the 99%. The results are analogous with dictionaries of smaller sizes.

The superiority of our approach can be seen when another metric is used for comparison: the execution time of the program. Recall that we account for one cycle when a hit takes place in a compressed word and two cycles when a hit occurs in an uncompressed word. As a consequence, smaller dictionaries increase the program's execution time, since more cache accesses follow the slow path (touch nonfrequent values). The slow down in execution time decreases as the size of the dictionary increases. Fig. 6 shows this trend. The darkest (bottom) bars represent a dictionary size of 256 entries (minimal slow down) and every additional darker segment on top shows the increase in the execution time when a 128-, 64-, and 32- (lighter bar) entry dictionary is used. Fig. 6 shows that our approach results in an increase in execution time less

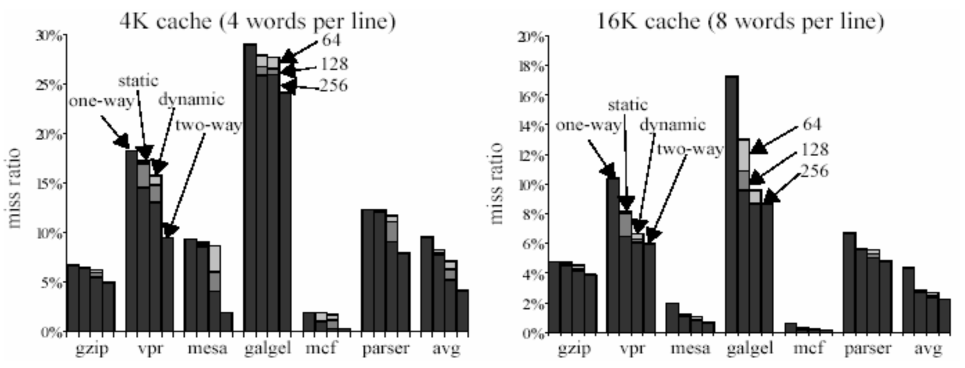

Fig. 8. Reduction in miss rates using static and dynamic dictionary

than 1% (average) even when a dictionary of 32 entries is used. In *parser*, which experiences the largest increases in execution time, the difference in execution time of the dynamic technique and the training technique is almost 3%.
We also use Wattch and XCACTI to estimate the relative Energy-Delay Product (EDP) reduction for the cache and for the three approaches we examine. The savings in the relative EDP, shown in Fig. 7, are substantial for the two cache configurations we examine. As we can see, our solution achieves up to 64% reduction in EDP compared to a non-decayed cache and nearly 10% reduction relative to the two other approaches, indicating that our solution is better for power-sensitive systems (i.e. portable devices).

5.3 Evaluation of the High-Performance DFVC (HP-DFVC)

In this section, we evaluate the effectiveness of the dynamic dictionary as opposed to increasing the effective capacity of the L1 data cache. As we explain in Section 4.3, the idea in this case is to store a cache line in a compressed form so that a greater number of cache lines can fit in the cache simultaneously and thus lower the miss rate.

Again, we consider two compressed cache configurations: a 4KB, 16-byte-per-line, and a 16KB, 32-byte-per-line. The compressed caches can accommodate up to two compressed lines per cache line. The targets for comparison are a conventional direct-mapped cache (DM) of equal size and a conventional 2-way cache of double size. Recall that, in the high-performance case, the design of the compressed cache necessities a doubling of the tag array and an increase of the data array by 2 bytes per cache line. We assume 3 cycles for the compressed cache, because in this case the compression/ decompression latency cannot be hidden by the tag comparison. We validated this model using the XCACTI simulator. Thus, in the first configuration the comparison is among a 4K, 2-cycle, direct mapped cache, our 3-cycle compressed cache (which yields an effective capacity of about 5K) and an 8K, 2-way, 3 cycle uncompressed cache.

The cache miss rates (absolute numbers) for the six benchmarks are shown in Fig. 8. The right graph shows the results for the 4KB cache, while in the left graph presents the results for the 16KB cache. There are four bars per benchmark: the leftmost bar shows the miss rate for the DM cache, the next two bars represent the compressed cache using the static and the dynamic dictionary respectively, and the rightmost bar represents the 2-way uncompressed cache (of larger capacity). Similarly to previous graphs, for the compressed caches, the darkest (bottom) bars correspond to the 256-entry dictionaries and the two additional darker segments above show the increase in miss ratio with 128- and a 64-entry dictionaries.

For the majority of the benchmarks, the compressed-cache miss rate —for both the static and the dynamic dictionaries— is very close to the 2-way cache and is clearly better than the DM cache. Of course, our approach offers better improvements over the static case especially for the 4KB cache (left graph). For example, in *parser*, the compression using a static dictionary offers only a slight drop in miss ratio (< 0.2%) compared to the DM cache, while the dynamic approach manages to lower the miss ratio by 3%. This is a relative improvement of 26.5%. On average the DFVC improves the miss ratio by 45% for the 4KB cache and 44% for the 16KB cache, while the static approach achieves a 18% and 36% improvement for the two configurations we examined.

6 Related Work

Compression in Memory Components. Most schemes for cache compression are proposed for power/energy savings rather than performance. The idea behind such schemes is simple: unused storage cells and wires provide a benefit simply by not consuming power. In the Dynamic Zero Compression scheme [22], each zero valued byte is represented by a single bit. Another approach for power/energy reduction was by Kim et al. [13]. The authors exploit small sign-extended values by compressing the upper portion of a word to a single bit if it is all 1s or all 0s. Recently, the idea of compressing the sign-extended values was further refined [18]. The goal in this case was to increase the apparent capacity of the L1 data caches.

As we have already mentioned, our work is inspired by frequent value locality shown by Zhang et al. [26] and subsequently by Yang et al. [23,25]. This value locality motivates their initial approach to increase the effective capacity of the L1 cache [23] and their latter approach to reduce the power consumption of the cache [25]. Alameldeen and Wood exploit value locality in their Frequent Pattern Compression algorithm applied to L2 caches [1]. They observe that some data patterns, are frequent and compressible. This work can be considered as the only adaptive compression mechanism for hardware caches but relies on a compression mechanism that is too slow, expensive, and power-consuming for L1.

Lee et al. [14,15] propose a compressed memory hierarchy, called Selective Compressed Memory System (SCMS), that selectively compresses L2 cache and memory blocks that can be reduced to half their original size. The idea of the SCMS was recently further investigated by Hallnor and Reindardt [11]. Their design allows blocks to be compressed in variable amounts of storage according to their compressability. Their results show a significant benefit from this flexibility. Chen et al. [4] propose a scheme that dynamically partitions the cache into sections of different compressability.

The compression technique was applied in many commercial products too. In IBM's Memory Expansion Technology (MXT) [21], all main-memory data is stored in compressed form. A hardware engine built into the memory controller manages compression/decompression transparently to software. However, to reduce decompression latency for misses in the on-chip caches, the MXT memory controller includes a large (32 MB) off-chip uncompressed cache. Recently, Ekman and Stenstrom [6], attacked the problem of the long decompression latency in main memory compression schemes by using a simple but very effective compression technique. Their mechanism (inspired by the frequent value approach) introduces negligible decompression latency. Thus, their method does not rely on huge caches.

Compression in Communication Channels. There has been a significant amount of research on reducing address/data bus swithching activity. Work dedicated to address buses such as Bus Expander [5], Dynamic Base Register Caching [7], and Working Zone Encoding [17], is based on the sequentiality of program counters and regularity of memory accesses . These techniques have been re-evaluated for data buses. In this case, the benefits were significantly reduced, since these schemes fail to exploit locality in non-contiguous bit positions.

The work that applies to data buses includes variants of directory-based solutions. Frequent Value Encoding [24] is a data bus encoding scheme capable of encoding entire data values. FVMSBLSB [20] stores the MSB portions and the LSB portions of values in separate tables. While encoding MSB/LSB portions alone, the remaining portion of the data are sent unencoded. Recently, Suruch et al. [19] proposed a scheme, called TUBE, which captures chunks of varying widths from data values. Finally, Basu et al. [2] proposed a value cache at both ends of a memory channel. During a hit, the index to the cache entry is sent instead of the whole word.

7 Conclusions

In this paper, we propose the first mechanism —to best the of our knowledge— for dynamic dictionary-based compression for L1 data caches. Our approach relies on the frequent value locality. In contrast to the previously proposed dictionaries for cache compression, the context of our dictionary dynamically adjusts to the requirements of a running program. We solve the problem of keeping the cache state and the dictionary state consistent by decaying the cache/dictionary in exactly the same way. Decayed entries in the dictionary are available for replacement by new frequent values without worrying about dependencies with the cache compressed data (no live cache line can possible refer to a decayed dictionary entry). Thus, we adapt the contents of the directory to the set of frequent values that are most relevant at any point in the execution.

We evaluate our adaptive compression technique using full system simulation and a range of benchmarks. Our dynamic scheme provides an improvement in the relative EDP of the cache up to 10% compared to the static and training approaches leading to a more power-efficient solution. When high performance is the target for optimization, our proposal yields 45% reduction in miss rate compared to a conventional cache of the same capacity and up to 27% improvement over the static dictionary technique.

References

1. A. Alameldeen and D. Wood. Adaptive Cache Compression for High-Performance Processors. 31st International Symposium on Computer Architecture, 2004.
2. K. Basu, et al. Power protocol: Reducing Power Dissipation on Off-Chip Data Buses. 35th International Symposium on Microarchitecture, 2002.

3. D. Brooks, et al. Wattch: A framework for Architectural-level power analysis and optimizations. 27th International Symposium on Computer Architecture, 2000.
4. D. Chen, et al. A Dynamically Partitionable Compressed Cache. Singapore -MIT Alliance Symposium, 2003.
5. D. Citron and L. Rudolph. Creating a Wider Bus using Caching Techniques. 1st Symposium on High Performance Computer Architecture, 1995.
6. M. Ekman and P. Stenstrom. A Robust Main-Memory Compression Scheme. 32nd International Symposium on Computer Architecture, 2005.
7. M. Farrens and A. Park. Dynamic Base Register Caching: A technique for Reducing Address Bus width. 18th International Symposium on Computer Architecture, 1991.
8. F. Gabbay and A. Mendelson. Can Program Profiling Support Value Prediction?. 30th International Symposium on Microarchitecture, 1997.
9. K. Ghose and M. B. Kamble. Reducing Power in Superscalar Processor Caches using Subbanking, Multiple Line Buffers, and Bit Line Segmentation. International Symposium on Low Power Electronics and Design, 1999.
10. M. Huang, et al. L1 Data Cache Decomposition for Energy Efficiency. International Symposium on Low Power Electronics and Design, 2001.
11. E. Hallnor and S. Reinhardt. A Unified Compressed Memory Hierarchy. 11th Symposium on High Performance Computer Architecture, 2005.
12. S. Kaxiras, et al. Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power. 28th International Symposium on Computer Architecture, 2001.
13. D. Kim, et al. Low-Energy Data Cache using Sign Compression and Cache Line Bisection. Workshop on Memory Performance Issues, 2002.
14. J.S. Lee, et al. An On-chip Cache Compression Technique to Reduce Decompression Overhead and Design Complexity. Journal of Systems Architecture, 2000.
15. J.S. Lee, et al. Adaptive Methods to Minimize Decompression Overhead for Compressed On-chip Cache. International Journal of Computers and Application, 2003.
16. M. Lipasti, et al. Value Locality and Load Value Prediction. 7th International Conference on Architectural Support for Programming Languages and Operating Systems, 1996.
17. E. Musoll, et al. Working Zone Encoding for Reducing the Energy in Microprocessor Address Buses. Transaction on VLSI Systems, 1998.
18. P. Pujara and A. Aggarwal. Restrictive Compression Techniques to Increase Level 1 Cache Capacity. International Conference on Computer Design, 2005.
19. D. Suresh, et al. Tunable Bus Encoder for Off-Chip Data Buses. International Symposium on Low Power Electronics and Design, 2001.
20. D. Suresh, et al. Power Efficient Encoding Techniques for Off-Chip Data Buses. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, 2003.
21. R. Tremaine, et al. Pinnacle: IBM MXT in a Memory Controller Chip. IEEE Micro, 2001.
22. L. Villa, et al. Dynamic Zero Compression for Cache Energy Reduction. 33rd International Symposium on Microarchitecture, 2000.
23. J. Yang and R. Gupta. Frequent Value Locality and its Applications. Transactions on Embedded Computing Systems, 2002.
24. J. Yang and R. Gupta. Frequent Value Encoding for Low Power Buses. Transanctions on Embedded Computing Systems, 2004.
25. J. Yang, et al. Frequent Value Compression in Data Caches. 33rd International Symposium on Microarchitecture, 2000.
26. Y. Zhang, et al. Frequent Value Locality and Value-centric Data Cache Design. 9th International Conference on Architectural Support for Programming Languages and Operating Systems, 2000.