Evaluation of Adaptive Insertion Policies, as Described by Qureshi et al.

Dan Hullihen

Introduction

The goal of this project was to take an existing paper and attempt to reproduce its results using a different set of benchmarks. The paper selected was Qureshi's "Adaptive Insertion Policies for High Performance Caching" (Qureshi, 2007). In this report we will review the paper, the methodology used for reproducing the results, and the results that were found.

Background

A common cache organization is one that uses a "least-recently used" (LRU) replacement policy where the oldest block/cacheline will be chosen during the eviction process. Upon allocation, a cacheline is marked as "most-recently used" (MRU), requiring time for it to "age" and become the oldest and thus eligible for eviction. Qureshi explains if such a large number of cache lines actually never get used, they are reducing the available cache, effectively resulting in trashing as older blocks are forced out and new ones are allocated.

Through experimentation, they found a trend where there can be a significant number of cachelines that get allocated, but never subsequently used again during its lifetime. This general architecture is quite common, and ideally could be fundamentally changed. However, Qureshi acknowledges the fact that there is significant design and verification effort to reorganize an entire cache to address this problem. Thus, in this paper, they explore simply modifying the mechanism used to "insert" the allocated block in the first place. Specifically they look into the idea of marking a new cache block as "LRU" right from the beginning.

The authors go through a few insertion policies:

1) MRU-Insertion Policy (MIP)
    a. The original design
2) LRU-Insertion Policy (LIP)
    a. A simple modification where you always insert a cacheline as LRU
3) Bi-Modal Insertion Policy (MIP)
    a. A compromise between LIP and MIP where you "randomly" use the MIP, else LIP
4) Dynamic Insertion Policy (DIP)
    a. A method that uses set-dueling to decide which insertion policy to use

They evaluate these implementations using SPEC CPU2000 benchmarks and a benchmark from the Olden suite. The evaluation metric was the number of L2 misses per 1000 instructions (MPKI) (they also separately accounted for compulsory misses). They do not go into details of the simulator used. However, the cache configuration that was used is described in the table below.

L1 I-Cache	16kB; 64B linesize; 2-way with LRU repl.
L1 D-Cache	16kB; 64B linesize; 2-way with LRU repl.
Baseline L2	1 MB; 64B linesize; 16-way with LRU repl.

Table 1: Cache Configuration Used in Original Paper

Methodology

The approach used for this project is not completely identical to that described in the original paper, partly due to lack of detail in their methodology as well as time-constraints; the methodology used here can be considered a baseline that could be improved upon.

The authors mention using a "trace-driven cache simulator" for their experiments, but we do not know which one, or how the traces were originally collected. Based on the rest of the paper, they were able to work with instruction traces, identify each memory transaction as well as the address of those transactions. This would be how they came up with their MPKI metric.

For this project, an address trace-driven cache simulator was written from scratch in Python to allow easy modification to evaluate the proposed implementations. The type of cache modeled was fully-associative (exact configurations are covered in the next section). The applications used were the benchmarks from the Bioperf suite (Bader, 2006). The traces were collected using Pin (Berkowits, 2012), specifically the "atrace" Pintool that allows you to collect a list of memory accesses during program execution. One drawback with this tool, though, is that it only generates addresses, so you do not have the corresponding instruction trace to derive the MPKI metric. Instead, the metric used for evaluation is the hit-rate in the L2 cache.

The following Bioperf benchmarks were used:

1) ClustalW
    a. Sequence alignment program for nucleotides/amino acids. This involves multiple comparisons across sequences and figuring out the optimal number of modifications to be made to each of them until they are the same (fewer modifications means they are more similar).
2) GRAPPA
    a. "Genome Rearrangements Analysis under Parsimony and other Phylogenetic Algorithms". Used for "phylogeny reconstruction". This benchmark is used to find common ancestors between sequences. This typically involves generating trees of relationships (i.e. this sequence is an ancestor of another sequence, which may be a common ancestor to multiple other sequences).
3) Hmmer
    a. Uses statistical models ("hidden Markov models" or HMMs) to align multiple sequences. This is similar to ClustalW, but a bit more robust.
4) Predator
    a. Used for locating protein sequences based on amino-acid pairs. This involves modifying protein structures to determine their similarity. They are considered similar if the structures can be rotated/moved such that the points of different structures can map to a set of "close-by" points.

5) Tcoffee
    a. A more accurate/complex version of ClustalW that uses an “internal library” to track sequences.

Each application has different “classes” of possible inputs, each progressively larger/more complex than the last. As a starting point, the smallest “class A” input set was used for this project. The first 15 million memory accesses were collected for each benchmark.

Reproducing the Issue

As a first step, we explored reproducing the original issue described by the authors, which was the phenomenon where applications would have a significant number of L2 cachelines allocated that never get used before getting evicted (i.e. the “zero reuse lines %” metric). In addition to tracking the number of cachelines that were not reused, we also tracked the hit rate of the L2. The results can be found in the figures below. While the original cache configuration mentioned in Table 1 was initially used, it appeared that the size of the caches may be too large to truly explore the effects covered in the paper. The applications were run again at 1/4 and 1/16 the capacity of the original configuration in order to observe the impact of the cache size on hit rate and cacheline reuse. The table below outlines the configurations that were explored.

Configuration	L1 Size	L2 Size
Original	16 kB	1 MB
1/4 Config	4 kB	256 kB
1/16 Config	1 kB	64 kB

Table 2: List of Cache Configurations Explored

Note that the cache modelled here was fully associative, whereas the authors looked at 2-way and 16-way caches for the L1 and L2 respectively. For simplicity we scaled the size of both caches. The cacheline size of 64B was unchanged.

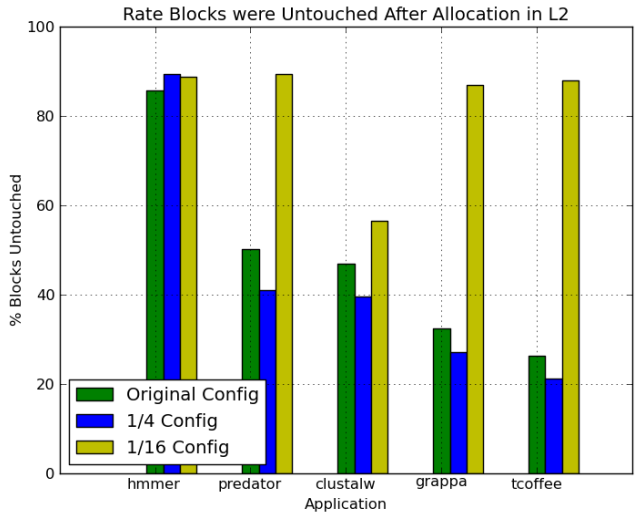

Figure 1: Rate of Untouched Blocks with Varying Cache Configurations

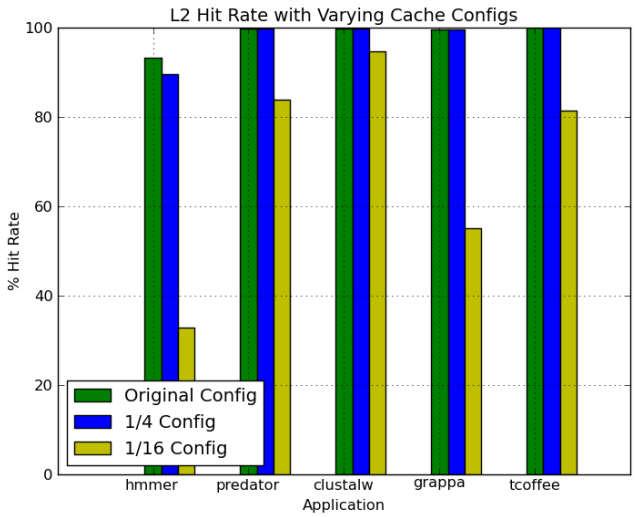

Figure 2: L2 Hit Rates Across Different Cache Configurations

With the original configuration, the authors found roughly half of their apps reaching 80%+ rates of unused cachelines, suggesting thrashing. We were not able to reproduce this with the original configuration, with the main difference here being 1) a fully associative cache was used and 2) different benchmarks were used. We did not observe similar results/thrashing until we reduced the size of the caches to 1/16 of the original design, suggesting these apps may have a significantly smaller working set and perhaps the cache being fully associative lessening the effects to begin with.

When accounting for the changes in the hit rate, it is interesting to see how the hit rate can get progressively worse, while the rate of unused cachelines remains the same. For example, with grappa and tcoffee, the rate of unused cachelines significantly increases with the 1/16 config, but tcoffee sees a much more significant reduction in the cache hit rate (from roughly 90% to 35% between the 1/4 and 1/16 config). On the other hand, the unused reuse rate in hmmer generally stays the same across configurations, but the hit rate begins to drop significantly with the 1/16 config.

The behavior observed in the hmmer app seems to be a more typical example of an app that does not have a large enough cache to contain the working set. Thus, the hit rate suffers with progressively smaller cache configurations, resulting in cachelines being more quickly evicted and thus not as likely to have been reused in the first place, hurting the reuse rate. It would be interesting to see whether looking at the reuse information in a cache controller would help the hit rate (i.e. sooner evict cachelines that have not been reused than ones that have been reused or vice versa).

Looking more closely at the data between tcoffee and grappa, it is ultimately a case where, once the hit rate goes below a certain threshold, depending on the size of your cache, we are more likely to evict an unused cacheline than a used one, reducing the amount of time a given cacheline has to be reused, worsening the reuse rate. It seems that hmmer was able to get away with a poor reuse rate but high hit rate because of the sheer size of the L2; the working set was well contained in the L2 to the point where there were a sizable portion of cachelines that would get loaded and never used through the end of the simulation.

Static Insertion Policies

Given the cache configurations that were initially explored, we were able to reproduce the original "problem" being addressed by the authors. As a first step, the configurations were run with just the MRU insertion policy ("MIP"). The figure below shows the results.

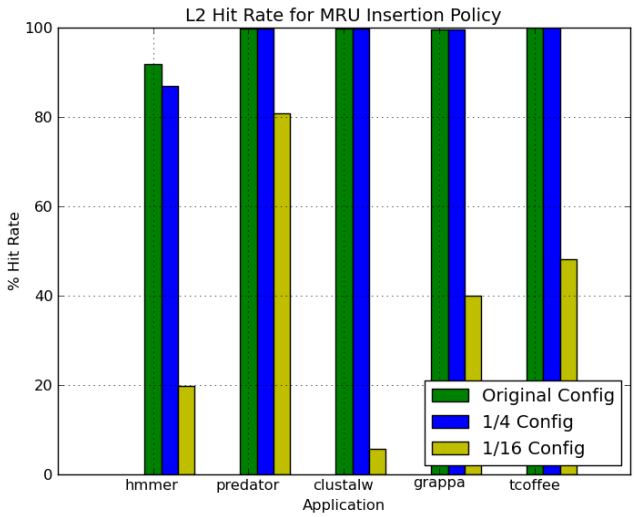

Figure 3: L2 Hit Rate across Cache Configurations for MIP

In general, the insertion policy did not impact the larger configurations (some impact was seen in the 1/4 config with Hmmer, though). The greatest impact was with the 1/16 config, showing a significant drop in the hit rate for all benchmarks.

Given it is the only configuration that has shown any reaction to these configurations, and as a further point of comparison, the figure below focuses on the 1/16 config, comparing the LRU and MRU insertion policies' impact on the L2 cache hit rate.

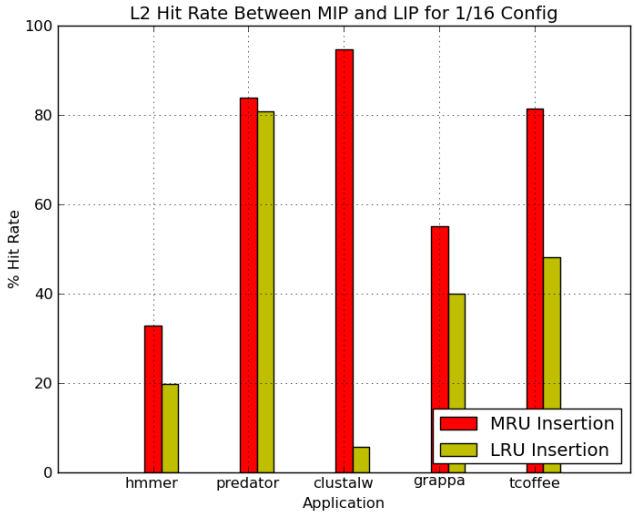

Figure 4: L2 Hit Rate Across Insertion Policies for 1/16 Config

It is interesting to see such a dramatic effect on clustalw. Most other apps see a significant drop, but not as much. The cache allocation eviction process works by allocating/evicting the first available cacheline (walking from first to last, essentially the cacheline with the lowest index). Thus, what can happen is, for the LRU insertion policy, the first cacheline allocated can now be first in line for eviction until some other cacheline is loaded in and also marked as LRU instead. Clustalw appears to need to have a larger enough working set such that, while it can be contained in the original cache, MIP causes cachelines to be evicted far too early. The app walks back over this address space, and we need the cachelines to be loaded in again. This "sweeping" appears to be within a small enough memory region to cause this significant thrashing and the size of the footprint used by clustalw leaves it most vulnerable to this effect.

The cache simulator was then modified to track the hit rate as it changes over the course of the address trace. The hit rate was tracked in windows of 10,000 L2 cache accesses. This was also done to see if there are any phases of execution that would benefit from one insertion policy or another. The figures below show this for each application, for all cache configurations, and the two insertion policies explored so far.

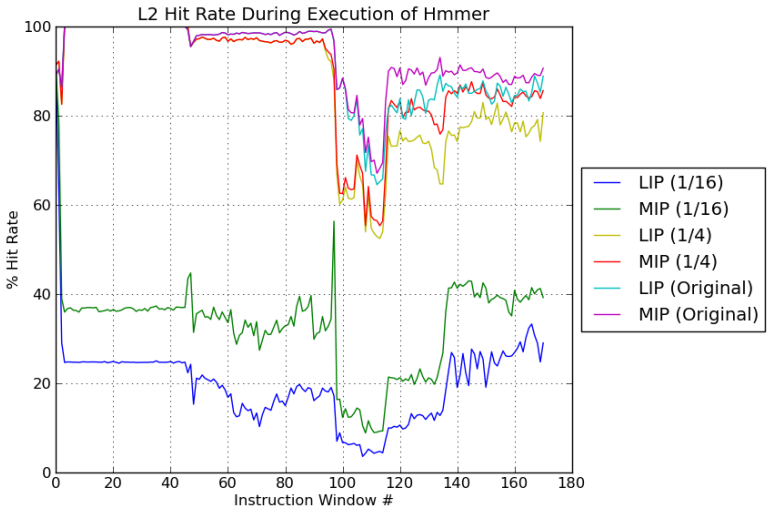

Figure 5: Hit Rate during Execution of Hmmer

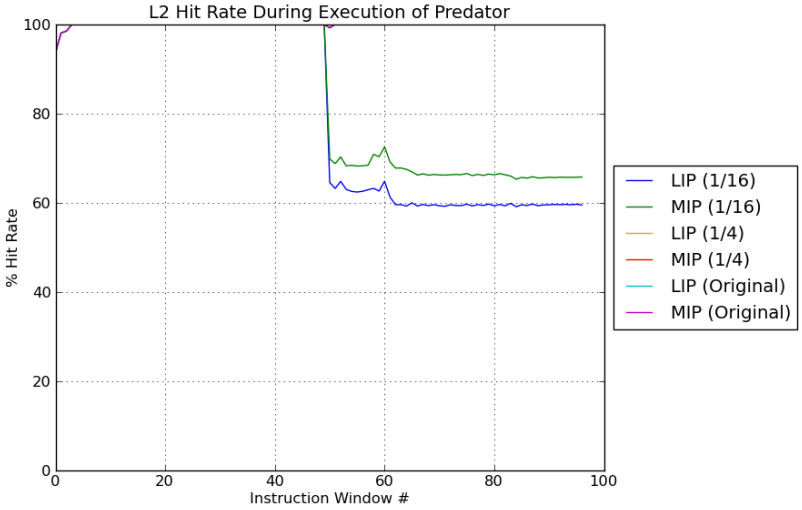

Figure 6: Hit Rate during Execution of Predator

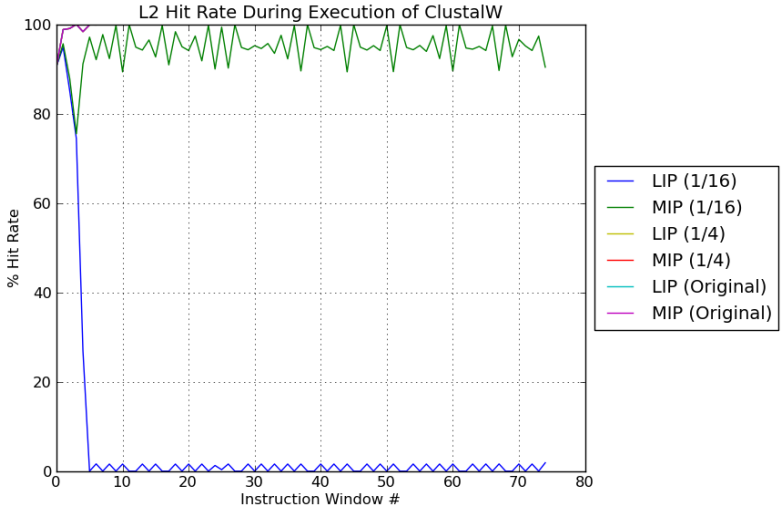

Figure 7: Hit Rate during Execution of ClustalW

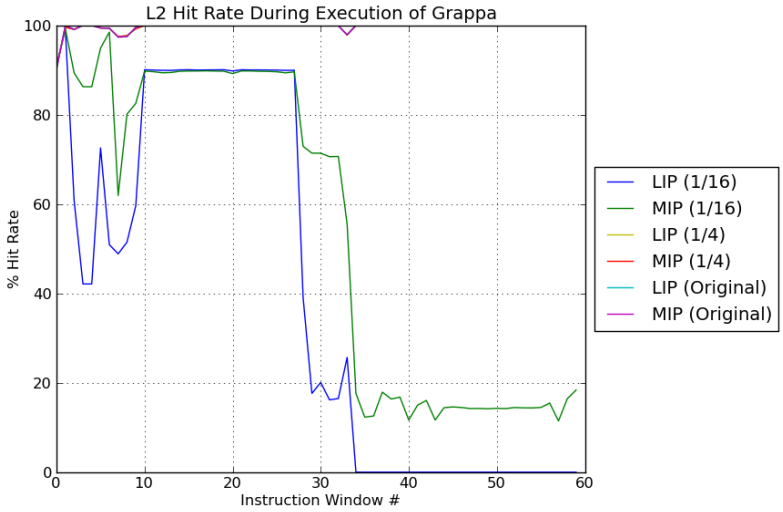

Figure 8: Hit Rate during Execution of Grappa

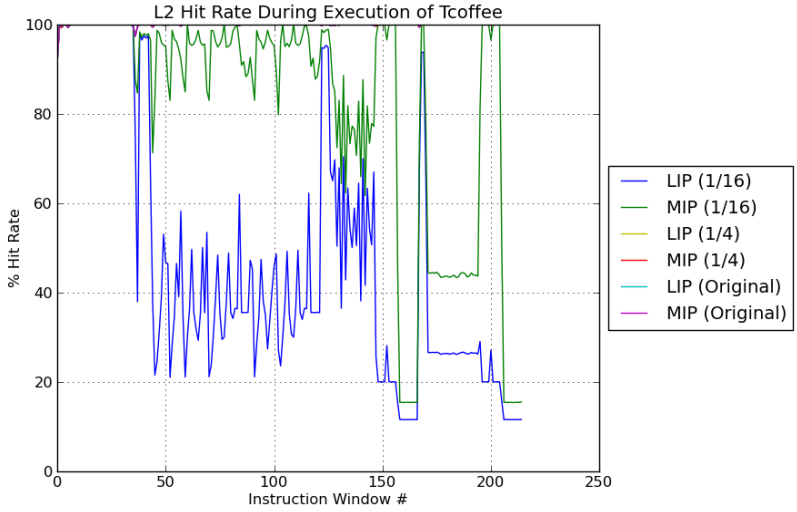

Figure 9: Hit Rate during Execution of Tcoffee

Most of the figures really only show meaningful information for the 1/16 configuration (yet all other configurations are plotted). This is because, for the larger configurations, the hit rate is so high that it is usually near 100%. In this case, the insertion policy makes little difference given the cache can already hold the working set to begin with. One thing that is worth noting here is, while it may be implied by the authors, this kind of research could be used to figure out how to get more out of a smaller cache configuration. Regardless of the application, we start seeing differences in performance once we make the cache small enough. It does not necessarily have to be about a cache not being able to contain the working set and the behavior of the application so much as how can we make the cache smaller and still get decent performance out of it.

Clustalw shows a sharp drop in the L2 hit rate very early on in the trace and continues to stay poor for the rest of the trace. In all the applications, regardless of the cache configuration, the LRU insertion policy

either has no impact or degrades the L2 hit rate. One interesting twist: hmmer appears to be more sporadic when it comes to its phases of execution and hit rates, generally having a poor hit rate for the smaller cache configurations. Generally, LIP with the original cache config performs better than both LIP and MIP of the smaller 1/4 configuration. However, there is actually a point where MIP of the 1/4 configuration performs *better* than the LIP of the original, larger configuration. Nothing like this is seen for the 1/16 configuration. This may be a unique to cases where the working set generally does fit within the cache.

Predator and Grappa each do have a phase where neither insertion policy impacts the hit rate. Predator seems to have this phase because the working set is small enough that no evictions take place, yielding the near 100% hit rate. This is arguably equivalent to running with a larger cache anyway. Grappa appears to encounter a phase where, while some evictions are happening, the blocks being evicted were not being used anyway.

Tcoffee generally shows no improvement. LIP effectively reduces the overall hit rate and we see larger volatility in the hit rate compared to MIP. This may be a result of tcoffee repeatedly walking different parts of its "internal" library, regularly needing to load in new data that is subsequently pushed out to load another part of the library.

Ultimately, it is generally not viable to entirely change the insertion policy to the other extreme. The L2 hit rate in all applications suffered greatly. Both on average and throughout the trace, suggesting little room for extremes. It appears, though, that there is a "sweet spot" to hit. Clustalw shows one extreme where the memory accesses pattern is such that LIP completely ruins the hit rate. Less disastrous applications such as Predator show a significant degradation in performance. Some applications are more sensitive than others depending on their access patterns.

Given the two extremes of insertion policies, a sort of "middle ground" can be found with the bimodal insertion policy (BIP). A weight "1/e" is specified such that for every *e* cacheline allocations, the cache will use the LRU insertion policy; else use the original MRU insertion policy. The same values of *e* were evaluated as in the paper: 16, 32 and 64. The figure below shows the L2 hit rate using this config. It also includes the original MIP and LIP data for comparison.

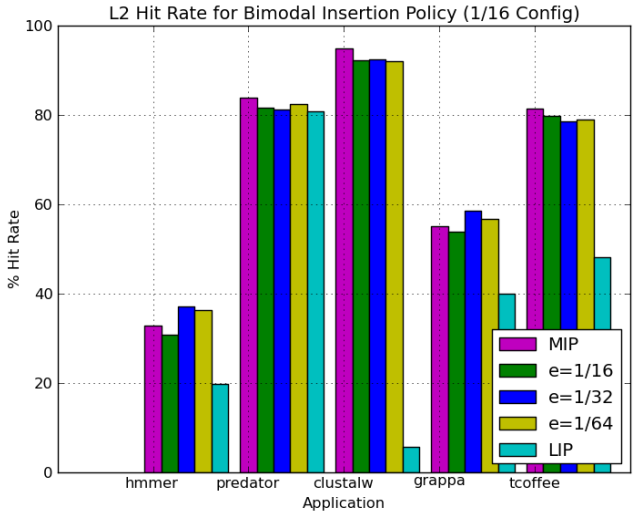

Figure 10: Comparison of LIP/Bimodal/MIP Insertion Policies for 1/16 Config

In some cases BIP actually does result in a better cache hit rate than the original MIP. In two cases where BIP results in better hit rates (hmmer, grappa), e = 1/32 shows the best improvement. However, the majority of the applications show some performance degradation, albeit nowhere near as bad as LIP, and the improvement here is not on the grand level seen by the authors. This stems from the fact that we are far more likely to preserve some of the working set, helping reducing the impact of either insertion policy. Nevertheless, given some applications perform better with BIP vs. MIP, it would be good if we could be able to choose between the two. This is where the dynamic insertion policies proposed by the authors come in.

Dynamic Insertion Policies

It is worth reiterating that a major difference with the cache configuration being used here is that we are using fully associative caches. For the dynamic insertion policies, the authors proposed *set dueling* where a fraction of the sets of a cache are dedicated to either BIP or MIP and their performance is tracked. Based on the performance of these sets, the remaining *follower sets* would use the insertion policy that performed better. This approach is referred to as dynamic-insertion policy set-dueling (DIP-SD).

However, a fully associative cache inherently does not have sets in this sense. A given cacheline may hold data for different addresses at different times. In an attempt to model this, 10% of the cachelines were marked to always use LIP, while another 10% were marked to always use BIP. The rest were follower cachelines. In order to determine what the followers should use, we would look at the average hit rate between the two sets we created. Each non-follower cacheline would track its hit rate. Whichever has the highest hit rate over the past 1024 accesses would dictate what policy the follower cachelines should use. This is admittedly a sizable amount of history being tracked, resulting in 10 extra bits for each block of 10% of the cache. Support was also added in the simulator to track how many follower cachelines were using one policy over another at the end of each window of 10,000 L2 accesses.

The results of the dynamic insertion policy was added with all of the other information, resulting in the figure below.

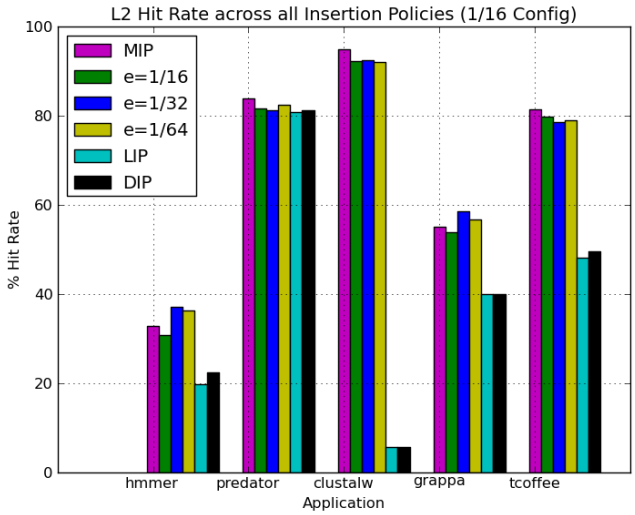

Figure 11: Aggregated Results of Static and Dynamic Insertion Policies for 1/16 Config

Generally, it did not appear the dynamic insertion policy proved much better than LIP to begin with. One natural question that comes up is how often each application used one insertion policy over another.

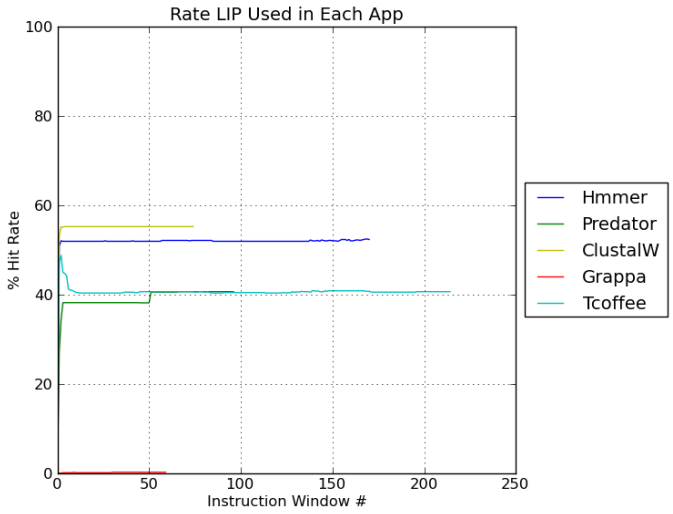

Figure 12: Rates LIP is Used in Dynamic Insertion for 1/16 Config

It appears that generally all apps but Grappa results in half of the eligible cachelines using LIP. Grappa winds up never using LIP, electing always to use BIP, yet it has the same performance as LIP based on the previous figure.

Generally, this implementation cannot be considered well-vetted due to time limitations, but results are included here for completeness. Despite the poor hit rate, why were the insertion policies used roughly split half way for most of the applications? It may be that the selection of the original, dedicated LIP and BIP cachelines was not uniform enough to provide a representative sample. It is also possible there is a bug in the simulator.

Other Discussion

One thing that is worth noting here is that the authors do not explore the idea of only using BIP and dynamically changing the value of e. The value of e can be adjusted to model either LIP or MIP by setting it to 1 (always use LIP) or infinity (never use LIP), respectively. In this case one could have a spectrum over which to adjust e. As the BIP results showed, when performance improved, it not only when e = 32. There may be other values that show better results.

Another thing to consider here is the complexity we are adding through these implementations and the expected pay-off. The authors show some sizable improvements (21% improvement in MPKI), at least for their apps and metrics; however, it is generally not observed here. Here we observe marginal (5% for BIP) improvements only some of the time (2 of the 5 apps). Given the results, it brings into the question trying to be solved in the first place. Is the issue revolving around the cache not being able to contain the working set for a given application? Could this be more easily addressed by expanding the caches or adding another level of cache? It would certainly take up more area/power, but maybe it is simpler, and efforts could be directed elsewhere to save power elsewhere for example.

Future Work or "What are known weak spots?"

There is a list of things that could be looked further into and improved upon when it comes to this existing work.

1. Larger instruction traces, possibly using SimPoint
    a. All traces are ~15 million memory accesses. The authors work with ~250 million instructions (which include memory accesses) with SimPoint. It would also be good to use larger class data sets from Bioperf (not just the smallest "class A" used here). There may be phases of execution dominating these apps that we could not observe.
2. Full validation of cache simulator
    a. We should run side-by-side comparisons of this simulator with another vetted simulator, such as Dinero as part of general validation and confidence in the program.
3. Explore non-fully-associative caches
    a. The simulator only models fully associative caches. It should also support set-associative caches, all the way down to direct-mapped caches.
4. Tie instructions with memory accesses.
    a. Current traces only have memory accesses. It would be better to collect the corresponding instruction trace (via another existing Pintool or creating a new one) and, if possible, being able to map that back to the original C code to better explain phases of execution.
5. Investigate different cache configurations.
    a. What if different levels of cache used different policies? Would the organization of policies change as more levels of cache were added? Are there interactions?
6. Investigate cases where BIP is used, dynamically changing e during program execution.
    a. We would also need to consider the feasibility of such a design in hardware.
7. Investigate other approaches to implementing DIP-SD with a fully-associative cache.
    a. The current implementation was a "first attempt."
8. Investigate impact of the number of LRU bits used on L2 cache hit rate.

a. All caches here assume 4 bits for the LRU counter. That means the oldest cacheline can have a value of 2^4 = 16, which may be a small granularity. It would be interesting to see if adding more bits to this would improve any of the implementations explored here.

Conclusions

The purpose of this project was to evaluate adaptive cache insertion policies as described by Qureshi et al. A different set of applications were chosen, a different cache simulator was written and used, and a different metric was used for evaluation. This can be considered a first foray into evaluating the insertion policies, as there are several basic things that could be done to instill more confidence in the results. Nevertheless, based on the current data, there appears to be some minor benefit to L2 cache hit rate when using the bimodal insertion policy (BIP), roughly a 5% improvement for 2 of the 5 applications (and about 5% reduction in hit rate for other apps). Memory access patterns may change over the course of program execution, and depending on the size of their working set with respect to the cache, one policy may be more appropriate than the other. If a cache is known to be relatively small for its expected workloads, then there may be some benefit to be had in the adaptive insertion policies. The dynamic insertion policy explored here with a fully associative cache showed no benefits, but requires further investigation.

References

Bader, S. e. (2006, 10 8). *BioPerf: A Benchmark Suite to Evaluate High-Performance Computer Architecture on Bioinformatics Applications*. Retrieved from bioperf.org: http://www.bioperf.org/

Berkowits. (2012, 6 13). *Pin - A Dynamic Binary Instrumentation Tool*. Retrieved from https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

Qureshi, M. K. (2007). Adaptive insertion policies for high performance caching. *ACM SIGARCH Computer Architecture News*, 381-391.