

Deconstructing the Inefficacy of Global Cache Replacement Policies

Rahul V. Garde Samantika Subramaniam Gabriel H. Loh

Georgia Institute of Technology
College of Computing
{garde,samantik,loh}@cc.gatech.edu

Abstract

In a conventional two-level cache hierarchy, L1 cache hits do not propagate to the L2 cache; as a result, the L2 cache only observes a “filtered” memory access stream. A frequently accessed address may hit in the L1, but since these accesses never make it to the L2, the corresponding copy in the L2 will “decay” with respect to its replacement policy state and may eventually get evicted. Previous studies have advocated the use of global replacement policies where the L1 access information propagates to the L2 to maintain a replacement policy state that is consistent with the overall global memory access stream. We first attempt to duplicate previously reported results on global cache replacement policies. Despite the intuitive explanation for why a global scheme should work, our experimental results show that the performance potential of global replacement is very limited. We deconstruct the problem with reuse-distance analysis and show that only under very specific reuse-distance profiles will a program be able to benefit from global replacement. Our experiments include the evaluation of multi-core shared caches, inclusive cache hierarchies, and a wide spectrum of cache sizes and associativities; we show that global replacement fails to provide significant performance benefits for any of these scenarios.

1. Introduction

Cache replacement policies considerably impact the performance of many applications. Since these policies choose which lines to evict from the cache, they have a direct effect on miss rates which is usually directly related to performance. Over the decades, there has been a large body of work on basic replacement policies [16] and several recent proposals have shown that combinations of basic policies can improve performance by adapting to different program and phase behaviors [7, 15, 19]. All these research proposals, however, only make the replacement decisions *locally*; that is, the second level cache (L2) is oblivious to the state of the first level cache (L1), and vice-versa. Caches employing local replacement policies are easy to implement, but recent research suggests that limiting the replacement policy’s scope may have drawbacks [21].

The basic problem is that not all levels of the cache hierarchy see the same access patterns. Since the L2 is only accessed when we miss in the L1, the L2 cache only observes a “filtered” version of the global memory access pattern. As processors integrate L3 caches [20], the disconnect between the observed access patterns at different levels of the cache hierarchy could become even more dramatic. In the context of cache replacement policies, this means that even if a line is accessed repeatedly in the L1 cache, the line might still make its way to the least-recently-used or LRU position of the L2 cache and thus risk being evicted (assuming a LRU replacement policy, but this observation holds for most replacement policies).

An alternative to such local replacement decisions is a system that exposes the entire, unfiltered, global access stream to all caches in the hierarchy. Zahran described and evaluated several global replacement policies [21]. We believe that it intuitively makes sense that an L2 cache with a global view of the unfiltered L1 access stream can make better replacement decisions because it has more accurate and complete information about the true, program-level, memory access patterns. While Zahran’s preliminary results showed that a global replacement policy does not provide a substantial performance improvement, he described several characteristics of programs as well as potential processor configuration settings where a global policy would be beneficial. These include newer benchmark suites with larger memory working set sizes, different cache sizes and associativities, inclusive cache hierarchies with different line sizes at the different cache levels, and multi-core processors with shared L2 caches.

In this work, we explore these other possible scenarios, and disappointingly show that even in all of these other cases, global replacement policies still fail to show any promise. We then take a step back to try to explain the observed results. Using analysis based on memory access reuse distances, we describe the necessary conditions for a global replacement policy to provide benefit. We then quantify the actual reuse-distance histograms of our benchmarks and show that nearly *none* of the programs ever exhibit such behaviors, and due to the rather stringent set of conditions, we conclude that it is very unlikely that global replacement policies will ever be useful (unless there are some drastic

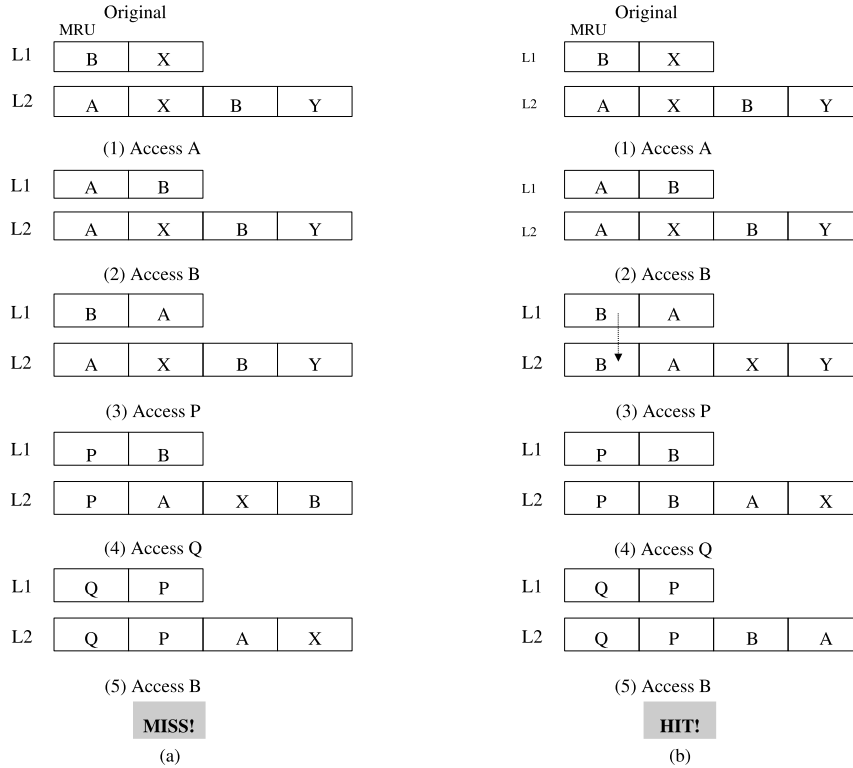


Figure 1. Example highlighting the problems with local policies (a) as opposed to global policies (b)

changes in how cache hierarchies are designed). In the next section, we provide the background for local and global policies. Section 3 then details our simulation methodology. Section 4 attempts to find situations where global replacement policies are useful by exploring a wide range of cache sizes, line sizes, associativities, inclusion properties and multi-core scenarios. After all of these negative results, Section 5 employs reuse-distance analysis to deconstruct the observed phenomena, and Section 6 draws some final conclusions.

2. Background and Motivation for Study

In a traditional multi-level cache hierarchy, accesses that hit in the L1 need not be propagated to the L2, which reduces the port requirements for the L2 structures as well as the power consumption. For programs exhibiting good locality, accesses are satisfied directly by the first level cache and thus these lines, if present in the higher levels, are touched very infrequently. Due to this cache access property, the higher levels see only a filtered stream of accesses which is dependent on the miss rates observed by the lower levels. The L1 is not always able to satisfy all memory requests due to large working sets and changing application phases; these situations present a potential opportunity for global cache replacement schemes to be useful.

To understand this better, consider a processor with two levels of cache as shown in Figure 1. In this example we walk through a short snippet of memory accesses to elucidate the problems of local replacement policies. We assume a 2-way set associative L1 cache and a 4-way set associative L2 for simplicity. Both the levels use an LRU replacement policy. The original cache data shows that B and X are two cache lines present both in the L1 and the L2. In this example we do not enforce inclusion among the caches. Consider Figure 1(a) first which describes the working of a local LRU replacement policy. The first memory access is to A. This access hits in the L2 and is brought into the L1 at the most-recently-used or MRU position. The second memory access is to B. This access hits in the L1 itself and is hence not seen by the L2. In this step we can see that B is moved to the MRU position in the L1 but it is untouched in the L2. The third access, which is to P misses in both the levels and needs to evict Y from the L2 and A from the L1. Similarly when the fourth memory access to Q is issued, which misses in both the levels, it needs to choose a line to evict from both the caches. Since the L1 is only 2-way associative B is chosen as the LRU line to evict from the L1. The L2 is 4-way associative but since the access to B was never seen by the L2 cache, it is in the LRU position in the L2 as well and is replaced. Finally when the memory access to B

is issued it misses in both the cache levels and needs to be fetched from memory.

The filtering effect shown in Figure 1(a) may lead to inefficiencies in the cache replacement policy as evidenced by the eviction and subsequent miss on line B. Figure 1(b) shows the exact same access sequence, but this time using a global scheme where the L2 can observe and update its replacement state based on the global memory access stream. The original cache data is the same as in part (a). The first access to *A* brings it in the MRU position in the L1 similar to the local scheme. Since we employ a global scheme the L2 sees this access too; however *A* is anyway in the MRU position, hence there is no change in the L2. The second memory access, which is to *B* brings *B* in the MRU position in the L1. This access is also propagated to the L2 denoted by the arrow in the figure which now brings in *B* into the MRU position in the L2 as well. The next memory access, which is to *P* is similar to the local scheme and evicts *Y* from the L2. The access to *Q* however replaces *X* from the L2 instead of *B* since *B*'s position was updated when it was accessed earlier. Hence the final memory access which is to *B* now hits in the L2 and we can avoid the miss to main memory.

By employing a global replacement policy, such as that shown in Figure 1(b), all levels of cache are able to implement true LRU-evictions based on the actual program-level memory access sequence. With such a scheme, lines having a memory access pattern like *B* as shown in the example, have a higher probability of being found in the cache hierarchy. Having explained the qualitative benefit of a global cache replacement policy, in the following sections we evaluate its usefulness.

3. Experimental Evaluation

3.1. Simulation Parameters

Our baseline processor configuration is modeled after a contemporary x86 microarchitecture. We use a cycle-level simulator built on top of a pre-release version of SimpleScalar for the x86 ISA [2]. Our simulator models many low-level details of the microarchitecture including limited decoding bandwidth for complex instructions that require microsequencer assistance, decomposition of x86 macro instructions to micro-op (uop) flows, wrong-path fetching past multiple branches, additional latencies between branch misprediction detection and front-end recovery, and many other details. Particularly important to this work, our simulator includes detailed models of the cache hierarchy including contention for inter-level buses, queuing delays, finite MSHR capacities, and memory controllers that reschedule accesses to improve page-level locality [17].

Our baseline configuration uses a 96-entry reorder buffer, 32-entry issue queue (RS), 32-entry load queue, 20-entry store queue, and a 14-stage pipeline depth (minimum

branch misprediction latency). The processor is 4-wide throughout its in-order pipelines (fetch/dispatch/commit), and 6-wide at issue (similar to the Intel Core 2[6]), with a 4K-entry TAGE branch predictor [18] and 2048-entry/4-way BTB, 3 integer ALUs, 1 integer multiplier, 1 load port, 1 store port, 1 FP adder and 1 FP complex unit. The default memory hierarchy has 32KB/8-way/3-cycle L1 data and instruction caches, a 4MB/16-way/9-cycle unified L2 cache, a 32-set/4-way instruction TLB, and 64-set/4-way DTLB with 250-cycle main memory latency. To implement a global replacement policy, we allow every access to the L1 cache to be automatically visible to the larger L2 cache (specifically every hit in the L1 results in an update being sent to the L2) without incurring any additional overhead due to bus contention, port contention, etc. If the accessed line is present in the L2 it is updated to be the most-recently-used line in the set. In case the line is not present, we simply drop the access.

We simulated a variety of applications from the following benchmark suites: SPECcpu2000, SPECcpu2006, MediaBench [9, 11], Physics [14], BioBench [1] and BioPerf [3]. All applications use the reference inputs. For applications with multiple reference inputs, we weight the separate runs such that each individual benchmark weighs equally in the final comparisons.¹ To reduce simulation time, we used the SimPoint 3.2 toolset to choose representative samples of 100 million instructions [13]. All simulations warm the caches for 500 million instructions and then perform cycle-level simulation for 100 million instructions. We report performance results based on the committed micro-ops per cycle rates.

4. Results

4.1. Baseline Experiments

The performance results of updating the L2 cache on each and every memory access is presented in Figure 2. We present average performance numbers for all the suites as well as per-benchmark results for the SPEC applications. The first bar shows the performance for the baseline (local) cache configuration, while the second bar is for a global cache in which all accesses to the L1 update the L2. As mentioned earlier, the performance of the global scheme does not include the negative effects of bus and cache contention due to the additional updates from the L1 to the L2. Despite the intuitive argument for why global policies should be effective, we observe practically no performance improvement using a global cache for all of the benchmarks. These results corroborate Zahran's observations that the global update scheme is not effective on SPECcpu2000 applications for a conventional cache hierarchy [21]. Zahran

¹For example, `eon` has three input sets, and so each would receive a weight of 0.33, whereas `mcf` has only a single reference input and we assign it a weight of 1.0.

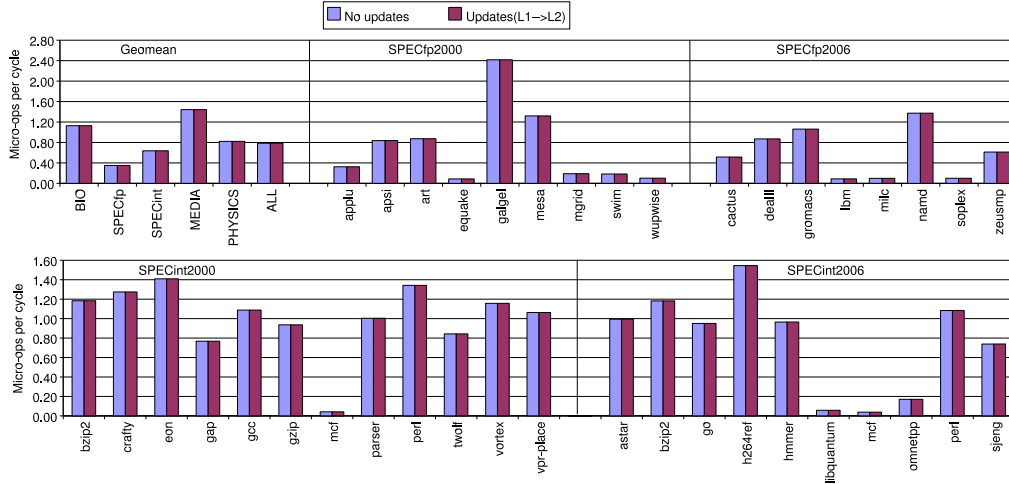


Figure 2. Performance results for all simulated applications

hypothesized that the larger working sets of SPEC2006 might stress the cache hierarchy a lot more and generate more opportunities for global policies to be effective. Our results include these benchmarks, but still show no real impact. One difference between our results and Zahran’s is that he assumes a small 256KB L2 cache for his simulations, whereas we employ a more contemporary 4MB L2 cache. Our larger cache results in many fewer L2 misses than when a smaller 256KB cache is considered, but we do not feel that 256KB is a particularly interesting design point when modern processors already contain caches as large as 6MB [10]. In the rest of this section, however, we do attempt to explore a much larger range of the cache design space in search of configurations where global schemes demonstrate some benefit.

4.2. Basic Cache Parameters

A cache’s miss rate is very sensitive to the total size of the cache (impacts capacity misses) and the set-associativity of the cache (impacts conflict misses). As the total number of sets decreases in the cache, a given cache set will have more lines mapping to it. This in turn may result in cache lines being more rapidly moved to the less recently used end of the LRU stack and thereby risk an earlier eviction. This early eviction may occur even though the copy of this line in the L1 may be frequently used. Such a scenario may benefit from global cache replacement. Similarly, as set-associativity decreases, a given cache line has fewer “chances” before it gets evicted; if the line is frequently accessed in the L1, then this situation may also benefit from a global scheme.

We explored these possibilities by decreasing the L2 cache size in multiples of two starting from our baseline 4MB cache all the way down to a 256KB cache to be con-

sistent with Zahran’s original study. We also varied the set-associativity from 4 to 64 ways while holding the total cache capacity constant at 4MB. We do not present a detailed graph of the results, as the relative trends are nearly identical to those shown earlier in Figure 2. Even for the smallest cache configuration (256KB), only on a handful of applications did we observe the L2 miss rate change by more than a few percent. The SPECfp2006 benchmark *zeusmp* showed a 2.5% decrease in its L2 MPKI rate; this did not translate into any substantial performance benefit since its baseline MPKI rate was only 1.46 to begin with. The only benchmark which showed a performance improvement was *wupwise*. For a 1MB/16-way configuration, the L2 MPKI miss rate went down by 6% and the performance went up by 7.17%. But on average there was negligible improvement in performance with updates. Some applications, for example *perl* actually showed a 1% performance degradation.

4.3. Maintaining Inclusion

In Zahran’s original study, he modeled a cache hierarchy that enforces the inclusion property. An inclusive cache maintains the property that any cache line stored in the L1 must also be found in the L2. That is, the L2 maintains a strict super-set property. To enforce this property, any line evicted from the L2 cache must also forward an invalidation to the L1 to ensure that the copy in the L1 (if it exists) is also removed. If the L2 line size is larger than the L1 line size, then a single L2 eviction may require multiple L1 invalidations. Avoiding an eviction of the L2 line can potentially save quite a few invalidations in the L1, as well as the subsequent cache misses that the invalidations would induce. A global replacement policy is one potential way to help prevent the eviction of such L2 cache lines.

In our simulator, we did not explicitly model a strictly

inclusive cache hierarchy. Instead, we simply measured the potential impact that maintaining inclusion would have. For every line evicted from the L2, we probe the L1 to see if the same line is present. If we find that the line is in fact present in the L1, then an invalidation would have been required to maintain the inclusion property. For each benchmark, we counted the number of evictions from the L1 that are a result of evictions from the L2. For our baseline 4MB/16-way L2 cache, the percentage of invalidated L1 lines, with respect to all L2 evictions is approximately 0.01%. For a very small 256KB/16-way cache, this number increases to 0.08%; in both cases, the impact in terms of the increase in L1 miss rates is negligible. As a result, we fully expect that a global replacement policy still will not help here (as was the case in the original study). For our final experiment to study the impact of inclusion, we modeled an L2 line size that was twice the size of the L1. In this configuration, an eviction from the L2 has the potential to evict two lines from the L1. In this experiment too, for the baseline configuration with a 4MB/16-way cache the percentage of L1 lines evicted due to an L2 eviction with respect to all L2 evictions is 0.01%. For a smaller configuration with a 256KB, 16-way L2 cache, this percentage increases slightly to 0.28% which is again very small.

4.4. Multi-Core Shared Caches

Modern processors now contain multiple execution cores that often share a single on-chip L2 cache. This presents new opportunities for global replacement policies to provide some benefit. In particular, consider a situation where core 1 executes a program that has a high L1 hit rate, and core 2 executes a program with low L1 and L2 hit rates. What could happen is that the program running on core 2 will constantly miss in the L1 and L2 and consequently bring in many cache lines from main memory. Since core 1 is hitting in its L1 cache, it does not need to access the L2. In the meantime, all of core 2's newly installed cache lines quickly push the L2 copies of core 1's lines to the LRU positions and then evict them from the cache. If core 1 then needs to reaccess one of these lines, it will already have been evicted, leading to unnecessary misses. With a global scheme, however, each of core 1's accesses will be propagated to the L2, thereby maintaining these lines closer toward the MRU positions of the LRU stack. This would help to protect core 1's line from premature eviction due to core 2's poor cache locality.

We measure the impact of global updates in a multi-core scenario. We simulated a two-core system with the same baseline configuration as described in Section 3.1. The two cores have private L1 caches, but share the L2 cache. One benchmark is run on each core. All the simulations warm the caches for 500 million instructions and then perform cycle-level simulation for 100 million instruc-

tions *per benchmark*. If one of the benchmarks completes execution earlier, the statistics for that core are frozen but the benchmark loops till the other benchmark completes execution, thus maintaining contention for the shared L2 cache. We simulated twenty-eight pairs of benchmarks from the SPECcpu2006 suite. The pairs were carefully chosen such that benchmarks with varying L2 cache behaviors and requirements were paired with each other. Here too, for a 1MB/16-way and 4MB/16-way L2 cache size, we see no significant performance difference with and without updates. One benchmark-pair comprising of *lbm* and *astar - rivers* showed a 1% performance improvement when global updates are employed. Here *lbm* is an application which has a very high miss rate and *astar - rivers* is an application whose miss rate is very sensitive to the number of ways it is allocated. So any update that keeps *astar - rivers'* data in the cache helps its performance. The updates that belonged to *lbm's* data, however, do not help at all, hence we see only a slight improvement. But for all the other application pairs the performance difference is negligible.

5. Why Do Global Updates Not Matter?

The results in the preceding section indicate that global caches do not provide any performance improvement under almost *any* circumstances. This result seems counterintuitive given that a global cache hierarchy has more information about memory access patterns and therefore should be able to make better replacement decisions. In this section, we first observe that even with global policies, the overall L2 cache miss rates do not decrease by very much. We then attempt to deconstruct the observed results by employing reuse-distance analysis to characterize the required conditions for global policies to provide benefit.

5.1. Impact on Miss Rates

One quick test for any new cache management scheme is to check the impact on cache miss rates. What we observed is that even with global updates to the L2 cache, the overall L2 MPKI miss rate does not change much. One problem is that while global update policies may prevent misses on some cache lines, each miss removed may come at the cost of some other line missing later. For example, by keeping a line J in the cache for longer, J ends up consuming space that could have been used to store some other line K . By caching J , we prevent a miss on the next access of J , but in turn introduce a new miss on the next access of K . The net result may be that the *total* number of misses is unaffected. Referring back to our original example in Figure 1(b), the access to B that was propagated to the L2 saved us a miss on the next access to B . This update, however, ended up evicting X from the L2. Following this code snippet, if there is an access to X now this will miss, keeping the absolute

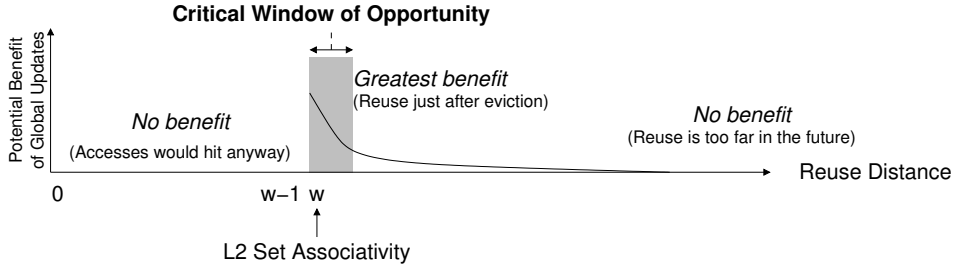


Figure 3. Illustration of a range of memory access reuse distances annotated with the potential impact of global updates

number of misses unchanged. Extending this example, we can potentially see scenarios where global updates can actually cause the performance to go down.

Another interesting result which helps explain the lack of performance improvement comes from our examination of the L2 state when we propagate L1 updates to the L2. In particular, we ran an experiment where if an update “hits” in the L2, we make note of the line’s position in the L2 recency stack. We found that for the vast majority of situations, the updates from the L1 accesses would find that the corresponding copy in the L2 was already at, or very close to, the MRU position in the L2 recency stack. As a result, the global updates from the L1 cache did very little to perturb (for better or for worse) the replacement information viewed by the L2. This also leads us to conclude that the misses are dominated by capacity or compulsory misses; neither of which would benefit much from knowing more about the global access patterns.

5.2. Reuse Distance Requirements

Reuse distance is another useful metric for characterizing a program’s memory access patterns [4, 8, 12]. We start with Ding and Zhong’s definition of reuse distance [5], but we generalize it to measure reuse distances on a per-set basis. More precisely, our reuse distance for an access to a cache line is the number of distinct (unique) data elements accessed (in the same set) before the same cache line is accessed again. A reuse distance of 4 for cache line A means that between two uses of A there were 4 unique or distinct references in the same set.

Consider the following example access sequence to a single cache set:

$$A_1 B_2 B_3 B_4 C_5 B_6 D_7 D_8 C_9 C_{10} E_{11} B_{12} B_{13} C_{14} D_{15} A_{16}$$

where the letter “A” is the address of a cache line and the subscript is just used to indicate the ordering/sequence of the accesses, so that A_1 and A_{16} are both accesses to the same cache line A, but just at different locations in the overall access pattern. In this example, there is a total of fourteen accesses seen between A_1 and its reuse A_{16} , but only

four of these (B , C , D and E) are unique. Thus, the reuse distance for A_1 is four. Note that for the accesses B_2 and B_3 , there is no unique access between these two requests, and therefore the reuse distance for B_2 is zero.

For a 16-way set-associative cache, any access with a reuse distance of 15 or less will not result in a cache miss on the reuse. For example, when we first access a line X , it will be placed in the MRU position of the recency stack. A reuse distance of 15 means that we will first observe 15 other unique addresses to this set, which would end up placing address X in the LRU position. Line X will not leave this position until the next access of X at which point the line will be re-promoted back to the MRU position in the recency stack. Therefore, for any accesses with reuse distances less than the set associativity of the cache, a global replacement scheme will not provide any benefit because the line would not have been evicted from the L2 in the first place.

A cache access with a reuse distance of exactly 16, however, would be the perfect candidate for a global replacement scheme. Such a line gets evicted and then is immediately recalled. In such a case, even a single update from the L1 may promote the line back to the MRU position and keep the line in the L2 cache for long enough to provide a cache hit when we get to the reuse. As the reuse distance increases, however, the cache line needs to be kept around for longer and longer to prevent the miss when eventually reused. One danger of keeping the line around for a long time is that the cache has no way of knowing *a priori* whether or not the line’s reuse distance is 16 or 1000. Caching a line with a long reuse distance may end up being very inefficient, as the line occupies space that might have otherwise been used by another line with a possibly much shorter reuse distance.

Figure 3 illustrates a plot of reuse distances and highlights the range of reuse distances that provide opportunities for a global replacement policy to be effective. Based on our earlier discussion, any accesses with reuse distances less than the set-associativity will not benefit from knowing about the L1 access patterns. Accesses with reuse dis-

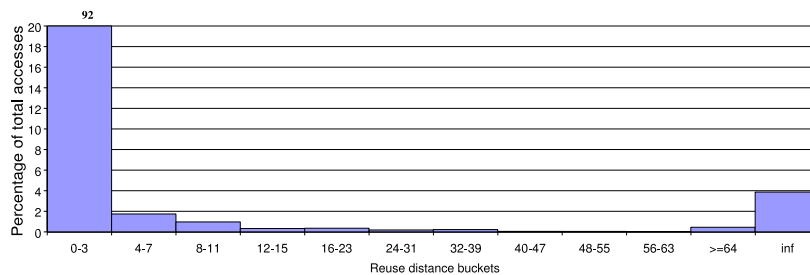


Figure 4. Histogram of the reuse-distances of memory accesses averaged across all SPECcpu2000 and SPECcpu2006 applications

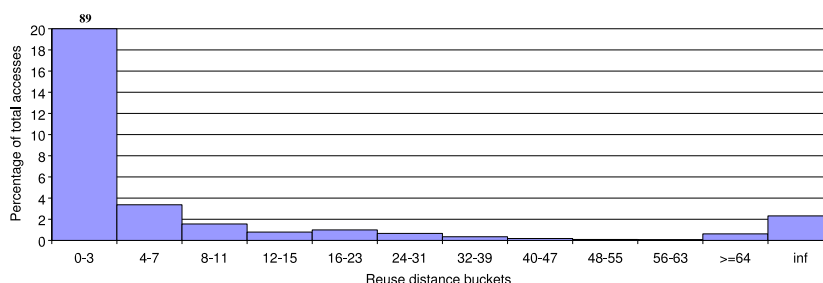


Figure 5. Reuse distance histogram showing the spectrum of memory accesses plotted for all SPECcpu2000 and SPECcpu2006 applications for a two-core processor

tances equal to and just barely greater provide the greater potential opportunity for benefit. Keeping such lines in the cache reduces misses due to the reuses while minimizing the time spent preventing other lines from being cached. As the reuse distance increases, the potential benefit rapidly reduces. Each line that with a large reuse distance kept in the cache can at most eliminate a single miss (on its next reuse), but the longer it is kept in the cache, the higher the probability that it causes misses due to other more useful lines not being able to be kept in the cache. Therefore, there exists only a small window of opportunity on the reuse chart shown in Figure 3. If programs exhibit a relatively large number of accesses with reuse distances that fall in this critical range, then global replacement policies should be very beneficial. Unfortunately, this window of opportunity is very narrow and it is not likely that many accesses would fall in this range.

We collected reuse-distance histograms for all the SPECcpu benchmarks to corroborate this assumption. The processor configuration used to collect these reuse distances same as described in Section 3.1. Figure 4 shows the histogram of the average reuse distances, and we can see that the fraction of cache accesses that fall in the critical range just beyond the set-associativity is very small. Most reuse distances fall in the 0-3 bucket indicating that most of the updates access lines that are very close to the MRU position and as such are in little danger of being evicted. This is

one of the primary reasons why we see no performance improvement. Additionally, the bucket with the next-largest fraction of accesses is the last one which encompasses memory accesses that touch an address only once and then never again, indicating that there simply is no opportunity for reuse. Keeping such lines in the L2 cache through a global policy is guaranteed to have no benefit.

A multi-core processor with a shared L2 cache increases the likelihood of a line getting evicted prior to reuse. The larger number of unique cache lines from the different cores mapping into the same cache sets can increase reuse distances. With enough cache contention, this may cause the number of accesses with reuse distances in the critical window of opportunity to increase. Figure 5 shows the reuse-distance histogram (averaged across all of our dual-core workloads). While the fraction of accesses in the critical reuse-distance range does in fact increase in this multi-core scenario, the absolute number of such accesses is still too few to have any significant impact on performance.

5.3. ‘Global’ Reuse Distance Pattern and its Impact on Global Cache Replacement Policies

The histograms presented above may seem a little surprising, given the fact that some earlier studies on reuse distance analysis have indicated that the histograms might be more flat. In this sub-section we explain this anomaly. Our earlier discussion on reuse distances used an adapted defi-

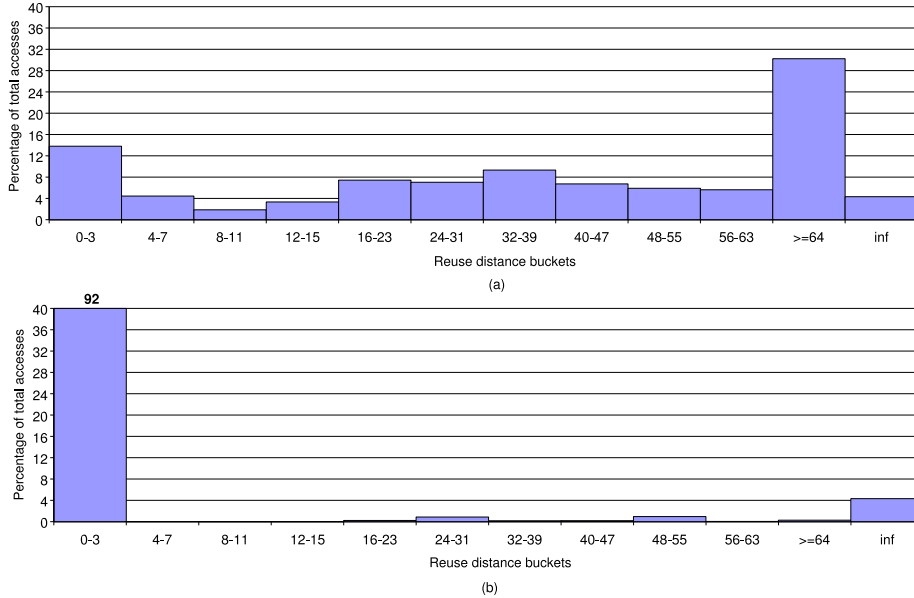


Figure 6. Reuse distance histogram for cactus presented on (a)global memory access pattern (b) per-set access pattern

inition of reuse distances which collected information on a **per-set** basis whereas most of the previous work considered reuse distances based on the global memory access pattern. A global reuse distance study treats the cache as having a single set (fully-associative cache) where a line may be placed anywhere in the cache. Hence, there are most certainly going to be more number of unique accesses between two accesses of a line which would increase the reuse distances. We measured the global reuse distances and found that quite a few applications show a more spread out histogram where not all the accesses were concentrated near the MRU position. Figure 6 presents the reuse distance pattern for *cactus*, a SPECfp2006 application. In (a) we present reuse distances across the global access pattern and in (b) we present reuse distances collected on a per-set basis. We can clearly see that in contrast with the per-set reuse pattern, the global reuse pattern is more spread out with a considerable number of reuse distances falling outside the 0-3 bucket. This global reuse pattern however is immaterial for the purpose of measuring the benefit of global cache replacement policies for a set-associative cache. The global updates are supposed to help prevent the eviction of lines which are likely to be replaced from their set thereby reducing the occurrence of conflict misses. A fully-associative cache has no conflict misses; the misses are converted into capacity misses which as explained earlier cannot benefit from global updates. Thus we can see that even if the global histograms show more spread out reuse distances, there is still no scope for a cache replacement policy which uses global updates to work.

6. Conclusions

In this paper, we have explored a very wide range of possible cache configurations in search of scenarios where a global replacement scheme will provide a performance benefit. After searching through many different cache sizes, associativities, inclusion properties, and multi-core arrangements, we are unable to find any situations where managing an L2 cache with global updates from the L1 proves to be useful. Despite the initially intuitive motivation for this idea, we conclude that global update policies are critically flawed for two main reasons. The first is that only cache accesses that fall in a very narrow range of reuse distances stand to have any chance of providing a benefit with global updates, and in all applications that we studied, only a trivial fraction of accesses actually fell in this range. The second reason is that even when a miss can be averted, it is likely that we are only trading misses; reducing a miss on one address may create a miss on a different address, leading to no net improvement.

Our characterization of the window of opportunity using reuse-distance analysis is particularly condemning for this idea. Even with different L1 and L2 cache sizes, associativities, and other variations on organization, such changes will only slide the window of opportunity to the left or right in our reuse histograms, but these changes do not fundamentally increase the size of this window. Unless we can find applications with reuse-distance characteristics that are *drastically* different from what we have observed, we conclude that it is highly unlikely that a global scheme will ever provide any significant advantage to warrant its incorpora-

tion into any real cache hierarchy implementation.

References

- [1] Kursad Albayraktaroglu, Aamer Jaleel, Xue Wu, Manoj Franklin, Bruce Jacob, Chau-Wen Tseng, and Donald Yeung. BioBench: A Benchmark Suite of Bioinformatics Applications. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, pages 2–9, Austin, TX, USA, March 2005.
- [2] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Micro Magazine*, pages 59–67, February 2002.
- [3] David A. Bader, Yue Li, Tao Li, and Vipin Sachdeva. BioPerf: A Benchmark Suite to Evaluate High-Performance Computer Architecture of Bioinformatics Applications. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 163–173, Austin, TX, USA, October 2005.
- [4] Kristof Beyls and Erik D’Hollander. Reuse Distance as a Metric for Cache Behavior. In *Proceedings of the 13th IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 617–622, Anaheim, California, August 2001.
- [5] Chen Ding and Yutao Zhong. Predicting Whole-Program Locality through Reuse Distance Analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–257, San Diego, California, June 2003.
- [6] Jack Doweck. Inside Intel Core Microarchitecture and Smart Memory Access. White paper, Intel Corporation, 2006. <http://download.intel.com/technology/architecture/sma.pdf>.
- [7] Haakon Dybdahl and Per Stenstrom. An Adaptive Shared/Private NUCA Cache Partitioning Scheme for Chip Multiprocessors. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture*, Phoenix, AZ, USA, February 2007.
- [8] Changpeng Fang, Steve Carr, Soner Onder, and Zhenlin Wang. Reuse-distance-based Miss-rate Prediction on a Per Instruction Basis. In *Proceedings of the 2nd Workshop on Memory System Performance in conjunction with ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 62–68, Washington DC, USA, June 2004.
- [9] J. E. Fritts, F. W. Steiling, and J. A. Tucek. MediaBench II Video: Expediting the Next Generation of Video Systems Research. 5683:79–93, March 2005.
- [10] Intel Corporation. Introducing the 45nm Next Generation Intel Core Microarchitecture. *Technology@Intel Magazine*, 4(10), May 2007.
- [11] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 330–335, Research Triangle Park, NC, USA, December 1997.
- [12] R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. In *IBM System Journal*, pages 78–117, 1970.
- [13] Erez Perelman, Greg Hamerly, and Brad Calder. Picking Statistically Valid and Early Simulation Points. In *Proceedings of the 2003 International Conference on Parallel Architectures and Compilation Techniques*, pages 244–255, New Orleans, LA, USA, September 2004.
- [14] Physicsbench. <http://sourceforge.net/projects/physicsbench>.
- [15] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely Jr., and Joel Emer. Adaptive Insertion Policies for High-Performance Caching. In *Proceedings of the 34th International Symposium on Computer Architecture*, San Diego, CA, USA, June 2007.
- [16] Kaushik Rajan and Govindarajan Ramaswamy. Emulating Optimal Replacement with a Shepherd Cache. In *Proceedings of the 40th International Symposium on Microarchitecture*, pages 423–434, Chicago, IL, 2007.
- [17] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory Access Scheduling. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 128–138, Vancouver, Canada, June 2000.
- [18] André Sez nec and Pierre Michaud. A Case for (Partially) TAgges GEometric History Length Branch Prediction. *Journal of Instruction Level Parallelism*, 8:1–23, 2006.
- [19] Ranjith Subramanian, Yannis Smaragdakis, and Gabriel H. Loh. Adaptive Caches: Effective Shaping of Cache Behavior to Workloads. In *Proceedings of the 39th International Symposium on Microarchitecture*, pages 385–396, Orlando, FL, December 2006.
- [20] Joel M. Tandler, Steve Dodson, Steve Fields, Hung Le, and Balam Sinharoy. POWER4 System Microarchitecture. *IBM Journal*, pages 25–33, October 2001.
- [21] Mohamed Zahran. Cache Replacement Policy Revisited. In *Proceedings of the 6th Workshop on Duplicating, Deconstructing, and Debunking*, San Diego, CA, USA, June 2007.