Novel Vantage- Scalable cache Compression Scheme

Prof. B.Padmavathi, Poonam P.Aswnani
Department of Computer Engineering GH Raisoni College, Pune, India
Email: bpadma_cse@yahoo.com, aswanipoonam41@gmail.com

Abstract

In today’s world speed is one of the important factor that is considered for selecting any electronic component in the market. Speed of a microprocessor based system mainly depends on the speed of the microprocessor which in turn depends on the memory access time. cache compression is one of the way to increase speed of a microprocessor based system since it increases cache capacity and off-chip bandwidth. Storing compressed lines in the cache increases the effective cache capacity. For example, a compressed L1 cache design where each set can store either one uncompressed line or two compressed lines. Increasing the effective cache size can eliminate misses and thereby reduce the time lost to long off-chip miss penalties. However, compression increases the cache hit time, since the decompression overhead lies on the critical access path. Depending upon the balance between hits and misses, cache compression has the potential to either greatly help or greatly hurt performance.

1. INTRODUCTION

Cache memories have long been used to reduce average memory latency and bandwidth. Current processors typically provide two levels of on-chip caches separate L1 instruction and data caches and a unified L2 cache. Compression can improve cache performance by increasing effective cache capacity and eliminating misses. Depending upon the balance between hits and misses, cache compression has the potential to either greatly help or greatly hurt performance. In this paper, an adaptive policy that dynamically adapts to the costs and benefits of cache Compression is introduced. A two-level cache hierarchy is considered where the L1 cache holds uncompressed data and the L2 cache dynamically selects between compressed and uncompressed storage. The key insight is that the LRU stack depth and compressed size determines whether a given reference hits because of compression, would have missed without compression, or would have hit or missed regardless. The controller updates a single, counter on each reference, incrementing by the L2 miss penalty when compression could have or did eliminate a miss and decrementing it by the decompression latency when a reference would have hit regardless. The controller uses the predictor when the L2 allocates a line: storing the line uncompressed if the counter is negative, and compressed otherwise.

2. Compressed Cache Hierarchy

A two-level cache hierarchy consisting of uncompressed L1 instruction and data caches, and an optionally compressed L2 unified cache. The goals of this policy include:
• Using compression to increase effective L2 cache capacity in order to reduce L2 misses.
• Limit the impact of cache decompression overhead.
• Enable the policy to dynamically control compression based on workload demands.

2.1. Overview

Figure 1 shows the considered cache hierarchy. L1 instruction and data caches store uncompressed lines, eliminating the decompression overhead from the critical L1 hit path. The L1 data cache uses a write-back, write allocate policy to simplify the L2...
compression logic. On L1 misses, the controller checks an uncompressed victim cache in parallel with the L2 access. On an L2 hit, the L2 line is decompressed if stored in compressed form. Otherwise, it bypasses the decompression pipeline. On an L2 miss, the requested line is fetched from main memory.

![System Architecture for cache compression](image)

**Fig. 1: System Architecture for cache compression**

### 2.2 Decoupled Variable-Segment Cache

The data array is broken into eight-byte segments, with 32 segments allocated to each cache set. Thus, each set can hold no more than four uncompressed 64-byte lines, and compression can at most double the effective capacity. Each line is compressed into between one and eight segments, with eight segments being the uncompressed form. A separate cache state indicates the line’s coherence state, which can be M (modified), S (shared), I (invalid), or NP (not present). The compression tag indicates whether or not the line is stored in compressed form (CStatus) and the compressed size of the line (CSize). Data segments are stored contiguously in address tag order. That is, the offset for the first data segment of line k is:

$$\text{segment_offset}(k) = \sum_{i=1}^{k-1} \text{actual_size}(i)$$

Because the L1 and L2 caches maintain exclusion, an L1 replacement writes back both clean and dirty lines to the L2 cache. In many cases, the writeback finds a matching address tag, with space allocated, in state NP. If the compressed size is the same as before, this writeback is trivial. However, if the address tag is not found, or the compressed size has changed, the cache controller must allocate space in the set. This may cause replacing one or more L2 lines or compacting invalid/not present lines to make space. More than one line may have to be replaced if the newly allocated line is larger than the LRU line plus the unused segments. In this case, two lines are replaced by the LRU line and searching the LRU list to find the least-recently-used line that ensures we have enough space.

### 3. Compression Algorithm

The challenges to cache Compression algorithm are 1) Cache compression requires algorithm should de/compress a word in only a few CPU clock cycles. This rules out software implementations and has great influence on compression algorithm design. 2) Cache compression algorithms must be lossless to maintain correct microprocessor operation. 3) Good compression ratio is challenge. 4) The complexity of managing the locations of cache lines should be low.

#### 3.1 C-Pack Algorithm

C-Pack achieves compression by two means: (1) it uses statically decided, compact encodings for frequently appearing data words (2) it encodes using a dynamically updated dictionary allowing adaptation to other frequently appearing words. The dictionary supports partial word matching as well as full word matching.

The patterns and coding schemes used by C-Pack are summarized in Table I, which also reports the actual frequency of each pattern observed in the cache trace data file mentioned. The ‘Pattern’ column describes frequently appearing patterns, where ‘z’ represents a zero byte, ‘m’ represents a byte matched against a dictionary entry, and ‘x’ represents an unmatched byte. In the ‘Output’ column, ‘B’ represents a byte and ‘b’ represents a bit. The C-Pack compression and decompression algorithms are illustrated in Figure 2. We use an input of two words per cycle as an example in Figure 2. However, the algorithm can be easily extended to cases with one, or more than two, words per cycle. During one iteration, each word is first compared with patterns “zzzz” and “zzzx”. If there is a match, the compression output is produced by combining the corresponding code and unmatched bytes as indicated in Table I. Otherwise, the decompressor compares the word with all dictionary entries and finds the one.
with the most matched bytes. The compression result is then obtained by combining code, dictionary entry index, and unmatched bytes, if any. Words that fail pattern matching are pushed into the dictionary. Figure 3 shows the compression results for several different input words. In each output, the code and the dictionary index, if any, are enclosed in parentheses. Although a 4-word dictionary can be used, the dictionary size is set to 64B in our implementation. Note that the dictionary is updated after each word insertion. During decompression, the decompressor first reads compressed words and extracts the codes for analyzing the patterns of each word, which are then compared against the codes defined in Table I. If the code indicates a pattern match, the original word is recovered by combining zeroes and unmatched bytes, if any. Otherwise, the decompression output is given by combining bytes from the input word with bytes from dictionary entries, if the code indicates a dictionary match. The C-Pack algorithm is designed specifically for hardware implementation. It takes advantage of simultaneous comparison of an input word with multiple potential patterns and dictionary entries.

This allows rapid execution with good compression ratio in a hardware implementation, but may not be suitable for a software implementation. In general, software must process operations sequentially. For example, matching against multiple patterns can be prohibitively expensive for software implementations when the number of patterns or dictionary entries is large. C-Pack’s inherently parallel design allows an efficient hardware implementation, in which pattern matching, dictionary matching, and processing multiple words are all done simultaneously. In addition, we chose various design parameters such as dictionary replacement policy and coding scheme to reduce hardware complexity, even if our choices slightly degrades the effective system-wide compression ratio. In the proposed implementation of C-Pack, two words are processed in parallel per cycle. Achieving this, while still permitting an accurate dictionary match for the second word, is challenging. Let us consider compressing two similar...
words that have not been encountered by the compression algorithm recently, assuming the dictionary uses first-in first-out (FIFO) as its replacement policy. The appropriate dictionary content when processing the second word depends on whether the first word matched a static pattern. If so, the first word will not appear in the dictionary. Otherwise, it will be in the dictionary, and its presence can be used to encode the second word. Therefore, the second word should be compared with the first word and all but the first dictionary entry in parallel. This improves compression ratio compared to the more naive approach of not checking with the first word. Therefore, we can compress two words in parallel without compression ratio degradation.

4. Adaptive Cache Compression

While compression reduces L2 misses, it increases the latency of the (usually more frequent) L2 hits. Ideally, a compression should compress data when the benefit (i.e., avoided misses) should be greater than cost (i.e., penalized L2 hits). An adaptive predictor that checks the actual effectiveness of compression and uses the feedback to dynamically determine whether to store a line in a compressed or uncompressed form. L2 cache compression will help if:

\[(\text{avoided L2 misses} \times \text{L2 miss penalty}) > (\text{penalized L2 hits} \times \text{decompression penalty})\]

Where penalized L2 hits are those that unnecessarily incur the decompression penalty. Rearranging terms yields:

\[(\text{avoided L2 misses} - \text{penalized L2 hits}) > (\text{decompression penalty} - \text{L2 miss penalty})\]

Thus adaptive policy obtains most of the benefit of compression when it helps, while never performing much worse than not compressing.

4.1 Classification of Cache References

The key feature of adaptive compression policy is that the LRU stack depth and compressed size determine whether compression helps or hurts a given reference.

Classification of hits:
• A reference to Address A hits at stack depth 1. Because the set can hold four uncompressed lines and the LRU stack depth is less than or equal to four, compression provides no benefit. Conversely, since the data is stored uncompressed, the reference incurs no decompression penalty. This case is called an unpenalized hit.

• A reference to Address C hits at stack depth 3. Compression does not help, since the line would be present even if all lines were uncompressed. Unfortunately, since the block is stored in compressed form, the reference incurs an unnecessary decompression penalty. This case is called a penalized hit.

• A reference to Address E hits at stack depth 5. In this case, compression has eliminated a miss that would otherwise have occurred. This case is called an avoided miss.

4.2 Classification of misses

• A reference to Address G misses in the cache, but matches the address tag at LRU stack depth 7. The sum of the compressed sizes at stack depths 1 through 7 totals 29. Because this is less than 32 (the number of data segments per set), this reference misses only because one or more lines at stack depths less than 7 are stored uncompressed (i.e., Address A could have been stored in two segments). This case is called an avoidable miss.

• A reference to Address H misses in the cache, but matches the address tag at LRU stack depth 8. However, this miss cannot be avoided because the sum of compressed sizes exceeds the total number of segments (i.e., 35 > 32). Similarly, a reference to Address I does not match any tag in the stack. This case is called an unavoidable miss.

The cache controller uses the LRU state and compression tags to determine the class of each L2 reference.

4.3 Compression Controller

The adaptive compression policy uses past behavior to predict the future. Specifically, the controller uses the classification above to update a global counter—called the Global Compression Counter (GCC)—to estimate the recent cost or benefit of compression. On a penalized hit, the controller biases against compression by subtracting the decompression penalty. On an avoided or avoidable miss, the controller increments the counter by the (unloaded)
L2 miss penalty. To reduce the counter size, we normalize these values to the decompression latency, subtracting one and adding the miss penalty divided by decompression latency. The controller uses the GCP when allocating a line in the L2 cache. Positive values mean compression has been helping eliminate misses, so we store the line in compressed form. Negative values mean compression has been penalizing hits, so we store the line uncompressed. All allocated lines—even those stored uncompressed—must run through the compression pipeline to calculate their compressed size, which is used to determine avoidable misses.

5. Conclusion

The objective of adaptive cache compression is to achieve performance comparable to the best of *Always* or *Never*. Using C-Pack, it is possible to compress and decompress the data into the cache in an efficient way without altering the processor performance. Vantage policy dynamically adjusts to the costs and benefits of compression. The L2 controller updates the counter based on whether compression could (or did) eliminate a (potential) miss or incurs an unnecessary decompression overhead. Maintains good compression ratio and area overhead and thus decreases memory latency and speeds up the processor and by making the system to work with high speed and thus helpful for mankind.

References


