Project Proposal
Cody Martin and Adu Bhandaru

1. Problem

The memory-bottleneck is well known problem in computer architecture. In general, we try to alleviate contention for memory, the bus, and data using caching, smart request scheduling, and many other techniques. Many of these methods maximize system throughput by providing higher quality of service and hiding more latency.

Using Base Delta Immediate compression (BDI), suggested by Gennady et. al., alleviates some of the penalties incurred by capacity and conflict misses. By compressing block data in the cache, we increase our potential cache utilization by the inverse of the compression ratio. This effectively increases the capacity, leading to fewer capacity misses. Similarly, lines in each set may occupy less space, possibly allowing greater associativity in some sets.

Caching is the bread and butter method for exploiting spatial and temporal locality in the memory accesses of an application. A cache miss usually incurs a high latency penalty and can stall the requesting processor, degrading system performance thusly. We categorize misses into three types: compulsory, capacity, and conflict. Capacity misses occur when the working memory set (data size) of the program exceeds the size of the cache. Conflict misses occur when blocks that map to the same set have interleaved accesses.

The primary drawback to BDI is the additional compression-decompression latency as data enters and leaves the cache. If the range of the data values in each block is large, we get very poor compression. High range in values typically occur when different data-types are stored adjacently in memory. For example, consider an array of structs each with pointer and an error flag. Because pointer and flag data are interleaved in memory, BDI compression performs poorly because of the high range in values.

We believe we can increase the effectiveness of BDI on these common-case data arrangements to make the technology more viable.

2. Novelty

There have been many papers on memory compression. We found BDI compression to have compelling advantages and use cases. However, most ideas on memory compression focus on modifying the underlying hardware.

Furthermore, the benchmark performance tests were performed with applications that are written and built without knowledge of the underlying compression techniques implemented in hardware. Our idea explores the possibility of exploiting this knowledge in our compilers and memory allocators to improve the compressibility of data. As far as we know, this has never been done before.

3. Idea

This idea is simple: optimize the arrangement of data in memory for BDI compressibility in the caching microarchitecture. To do this, we will need both application and compiler support. More specifically, we think certain modifications to the memory allocator library (such as malloc) will allow us to rearrange data favorably for BDI compression. And in turn, the compiler must we aware of this rearrangement and do the necessary pointer transformations on accesses to the data. The idea here to make these features transparent to the programmer, and maintain their view of memory.

We find that high ranges in data values occur when different types of data are adjacent. Consider calling malloc to allocate space for an array of structs. Each struct contains a flag (might contain the value 0 or 1) and a pointer. Normally these flag and pointer data values would be interleaved in memory and BDI compression would perform poorly.

Now consider a memory allocator memory allocator that had some information about the struct - not just the total size, but also the fields it has. From the above example, we could allocate memory for the structs in 2 arrays: one for the flags, and another for the pointers. In doing so we have decreased the data value ranges drastically within each array's contiguous memory. For instance, the delta between one flag and the next is at most 1, as opposed to the difference between a pointer and a 1 (orders of magnitude larger). This is great for BDI.

To identify similar values, we could use either static analysis tools or profiling. With regard to static analysis, we can do something as simple as type checking and ensuring we place similar data-types adjacently in memory whenever possible. Profiling on the other hand requires a representative set of inputs to run on an application during compilation. It could monitor data fields and record each one's respective range.

To support this memory rearrangement we will also need compiler support. For example, a simple base-index-scale-offset style load instruction is no longer sufficient to access different members of a struct. Instead, we need to compute the base of the array for this particular member (within our contiguously allocated memory) and then return the value at the given index. The stride here would depend on the type of the member. We think the compiler should do these transformations behind the scenes.

Finally, we note that there are certain to be cases where data

rearrangement will break a program. We intend to identify these areas and usages, and optimize around them conservatively.

4. Hypothesis

We expect basic implementation of this idea to realize overall system performance gains. This assumes all applications running on the system use our memory allocator library, and have been compiled with the modified compiler.

More specifically, we expect to reduce the overall cache miss rate by some substantial amount. By improving BDI compressibility, we are enabling better cache utilization and more associativity. Essentially, we can fit more blocks in the cache. The resulting number of capacity and conflict misses should decrease thusly.

On a lower level, our data rearrangement scheme places similar data next to each other. A block that comprising similar data has high potential for BDI compressibility. The range in values is small and the deltas can be expressed in a much smaller space. This frees up space for more blocks in each set, and more therefore more blocks can fit in the cache.

Because we have more addressable data in the cache, we expect a significant reduction in misses. This reduction in misses minimizes processor stalling and improves system throughput. Without data rearrangement, we expect the benefits of BDI compression to be less substantial.

Finally, we acknowledge that the different data-structures and algorithms used in applications will have different sensitivities to our optimizations. This includes basic system parameters such as 32-bit vs. 64-bit.

5. Methodology

The initial step will be a proof of concept. We will write a small memory simulator that implements BDI compression in the L1 cache. It will be modular such that we can feed it a trace of memory accesses and get hit-miss-eviction data on output. We will check this for correctness - both for the BDI implementation and the fidelity of the output data.

Second, we will then write a small program (kernel) that allocates an array of structs and performs some simple computation over them. We will profile the memory accesses and look for patterns. We will then feed this trace to our BDI simulator and look at the hit-miss-eviction data. We may do this for a number of small programs.

Third, we will modify the above program such that we allocate and place the struct data in a manner consistent with the ideas we have presented. In this simple case, we will just separate similar data into their own arrays. We will run this program on the same BDI simulator and compare the hit-miss-eviction performance to that of the previous program which did not implement data rearrangement.

If the above steps result in considerable improvement, we will continue. The next step is to generalize our data rearrangement scheme, and implement it as part of a memory allocator library. We would then test our programs (now using our allocator) on the BDI simulator for performance. Note that the compiler is still unchanged, and there is still a considerable amount of work we are doing by hand for the purpose of testing (such as the pointer transformations).

The next step is to implement BDI-compression as part of a cycle accurate simulator. Once this is complete and has been checked for correctness, we can move forward with running our test binaries and comparing cycle counts with the full memory hierarchy implemented. The first binaries we will test are those created above.

Before we can run our simulator on traditional benchmark programs, we would like to automate the pointer transformations. We can do this by augmenting the LLVM compiler. Once we have correctly implemented a pointer transformation module in LLVM, we can rebuild all our benchmark binaries, and run them on our cycle accurate BDI simulator.

6. Plan

The following list are some general goals we wish to meet as we conduct our research. These steps are largely derived from our methodology.

Milestone 1: Write a memory simulator that implements BDI compression in the L1 cache. Test it for correctness. Create a couple of simple programs to run on it and look for patterns in the hit-miss-eviction data.

Milestone 2: Modify the programs in our existing test suite to use data rearrangement. We will do this by hand. Check that all other program semantics are the same, and then run on our BDI simulator. Compare these results with those from the unmodified tests in the suite.

Milestone 3: Implement a memory allocator that generalizes the algorithm for data rearrangement. Use this memory allocator in the test programs from milestone 2 and verify the results. Also, implement a BDI compression as part of a cycle accurate simulator. Run both versions of tests in our test suite on the cycle accurate simulator and compare results.

Milestone 4: Hopefully we make it this far. Modify the LLVM compiler to do the pointer transformations for rearranged data. Build unmodified tests from milestone 1 with the the modified LLVM compiler and verify for correctness. Finally, build a few traditional benchmark applications using the modified compiler and observe their performance on the BDI cycle accurate simulator.

Final Report: Summarize the basic ideas of data rearrangement. Show how simple programs and benchmarks perform using just BDI compression. We can show the data for both the simple memory simulator and the full blown cycle accurate simulator. Then show comparisons for how these programs and benchmarks perform using data rearrangement. Ideally we will have a compelling case for using data rearrangement on systems that implement BDI compression.