Enhanced Base-Delta Compression with Data Splitting and Memory Pooling

Aditya Bhandaru (akbhanda@andrew.cmu.edu) Gennady Pekhimenko (akbhanda@andrew.cmu.edu)
Onur Mutlu (akbhanda@andrew.cmu.edu)

Abstract

Recent literature on cache compression has shown great potential for increasing the effective cache capacity on chip. Specifically, a technique called Base-Delta (B+Δ) compression has presented excellent compression (about 1.4X) and improvements in overall performance. However, B+Δ suffers from poor compressibility when adjacent data in memory have a high range in value.

We show here, as proof of concept, that existing techniques such as ***Data Splitting and Memory Pooling can enhance B+Δ compressibility*** *for data in memory. Our simulations over various micro-benchmarks show that B+Δ with pooling results in an 8% reduction in MPKI, and a compression ratio of 2.6X over the baseline.*

1. Introduction

1.1. Background

The memory bottleneck is a well known problem in computer architecture. Caching has become a standard for alleviating contention for data, the bus, and memory. As we trend to more cores, more applications, and larger computing problems, there is a much greater demand for data. Simply scaling cache size to compensate is too expensive, both in power and chip area.

Data compression in the cache is a promising alternative to increasing effective on chip cache capacity. For the same physical cache space, we can store more blocks per set. In general, compression algorithms look for patterns in data to exploit. Therefore, they are very sensitive to how data is laid out in memory and the kind of data a program manipulates.

The ideal cache compression implementation would be fast, simple, and offer a high compression. These design points are largely at odds with one another. For example, many ideas from older literature on cache compression suffer from either poor compression or incur high hardware complexity or long decompression latencies.

Why is fast decompression more important than fast compression? Decompression is on the critical path for a read. In order to supply the requested word, we must decompress the cache line. During a cache fill, compression can occur in the background while we bypass the requested word.

1.2. Motivation

Recently, a paper on Base-Delta-Immediate compression [2] suggested technique called Base+Delta (B+Δ) compression. Base-Delta-Immediate (BΔI) compression is the final iteration of the technique. In comparison to preceding work on cache compression, it hits the fast-simple-high-compression trifecta nicely.

Their work yielded largely positive results, making B+Δ compression worthy of further study and improvement. In particular, we observe that improving data layout in memory to leverage B+Δ compression in hardware may result in substantial performance gains. This paper focuses on two existing techniques: data splitting and memory. As a proof of concept, we show that applying these transformations to programs running on B+Δ architectures will realize considerable gains.

2. Previous Work

2.1. Cache-Aware Data Placement

There is a substantial body of work on optimizing the arrangement of data in memory to improve temporal and spatial locality. Notably these approaches often involve compiler hints or runtime directives to a runtime library [6] or allocator [1] for cache conscious data placement.

Further studies on dynamic analysis using profiling [3] and even runtime structure splitting [7] were introduced to reduce programmer effort. The latter technique offers great adaptability to the dynamic needs of the program, but incurs some overhead due to safety checks.

The idea of cache conscious data placement is not new. *However, none of these previous works target systems that implement B+Δ cache compression.*

2.2. Base-Delta Compression

B+Δ compression [2] leverages the observation that for many cache lines, the values contained have a low dynamic range. That is, they could be encoded using a common base-value and a number of much smaller delta-values.

B+Δ compression does have its shortcomings. Every access and fill incurs and additional decompression and compression latency penalty respectively. Furthermore, certain benchmark programs manipulate data with very high dynamic range, particularly pointer based algorithms. B+Δ compression performs poorly here, and does not justify its inherent latency overhead.

For architectures that implement B+Δ cache compression, a cache conscious data placement policy would then place values with low dynamic range together, therefore minimizing the deltas. We anticipate that previously, poorly performing benchmarks would greatly benefit from such a data-layout transformation.

2.3. Data Splitting and Memory Pooling

We investigate two such mechanisms that place similar values together in memory: **data splitting and memory pooling**. As a proof of concept, this paper does not address the non-trivial implementation challenges that face the splitting and pooling of data. There is however, compelling work in the area such as MPADS [5] and Forma [4].

The basic idea is to analyze an object (perhaps a struct), and determine whether you can *safely* allocate its members separately. This is the splitting phase. Next, we determine where we want to allocate the members. This is the pooling phase. In relation to B+Δ compression, we want to pool similar values together–allocate them adjacent to or near one another.

Determining whether or not values are *similar* can be done in a number of ways. Static type analysis, profiling, and runtime analysis are all possibilities. Although, this is not a trivial problem.

2.4. Why B+Δ and not BΔI?

This paper focuses on B+Δ compression over BΔI compression for a couple reasons. First, we suspect that applying techniques such as data splitting and memory pooling will alleviate many of the low compressibility cases that B+Δ suffered on the benchmark tests. The second is simplicity. Encoding cache lines with only one base requires less metadata in the tag store and simpler hardware.

3. B+Δ with Splitting and Pooling

Our idea is simple. Arrange data in memory optimally for BD compression. Data splitting and memory pooling techniques can achieve this by placing data with similar value-behavior adjacently in memory.

3.1. The Compression Algorithm

The key premise behind B+Δ compression is that cache lines with low dynamic range are abundant during runtime. That is, the mathematical differences between the word values are small compared to the size of the word.

When this is the case, a cache line can be represented as a single *base* value and an array of much smaller *delta*-values (often half the size). As a result, we can encode the data in fewer total bytes. It follows that cache lines with high dynamic range cannot be compressed with B+Δ encoding.

Benchmarks using pointer manipulation algorithms with large objects often suffer from poor compressibility. For instance, a linked-list traversal program may have many node structs allocated next to one another in memory. If each node consists of a flag, a counter, and a pointer, we might see boolean, integer, and pointer values interleaved in memory. Generally, a cache line with these types of interleaved values is incompressible.

3.2. Low-Compressibility Cases

Let us start with a favorable case for B+Δ compression. For example, placing pointer-like values with other pointer-like values is preferable to having a pointer next to a boolean flag. The mathematic difference between a pointer-value and a boolean-value is obviously some large delta, which is useless for compression. Less obvious, perhaps, is that the most significant bits of pointers are usually the same. If the first 32 bits are the same, we can use 4-byte deltas to encode a string of pointers relative to some 8-byte base (assuming a 64-bit architecture).

However, benchmarks with heavy pointer manipulation and large objects often suffer from poor compressibility. For instance, a linked-list traversal program may have many node structs allocated next to one another in memory. If each node consists of a flag, a counter, and a next-pointer, we might see boolean, integer, and pointer values alternating in memory. Generally, cache lines with such interleaved values are incompressible.

3.3. Data Splitting and Pooling

For such programs, we want to allocate objects such that we maintain spatial locality of similar-valued members. More elaborately, **split** an object up into its respective members. Allocate space for those members based on what kinds of values they hold–these decisions may be during compile time or runtime, depending on the pooling implementation. Members with similar value-types should be **pooled** (allocated) together.

Expanding on the example from the previous section, we might have separate memory pools for flags, counters, and pointers. This creates contiguous regions of memory with much lower dynamic range. Now, when the next-pointer member of a node is read, the likelihood that the resulting cache fill is B+Δ-compressible is far greater.

The key observation here is that B+Δ compression can strongly leverage any data transformations that create regions of low dynamic range.

3.4. Caveats

Some simplifications were made to the above examples for that sake of brevity. For example, a long string of boolean flags in memory is seldom B+Δ-compressible. This is because B+Δ compression operates at 4-byte and 8-byte granularities. A string of 32 or 64 booleans has high entropy, even for biased distributions for true and false values. Interpreting these strings as integers and computing the differences may result in very large deltas, and low compressibility.

References

[1] S. John B. Calder, K. Chandra and T. Austin, "Cache-conscious data placement." *Proc. ASPLOS-VIII*, 1998.

[2] O. Mutlu M. Kozuch P. Gibbons T. Mowry G. Pekhimenko, V. Seshadri, "Base-delta-immediate compression: Practical data compression for on-chip caches," *PACT*, 2012.

[3] Chris Lattner and Vikram Adve, “Automatic pool allocation: Improving performance by controlling data structure layout in the heap.” *PLDI*, 2005.
[4] Y. Gao R. Silvera J. Amaral P. Zhao, S. Cui, “Forma: A framework for safe automatic array reshaping,” *ACM*, 2007.
[5] J. Amaral Y. Gao S. Cui R. Silvera R. Archambault S. Curial, P. Zhao, “On-the-fly structure splitting for heap objects,” *ISMM*, 2008.
[6] M.D.Hill T.M. Chilimbi and J. R.Larus, “Cache-conscious structure layout.” *PLDI*, 1999.
[7] PC. Yew J. Li D. Xu Z. Wang, C. Wu, “On-the-fly structure splitting for heap objects,” *ACM*, 2012.