Cache compression on inclusive caches

—

L2$ Dictionary Sharing within the Sniper x86 simulator

Guillaume Didier and Daniel Stiffler

May 7, 2018

1 Introduction

For the past sixty years, Moore's Law has graced the semiconductor industry with faster and more efficient devices each generation. Microprocessors and other products throughout the computing stack have benefited immensely from these gains. Historically, however, memory has struggled to keep up. As a result, modern processors rely heavily on caches and high-speed interconnects to feed their execution units. Strapping on these complexities to the memory hierarchy, naturally, comes with a steep power bill. Computer architects have experimented in the literature with numerous ways to mitigate the downsides of caching, such as implementing more accurate policies [JBB+10]. Recently, in-cache data compression was proposed as a solution to many of these common pathologies in computer memory systems.

In this study, we implemented a compression algorithm using an x86 simulator to investigate its impact on a model of a real-world CPU. Our original plan was to improve upon a particular scheme described in the literature [PS16]. Due to unexpected complexities in the software, however, we scaled down the algorithmic side of our project to get a better view of the consequences of inclusive caches and scheme-specific restrictions on compressibility.

We thus show that compressed caches are unsuitable for use in an inclusive cache hierarchy without significant modifications.

2 Motivations

At its core, cache compression is about allowing memory arrays to store more data within a given area, at the expense of read and write latencies. This technique breaks down the traditional *memory wall* by providing the illusion of larger and more inclusive caches where latency is not particularly important – L2$ and L3$ are prime candidates for compression. Compression algorithms have additional benefits for power as well. Many implementations permit lines with spatial and content locality to be prefetched for free. Finally, and most importantly for scaling, compression brings more data closer to the cores, thereby reducing traffic out to the increasingly-expensive levels of the memory hierarchy. There are significant hurdles to overcome, yet the premise of data compression is sound and has been successfully applied in other aspects of computer systems.

The idea of compacting multiple cache lines has been an active research topic since the introduction of CPACK+Z in 2010 [CYD+10]. In the subsequent years, several alternate schemes have emerged, some favoring efficiency and others speed (among other considerations). One of the universal questions, nonetheless, is how to extract enough content locality to encode – without loss – a particular line in fewer bits than it started with. Early methods, like CPACK+Z [PSM+12] and $B\Delta I$ [PSM+12] focused on recognizing primitive patterns in the data using complex decision trees; these algorithms produced modest gains in some workloads, but ultimately suffered in

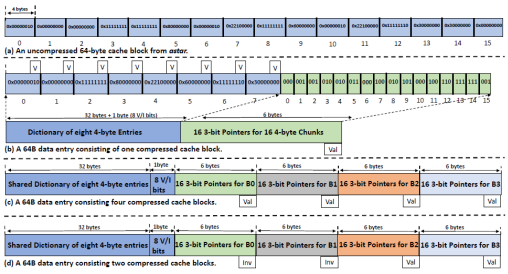

Figure 1: Compression using inter-block data locality, from [PS16]

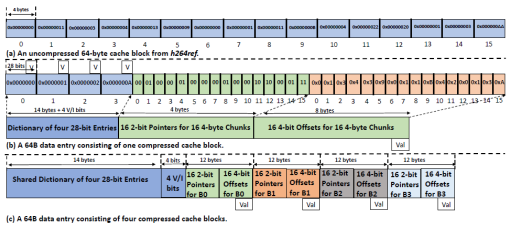

Figure 2: Compression using inter-block MSB locality, from [PS16]

general-purpose applications. State-of-the-art methods, such as DIctionary SHaring (DISH) [PS16], on the other hand, consider *Superblocks* of up to four consecutive cache lines and re-encode the values as 2- or 3- bit pointers to a shared dictionary.

As mentioned previously, we aimed to tweak the recent state of the art compression algorithm, DISH, to see if it is possible to obtain more locality through shuffling cache lines around. But we eventually discovered that our simulator had a irreconcilable difference with the one used by Panda & Seznec (2016): they had been using exclusive caches (without great details why), while our simulator only supported inclusive caches (correlated with the fact that Intel only uses inclusive caches) [PS16]. This turned out to be a crucial oversight that crippled the algorithm despite our best attempts at improvement.

3 Compression Algorithm

The study implements the DISH algorithm described in *Dictionary Sharing* [PS16], of which we will recall here some of the most prominent features.

The figures above, reproduced from the DISH paper [PS16], illustrate the cache compression schemes that we built upon for this project. Scheme-I (1) works by encoding repeated 4-byte values within a cache block as 3-bit pointers to slots in an 8-entry dictionary. Studies throughout the cache compression literature have demonstrated that this approach works very well for compacting uninitialized structures, small integers, and occasionally global variables. However, the downside is that the presence of pointers in a cache line – usually localized to a region of memory with an offset – can quickly render blocks incompressible: the upper bits may be shared, but there is not enough content locality to leverage the dictionary.

One way to circumvent this problem is to instead consider patterns in the upper and lower bits separately, when the line is divided into chunks. The second (2) figure illustrates this approach, referred to as Scheme-II in the paper. In this setting, the shared dictionary is reduced to 4 entries and the now-free bits used as offsets to the encoded values: the cache line is still split into 4-byte chunks, but the upper 28 bits are substituted for a 2-bit dictionary pointer and the lower 4 bits are moved to a different part of the line for storage.

There is a lot of work on behalf of the cache controller to keep track of how data are encoded, but the key idea here is that it is free to select whichever strategy is most efficient. Otherwise, the fall-back is to leave the block uncompressed at no additional cost over a conventional cache.

In order to utilize the empty space left in compressed cache blocks, DISH allows up to four adjacent dictionaries to merge and form a *Superblock*. This can only occur for lines with consecutive addresses, so that additional storage requirements for the tags are reduced, and may lead to some special behavior on writes and insertions.

The main differences between our paper and the original implementation is how we decide which scheme to use when both are potentially valid. Our contribution entails an arbiter at the level of cache sets, which tracks how many *Superblocks* are currently compressed with each scheme, and chooses whichever one that has the greatest num-

ber if both scheme are possible. Should a write or insertion render the block incompressible with the initial scheme, the arbiter can allows for on-the-fly switching as well as dictionary entry pruning for allocated, but unused entries. Those two possibility can be toggled with a boolean setting, but we made our simulation with both of them enabled.

Lastly this compressed cache is used differently. We continue using compression in the L2 cache (of the traditional 3 level of cache hierarchy of recent processors), but we work with inclusive cache instead of the exclusive caches of the original paper. (This is due to Intel exclusive use of inclusive caches in their systems, hence Sniper support for only inclusive caching). This last difference will prove to be the cause of wildly different result in our simulations

4 Implementation

4.1 The Sniper x86 Simulator

Our implementation was built using the Sniper multi-core simulation framework [CHE11]. This tool is based on an interval core model and leverages the GRAPHITE [MKK+10] engine and Intel PIN [LCM+05] instrumentation to approximately simulate x86 programs. In our test configuration, Sniper modeled a full memory hierarchy, consisting of address translation, L1$ through L3$ caches, and MESI coherence protocols. The core execution was divided into regions of roughly constant CPI bounded by the presence of miss events like branch mis-predictions, cache and TLB misses, serialization instructions, etc. Because the analytical model only derives timing for each interval and not the actual data, we incorporated hooks into the cache controller objects that allow them to either retrieve real data from the simulated executable or pass around internal buffers containing potentially-stale copies.

4.2 Setting up Sniper

We faced some initial difficulties building the Sniper simulator, as it required older versions of software like GCC 4.8 and PIN 71313. Fortunately, we were able to solve these issues by switching our development over to virtual machines (VMs) running Ubuntu 14.04 LTS. We also faced issues with its extremely poor documentation, which meant that we had to spend a lot of time sifting through the code to understand its behavior. Some of our worst surprises were discovering that the cache controllers bypass the actual cache objects to manipulate coherency states of lines they contain, also the data buffers are never filled with real data. For single-threaded applications, the program, PIN tool, and the cache simulator all reside in the same address space. In that case, the addresses that are used to index the cache are valid addresses of the corresponding data in the program, and after a bold cast and dereference of this pointer we can get the data we need. Despite our best efforts, unfortunately, this cannot be applied in multi-threaded programs due to violation of the memory model and possible data races.

4.3 Modifications to Sniper

In order to accurately model cache compression, we made extensive changes to the memory subsystem used by Sniper. We attempted to preserve as much of the cache controller and coherence engines as possible, in order to cut down on the project scope and maintain compatibility with other features. However, limitations in the way Sniper shuttles data between cache levels as well inflexibility of functional interfaces meant we had to rewrite essentially everything below the `CacheCntlr` class.

Our first significant overhaul was to the `Cache` class implementation (including several classes representing different internal organizations). In the original framework, the `CacheCntlr` objects issued the various kinds of load and store requests to `Caches` through a multi-purpose method, `accessSingleLine`. However, this functional interface did not provide a clean way to back-propagate evictions. This is not an issue for conventional caches, but writes to compressed caches may cause evictions on valid writes if the new data renders a line

incompressible with its current *Superblock*. We addressed this possibility by adding an optional parameter of type `WritebackLines` to the access and insertion functions of the cache. This parameter passed down, in both cases, to the underlying mechanisms of the class so that they can specify the address, metadata, and fill data for blocks that were evicted by the current operation. In DISH compression, evictions always happen at a *Superblock* granularity, so the parameter always has at least four free slots to hold writebacks. Finally, we made some additional modifications to the other `Cache` methods, but they were insignificant in comparison to the access and insertion functions.

Compressed cache objects also have a different set of rules when it comes to how they handle the core set of operations: LOAD, STORE, EVICT, INVALIDATE, and INSERT.

For access-type operations (LOADs and STOREs), the `Cache` first splits the address into its constituent parts, queries a vector of `CacheSet` objects for the requested line, and if it is present, executes the operation. From this point, LOADs proceed similarly to the original framework. STOREs work a bit differently. Sniper does not model real data out-of-the-box, so the `Cache` initiates a mux-ing process between the data that was passed in from the controller and the real program data; in the latter case, the object peeks into the executable by reconstructing a pointer and hopping across into the other address space. This approach effectively hides the complexity of cache functions behind a single function.

We also changed how the residency operations (EVICTs, INVALIDATEs, and INSERTs) are processed. The first two were largely untouched from the original implementation. The third, INSERT, was completely overhauled and shares some similarities with STOREs in our implementation. For instance, INSERTs may involve fetching real data when they are initiated by the Sniper Core, otherwise a `is_fill` flag can be set to indicate that the `Cache` should perform a fill from the next level up in the hierarchy. We also use a modified version of the least-recently-used (LRU) replacement policy described by Jaleel et al. (2010) called query-based-selection (LRU+QBS) to address performance and correctness issues that appeared some edge cases.

One of our goals through this project was to make it easier for future researchers to modify Sniper, so we spent a great deal of time cleaning up the aforementioned interfaces between the controllers and cache objects. Keeping with this spirit, we implemented the guts of data storage using a trio of modular classes: `CacheSet`, `SuperblockInfo`, and `BlockData`.

The first helper, `CacheSet`, sits directly below the `Cache` objects and handles the mechanical side of core operations. It contains a specific function for each task and provides the bridge through which the functional interface on the controller-side can interact with both data (BlockData) and meta-data (SuperblockInfo) storage. The latter two objects are responsible for implementing DIctionary SHaring compression through ask-then-proceed interfaces. For instance, sets need to check both their *Superblocks* and block storage objects using `SuperblockInfo::canInsertBlockInfo` and `BlockData::canInsertBlockData` before committing to an operation. There is a lot of complexity in these three classes that is not worth going into here. For all intents and purposes, these three classes form the compression engine.

We also cut out a lot of extra feature that we didn't use and that updating to use the new Cache interface would have been a useless loss of time. (We notably removed most eviction strategy outside of the LRU strategy we used)

4.3.1 Cache Creating Changes

We additionally had to add parameters to control our new features, as three boolean : `compressible`, `change_scheme_on_the_fly` and `prune_dish_entry`.

4.3.2 Adaptation to the controller

The controller had to be adapted to use the new multiple eviction interface, and thus issue the required recall and write-backs when necessary. We also had to build in support for bypassing levels of the cache where back-invalidated data could not be safely written.

4.3.3 Other Modifications

The TLB were modeled as a cache containing no data but with a line size corresponding to a page. We had to change its working in order for it to avoid consuming a wasteful amount of memory, and also make its use clear enough so as to avoid issues with the lack of data.

4.4 Known Limitations

The way Sniper implements write backs and some cache coherence transition causes us some trouble and still results in some bad performance issues.

The first case is when a line A is stored in L2$ compressed along with another line B. The line A is the modified to A' in L1$. Upon eviction of that line from L1$, it is possible that the A' line doesn't compress anymore in the cache. The cache thus has to insert A' in it, which may result in an eviction. If the victim line C is itself a line stored in L1$ and dirty the eviction of C result in a second write back C', in the same set. The way eviction is implemented in Sniper mean that the write-back occurs before eviction, which mean that insertion of C' in L2$ can actually result in line A being evicted and thus issue another recall of A', creating an infinite loop.

To control this phenomenon we changed the eviction policy of the L2 cache to LRU with Query-Based-Selection (LRU+QBS), that tries (up to 8 times) to find the least recently used line that isn't cache in L1 [JBB+10]. An extra parameter decides whether when none are found it should return a line that is cached in L1$ or simply return no line, causing a by-pass of the cache.

To prevent the infinite loop, the write back situation thus refuses to evict line that are cached in L1$ and prefers to bypass L2$. But this causes other issues :

- First case : another core makes a read exclusive request for a line A, modified to A' in L1$, compressed in L2$. This first causes write-back of A' in L2 and L3, then an invalidation in L1$ then L2$. But when the previous condition arises, the line is not stored in L2$, which violate the inclusive caching and thus isn't anymore in L2$ when the invalidation occurs. We removed some assertion, to make thing working, and the issue is only transient but this is a clear issue that the Sniper implementation of those operation make us break the invariant transiently.

- Second case : Line A is stored in L2$ and L3$, L1$ has a dirty A' version. An L3$ insertion result in evicting A, issuing a write back. Thus the L2$ cache tries to insert A' but fails. Thus after the write-back, L1$ and L3$ contain A', but L2$ doesn't, during the invalidation of those line, it similarly has an issue when invalidating the line in L2$. But in that there are no work around and thus results in crashes and/or hangs in the simulator.

These issue of cache coherence would not occur if the recall was simulated as packet exchanges with the eviction ad write-back occurring at the same time as L2$ could easily forward the L1 packet with the correct data and evict stale line A.

In addition to those, we also are limited by the hacks used to access cache content to single threaded programs and some short multi-threaded program that can luckily avoid the potential data races.

5 Experimental setup

Simulation was done on an Ubuntu 14.04 LTS VM which was allocated 4 cores on an Intel i7 6770k CPU and 8GB of DDR4 memory.

We used the Intel Gainestown 2.66GHz preset of Sniper with a slight modification to the Nehalem configuration file it includes to enable compression on the L2 cache.

Cache	Size	Ways	Evict. Pol.	Time in cycle	Location
L1-d	32KB	8	LRU	4	private
L2	256KB	8	LRU with 8 attempts QBS	8	shared with 2 cores
L3	8192KB	16	LRU	30	global

Table 1: Simulated CPU configuration

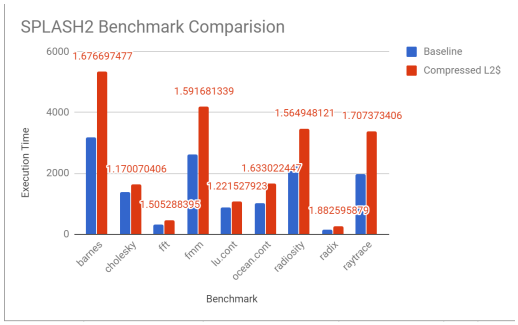

Figure 3: Performance results using Intel Gainestown CPU Model with L2$ Compression Toggled OnOff

We first ran the `fft` test included in Sniper. This test performs blocked fast Fourier transform with 65536 cache lines containing double-precision imaginary number. After another round of bug fixing we were able to run part of the SPLASH2 benchmark suite.

6 Experimental results

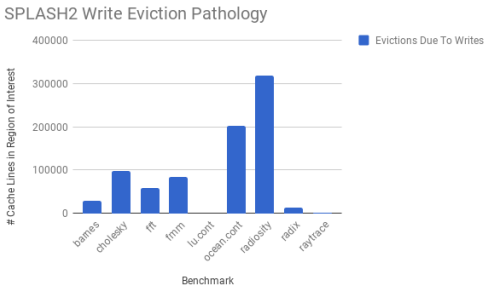

Figure 4: Evictions in L2$ caused by writes rendering the data incompressible using Intel Gainestown CPU Model

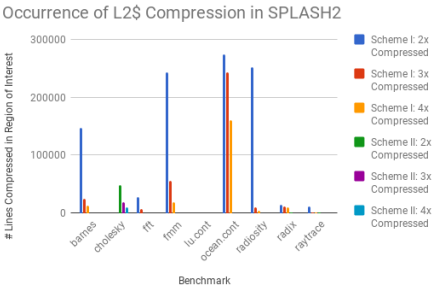

Figure 5: Occurrences of L2$ Compression using Intel Gainestown CPU Model

7 Conclusions and future work

The first result of this study is that cache compression can actually cause some nasty eviction patterns that, if implemented in an inclusive caching causes an excessive number of recalls from lower caches and thus degrades performance. This is demonstrated by our average 1.55x slowdown of the tested applications in Figure 3. As expected, writes that lead to evictions were a significant source of performance loss, in addition to other effects that were not well captured by our metrics. Cache compression may still have merits according to 5, since several applications exhibited high compressibility. Our contribution is therefore limited to demonstrating that DIctionary Sharing should not be applied to inclusive cache with the current state of the art.

Another conclusion is that cache compression studies require the following feature from a simulation framework, and that one should not attempt a study on a tool that doesn't provide these fundamental features, unless prepared to implement them:

- Modeling of the real data stored in the cache
- Support for arbitrary number of eviction, and eviction at unusual places.
- Ability for write-backs to by-pass a cache/be buffered when the cache is unable to insert this data.

Also we noticed that, as those issue are caused by modification and write-backs,

it may be worth investigating application of cache compression to caches containing read-only data.

References

[CHE11] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2011.

[CYD+10] Xi Chen, Lei Yang, Robert P Dick, Li Shang, and Haris Lekatsas. C-pack: A high-performance microprocessor cache compression algorithm. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 18(8):1196–1208, 2010.

[JBB+10] Aamer Jaleel, Eric Borch, Malini Bhandaru, Simon C. Steely Jr., and Joel Emer. Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (tla) cache management policies. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2010.

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, New York, NY, USA, 2005. ACM.

[MKK+10] Jason E Miller, Harshad Kasture, George Kurian, Charles Gruenwald, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. Graphite: A distributed parallel simulator for multicores. In *The Sixteenth International Symposium on High-Performance Computer Architecture*, 2010.

[PRH+17] Ardavan Pedram, Stephen Richardson, Mark Horowitz, Sameh Galal, and Shahar Kvatinsky. Dark memory and accelerator-rich system optimization in the dark silicon era. *IEEE Design & Test*, 34(2):39–50, 2017.

[PS16] Biswabandan Panda and Andre Seznec. Dictionary sharing: An efficient cache compression scheme for compressed caches. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[PSM+12] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillib B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. Base-delta-immediate compression. In *21st international conference on Parallel architectures and compilation techniques*, 2012.

[SSW14] Somayeh Sardashti, Andre Seznec, and David A. Wood. Skewed compressed caches. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[SSW16] Somayeh Sardashti, Andre Seznec, and David A. Wood. Yet another compressed cache. *ACM Transactions on Architecture and Code Optimization*, 13(3):1–25, 2016.