Qualitative Cache Performance Analysis

Andrew Carter Max Korbel Paula Ning Josef Spjut

Department of Engineering, Harvey Mudd College

September 23, 2013

Abstract

The effectiveness of caching policies has been measured by a number of metrics. The ultimate quantitative measure is overall system performance. Other metrics such as hit rate, misses per thousand instructions and instructions per cycle are also regularly used in the literature to compare cache behaviors. In this work we propose a novel class of metrics based on the idea that any memory element should ideally be kept out of the cache for as long as possible before being fetched again, an idea inspired by Belady's algorithm. In general we measure the Time to Recache for each evicted element as a qualitative measurement of a caching policy's performance. Among this set of metrics, we analyze the number of Memory Accesses and Memory Misses to Recache, as well as a rough approximation of Wall-Time to Recache. We believe that these metrics will not only be useful for comparing the performance of two different caching policies, but allow designers of such policies to identify memory access patterns that are problematic for existing policies. We also provide our simulation and testing methodology and source to assist others in applying these metrics to their own studies.

1. Introduction

Caching is used in processor design to keep data close to the computational units. This reduces memory access latency, which in turn reduces the cycles per instruction (CPI) of a program, ideally improving system performance. In an effort to improve the effectiveness of caches, a variety of cache policies have been proposed. Laszlo Belady [3] proposed an optimal algorithm that involves always evicting the cache block that will be next used furthest in the future. Unfortunately this requires knowing ahead of time what memory the program will access and the order in which that memory will be accessed. Online algorithms have been developed to approximate Belady's algorithms, these include Least Recently Used (LRU), Not Recently Used (NRU), and more recently Re-reference Interval Prediction (RRIP) [6], in the forms of Static RRIP (SRRIP), Bimodal RRIP (BRRIP), and Dynamic RRIP (DRRIP). LRU and NRU were proposed on the basis that if a processor just used a cache block, it is likely to be used sooner than a cache block it has not used in a while. The RRIP policies attempt to determine the future usefulness of cache blocks, and evicting cache blocks that are not likely to be used in the future. These policies are discussed at length in Section 3.2

In this paper we propose the use of a class of Time to Recache (TTR) metrics [11] in offering insight into why different cache management policies perform better than others. There are a variety of different ways to measure TTR, including Memory Accesses to Recache (MATR), Memory Misses to Reacache (MMTR), Wall-time to Recache (WTTR). This class of metrics refers to the time spent by a cache line after it has been evicted from the cache and before being fetched again. It is related to, but distinct from, the notion of "reuse distance," which refers to the amount of time between successive accesses to a given cache block. The TTR-based metrics are discussed in detail in Section 2.3.

2. System Performance Metrics

Computer performance is dependent on a variety of factors, ranging from very low level features like the latency of individual functional units, bypass networks, and out-of-order hardware, up through very high level features such as operating systems, disk I/O, and network latency. In this paper we focus our attention on the performance of the CPU cache, though other caching systems could also make use of TTR metrics. The quality of a cache is typically gauged by two factors, the improvement in performance, measured in Instructions Per Clock (IPC), that it affords to the processor it backs, and the reduction of Misses Per 1000 Instructions (MPKI), which equates to a reduction in the number of longer latency memory accesses from upper level caches.

2.1. IPC

The number of instructions a CPU can complete in a single clock cycle can be equated with its absolute performance. If a processor can complete more instructions in a given clock cycle than another, then the program will complete execution sooner and its performance is better. Hardware caches play a critical role in boosting this number. The closer that data sits to the functional units, the higher performance can be. In a typical three level cache hierarchy, it can take 10x longer to access the third level of cache than the first, and another 10x longer to access main memory. Finding data as close to the processor as possible is critical for high performance.

2.2. MPKI

One of the cache's main jobs is to reduce the number of memory accesses that are performed on the next level of cache. Each cache miss includes accessing the next level cache, and possibly writing back if that cache block is dirty. If the next level cache is shared, then there may be contention between the different processors. There is a lot of work that has to be done in the event of a cache miss, so reducing the MPKI of a cache is a popular and important metric to look at.

While both IPC and MPKI are good for comparing how one caching policy compares with another, neither provides insight on what sort of memory accesses the cache could improve to increase performance. Furthermore, a more detailed look at the impact of an application on cache behavior could be very useful for cache designers.

2.3. TTR

In this work we expand and analyze the previously proposed [11] Time to Recache (TTR) methodology for examining the behavior and effectiveness of the caching policy. TTR is defined as the amount of time (measured in a variety of different manners) that a cache block spends outside of the cache after it has been evicted and before it is accessed again. Note that this is distinct from the concept of reuse distance. Reuse distance is the time between successive accesses to a piece of data or cache block. TTR does not take into account the amount of time a cache block spent in the cache before it was evicted. Unlike TTR, reuse distance is independent of a replacement policy, and is therefore not very useful in analyzing a replacement policy. TTR only tracks the time spent after eviction and before reuse, making it dependent on the replacement policy based on when the cache block was evicted. In this paper, we propose the use of three variants of the TTR metric.

WTTR: Wall Time to Recache (WTTR), is defined as the amount of real world time the CPU takes (either in cycles, or in seconds for a known clock frequency) before a cache block is accessed again. Due to limitations in our trace collection and simulation methodology, we approximate WTTR by assigning a “wall time” cost to instructions and cache misses. Note that total wall time is the ideal measurement for total system performance and is therefore a desirable metric for performance analysis.

MATR: Memory Access to Recache (MATR), is defined as the number of memory accesses that occur before the cache block is accessed after eviction. This is much easier to gather in our simulations but is not as close to the real performance as WTTR.

MMTR: Memory Misses to Recache (MMTR), is defined as the number of cache misses from the time that a given cache block is evicted until it is returned to the cache. This is also straightforward to gather even in trace-based simulations, but is affected by the hit rate of the caches.

Belady’s optimal algorithm [3] for cache eviction always evicts the cache block whose reuse is furthest in the future, allowing that free space to be used as long as possible by other data before being recached. It is impossible to know at runtime for general workloads which cache block has the furthest reuse distance, hence why there are so many different caching policies that use various heuristics in an effort to approach the effectiveness of this optimal algorithm.

Measuring the IPC and MPKI of a workload using one caching policy, and comparing that to the IPC and MPKI of running that workload with a different caching policy can give you some sense of how close each of those caching policy comes to the optimal solution. This is, however, an indirect approach to quantifying how well a caching policy is performing. Tracking the TTR is a direct means of comparing two caching policies.

TTR is an effective metric because it asks the question every time a cache block is brought into the cache, “have I seen this block before, and if so, how long ago was it?” If caching policy A answers this question with “4000 cycles ago,” and caching policy B answers this question with “6000 cycles ago,” then caching policy B has done a better job at evicting that cache block early and allowing that space to be used by other data. With TTR, a higher number is better. A low TTR number means that the cache block was evicted and then recached very soon afterwards, suggesting that it should not have been evicted in the first place.

However not one TTR metric tells the whole story. In aggregate, it is hard to tell the difference between recaching after CPU intensive calculations, a small working set, and a large number of evictions. By having all three metrics, we can perhaps determine that a low WTTR actually had a relatively high MMTR, indicating that the working set was simply too large for our cache. Thus despite a low WTTR, it is probably hard to improve the cache over that range. Similarly if MATR or MMTR is very low, but WTTR is very high, then that portion of code is probably CPU bound, and there is no need to improve the caching algorithm over that region.

3. Methodology

We simulate a set associative cache using a number of different replacement policies. We use a statistical sampling technique to keep our total simulation time down while still ensuring our results are representative of the overall behavior for the application in question. Our simulation framework can be found on Github [4].

We chose to use the NAS Parallel Benchmarks [1] as a set of reference programs to explore our metrics. We specifically ended up using sp_omp with only one thread in this paper because it showed the most interesting results. PIN [8] was used to get a memory trace of the program, then this memory trace was sampled in a manner that will be discussed in 3.1, and then the sampled traces were run through a Cache policy simulator, which we wrote in Python. Using the cache replacement results and the original sampled traces, we then constructed MATR, MMTR, and WTTR for each policy.

3.1. Sampling Technique

We sampled the file to minimize the data set we operated on, while maintaining the integrity with results. We took 100 samples of 2 million memory accesses generated by sp_omp instrumented by PIN. By the central limit theorem we can approximate samples of this size as having a Gaussian distribution. Each sample was divided into 3 sections, a warm-up period of 100 thousand memory accesses, a sampling period of 1.4 million memory accesses, and a cool-down period of 500 thousand memory accesses. For each metric we warmed up the cache during the warm-up period, then measured how long it took for any cache line evicted during the sampling period to be recached. Cache lines that took longer than the cool-down period to recache were ignored. We felt that cache lines that took more than 500 thousand memory accesses to recache were unlikely to be interesting.

3.2. Caching Policies

Accessing a cache line for the first time causes it to be added to the cache. Each cache line contains 64 bytes of data. We use 7 of the cache line address bits to decide which set the cache line maps to. Each set is a 8-way associative set, and when all of the ways in a set are filled, and we want to insert another cache line into the set, then one of the old cache lines must be selected for eviction to make way for new data. We test several different cache replacement policies which determine the cache line to be evicted in the event of an associativity conflict. The new cache line is always placed in the same cache way that was vacated by the evicted data.

3.2.1. Belady Belady's algorithm [3] was proposed in the context of page replacement, but its principle also applies in cache replacement. The algorithm is to choose the cache line to be evicted whose next access is furthest in the future. This is the optimal cache replacement strategy, but it requires oracle knowledge of the total order of all memory accesses in the program, which is not generally knowable at the time of cache replacement. We show TTR results for Belady's algorithm anyway as an optimal baseline to compare the other cache replacement algorithms against.

3.2.2. Random While Belady's algorithm requires perfect knowledge of all memory accesses in a program, the Random cache replacement algorithm requires no knowledge whatsoever about the order of memory accesses in the program. When each cache replacement decision is to be made, a random number is generated that decides which way of the set to evict.

All of the other evaluated cache replacement policies will behave differently because of, and respond to, program behavior in one way or another, but the Random algorithm is not affected in any way by program behavior. This algorithm requires the storage of no additional state per cache line to implement.

3.2.3. FIFO First In First Out, or FIFO, chooses the cache line to evict from the set that has been in the cache longest. Neither reuse, nor any other factors, has any bearing in deciding which cache line to evict in the FIFO algorithm. This algorithm requires only enough per-set state to point to the cache line in that set that will be evicted and replaced next.

3.2.4. LRU Least Recently Used, or LRU, choose to evict the cache line from the set whose most recent access was furthest in the past. Conceptually this works like a queue, where eviction candidates are always chosen from the tail of the queue, and both initial use and subsequent reuse promotes cache lines to the front of the queue, furthest away from being in danger of eviction. In the absence of cache line reuse, this algorithm behaves identically to FIFO. This algorithm requires enough state per cache line to enforce an absolute order between all members of a set, typically $\log_2(N)$ bits per cache line, where N is the associativity of the set.

3.2.5. NRU Not Recently Used, or NRU, is a simplification of the mechanism and storage overheads in LRU, and is similar in operation to the clock algorithm in page replacement. There is 1 bit of state maintained per cache line in a set. A value of "0" means that the cache line has been "recently used," and is therefore not a candidate for eviction. A value of "1" means that the cache line has "not been recently used," and may be evicted. Re-referencing a cache line causes its NRU state to be set to "0".

The eviction victim is found by linearly scanning through the NRU state bits in a set (in a left-to-right fashion) and evicting the cache line corresponding to the first "1" encountered. If no "1"s are encountered, then all NRU bits are changed to 1 and the same victim selection process is repeated, i.e., the cache line in way "0" will be evicted.

3.2.6. SRRIP Static Re-Reference Interval Prediction [6], or SRRIP, is similar to NRU, but it uses multiple bits to encode how recently a cache line has been used, and therefore how soon it is predicted to be used again. For this study, we use 2 bits to encode the Re-Reference Prediction Values (RRPVs) for all RRIP-derived cache replacement algorithms, meaning that RRPVs are in the range of 0 to 3. An RRPV of 0 means that the cache line was recently used, and is in little danger of eviction, and an RRPV of 3 means that the cache line was not recently used, and is in danger of imminent eviction.

SRRIP works by assigning an RRPV of 2 to each cache line when it is initially brought into the cache. Reuse of a cache line promotes its RRPV to 0. Eviction victim selection is done by scanning through the set, left-to-right, and looking for a cache line whose RRPV is 3. The first encountered 3 is evicted. If no 3s are found, then all RRPVs in the set are incremented, and the scan to look for a 3 repeats. This process may repeat several times until a 3 is found and evicted.

3.2.7. BRRIP Bimodal RRIP [6], or BRRIP is similar to SRRIP, and varies only in the initial RRPV of cache lines when they are first brought into the cache. Whereas SRRIP always assigns an RRPV of 2 to incoming cache lines, BRRIP assigns an RRPV of 3 95% of the time, and an RRPV of 2 only 5% of the time. This offers resistance to scans through memory that may otherwise thrash the cache. In the absence of cache line reuse, BRRIP results in only a single way of each set being thrashed, rather than every way, as can happen in LRU, NRU, or SRRIP.

The mechanisms of scanning through the set to find an RRPV of 3, and incrementing all RRPVs if a 3 is not found, are identical to SRRIP.

3.2.8. DRRIP Dynamic RRIP [6], or DRRIP, chooses to use either SRRIP or BRRIP at run-time, depending on which cache replacement policy is currently more effective at minimizing cache misses. This is done by permanently assigning a small number of sets in the cache to always follow the SRRIP policy, and permanently assigning a small number of other sets to always follow the BRRIP policy. Each cache miss in a SRRIP-only set will increment a policy selection counter, and each cache miss in a BRRIP-only set will decrement the policy selection counter.

Depending on the value of the policy selection counter, either SRRIP or BRRIP will be considered to currently be causing more cache misses than the other. The remaining cache sets that are not permanently assigned to follow a specific cache replacement policy will then use the cache replacement policy of whichever candidate policy is currently causing the fewest cache misses.

The mechanisms of scanning through the set to find an RRPV of 3, and incrementing all RRPVs if a 3 is not found, are identical to SRRIP and BRRIP for all sets, regardless of which replacement policy they are using.

4. Results

We present results from Belady, Random(rand), FIFO, and all of the RRIP policies (SRRIP, DRRIP, BRRIP). These polcies were run against a trace of sp_omp run on a single core, with a 8-way set associative cache, with 7 bits of direct mapping within each set, and 64 byte cache lines. We limited the graphs to only show MATR because we believe that for this particular trace, and this set of replacement polcies, this TTR best shows the usefulness of the TTR metrics in general.

Figures 1-6 show each individual replacement policy, as well as the error in measurement due to our sampling strategy. Figures 7-13 show a side-by-side comparison of two different replacement policies.

In each graph, any (x,y) point corresponds to y evictions that took x thousand memory accesses to be recached. To smooth out the graph, these points have been collected into 1000 buckets, each bucket represents an area of 500 Memory Access.

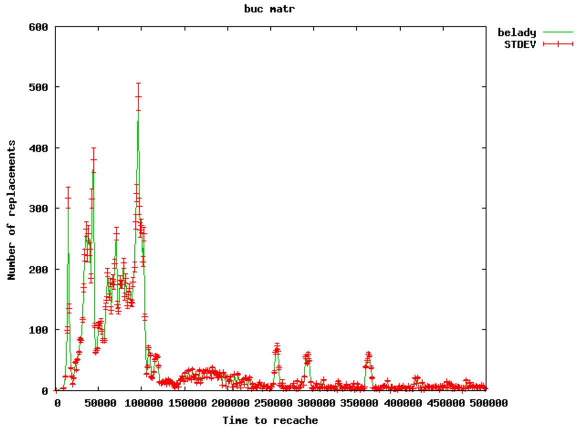

Figure 1: Bucketed MATR for BELADY

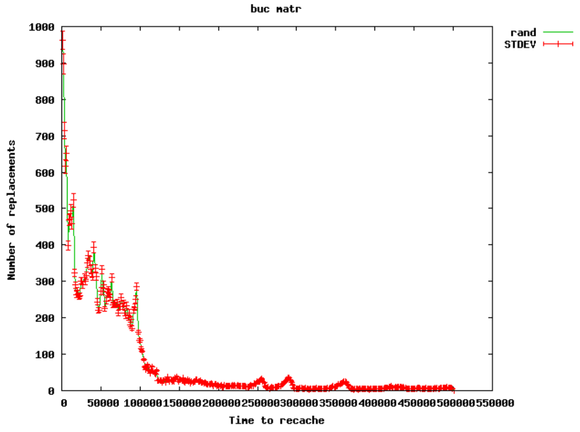

Figure 2: Bucketed MATR for RAND

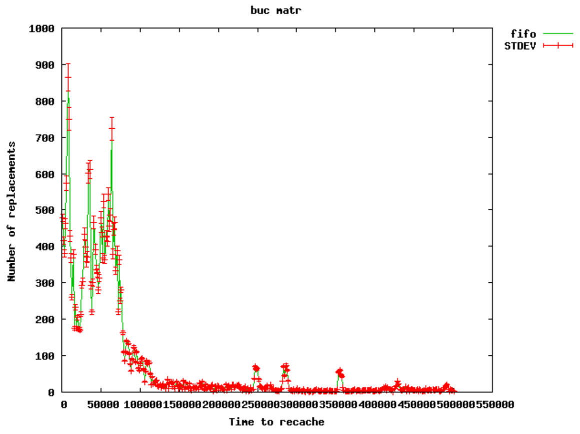

Figure 3: Bucketed MATR for FIFO

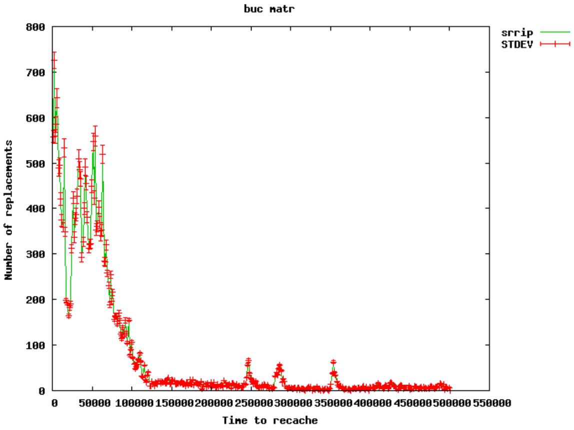

Figure 4: Bucketed MATR for SRRIP

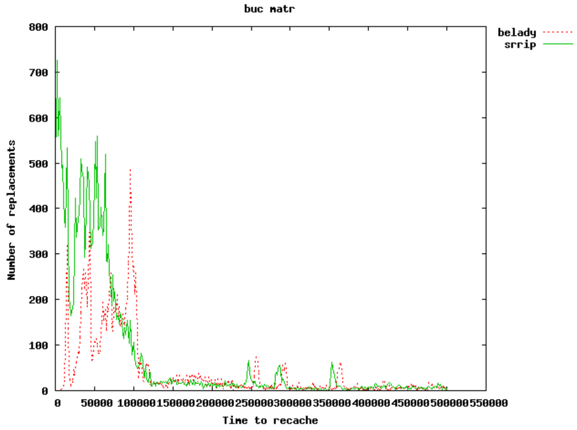

Figure 7: Bucketed MATR for BELADY vs. SRRIP

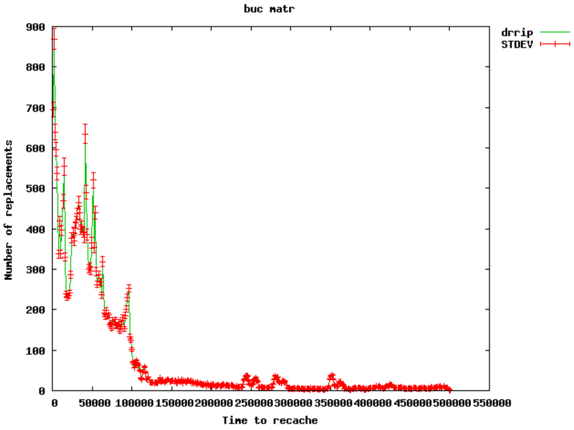

Figure 5: Bucketed MATR for DRRIP

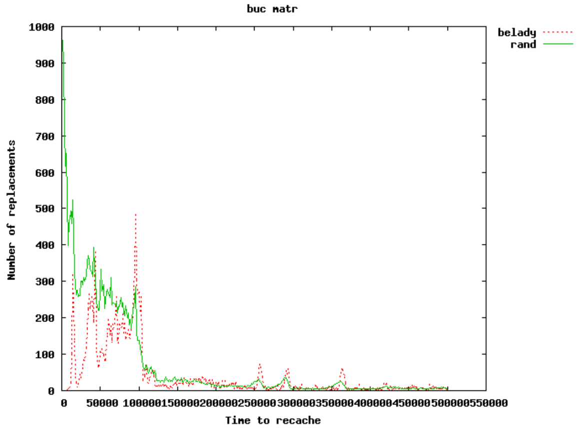

Figure 8: Bucketed MATR for BELADY vs. RAND

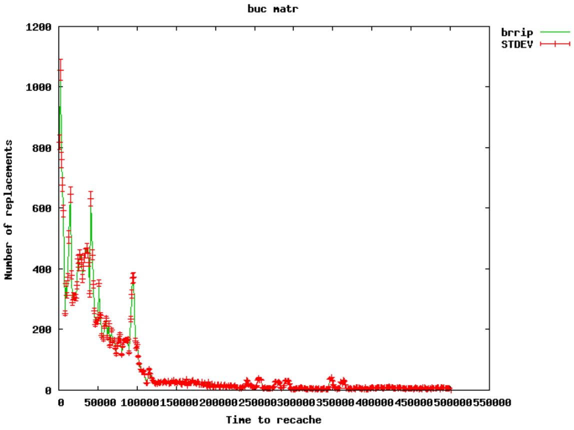

Figure 6: Bucketed MATR for BRRIP

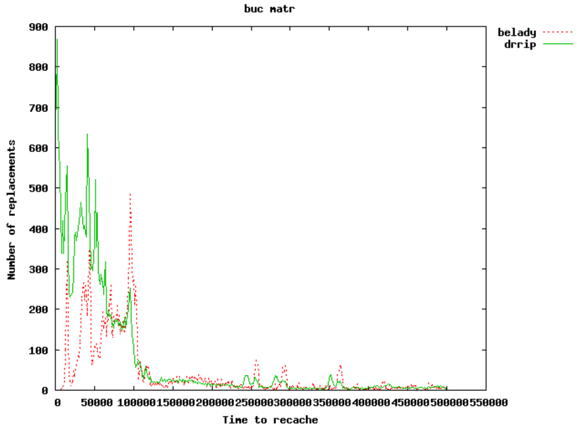

Figure 9: Bucketed MATR for BELADY vs. DRRIP

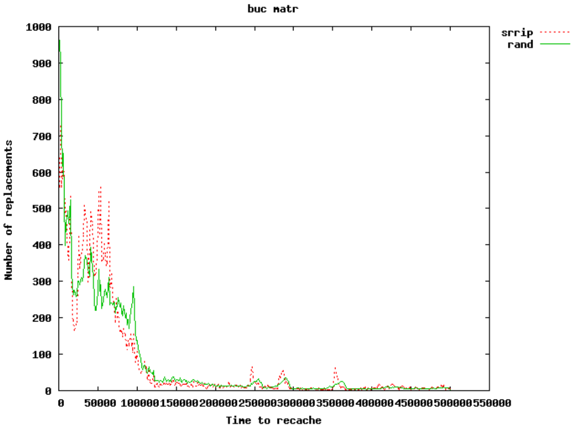

Figure 10: Bucketed MATR for SRRIP vs. RAND

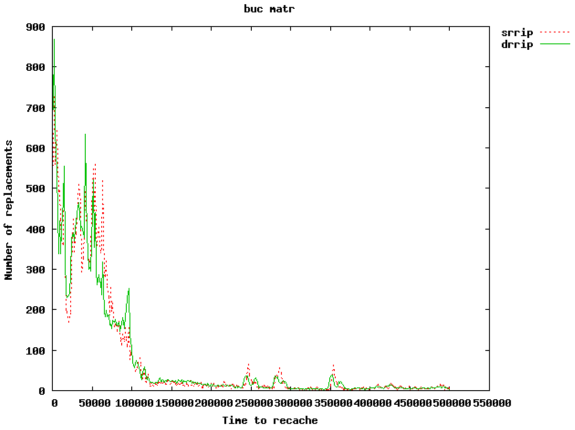

Figure 11: Bucketed MATR for SRRIP vs. DRRIP

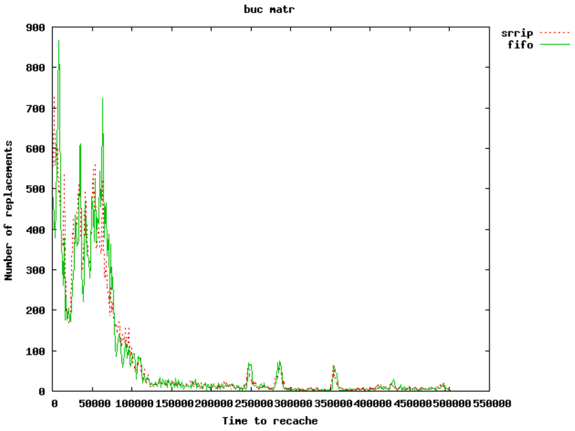

Figure 12: Bucketed MATR for SRRIP vs. FIFO

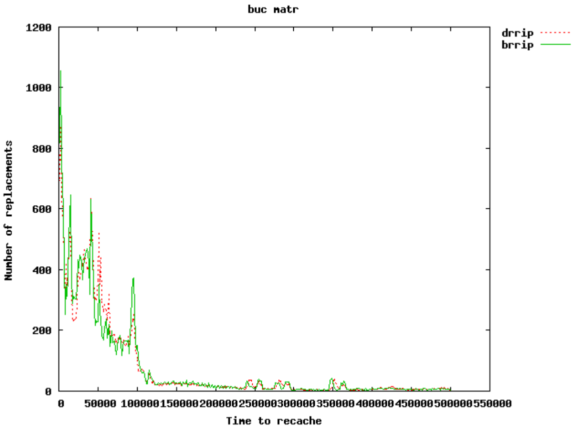

Figure 13: Bucketed MATR for DRRIP vs. BRRIP

5. Analysis

We chose sp_omp because it showed an interesting distribution of MATR results. We have plotted several replacement policies against each other to show the difference between these policies. Specifically we have plotted replacement algorithms against Belady's Optimal Algorithm to find out what these policies could do better. As you can see all cache algorithms have a lot of replacements under 100 K memory accesses from eviction. However, there are also three peaks around 250 K, 300 K, and 350 K memory accesses. The intuition of how to read a TTR graph is as follows. A high *y* value is generally bad, because it means there were many evictions and recaches, however if Belady also has a high *y* value, than this is a property of the program, and your caching policy cannot do better, (the program however could possibly be rewritten). Many TTR graphs include humps in their distribution, humps that appear significantly later than any equivalent hump in Belady are bad, this means that you are holding onto data for too long. Humps that do no appear at all in Belady, means that it would be optimal to hold the data long enough for the next memory access to occur, in this case your caching policy does not hold onto data long enough.

For instance in Figures 7,8,9 we see many peaks at less then 100 K memory accesses in SRRIP, RAND, and DRRIP. However in general we see much smaller peaks in Belady, indicating that our caching policies are holding onto the wrong data. In fact Belady has a relatively large peak just short of 100K indicating that none of the caching policies are correctly evicting those cache blocks, although RAND and DRRIP do a much better job than SRRIP as seen in Figures 10,11. As seen in Figure 7, for the three humps mentioned before (at 250, 300, and 350 K memory accesses), SRRIP evicts much too late, and in Figure 12 we can see that it chooses evicts around the same time as FIFO. As seen in Figures 9,11, DRRIP appears to split the difference between Belady and SRRIP. It has two smaller peaks for each peak in Belady, one around were SRRIP's peak is and one near Belady's. Similarly Figure 13 shows a similar distribution for BRRIP. This means that DRRIP and BRRIP are able to correctly predict some of the time that a cache block will not be used again significantly earlier than SRRIP and FIFO are able to make that prediction. Finally RAND, spreads out the peak accross the interval from Belady to SRRIP as seen in Figures 8,10. This is expected because of the random nature of the policy, sometimes it will evict as early as Belady, and sometimes it will evict as late as or later than FIFO, but most often it will evict some time in between, this reinforces the idea that DRRIP and BRRIP are able to make informed decisions about these cache blocks, as they, in contrast show a trough in the middle.

These graphs indicate, at least for sp_omp, that replacement policy designer's should look into ways to evict the cache blocks of the three humps as early as Belady does above and beyond what DRRIP and BRRIP already do. Furthermore they should also try to keep a lot more of the cache blocks that are accessed within 95K memory accesses, as Belady outperforms the other cache algorithms by quite a bit in that range.

6. Related Work

Effective management of the contents of the cache can be greatly beneficial to the performance of a processor. It can lower average memory access latency. , and conserve off-chip memory bandwidth. In this section, we discuss some recent attempts at improving the contents of caches, and how they relate to the TTR metric. We also discuss other tools and visualization techniques that have been proposed to improve cache performance by influencing the way code is written.

Lai and Falsafi [7] use the idea of dead block prediction to anticipate when the last touch of a cache line would be before it is naturally evicted from the cache. Their main observation is that the last access to a cache line occurs long before it becomes the LRU cache line in a set and is chosen for eviction, so they instead try to evict a cache block as soon as it is predicted to no longer be useful. This can directly influence the TTR of cache lines by evicting a line before a regular replacement policy would choose to do so, thereby giving it more time to spend outside of the cache before it is recached.

Basu et. al. [2] investigate the concept of eviction-use distance as motivation for their Scavenger LLC architecture, which identifies cache blocks that are recently missed in the LLC, and then puts them into a separate region of cache that protects them from their frequent eviction. Seshadri et. al. [10] do something similar with their eviction address filter. They use a Bloom filter to identify cache lines that are returning to the cache recently after having been evicted, and these cache lines are given lower initial RRPVs. These technique can improve TTR metrics by identifying and offering eviction resistance to a subset of the most often thrashed cache lines, favoring instead to evict cache lines that are unlikely to be immediately reused.

Manikantan et. al. [9], identify delinquent memory operation instruction addresses and track histograms of the next-use of the cache blocks brought in by them. Some of the ways of the LLC are dedicated to blocks brought in by the delinquent PCs, so that they do not pollute the rest of the ways. Their work takes a different approach to improving the quality of the contents of the cache by trying to prefer to evict offending data rather than to explicitly protect useful data. This would improve TTR data by evicting useless data early, and allowing useful data to have longer cache residence.

Some research has also been done in the area of cache performance visualization. Choudhury and Rosen [5] developed a visualization tool that tracks individual cache lines as they move from main memory through the various levels of the cache, and as they are inserted, reused, and evicted from each cache level. This tool was developed in the context of trying to see with a graphical interface how the code a programmer wrote interacts with the cache hierarchy, and is not directly meant to be a means to evaluate the quality of cache management strategies, as TTR is.

7. Conclusions

In this work we have presented a class of metrics for cache studies that provide deeper insight into the behavior of the replacement algorithm when used for a particular application. When Belady's algorithm is used, one can discover inflection points that cause poor cache performance in the application. A comparison between TTR for Belady's algorithm and any set of replacement policies can provide insight for cache designers to choose replacement policies that are appropriate for the class of applications that the designer deems important. Furthermore, our sampling technique allows for quick iteration in policy design with rapid feedback to the developer. MATR was reasonably straightforward to capture in our simulations, but other types of TTR may be more useful or easy to record in other contexts.

TTR differs from traditional metrics like IPC and MPKI by providing a qualitative resource for cache performance. IPC and MPKI only indicate how well a replacement policy performs, but not where it performs badly. In contrast, TTR provides replacement policy designers insight on how to improve their policy.

While we have presented these metrics in the context of CPU caches, the potential application for these techniques extends beyond CPU cache design. For example, large web applications employ DRAM as a cache for large databases held on arrays of disc drives as a method for increasing input and output operation throughput. Additionally, other application specific integrated circuits and processors that utilize caching may find TTR useful. In order to assist others in using our techniques, we have released our source code publicly (reference withheld) and encourage revisions and additions as appropriate for other domains.

8. Acknowledgements

Thanks to Seth Pugsley for helpful discussions and reviewing a version of this document.

References

[1] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, and V. V. andd S. Weeretunga, "The NAS Parallel Benchmarks," *International Journal of Supercomputer Applications*, vol. 5, no. 3, pp. 63–73, Fall 1994.

[2] A. Basu, N. Kirman, M. Kirman, M. Chaudhuri, and J. Martinez, "Scavenger: A New Last Level Cache Architecture with Global Block Priority," in *Proceedings of MICRO*, 2007.

[3] L. Belady, "A Study of Replacement Algorithms for a Virtual-Storage Computer," *IBM Systems Journal*, 1966.

[4] A. Carter, M. Korbel, P. Ning, and J. Spjut, "TTR Github Repository," 2013, [Online; accessed 16-February-2017]. [Online]. Available: https://github.com/Clay-Wolkin-Fellowship/spock

[5] A. Choudhury and P. Rosen, "Abstract visualization of runtime memory behavior," in *6th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, ser. VisSoft, 2011, pp. 22–29.

[6] A. Jaleel, K. Theobald, S. Steely, and J. Emer, "High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP)," in *Proceedings of ISCA*, 2010.

[7] A.-C. Lai and B. Falsafi, "Selective, Accurate, and Timely Self-Invalidation Using Last-Touch Prediction," in *Proceedings of ISCA-27*, 2000, pp. 139–148.

[8] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proceedings of PLDI*, 2005.

[9] R. Manikantan, K. Rajan, and R. Govindarajan, "NUcache: An Efficient Multicore Cache Organization Based on Next-Use Distance," in *Proceedings of HPCA*, 2011.

[10] V. Seshadri, O. Mutlu, M. Kozuch, and T. Mowry, "The evicted address filter: A unified mechanism to address both cache pollution and thrashing," in *Proccedings of PACT*, 2012.

[11] J. Spjut and S. Pugsley, "Time to Recache: Measuring Memory Miss Behavior," *Technical Report*, Sep. 2011.