Adaptive Cache Compression in Gem5

Kumar Alabhya
Department of Electrical and Computer Engineering
Purdue University
West Lafayette, USA
kalabhya@purdue.edu

Vivek Mahesh Nair
Department of Electrical and Computer Engineering
Purdue University
West Lafayette, USA
nair185@purdue.edu

Yatharth Agarwal
Department of Electrical and Computer Engineering
Purdue University
West Lafayette, USA
agarw414@purdue.edu

***Abstract*—Modern Processors rely heavily on caches to mitigate the memory bottleneck. However, the small size of these caches often proves fatal for data-intensive workloads. The fast access and area requirements put a size limit on the caches, which can be incorporated into SoCs, making it a big problem. In this report, we evaluate adaptive compression caches in the Gem5 simulator. We implement a Variable Segment Cache, which, combined with data compression, provides double the capacity compared to a normal uncompressed cache. Further, to avoid decompression penalization on data sparse programs, we implement an adaptation algorithm that stores data in compressed form only if there is a benefit. For our performance analysis we ran the simulation on SPEC 2k17 benchmarks. We belive that the Adaptive Cache compression Policy is ineffective for modern workloads. The code for our implementation can be found at: https://github.com/yathAg/gem5Project**

1. Introduction

The steep increase in the frequency of CPUs and the memory not being able to cope with this increase have made the memory bottleneck a cumbersome issue to be solved. Caches play a pivotal role in overcoming this bottleneck up to some extent. The caches must be faster than the main memory and are maintained much smaller than the main memory.

In today's world, Machine Learning and Artificial Intelligence dominate major use cases, dramatically increasing memory access. This has led to increasing misses in the cache due to their small sizes. Hence, there is a requirement for larger caches to support such data-intensive applications. So, on the one hand, we need faster data access, but on the other, we need larger cache sizes, creating a paradox. To resolve this paradox, we analyze adaptive cache compression [1] for the data stored in the cache for modern workloads. Since more data can be stored in a given cache size in a compressed format than in an uncompressed form, it allows us to improve the capacity of the cache without compromising the access time.
Modern processors use two or more levels of cache memories to bridge the rising disparity between processor and memory speeds. Hence, the next question is determining in which cache the adaptive compression should be implemented. Since compression comes with a decompression penalty and first-level cache (L1) caches are highly hit time critical, the L1 cache stores only uncompressed data [2]. The second level cache, or the L2 cache, needs a lower miss rate and would effectively benefit from compression logic.

Compression of block works well for memory-intensive workloads. However, we are penalized for workloads where memory access is not dominant due to the high decompression penalty. Therefore, a system that dynamically compresses data based on the workload requirements improves the performance of such workloads.

The remaining of the report is organized as follows:

- Section II discusses GEM5 Implementation Details of the Architecture
- Section III discusses the Evaluation methodologies and benchmarks.
- Section IV provides the results of miss rates and performance
- Section V Shares our conclusion and scope of further improvement

2. GEM5 Implementation

This project is designed to develop a cache system capable of storing data in compressed or uncompressed formats, depending on the cost-benefit analysis of compression. The implementation is divided into three main components, which are described in Figure 1

- Compression logic involves ensuring proper compression using FPC (Frequent Pattern Compression).
- Implementation of a decoupled variable segment cache to support compressed data.
- Implementation of the global predictor algorithm for Adaptive Compression.

2.1. Frequent Pattern Compression

The compression algorithm implemented for the L2 cache is Frequent Pattern Compression (FPC) [3]. FPC

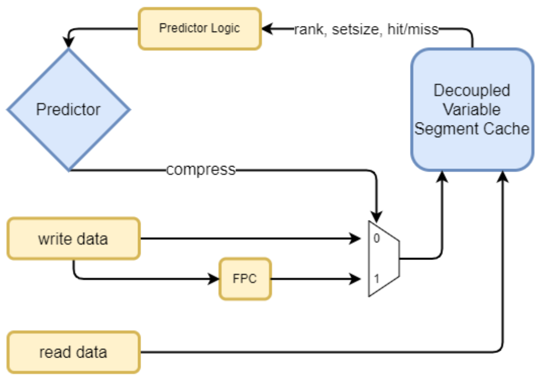

Figure 1: Structure of the Hardware implemented

compresses individual cache lines on a word-by-word basis by storing common word patterns in a compressed format accompanied by an appropriate prefix. The algorithm can be implemented on hardware with very little extra hardware and reasonable run-time latency, which makes it ideal for L2 cache compression. Gem5 already has the FPC compressor implemented and serves as the base for our project.

2.2. Decoupled variable segment Cache

For Adaptive Cache Compression, we implement a structure called the decoupled variable segment cache. The Decoupled Variable Segment Cache provides a cache structure that can be used to derive performance benefits out of the data compression through FPC. The main idea is that the data are stored in a long array consisting of segments, each 8 bytes wide. The uncompressed data block is eight segments wide (considering the cache line to be 64B long), and FPC can compress this data to anywhere between 1 and 7 segments (8 to 56 bytes). For an 8-way set associative tag structure, this cache consists of 32 segments and thus can hold a maximum of 4 uncompressed blocks and up to 8 compressed blocks (limited by tag structure). Hence, we obtain a two-time data storage compared to a normal uncompressed cache.

2.2.1. Decoupled Variable Segment Cache : GEM5 Implementation

We use the in-built base set associative tags for the GEM5 implementation of the cache mentioned above. We instantiate a n-way associative cache, where n is the maximum number of compressed lines that can be stored in a set (in our case, n=8). During the allocate phase the data is compressed by the FPC compressor. We compute the number of free segments by finding the sum of the sizes of all valid lines in the set. If the number of free segments (out of a total of 32) available is sufficient for the new block, it is allocated, and if not, we need to evict the required-sized block(s) to make space for the new block. If all eight tags are occupied, eviction will be required even if there is enough space.

To achieve the above, we implement three major modifications in GEM5:

1) Modify the cache block to hold compressed blocks and store its size and decompression latency,
2) A new getvictim function to handle the evicts for the new cache structure,
3) A new updateCompressionData function for writeBacks.

Cache Block Modifications: The first step is setting up the required attributes for a normal cache block. We implement a variable in the *CacheBlk* class for size and decompression latency. Further, we implement a function to allocate these values, namely *set/getSize* and *set/getDecompressionLatency*. These functions are called from the *base.cc* of the cache, which transfers the required values from the compressor to *CacheBlk*.

Algorithm 1: findVictimVariableSegment

```
entries ← getPossibleEntries(addr)
victim ← nullptr
curr_blk ← nullptr
valid_entries ← empty list
evict_blks ← empty list
set_size ← 0
for each entry in entries do
    if forUpdation and entry.matchTag(addr) then
        curr_blk ← entry
    else if entry is valid then
        valid_entries.append(entry)
        set_size ← set_size + entry.size
    else
        victim ← entry
diff_size ← max_set_size - set_size
if diff_size > 0 and !victim then
    victim ← LRU(valid_entries)
    evict_blks.append(victim)
    diff_size = diff_size - victim.size
if diff_size > 0 then
    new_valid_entries ← empty list
    for each entry in valid_entries do
        if entry.size >= diff_size and entry !=
          victim then
            new_valid_entries.append(entry)
    victim ← LRU(new_valid_entries)
    evict_blks.append(victim)
if forUpdation then
    return curr_blk
else
    return victim
```

getVictim implementation We implement a replacement unit that can evict the correct block based on the size requirement and victim block sizes. Hence, we approach the problem by creating a replacement function called the *getVictimVariableSegment*. The first part of the logic checks if the write is for updating an existing block's data or allocating the block to new data. If the data is being updated, the current block is returned as a victim. We calculate the free size available in the cache for data allocation and check if a block can be inserted in the given space. The pseudo-code for this

If all the segments hold valid blocks, we must make an eviction irrespective of available size. If there is enough space and at least one invalid block, we return that invalid block as the victim. If we do not have enough space, we evict the Least Recently Used block, and then calculate the free size. If this meets the size requirement, we exit the function; otherwise, we search for the least recently used block whose eviction provides us enough space for block allocation and to evict this block. At the end of the function, one of the evicted blocks is returned as the victim. The complete logic is demonstrated by the logic below.

Algorithm 2: UpdateCompression

```
if compressor  predictor then
    comp$_data$← compress(data)
    compression_size ← comp_data
      → $getSizeBits()$
    compression_size ← (compression_size + 63)
      & !63
    if compression_size < blkSize then
        decompression_lat $gets$ Cycles(5)
else
    Compression_size $gets$ blkSize
prev_size ← blk_size is_data_expansion ← false
  is_data_contraction ← false
if prev_size < compression_size then
    is_data_expansion ← true
prev_size > compression_size is_data_contraction
← true
victim ← blk evict_blks ← empty vector
if is_data_expansion then
    victim ← findVictimVariableSeg-
      ment(regenerateBlkAddr(blk)
      compression_size, evict_blks, true)
    if !victim then
        return false
if !handleEvictions(evict_blks, writebacks) then
    return false
setSizeBits(blk, compression_size)
  setDecompressionLatency(decompression_lat)
  return true
```

UpdateCompression implementaion Update compression data is required during writebacks when the block is already present in L2. After compressing the updated data, the compressed size can increase, decrease, or stay the same. In case of the compressed size decreases or stays the same, we can simply write the new data and update the compressed size in tags. But in case of an expansion, we need to ensure that there's enough free space available for the expansion, if not, blocks are evicted.

2.3. The Adaptive Logic: Global Predictor

The Predictor logic classifies each memory access into five different classes, namely, unpenalized hit, penalized hit, avoided miss, avoidable miss, and unavoidable miss. Below, we discuss these transactions in detail.

Algorithm 3: Pseudo-code for the Global Prediction Algorithm

```
Data: int hit, rank, setSize, i; Block *blk; Packet
      *pkt; Tags *tags; int global_predictor;
if blk is hit then
    hit ← 1;
    rank ← getLRURank(Addr, blk);
else
    setSize ← getSetSize(Addr());
if hit and blk→_isCompressed and rank ≤ 4 then
    global_predictor ← global_predictor -
      $decompression\_latency$;
else if hit and blk→_isCompressed and rank > 4
 then
    global_predictor ← global_predictor +
      $dram\_access\_time$;
else if not hit and setSize < max_size then
    global_predictor ← global_predictor +
      $dram\_access\_time$;
if global_predictor < 0 then
    compress_next ← false;
else
    compress_next ← true;
```

2.3.1. Unpenalized Hit

Memory accesses are regarded as an unpenalized hit when three conditions are met. First, the block is a hit. Second, the block is stored uncompressed in the cache, and third, the most recently used (MRU) rank derived from the LRU stack is less than or equal to $associativity/2$. In this situation, we do not update the global predictor because such transactions contribute no extra decompression latency and do not benefit from compression since it would have been a hit even in a regular uncompressed cache.

2.3.2. Penalized Hit

These accesses are similar to Unpenalized hits in that they are hits with an MRU rank less than or equal to $associativity/2$. They are, nevertheless, compressed blocks. Because they have a high MRU rank, they would have been a hit in the usual uncompressed cache, so compression provides no benefit. Additional cycles, on the other hand, are used to decompress the block on access. As a result, our logic penalized the performance. We deduct the decompression cycle from the Global Predictor variable in such cases.

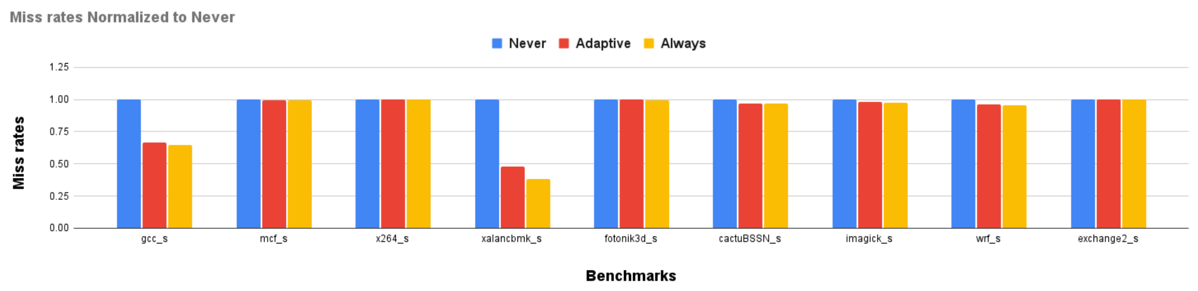

Figure 2: Miss Rates for Compressed Caches Normalized to Never Policy

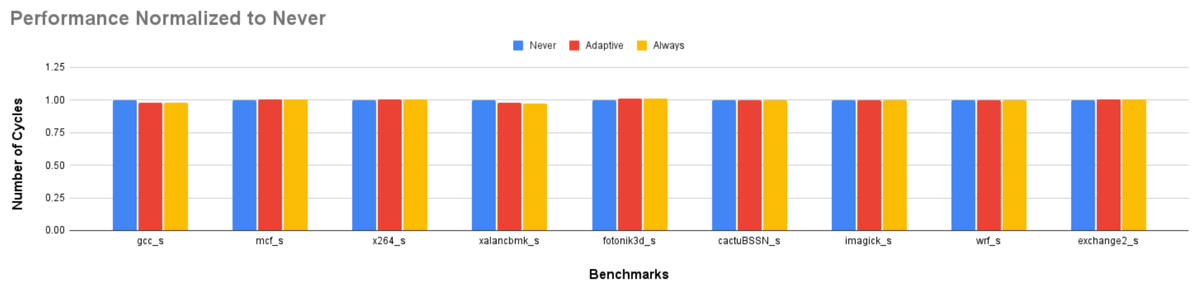

Figure 3: Performance of Compressed Caches Nomalized to Never Policy

2.3.3. Avoided Miss

This type of access is likewise considered a hit and is saved in a compressed manner. However, the MRU rank is greater than the $associativity/2$ value in this situation. Therefore, we profited from compression because it would have been a miss in the conventional cache. As a result, we added the DRAM access time that was saved to the global predictor.

2.3.4. Avoidable Miss

In the event of a miss, if there is still capacity inside the set to store the block in a compressed format, i.e., there is at least one free segment available in the set, it means if compression had been prioritized, this type of miss may have been prevented. As a result, in this scenario, we update the global predictor by adding the DRAM access time and prioritizing compression.

2.3.5. Unavoidable Miss

This is the final case of a missed access; however, the cache is too full to include this block even in its compressed form. As a result, it could not be avoided in any way, and hence, we do not penalize or promote the global predictor in this scenario.

The global predictor will have a negative value, ensuring that the new block allocated is not compressed. On the other hand, if there are a significant number of avoided or avoidable misses, the global predictor will have a positive value, making the new block allocated in a compressed format.

3. Evaluation of Adaptive Compression

We present an evaluation of adaptive compression on a dynamically scheduled out-of-order processor using full-system simulation of a subset of the SPECcpu2k17 benchmarks.

TABLE 1: System Parameters

Processor	BaseO3CPU, x86
L1 Cache configuration	64kB, 2-way assoc., 64B lines
L2 Cache configuration	256kB, 256B sets (32 segments)

3.1. Selected Benchmarks

In our comprehensive system performance evaluation, we conducted benchmark tests to assess the effectiveness of our implementation relative to the spec2017 benchmarks, which are widely recognized as representative of modern workloads. This evaluation included four integers(gcc, mcf, x264, xalancbmk) and five floating-point benchmarks (fotonik3d, wrf, imagick, cactuBSSN, exchange2). These carefully selected benchmarks provided a robust framework for evaluating and analyzing the computational capabilities of our implementation in the context of contemporary computing tasks. We fast forward each benchmark for 10 million instructions and simulate the following 10 million instructions. For better results, it is recommended to simulate for larger workloads. However, owing to the project's time constraints, we could not achieve the same.

4. Evaluation of Adaptive Compression

We evaluate the performance of adaptive compression to two extreme policies: Never and Always. Never represents a conventional 4-way set associative L2 cache design in which data is never compressed. The always scheme simulates a decoupled variable-segment cache but always stores compressed data. The proposed adaptive policy compresses the data only when the benefits outweigh the compression overheads.

4.1. Miss Rates for Compressed Caches

Compression to increase effective cache capacity should decrease the L2 miss rate. Figure 2 presents the average miss rates for the benchmarks. The results are normalized to never compress to focus on the benefit of compression. The differences concerning never compress have also been scaled by a factor of 10 to make them more apparent in the plot. Both Always and Adaptive have lower or equal miss rates when compared to Never. However, the differences were too small to have any practical benefits. We performed an analysis and found out that the differences in misrates in uncompressed cases after doubling cache capacity and associativity (i.e., the best case that compression can achieve) were not much to begin with. Therefore, the lack of improvement owes more to the lack of sensitivity of these benchmarks to L2 size than to the effectiveness of compression logic itself. Another flaw that we believe exists in the adaptive compression logic is that it doesn't consider compulsory misses. In our initial runs, we found out the predictor was always biased towards compression. This is because it treated most of the compulsory misses as avoidable misses, thus biasing itself towards prediction. And given the huge difference in memory access latency and decompression latency, no amount of penalized hits was enough to bias the predictor towards no compression. To overcome this, we initially biased the predictor towards no compression.

4.2. Performance

The performance (total cycles) is plotted in Figure 3. In benchmarks with relatively more significant differences between always compress and never compress miss-rates, such as gcc, the performance of always compress is the best. However, always-compress performs worse than never-compress in benchmarks with relatively minor differences in miss rates, such as cactuBSSN. However, adaptive compression can improve performance slightly. But, similar to miss rates, these differences were too low to have practical benefits. Another reason for the low differences is the low number of total instructions for which the simulation was implemented.

5. Conclusion

We successfully implemented the Adaptive Cache Compression policy in GEM5. Further, we could use the decoupled variable segment cache and FPC compression, replicating the reference paper. We also implemented the global predictor to make the compression adaptive based on the workload. To evaluate the performance of our architecture, we chose nine benchmarks from SPEC2k17. We found that the difference between miss rates and performances for all these benchmarks was insignificant. Therefore, we believe the cache architecture from the reference paper might not be effective for current workloads. Furthermore, we have provided our analysis of why we believe the differences in miss rates are too small.

Acknowledgment

We want to express our sincere gratitude to Professor Mithuna Thottethodi for his invaluable guidance and insightful input throughout this project. His expertise and mentorship have been instrumental in shaping the direction of this work.

We extend our heartfelt thanks to our Teaching Assistant, Ceasar, for his unwavering support and patience in clarifying doubts during various project stages. His dedication has contributed significantly to the successful completion of this project.

References

[1] A. Alameldeen and D. Wood, "Adaptive cache compression for high-performance processors," pp. 212–223, 2004.

[2] J. Yang, Y. Zhang, and R. Gupta, "Frequent value compression in data caches," *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, 2000.

[3] A. Alameldeen and D. Wood, "Frequent pattern compression: A significance-based compression scheme for l2 caches," 01 2004.