Analysis of Cache Compression via Clustering

Luciano Dagnillo
ECE 462/562
University of Arizona
Tucson, AZ, USA
lucianodagnillo@arizona.edu

Sophia Golota
ECE 462/562
University of Arizona
Tucson, AZ, USA
sgolota@arizona.edu

Jimmy Payan
ECE 462/562
University of Arizona
Tucson, AZ, USA
jimmypayan@arizona.edu

Cecilia Quevedo
ECE 462/562
University of Arizona
Tucson, AZ, USA
ceciliaquevedo@arizona.edu

***Abstract*—This analysis realizes the benefits that can be achieved through the utilization of dynamically clustered compressed cache lines. The particular focus of this compression application is Base-Delta-Immediate, or BDI, cache compression. The implementation is based on a dynamic cache clustering method introduced in relevant research papers and is modeled through a processor simulator. Performance parameters including workload runtime, compression ratio, hit and miss rates, and cache access latency were to be compared between the BDI compression simulation and the proposed method upon which the simulation was modeled. Though there was not a thorough analysis of these parameters, there was compression of the cache observed in an 8$\triangle$1 framework, which achieved a runtime of roughly 3ms.**

***Index Terms*—cache, cache compression, (dynamic) clustering, clusteroids.**

I. Introduction

Improving performance is always a key concern in the world of computing. Processor performance has grown exponentially since 1980 while memory performance has lagged behind, resulting in a performance bottleneck with memory accesses. Because the processor moves faster, memory requests are generated at a rate greater than that which the main memory can service. This issue has become increasingly relevant as computer architects transitioned from designing single-core systems to designing multi-core systems, thus further increasing the amount of memory requests.

The integration of caches has helped to remedy this bottleneck. Caches allow a processor to store frequently used blocks of data in a location near the core, and service memory requests at a higher frequency than main memory. This allows for reduced power consumption when repeatedly accessing frequently requested data, as the information does not have to travel from the main memory each time. Large caches should theoretically improve performance, as they limit the need to retrieve data from the slower main memory. The limiting factor is twofold; large, fast memory is difficult to manufacture, and it suffers from diminishing returns - as cache storage capacity increases, so does cache access latency.

To address the challenges presented by incorporating large caches, modern computer architects utilize multiple cache levels that decrease in size and access latency as their proximity to the core increases. Multilevel cache approaches can be further improved by compressing data in lower cache levels, decreasing the size of the data stored in a cache, thus increasing its effective capacity. This helps optimize the utilization of computer resources, as it allows more space to be allocated to computing resources of the system. Clustering is a specific method of cache compression, as it reduces the storage required to encode similar data by only storing one instance of the common data, alongside the differences.

This paper explores a solution to optimize system resource utilization via cache compression. The project aimed to reproduce a dynamic cache clustering method, as proposed in Thesaurus [1] and Two dimensional cache compression (2DCC) [2] using Base-Delta-Immediate (BDI) Compression [3]. In this method, the base entry in a cache line is compared to all subsequent entries to quantify the difference between each entry and the base as a delta. If this delta falls within the maximum tolerance of the chosen implementation, the new compressed cache line comprises the base entry and the sequential deltas. This technique was implemented with a processor simulator, ZSim [9], and results were evaluated in conjunction with the implementation of Thesaurus. In an ideal scenario involving eight uncompressed words differing by less than one byte, the maximum achievable compression ratio is 4 to 1. This scenario would allow 64B of data to be compressed down to 15B; the 8-byte base and seven 1-byte deltas (discussed in III). Notable performance parameters between the two methods, including workload runtime, compression ratio, hit and miss rates, and cache access latency were compared with those reported in Thesaurus.

Prior to simulation of the algorithm, various hypotheses and predictions were made with regard to performance of BDI compression: An ideal implementation of cache compression should feature a high compression ratio alongside unaffected hit rate, and a workload runtime, related to the cache access latency. The theoretical maximum compression ratio of BDI is 4:1, which is substantial. Hit rate should be entirely unaffected by cache compression, assuming the replacement policies implemented do not affect the integrity of the data stored in the cache. Workload runtime and access latency are somewhat related; if the cache can avoid a stall during data compression/decompression, then overall runtime should be the same. If a stall is required, compressed cache operations will increase system runtime.

A successful implementation of cache compression [1], [2] was found to reach compression ratios up to 2.25:1, at the cost of reducing hit rate to 94.8%. Simulating the cache was the

most challenging aspect of the project: the simulator utilized is optimized for research of many multi-core, multi-thread computers in parallel, and introduced programming overhead which prevented a working BDI algorithm from being simulated. A non-BDI cache was found to have a workload runtime sat of 3×10^6 ns, or 3 ms. As the experimental control, this cache has a compression ratio of 1:1. Cache access latency and hit rate were not measured.

II. Related Work

Previous papers have analyzed various methods of compressibility in caches, including frequent value compression (FVC) [7]. This method compresses the most frequently accessed data in a cache line, so that each cache line can hold one uncompressed cache line or two compressed lines. Unlike other compression techniques, FVC cannot group common patterns in an application's data, as the data must be identical to all other entries in the cache line. Deduplication [4] can be used to find identical cache lines and only store a single copy. However, this can be potentially ineffective if there exist few identical memory blocks. Using this concept, compression techniques have been developed to find nearly identical cache lines and store the differences.

Several papers have proposed a dynamic clustering method [1], [5], [6] for cache compression. Dynamic clustering is a compression technique that groups similar cache lines by defining a "clusteroid" or "base" as a reference for storing the differences of other cluster members. This method is designed to adapt to varying workloads and data patterns, enabling the system to handle different cluster sizes at runtime without additional parameters. Reference [6] is an example of this technique, where Alameldeen and Wood were able to develop a cache compression policy that would dynamically decide between compressed and uncompressed storage in a two-level cache to increase cache size. This "frequent pattern-based" compression could effectively double the size of a cache by dividing the cache line into 32-bit words and checks before comparing it to seven frequently occurring patterns. If the data matches the pattern, each line is then compressed into the pattern encoding, and the additional data is to decompress the word. As this method can substantially increase latency, they utilize an adaptive scheme that nullifies compression if the latency from decompression outweighs the benefits of the compression.

These compression and decompression latencies can be mitigated to an extent by utilizing BDI compression [3], with relatively modest hardware complexity. The idea behind BDI compression is that for many caches, the values stored within the cache line have very small differences, and as such a cache line can be represented with a base value and array of differences. Thesaurus and 2DCC demonstrate improvements to last-level cache (LLC) efficiency while reducing hardware overhead. 2DCC exploits two dimensions of redundancy, inter-block and intra-block, to enhance cache compression efficiency and performance [2]. Thesaurus builds upon 2DCC to allow for dynamic clustering at the hardware level with locality-sensitive hashing (LSH) [8] to produce the same hash for similar blocks where a unique code is generated and used to identify the blocks [8]. Only the difference is stored if other memory blocks with the same code are already cached. Thesaurus will then evict clusters that are not useful for adapting to changing workloads and form new clusters over time.

III. Methodology

A. Setbacks

The first iteration of this project sought to run an instance of Thesaurus/2DCC on a core similar to an i5-750 intel processor, with simulation occurring in ZSim. This action item proved to be prohibitively difficult due to a lack of detailed documentation throughout the research papers [1], [2], repository "README" files, and source code itself. The PARSEC benchmark test suite was the first choice for analysis, due to its high number of memory accesses and ability to run a few programs in parallel with varying cache access patterns [11], which would substantiate the effectiveness of compression on multiple cores.

The most notable difference between ZSim and Thesaurus lies in their utilization of intel's Pintools library. ZSim does not concern itself with the specific data values stored within its caches, though it does allocate the memory and write to the correct locations. Thesaurus leverages certain pin functions, specifically PIN_SafeCopy(), to read the values contained within the caches of ZSim. Utilizing the same functions in the BDI cache should grant access to the values in ZSim's caches.

Configuring ZSim was easier, with the caveat that it does not natively support cache compression. The ZSim [9] simulator is geared for researchers interested in conducting multiple realistic processor simulations in a short time, making it ideal for the research conducted by Ghasemazar *et al.* [1], [2]. Unfortunately, as the documentation is sparse, this required a significant amount of time and attention dedicated towards learning the ZSim environment.

B. Realization

Due to time constraints and issues with the obscurity of the chosen simulator, the scope of the project was changed to the more attainable goal of implementing the BDI cache compression algorithm on a basic core. By narrowing the scope of the project, confounding variables such as multi-core performance should be mitigated. Thus, the decision was made to rewrite parts of ZSim to allow for cache compression by cross-referencing the Thesaurus and 2DCC repositories with the ZSim repository. Additionally, PARSEC was determined to be low priority until cache compression was implemented. Instead, testing would be carried out with snippets of C++ code, which allows access patterns to be completely predictable.

The most simple processor supported by ZSim is a single wimpy core with an L1 cache split between data and instruction, and an L2 cache. This processor was modeled in ZSim [9], an x86-64 multi-core simulator, and performance

parameters were analyzed through micro-benchmarking. Given the increase in time overhead associated with the compression and decompression of frequently accessed data, the last-level cache was chosen for modification.

C. Base Delta Immediate Compression Algorithm

For this project, the team integrated BDI cache compression into an instance of ZSim. Compression was obtained by passing cache lines through a buffer before allowing them to be entered into the cache. This buffer utilizes a BDI algorithm to determine if the cache line can be compressed, and carries out the compression if possible. For this project, the BDI compression was specified to be 8Δ1. This indicates that each entry and base are 8 bytes in size, and the maximum tolerance between entries and the base (defined by Δ) is 1 byte.

Cache lines in this implementation have a line size of 64B, corresponding to eight 8-byte entries. Once a cache line of 8B values (V_1, V_2, ..., V_7) from the buffer is found, the next step is to verify compressibility for the chosen specification (8Δ1). The first value in the cache line will act as the base (B_0) with which the other values will be compared. If all of the remaining values are within the limit of the byte difference (Δ of 1, 2, or 3 bytes), then the line can be compressed. If one value in the cache line has a difference greater than the predefined Δ, then the line will not be compressed.

The compression method itself involves finding the difference between the 8-byte base and each other 8-byte entry in a verified-as-compressible cache line and representing it as a 1-byte base offset (Δ). The new compressed cache line will be stored as the base B_0 with each consecutive delta Δ_n stored sequentially, in place of the original line with B_0 followed by each 8-byte entry. The tag line will be stored in a tag array, which notably requires additional pointers compared to an uncompressed cache. This method can be modified to function for any specified delta, or difference, as shown in Fig. 1 where Δ is 3.

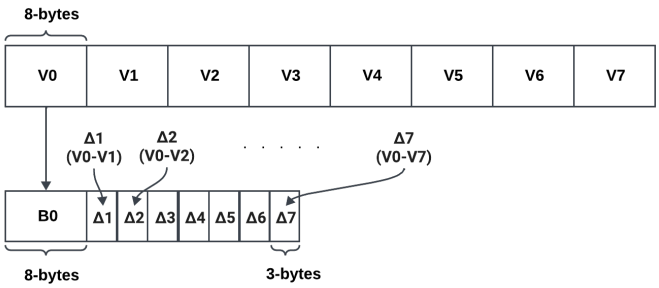

Fig. 1. BDI Compression with 8B Base & 3B Δ.

If implemented correctly, the BDI cache should function identically to a cache with increased storage capacity. Higher-capacity caches feature both additional data and additional tags. The BDI algorithm allows for compression of data within the cache, and additional memory was allocated for tags to properly index the cache. The amount of additional memory required for these tags must reflect the best-case performance scenario of the BDI cache; e.g. with a maximum compression ratio of 4:1, each cache line (regardless of compression) will have four tags associated with it instead of one.

Replacement policies are arguably the most complicated aspect of BDI compression. Due to a compressed cache featuring multiple line sizes, a careless replacement policy may result in the eviction of many compressed lines in the event that all "least recently used" lines are dispersed throughout different data cache lines. This can be remedied by utilizing a custom replacement policy that considers the size of the data being inserted and prioritizes evicting blocks with lower compression ratios; e.g. evicting a single 8Δ3 instead of two 8Δ1 blocks.

Another challenge associated with BDI compression is maximizing the amount of data stored per line during runtime. If possible, a compressed cache with a line size of 64 should fill all 64 lines at all times. This is difficult during runtime due to the constant nature of evictions, which results in data being sub-optimally allocated. The proposed solution is to periodically refactor the cache; this allows data to be entered into the cache as tightly as possible as shown in Fig. 2.

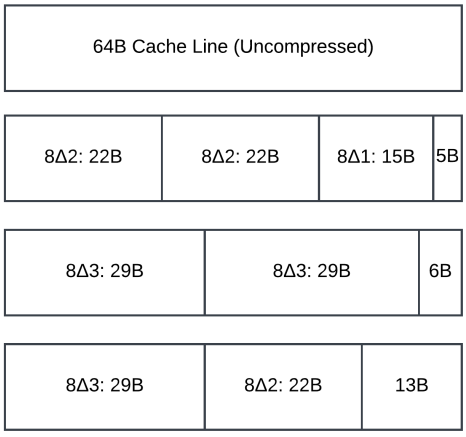

Fig. 2. Various degrees of compression efficiency.

To decompress the cache line, the compression process is reversed. The individual Δ_n, where n represents the number of the delta in sequential order with relation to the others, is added to the base value in order to obtain the original entry and this process is repeated for the entire cache line. This decompression is modeled with a delta of 3 in Fig. 3. BDI decompression is not computationally expensive; it can be performed in parallel and will add the latency of an adder to the critical path [3].

IV. Experimental Results

A. Experimental Setup

A ZSim configuration file used to implement the compression algorithm defined the system parameters as follows: a line size of 64B, a 64B L1 cache, and a 256B L2 cache. A lack of proper functional cache traces in ZSim presented a challenge – to address this, an output file was added to the cache access

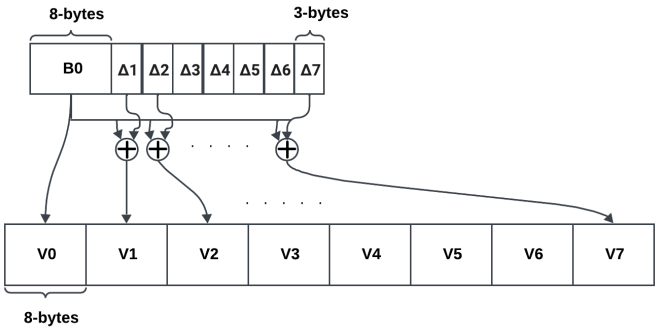

Fig. 3. BDI Decompression with 8B Base & 3B Δ.

function in order to track values accessed during runtime. Similarly, a custom microbenchmark was utilized to allow for a more precise view of cache access behavior. The open source code of the ZSim processor simulator was modified to test this created cache, using Data and Tag arrays defined for the cache to later analyze access patterns, and replacement policies were updated to track the movement/eviction of entries in the cache line.

The replacement policy framework was kept fairly similar to ZSim's original setup. However, to handle proper data replacement, a Data Replacement Policy class was added, which includes a new array of boolean values and returns a time stamp of the last time a data block was used and its validity. Each boolean value indicates whether data is valid and is to be modified by different functions. For instance, the update() function in the original replacement policy only updated the time stamp of the block, whereas the modified function sets the indexed boolean value in the valid array to true.

B. Results and Analysis

Table I portrays the different values observed for performance parameters in both the 8Δ1 cache compression method implemented for analysis and the dynamic cache clustering methods discussed in Thesaurus [1]. Parameters used for comparison include cache compression ratio, hit and miss rates, workload runtime, and cache access latency. The values recorded for Thesaurus' hit and miss rates are for a 512-entry (33KB) cache, where the miss rates are evaluated over all benchmarks. The corresponding values for the 8D1 method have been scaled accordingly, though hit rate was unable to be measured. The compression ratio of Thesaurus, which averaged 2.25:1 for last-level cache (LLC) working sets, could be compared to that of the 8Δ1 method in a proper implementation. A direct comparison would analyze how Thesaurus' compression behaved during the same micro-workloads utilized for the team's ZSim testing. On the other hand, BDI's hardware overhead should introduce lower relative delays to the overall access latency; the compression identification, as well as compression/decompression can occur on adders, shift registers, and or gates, most of which are in parallel .

TABLE I
8Δ1 COMPRESSION VS. THESAURUS

Compression Method	**Performance Parameter**			
	Ratio	***Hit Rate***	***Runtime***	***Latency***
Thesaurus	2.25:1	94.8%	27.2% faster	5 cycles
8Δ1	1:1 (non-BDI)	N/A	3 ms	N/A

V. CONCLUSIONS AND FUTURE WORK

This project details the merits of cache compression in modern processors. It proposes an implementation of base-delta-immediate cache compression, specifically 8D1, and the hypotheses proposed, such as the run time performance of the BDI cache and compression ratio, were confirmed by the analysis through ZSim.

These cache compression algorithms would be best studied in a simpler simulator, as ZSim is too robust for a single instance of cache compression, therefore introducing more problems than solutions during the implementation process. A single core RISC-V processor simulator such as Chipyard would have allowed more time to be dedicated towards the BDI compression algorithm, rather than learning the details of the simulator.

The implemented work can be further improved through an interpretation of BDI compression utilizing multiple deltas that encompass a range of differences rather than a single-constant delta. An example of this would be implementing deltas 1-3, where cache lines with differences of up to 3 bytes between entries would be considered compressible, and compressed data would be locally grouped with regards to the delta magnitudes. This would improve the rates of compressibility, and therefore meet the goals of cache compression more precisely, but introduces complications in replacement policies and loss of relevant data with eviction of least frequently used compressed data. Additional follow-up research related to cache compression should measure the impact dynamic clustering has on silicon area. For these purposes, Cacti or Verilog can be used to quantify the size of the hardware required to implement clustering, as well as the additional power consumption during workloads with high compression ratios and high decompression rates.

REFERENCES

[1] A. Ghasemazar, P. Nair, and M. Lis, "Thesaurus: Efficient Cache Compression via Dynamic Clustering," in Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Lausanne, Switzerland, March 2020.

[2] A. Ghasemazar, P. Nair, and M. Lis, "2DCC: Cache Compression in Two Dimensions," in Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France, March 2020.

[3] G. Pekhimenko, V. Seshadri, O. Mutlu, M. A. Kozuch, P. B. Gibbons, and T. C. Mowry, "Base-Delta-Immediate Compression: Practical Data Compression for On-Chip Caches," in Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT), Minneapolis, MN, USA, September 2012.

[4] T. Tian, M. Feng, and D. A. Jiménez, "Last-Level Cache Deduplication," in Proceedings of the 27th International Conference on Supercomputing (ICS), Eugene, OR, USA, June 2013.

[5] A. Bouchachia, "Dynamic Clustering," Evolving Systems, vol. 3, pp. 133–134, August 2012.
[6] A. R. Alameldeen and D. A. Wood, "Adaptive Cache Compression for High-Performance Processors," in Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA), Munich, Germany, June 2004.
[7] M. Zhang, K. Asanović, and B. C. Catanzaro, "Frequent Value Compression in Data Caches," in Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Monterey, CA, USA, December 2000.
[8] M. Ghayoumi, M. Gomez, K. E. Baumstein, N. Persaud, and A. J. Perlowin, "Local Sensitive Hashing (LSH) and Convolutional Neural Networks (CNNs) for Object Recognition," in Proceedings of the 17th IEEE International Conference on Machine Learning and Applications (ICMLA), Orlando, FL, USA, December 2018.
[9] D. Sanchez and C. Kozyrakis, "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems," in Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA), Tel-Aviv, Israel, Jun. 2013.
[10] D. R. Carvalho and A. Seznec, "Understanding Cache Compression," ACM Transactions on Architecture and Code Optimization, vol. 18, no. 3, pp. 1–27, June 2021.
[11] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT), Toronto, ON, Canada, October 2008.