Time to Recache: Measuring Memory Miss Behavior

Josef Spjut[†] and Seth Pugsley[†]

[†]School of Computing, University of Utah

September 2011

Abstract

Caches are integral to the performance of the memory subsystem for any processor. In contemporary literature, it is common to use metrics such as a count of memory misses over some number of instructions, or to use the rate of instruction issue as a proxy for true system performance as indicators of the performance of a last level cache. We introduce a novel metric called "Time to Recache" (TTR) that more directly represents the closeness with which any technique approaches the optimal caching scheme. As an example of how this metric might be used, we include an analysis of a few traditional and modern replacement policies using cycle-accurate processor and cache simulation.

1 Introduction

Caching is used in processor design to keep data close to the computational units, which improves memory access latency, and reduces off-chip memory bandwidth. Multi-level caches store data in progressively larger, and further away structures. The increased distance, and especially the increased size, of higher levels of cache mean that the higher levels of cache have longer latencies to access. If all of the on-processor caches fail to contain the data being accessed by the processor right now, then a long latency and high energy access to main memory is used to fetch the data into the caches so it will be available for the next time that data is accessed.

The Last Level Cache (LLC) is especially important for performance because it represents the last opportunity for a memory access to remain on-chip, without having to fetch data from the DRAM main memory. While an LLC cache hit may take 10x more cycles to complete than an L1 hit, an LLC miss takes an additional 10x longer to service. An LLC miss not only takes longer to complete, but activating the DRAM system also uses more energy to service the request. From an energy perspective, an LLC hit consumes only the energy to service the memory request by the LLC. On the other hand, an LLC miss requires the energy to check the LLC, notice that it's a miss, and then fetch the data from DRAM, possibly also needing to write back dirty data to DRAM which has been displaced by the incoming data.

In an effort to improve the effectiveness of the LLC and reduce the number of DRAM accesses, a number of cache management policies have been proposed, often based around having a set associative cache, such as the Least Recently Used (LRU) policy, Not Recently Used (NRU) policy, and others. Each of these cache management policies attempts to evict data that is no longer deemed useful in favor of newly accessed data which was not found in the LLC before. Recently the Re-reference Interval Prediction (RRIP) [6] cache management policy has been proposed as a way to improve LLC performance by attempting to predict the future usefulness of cache blocks. RRIP is represented by two distinct varieties, namely Static RRIP (SRRIP) and Dynamic RRIP (DRRIP), which each use a different insertion policy when dealing with incoming cache blocks. Both SRRIP and DRRIP are discussed at length in section 4.3.

In this paper we explore the effectiveness of the Time to Recache (TTR) metric in offering insight into why different cache management policies perfor better than others. This metric refers to the time spent by a cache line after it has been evicted from the LLC and before it is 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. A long TTR for a given cache line would indicate that the cache policy correctly identified that the line could be evicted without causing the memory system to bring the line back.

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 on the performance of the LLC. The quality of an LLC 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 long latency DRAM main memory accesses.

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, 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 LLC's main jobs is to redue the number of DRAM accesses that are performed. Each DRAM read access includes occupying a memory controller's read buffer, waiting for this access's turn, and then sending that read request across long wires to distant DRAM chips, activating those DRAM chips, and then finally sending the data back to the memory controller over several more cycles. This description of events omits what happens to the data after it gets back to the memory controller, and also ignores the consequences of needing to write dirty data from the LLC back to the cache. There is a lot of work that has to be done in the event of an LLC miss, so reducing the MPKI of a cache is a good goal, making MPKI a popular and important metric to consider.

3 Time To Recache

In this work we propose the Time to Recache (TTR) methodology for examining the behavior and effectiveness of the LLC. TTR is defined as the amount of time (measured in cycles or seconds) 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 doesn't take into account the amount of time a cache block spent in the LLC before it was evicted. TTR is only concerned with the time spent after eviction and before reuse.

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, and indirect approach to quantifying how well a caching policy is performing. Tracking the TTR is a direct means of comparing two caching policies. In particular, Belady's optimal algorithm would generate an optimal measure for TTR.

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.

In Figure 1, we see an example of a TTR graph for the CG benchmark for both a 4MB LLC and an 8MB LLC and across the four cache insertion policies we tested. A TTR graph is a line-graph representation of a histogram of TTR values. The majority of the results in this paper are presented in this format. The bins of a TTR graph, along the X-axis, are 10,000 cycle-long periods of time that a cache block has been absent from the cache before returning. The Y-axis of the graph is the number of cache blocks that were absent from the cache for that long before returning. For example, if 10 cache blocks had been recached after being absent from the cache for 40,000 cy-

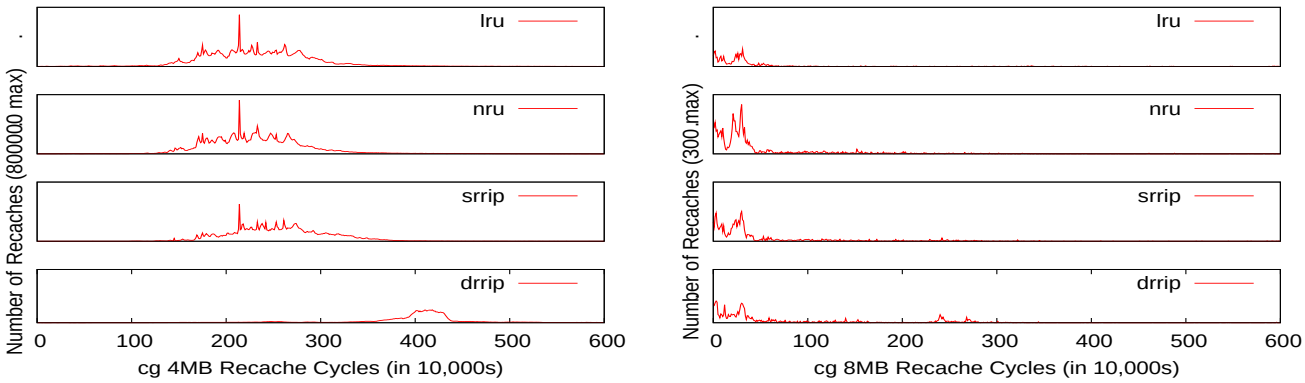

Figure 1: CG Time to Recache

cles each, then the 4th bin of the graph would have the value of 10. Note that the Y-axis for each chart is different to show the interesting parts of the charts. For example, the 8MB LLC results in many fewer misses than the 4MB LLC, so the Y-axis is much smaller in most benchmarks.

A TTR graph shows the distribution of how long cache blocks were absent. 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. Similarly, recaches that happen at a low TTR bin number (low X value in the graph) are considered bad because it means that when a cache block was evicted it was soon recached, suggesting that cache block should not have been evicted in the first place. Many TTR graphs include humps in their distribution. Humps that appear at low TTR values are generally worse than humps that appear at high TTR values, although the Y magnitude of the hump must also be considered. High Y magnitude humps are generally worse than low magnitude humps, although the X location of these humps is also important.

A series of charts appears at the end of this paper detailing a set of results from our simulations. Details of how these data were generated can be found in Section 4. CG is an interesting comparison for this section because one can clearly see the distinction between the different caching policies. In particular, LRU and NRU look very similar and SRRIP shifts the chart slightly to the right which corresponds with an increase in performance. DRRIP shows a significant change that will be discussed more later, but intuitively shows a drastic increase in caching behavior by reducing the number of misses and increasing the time between subsequent misses to the same line.

4 Methodology

We gathered our TTR and other performance statistics using the Virtutech Simics [8] full system simulator version 3.0.x.

4.1 Simulation Parameters

We model an 8-way Chip Multiprocessor of ultraSPARC III cores. Our processing cores are in-order with 32 KB 8-way L1 instruction and data caches with a 3 cycle latency. Each core also has a private, 256 KB 8-way L2 cache with a 10 cycle latency. The L2 and L1 private cashes are non-inclusive with respect to one another. All cores share a 16-way L3 cache, which is inclusive of the contents of the L2 and L1 caches, and is the last level of cache before DRAM. The size of this LLC varies between experiments, and we collected data using a 4 MB, and 8 MB L3 cache. Off-chip memory accesses have a static 300 cycle latency to approximate best-case multi-core memory timings as seen by Brown and Tullsen [4]. Only memory operations are modeled in detail and all non-memory operations are given a latency of 1. Since we use a three level cache hierarchy, a large number of memory accesses which would normally influence the LLC replacement policy are filtered out by the L1s and large L2s.

4.2 Benchmarks

We use a selection of benchmarks chosen from the Nas Parallel Benchmark(NPB) Suite [1] and Spec2k6 benchmark suite [5]. The NPB benchmarks are single program, multi-threaded, and use the W-size reference input. The Spec2k6 benchmarks are multi-programed, running eight copies of the given benchmark. As we move to larger LLCs, the differences between cache policies running the

NPB benchmarks becomes less pronounced, because the W-size working set becomes more and more cache resident. Many of the Spec2k6 benchmarks have much larger working sets and continue to experience many cache misses even with larger caches. There is no fairness policy or static partitioning of the LLC, so all threads and programs have equal access to LLC capacity.

For each benchmark, we fast-forward to the middle of program execution and then spend 100 million instructions per core warming up the caches. After this, we track our performance statistics and gather TTR samples for another 250 million instructions per core. Fast-forwarding is done to the beginning of the region of parallelized work for the NPB benchmarks, and for 20 billion instructions for each of the Spec2k6 benchmarks.

4.3 Cache Management Policies

In our experiments we analyze the differences between four LLC management policies, namely Least Recently Used (LRU), Not Recently Used (NRU), Static Re-reference Interval Prediction (SRRIP), and Dynamic Re-reference Interval Prediction (DRRIP) [6]. A brief description of each policy follows in the context of an LLC.

LRU works by maintaining a list with a head and a tail for each set in the LLC. When a cache line enters the LLC, it is placed at the head of the list for its cache set. When a cache line needs to be removed to make space for another, the tail is chosen and evicted. Every time a cache line is reused, it is promoted to the head of the list.

NRU works by maintaining a 1-bit counter for each cache block in the LLC. When a cache block is first brought into the LLC its counter is set to 0. When a cache line needs to be removed from a set, a scan is done to see if any have their counter set to 1. If a 1 is found, then it is evicted. If no 1 is found, then all counters are incremented and the scan to find a 1 repeats. Every time a cache line is reused, its counter is set to 0.

SRRIP and DRRIP both work on the basis of Rereference Interval Prediction, which is similar in concept to a 2-bit NRU-type policy, where all cache blocks have a 2-bit counter instead of NRU's single bit. When a cache block is brought into the cache, it is assigned an initial Rereference Prediction Value (RRPV) for its counter. When a cache block needs to be evicted, a scan is done to find a block with an RRPV of 3, and that block is evicted. If no 3 is found, all RRPVs in the set are incremented, and the process is repeated. This continues until a 3 is found and evicted. Every time a cache line is reused, its counter is set to 0.

SRRIP works by statically inserting blocks with an RRPV of 2. DRRIP has the option of inserting with an RRPV of 2 or 3, depending on which is performing better at the time. If DRRIP decides to insert a block with an RRPV of 3, there is a small chance that it will insert the block with 2 instead.

This paper does not seek to validate or debunk the claims put forth in the works that propose any of the above mentioned caching policies. The intent of this paper is to analyze and visualize the different behaviors of varied caching policies with respect to the TTR metric.

4.4 TRR Data Collection

We collect TTR data at run time during our Simics simulations. Every time a block is evicted from the LLC its address is added to a list that tracks all of the previous 8 million evictions from the cache, along with a timestamp of when the eviction occurred. When a cache block is fetched to be placed into the LLC, its address is compared against the list of 8 million recent evictions, searching from the most recent first. If there is a match, then we consider that to be a recache, and we increment the TTR bin corresponding to the amount of time the cache block spent out of the cache. We only track recaches that happen within 40 million cycles, as recaches within this window seem to have the greatest impact on performance. Depending on the application of TTR, it may be interesting to extend or reduce the window in which recaches are tracked.

5 Results

In this section we show the results of TTR tracking for our different benchmarks and cache management policies. As stated earlier, the purpose of this paper is not to showcase the relative performance of one caching policy over another, but is rather to show the effectiveness of the TTR visualization method at giving insight into system performance based on cache behavior. Furthermore, no attempt was made to ensure that the simulation windows we used in gathering these statistics is necessarily representative of program execution overall. For this reason, the performance numbers for the cache policies considered here may differ from previously published results for those cache policies. Our goal is merely to show that TTR visualization is a useful way to explain the performance we observed.

Our results are presented by showing the TTR graphs for LRU, NRU, SRRIP, and DRRIP for each of the benchmarks, using both 4 MB and 8 MB. When looking at the TTR visualizations for a given benchmark and a given cache size, the scale for all four component graphs (LRU, NRU, SRRIP, and DRRIP) is the same in both X and

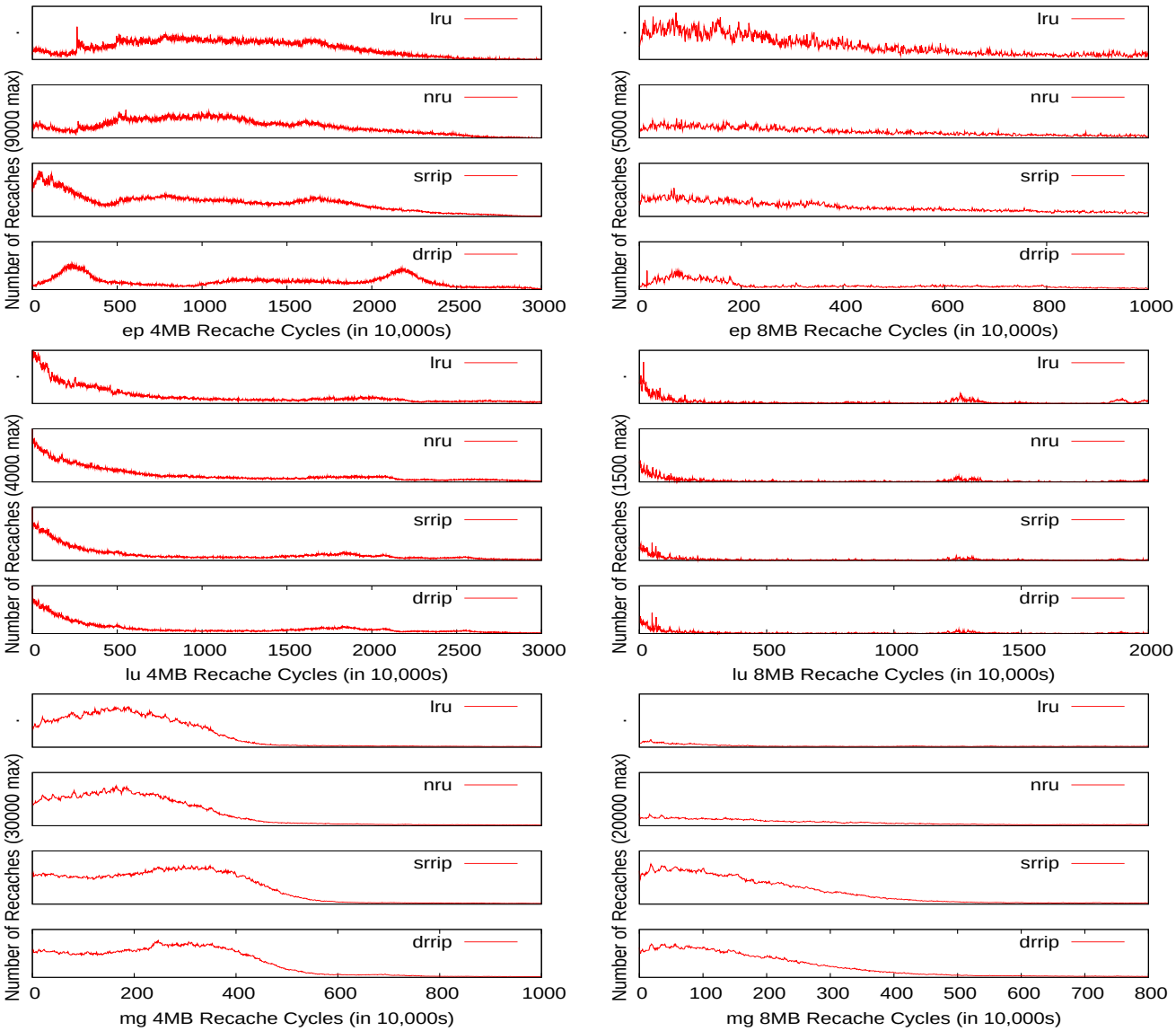

Figure 2: Time to Recache

Y directions, so they are all directly comparable. When moving between the 4 MB and 8 MB visualizatoins, the scales may be different. Figure 2 shows some of the most interesting results, and Figures 3-5 can be found at the end of the paper.

6 Related Work

Creating good cache line replacement policies is a well studied area of computer architecture, and several papers have addressed on this problem. In general it is the goal of every cache policy design to keep around truly reused data while evicting data that will not be reused for a very long time, but we mention here as related work the studies whose approach explicitly looks at reuse distance or some variation of it as a part of its design or motivation.

Scavenger [2] investigates 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. In their motivation for this work, Basu et. al. [2] mentions large Eviction-Use distance as one of the major contributors to the problem and presents static values for Eviction-Use distance for several benchmarks. This differs from TTR visualization in that they distilled the entire phenomenon down into a single value, and TTR visualization shows the spectrum of all eviction-use distances.

Keramidas et. al. [7], seek to genuinely predict the reuse distance of cache blocks by using the program counter (PC) of a memory access to index into a predictor structure, which is updated when a reuse has been detected. TTR visualization does not take the PC of memory operations into account when plotting out recaching time, although it could be interesting for future work to look at the TTR graphs for each individual memory operation PC in a program.

Manikantan et. al. [9, 10], identify delinquent memory operation PCs 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. This work only tracks reuse within a few dozen LLC misses of when a block is last accessed, so their notion of reuse is much more resitricted than that in TTR visualization.

7 Conclusions

We have presented a novel cache performance metric we call Time-To-Recache (TTR) and have demonstrated how it can be used to analyze the cache behavior of a given system. While we have not shown a quantitative measure based on the TTR plots that can be used in performance comparisons, we believe there are other ways that TTR can benefit computer architects. For example, TTR can be used to discover reuse intervals of a given program by inspecting the periods where TTR is hot. Additionally, by comparing different cache sizes a designer can see how the cache size affects the ability of the cache to work effectively.

We anticipate that TTR can be used for a variety of applications from cache design for computer architects, through low-level programmers laying out program memory to large distributed data center memory caches managing many terabytes of data on even slower and energy hungry drives. Each type of design will use TTR in a different way, but this new method for visualizing data can be used to great effect and success.

References

[1] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):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] J. A. Brown and D. M. Tullsen. The Shared-Thread Multiprocessor. In *Proceedings of ICS*, 2008.

[5] J. L. Henning. SPEC CPU2006 Benchmark Descriptions. In *Proceedings of ACM SIGARCH Computer Architecture News*, 2005.

[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] G. Keramidas, P. Petoumenos, and S. Kaxiras. Cache Replacement based on Reuse-Distance Prediction. In *Proceedings of ICCD*, 2007.

[8] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, February 2002.

[9] R. Manikantan, K. Rajan, and R. Govindarajan. NUcache: A Multi-core Cache Organization Based on Next-Use Distance. In *Proceedings of PACT (poster)*, 2010.

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

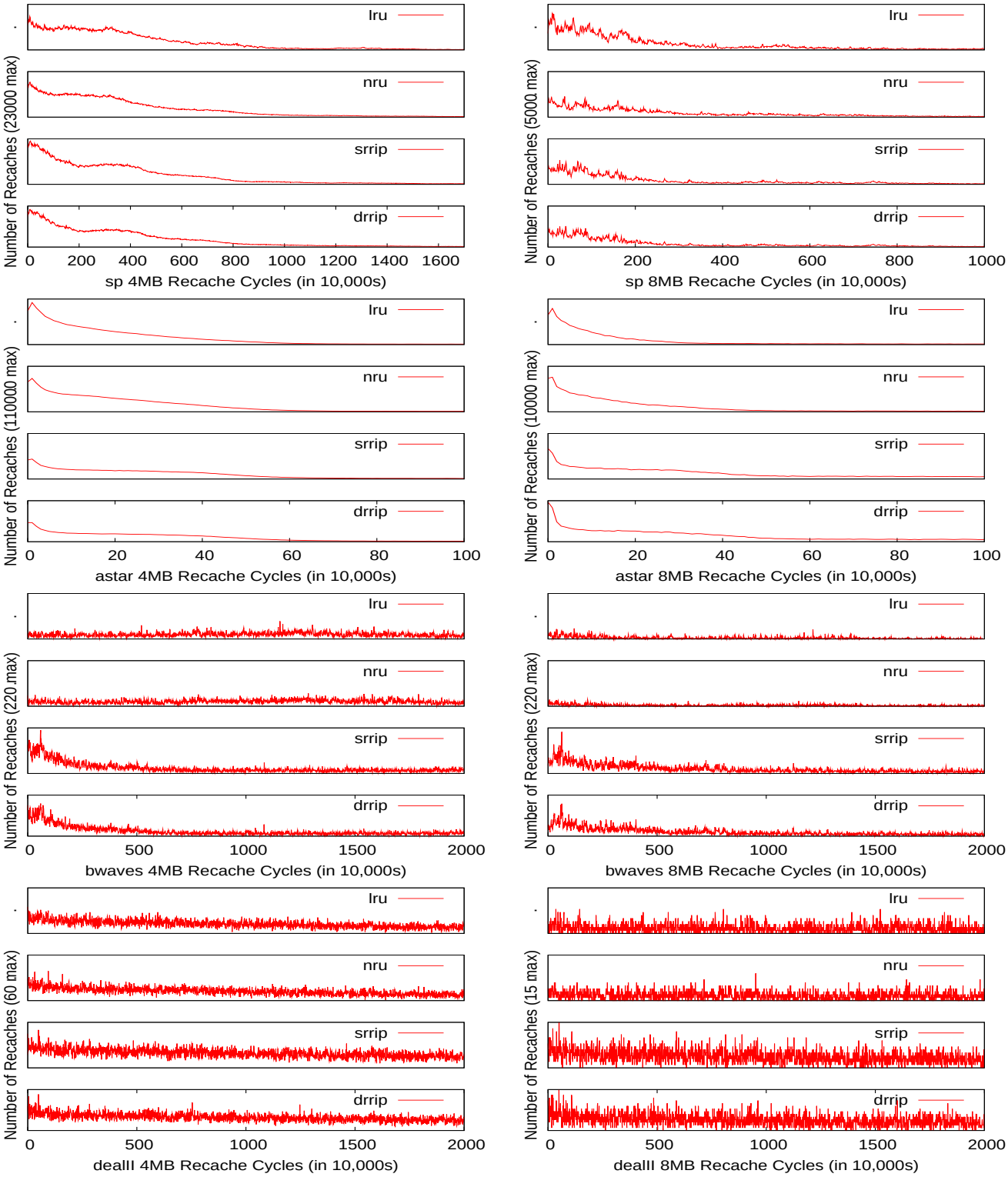

Figure 3: Time to Recache

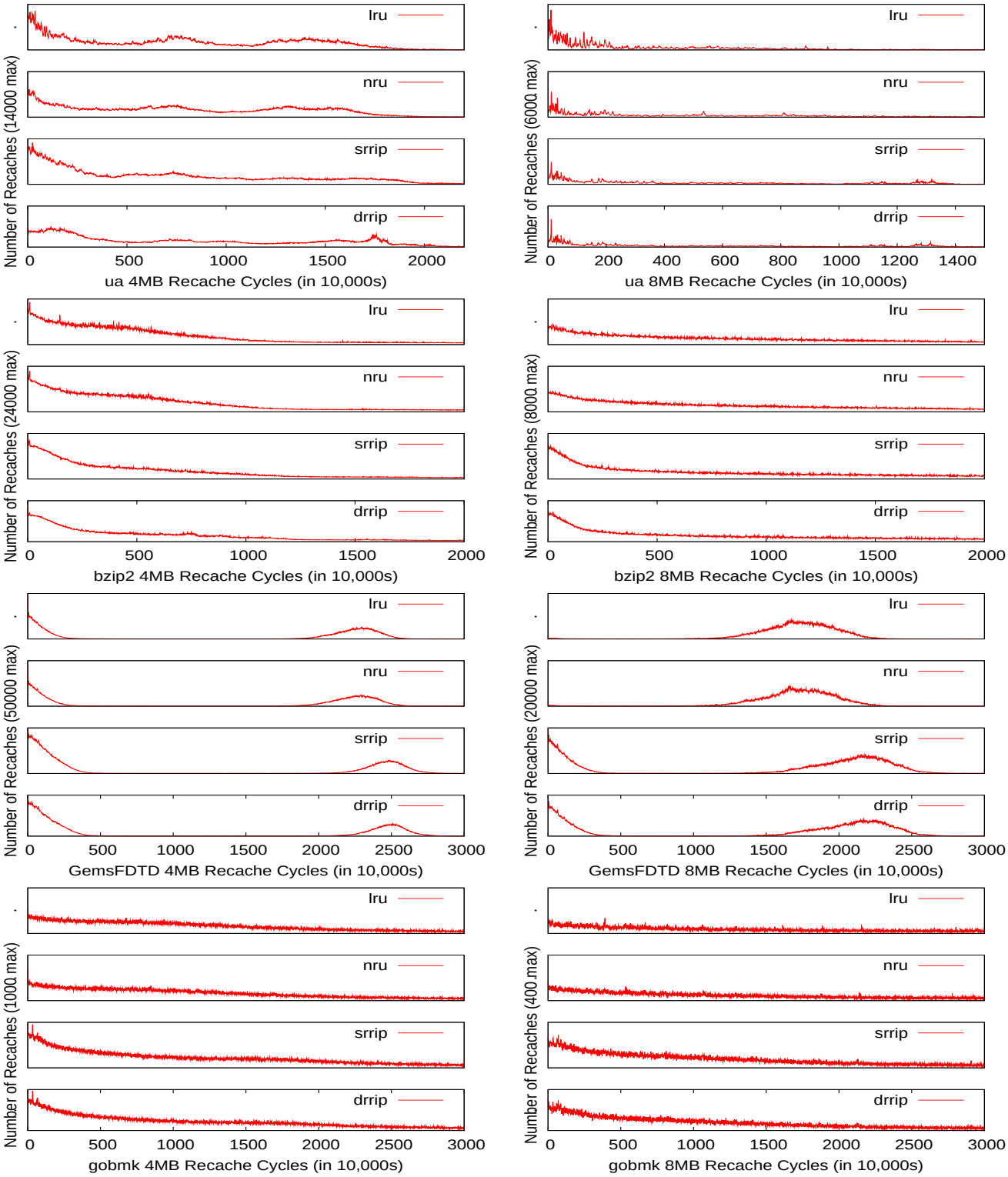

Figure 4: Time to Recache

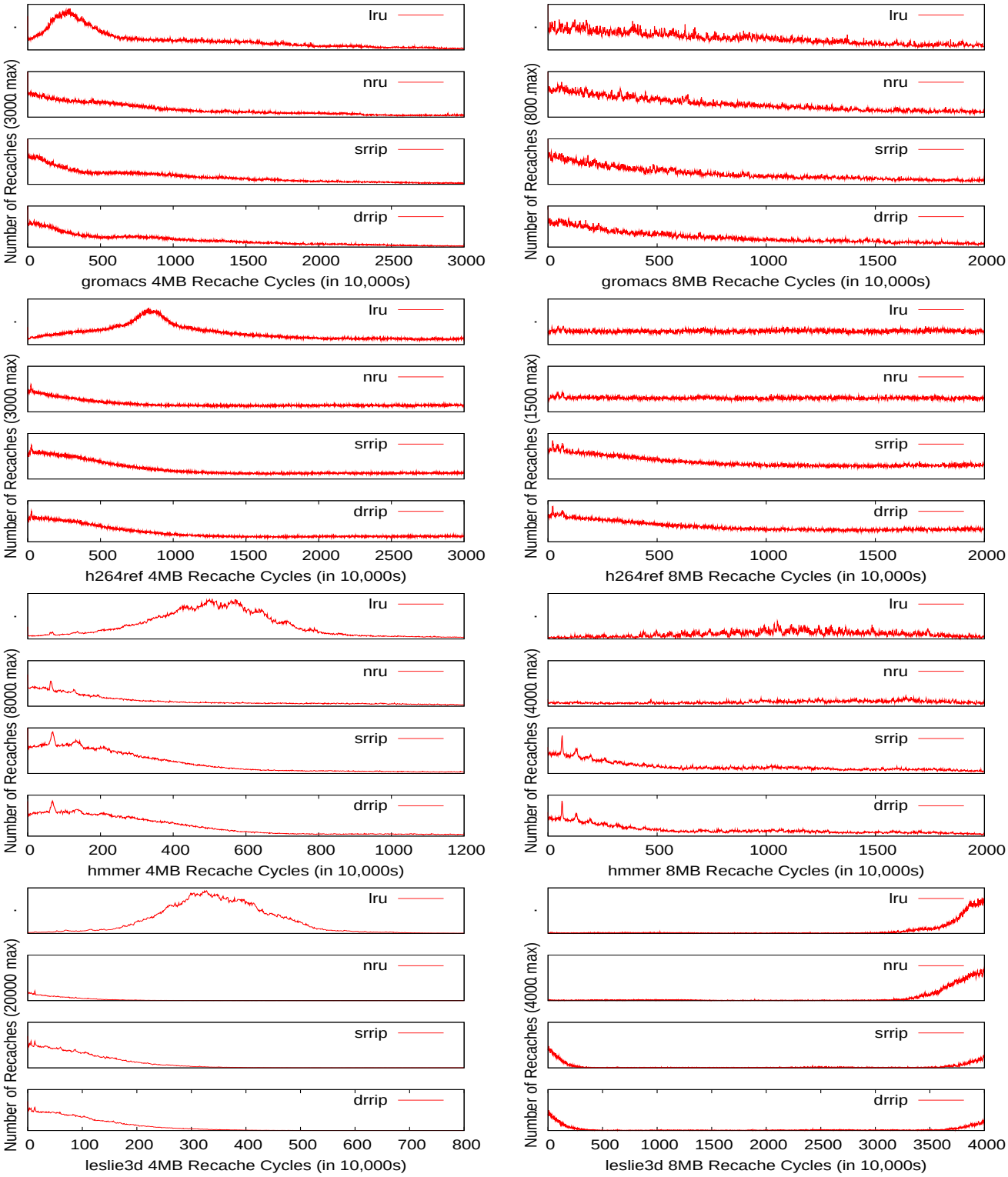

Figure 5: Time to Recache