Cache Replacement Policy Revisited

Mohamed Zahran
Electrical Engineering Department
City University of New York
mzahran@ccny.cuny.edu

Abstract

Cache replacement policy is a major design parameter of any memory hierarchy. The efficiency of the replacement policy affects both the hit rate and the access latency of a cache system. The higher the associativity of the cache, the more vital the replacement policy becomes. Therefore, a lot of work has been proposed, both in industry and academia, to enhance the performance of the replacement policy. However, all the proposed schemes are local in nature.

The memory system does not consist of a single cache, but a hierarchy of caches. If each cache in the hierarchy decides which block to replace in an independent manner, the performance of the whole memory hierarchy will suffer, especially if inclusion is to be maintained. For example, if the level 2 cache decides to evict an LRU block, one or more blocks at level 1, possibly not LRU, can be evicted, therefore affecting the hit rate of level 1 and the overall performance. So, is a *global* replacement algorithm needed?

In this paper we study the feasibility of having a global replacement algorithm. The contribution of this paper is threefold. First, we show the shortcomings of the localized replacement policies. Second, we propose several *global* replacement policies that tackle the shortcomings of the localized version. Third, we show when the proposed techniques do not work, and the characteristics of an application that makes use of these techniques. This will lead to a new way of thinking about replacement policies in general.

1 Introduction

As the gap between the processor speed and memory speed increases, the role of the cache hierarchy becomes more and more crucial. Having several levels of caches is currently the most efficient method to deal with the memory wall problem. However, the design of an efficient cache hierarchy is anything but a trivial task.

The design of a cache system involves making several decisions, in terms of the size of each cache in the hierarchy, the block size, the associativity, etc. One pivotal design decision is the replacement policy. Replacement policy affects the overall performance of the cache, not only in terms of hits and misses, but also in terms of bandwidth utilization and response time[1].

A poor replacement policy can increase the number of misses in the cache, cause a lot of traffic out of the cache and increase the cache miss penalty. Because of these reasons, a lot of research, both in academia and industry, is geared toward finding the best cache replacement policy. But almost all of the proposed work deals with replacement policy *within a single cache*.

These local replacement policies, although efficient within a single cache, do not take into account the interaction between the different caches in the hierarchy. For instance, when an L1 cache sends a victim dirty block to L2 cache, it affects the LRU stack at L2[2]. Both the private L1 instruction cache (il1) and private L1 data cache (dl1) affect the LRU stack at the shared L2 cache (ul2) in a random way. This makes ul2 behave in an unpredictable way, leading to many misses and increased traffic up and down the hierarchy. Moreover, this random behavior makes it difficult to study the overall performance of a cache hierarchy except through simulations, which is time consuming. Is a *global replacement policy* needed?

The contribution of this paper is threefold. First, we show that local replacement policies may not always be the correct way to go for obtaining the most efficient cache hierarchy. Second, we propose several global replacement policies and discuss their behavior with several benchmarks using a cycle accurate simulator. Third, we show that for some benchmarks, the global replacement schemes do not perform much better than their local counterparts, and we discuss the characteristics of an application that can benefit from the global schemes.

The rest of this paper is organized as follows. Section 2 discusses some of the problems associated with local replacement algorithms. Section 3 proposes several global replacement algorithms. The performance of the proposed global policies is compared to the local policies in Section 4. Some related work is discussed in Section 5. Finally, Section 6 summarizes the paper and

[1] Throughout this paper, we assume an inclusive cache hierarchy

[2] In this paper we use LRU replacement policy as an example. However, the proposed ideas can be applied to any kind of replacement policy, except the random of course.

shows our roadmap for the global policies.

2 Local Replacement Hazards

Local replacement policies keep track of blocks behavior within each cache set in order to be able to choose a victim block when a replacement is needed. The replacement policy gets its own information from accesses to its cache. However, a replacement policy p_i at cache L_i can indirectly affect the behavior of the replacement policy p_j at another cache L_j through a cache miss. This is because p_i affects the number of cache misses in L_i and whether the victim block is dirty or not. The problem is that both of these factors also affect the replacement policy behavior of the cache below L_i in the hierarchy.

Therefore, the *communication* between the replacement policies of two caches in the hierarchy is established only after a cache miss. This limited connection between caches in terms of replacement policies may lead to many inefficiencies.

The first of these inefficiencies is the replacement of a non-LRU block at L1 because its corresponding block at L2 became the LRU and needs to be replaced. Beside affecting the hit rate of the cache hierarchy, this scenario also increases the miss penalty. If L1 misses on an access, L2 is accessed. If L2 misses, it will need to choose a victim for replacement if the set is full. The victim must be replaced before the new block is brought. And if the victim has corresponding blocks at L1, these blocks must be invalidated, which takes time.

Another side effect occurs when the block size at L2 is larger than the block size at L1, in which case a single block replacement at L2 may lead to several replacements at L1. Moreover, the new incoming block, in case of L2 miss, may cause yet another replacement at L1 if it maps to a different set than the one that has been invalidated by the L2 replacement. This back-to-back replacement contributes to the increase in miss-penalty of the cache hierarchy. Figure 1 illustrates the above problems. The shaded blocks in L1 cache represent possible evacuation due to a single replacement at L2. It is to be noted that more than two blocks can get evacuated from L1 if the block size at L2 is larger than the block size at L1.

A major drawback of local replacement policies is the increase of traffic between L1-L2 caches, and between L2-main memory. This is due to the invalidation of dirty blocks in L1 as a result of a replacement at L2. In addition, the secondary effect of back-to-back replacement increases the traffic if the replaced block is dirty.

There are several reasons for the above hazards. First, L2 cache is *shielded* from program behavior by

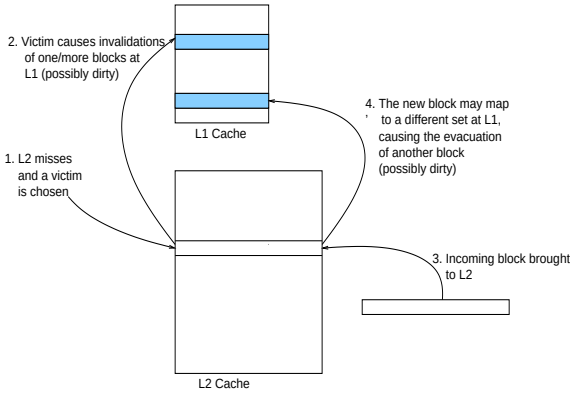

Figure 1: Several Evacuations at L1 from A Single Replacement at L2

L1 cache. So the LRU stack at L2 is updated only after a cache miss at L1. These updates sent to L2, in the form of block inquiries and/or dirty blocks write-backs, contain only partial information about the program behavior. This means the behavior of L1 cache affects the behavior of L2 cache. Second, if L2 cache is shared, it gets randomly updated by both L1 instruction cache and L1 data cache, leading to distorted information about the program behavior. Third, if the block size at L2[3] is larger than the block size at L1, a single replacement at L2 can have several effects at L1 cache. Finally, in the case of shared L2 cache, a single set at L2 may consist of blocks for instruction cache and blocks for data cache, each of which has different behavior.

Based on the above discussion, if the aforementioned scenarios frequently occur in an application, then local replacement policies will perform poorly and a new *global* scheme is needed.

3 Global Replacement

A global replacement policy is one that controls the replacement of blocks in all the caches in the hierarchy in an attempt to avoid the aforementioned problems of local policies.

3.1 Types of Global Policies

There are two main types of global policies; communication-based and centralized. In the communication-based scheme, each cache in the hierarchy has its own local replacement policy. However, they exchange information regarding the behavior of the program. The main advantage of this scheme is its hard-

[3]Throughout this paper, we assume two-level cache hierarchy. However, the work presented can scale to cache hierarchies of any height.

ware simplicity, as the hardware required to choose the victim block in each cache of the hierarchy is almost unchanged. The price we pay for that scheme is the extra traffic required to exchange information about the program behavior among the caches in the hierarchy.

The second category of global policies is the centralized scheme. In thi scheme, there is only one controller responsible for making replacement decisions for all caches in the hierarchy. This scheme is very flexible because it can easily work with inclusive, non-inclusive, and exclusive cache hierarchies. However, due to its centralized behavior, it can be a bottleneck of performance and can suffer from scalability problem. Figure 2 shows the two types of global replacement. For the centralized type, there are several implementations. For instance, there may be a centralized cache controller for the whole hierarchy. A different implementation is to have separate cache controllers for each cache, as in the traditional systems, but the part responsible for replacement is centralized for the whole hierarchy.

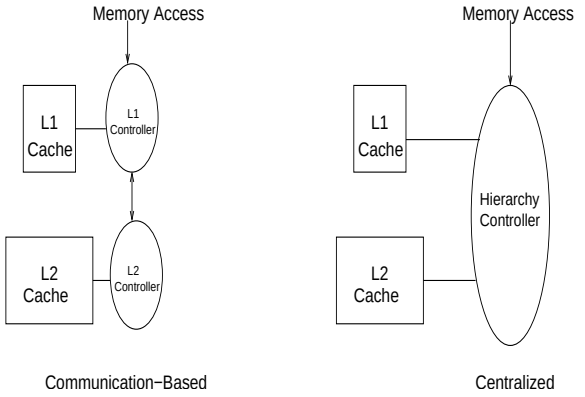

Figure 2: The Two Types of Global Replacement Schemes

In this paper, we explore several communication-based global replacement algorithms.

3.2 Proposed Schemes

The main cause of the problems faced by local replacement policies is that the different caches in the hierarchy are not equally updated with the behavior of the application. The cache at L1 is getting the exact behavior of the program. As we go higher in the cache hierarchy, that is, toward the main memory, caches at each level are getting diminishing information about the application behavior. Therefore, the first technique we propose, called **global-both**, updates the LRU stack of *all* the caches in the hierarchy after each cache access at L1. This means that all cache accesses are made visible to the LRU mechanism of all caches in the hierarchy. In this case, all the caches in the hierarchy will have full information about the program behavior, and the replacement decision will be beneficial to the program execution. These updates, propagated in the cache hierarchy, are not in the critical path of the cache system and therefore do not affect the cache access time.

In order to see whether all the updates are required for all caches, we propose another scheme called **global-filtered**. In global filtered, not all cache accesses to L1 are sent to L2. Only accesses serviced by the MRU block at L1 are sent to L2. In this case we guarantee that the MRU block at L1 will never be replaced due to a replacement at L2 cache. In addition, this scheme does not affect the cache access time.

Another scheme, aimed at reducing the back-to-back eviction mentioned in section 2, is to avoid replacing an LRU block at L2 unless the new incoming block is in the same set at L1 as the block that will be evicted from L2. If the LRU at L2 does not satisfy the above condition, the following block in the LRU stack is checked for the condition, and so on. If we reach the MRU block, we do not replace it even if it satisfies the condition, but instead replace the LRU block as in a conventional system. We call this system **global-sameset**. This scheme is activated when L1 misses. So by the time L2 is accessed, the set at L1 of the incoming block will be already known. The corresponding set at L1 of each block at L2 is known and can be stored with the tag at L2. Once L1 misses and L2 access starts, and while tags at L2 are compared, the set number of L1 is also compared to each set number at the corresponding set at L2. Therfore if L2 misses, we will already have the comparison result of the set numbers and the replacement choice can be done. The access time is not affected in that scheme, albeit with extra hardware.

Finally, the last scheme we propose, called **global-mru**, does not replace an LRU at L2 that has a corresponding MRU at L1. This scheme aims at eliminating MRU eviction at L1 due to an LRU eviction at L2. When a block becomes MRU at L1, the address of that block is sent to L2 cache, and the corresponding block at L2 is flagged as non-replaceable. When a block is flagged, the other block in the same set, if any, is unflagged.

The four schemes described above vary in their complexity and the type of scenarios they deal with.

3.3 Hardware Complexity

The proposed techniques require very little hardware. The global-both scheme does not require any hardware. The memory accesses are sent to L2 on the address bus at the same time as L1, just to update the LRU stack. The same is done for the global filtered method. For

the global-sameset, an address decoder, adjusted for L1 cache, is used at L2 to see whether the incoming block is in the same set as the L1 victim. The global-mru requires associating a bit with each block at L2 to indicate whether it corresponds to an MRU at L1.

4 Experimental Results

In this section we present preliminary results obtained from our simulations, to assess the efficiency of global replacement policies. First we show our experimental setup, then we present the results, and discuss their implications.

4.1 Experimental Setup

To conduct our experiments, we use the Simplescalar [1] with the PISA instruction set, modifying it to generate the statistics we need and implementing our set of proposed cache management modifications. Both the instruction (il1) and data (dl1) L1 caches are 32KB, 2-way set associative, with 64B lines. L1 data cache is write-through. We have chosen it to be write-through in order to be the closest to the global schemes in terms of frequency of ul2 update. Unless stated otherwise, all the caches in the global schemes are writeback. The unified L2 (ul2) cache is 256KB, eight-way associative, with LRU replacement, and 64B lines. We choose to have a fixed block size among all the caches in the hierarchy, similar to IBM POWER series [2], in order to isolate only the problems related to local replacements, and not problems related to different block sizes. We choose these numbers to represent current high-performance systems. Our hierarchies assume inclusion among the levels. Table 1 shows the fixed simulator parameters. Parameters not in the table use the default simulator values.

Parameter	Setting
Instruction Fetch Queue *Decode Width*	8 8
Issue width	8
Instruction Window	128
Load/Store Queue	8
Branch Predictor	combination of bimodal, 2048 table size and 2-level predictor
L1 Icache/Dcache	32KB, 2-way associative, LRU 64B line size,2 cycle latency
L2 Unified	256KB, 8-way associative, LRU 64B line size, 15 cycle access latency
Memory Latency	500 cycles for the first chunk
Memory Width	64B
Write Buffer	8 entries (between L1 and L2)
Execution Units	2 int and 1 fp ALUs 2 int mult/div and 1 fp mult/div

Table 1: Simulator Parameters

Benchmark	Total References	Skipped Instructions (in Millions)
bzip2	660852200	200
gcc	1116670488	1000
gzip	447876500	40
mcf	848274728	2100
parser	802472443	250
perl	672441228	500
twolf	765195129	400
vortex	826745934	0.5
vpr	653145629	500
ammp	830965138	2000
applu	382821775	500
apsi	474817653	30
art	638128299	2100
equake	485119040	2100
mesa	747598802	500
swim	235320480	250
wupwise	509670956	2100

Table 2: Simulated Applications

We use 17 benchmarks from SpecCPU 2000, we have problems compiling the others. After skipping the startup part as indicated in [3], we simulate each benchmark for 1.5 Billion instructions. Table 2 shows statistics per application, including the number of instructions skipped, and the total number of committed references (loads and stores). We use reference inputs except for `applu`, for which we use the train input.

4.2 Results and Analysis

In the following set of experiments, we compare the traditional LRU replacement policy to the proposed global algorithms. Although the LRU is the most popular and efficient replacement algorithm, it is not the cheapest in terms of hardware cost. Several pseudo-LRU algorithm have been proposed. We have implemented one of the pseudo-LRU replacements, which is the PLRUm [4]. PLRUm is a pseudo-LRU policy based on most-recently-used (MRU) bit. Each block has an MRU bit. Whenever a cache access hits on a block, the MRU bit of that block is set. When all the MRU bits in a set become 1, they are all cleared except the MRU. The cache controller replaces the block whose MRU bit is 0.

In the first set of experiments we look at the total traffic going between dl1 and ul2 caches. Figure 3 shows the total traffic in terms of MBytes. As expected, the LRU has the highest traffic because of its write-through. However, with PLRUm, we designed the system to be writeback. The global-both (gboth) has the lowest traffic, especially for SPEC INT benchmarks. This is because gboth keeps both LRU stacks, at L1 and L2, aware of the application behavior. Therefore, several of the evictions mentioned in Section 2 have been eliminated. But, as we will see later, these scenarios happen with low frequency in the SPEC benchmarks, or at least in the 1.5 billion instructions we simulated. The other

global schemes keep L1 LRU stack and L2 LRU stack aware of the application behavior but to different extends. For instance, the global-filtered (gfiltered) keeps L2 aware of the MRU behavior at L1 only. This eliminates most of the evictions, but because L1 is only 2-way set associative, most of the updates accesses at L1 get propagated to L2. The global-sameset (gsamset) and global-mru (gmru) put restrictions on the block to be evicted from L2, but do not prorogate updates about application behavior from L1 to L2, therefore their performance is lower than the other global schemes.

Figure 4 shows the total traffic between L2 and main memory in MBytes. Both gboth and gfiltered are doing slightly better than the other schemes, especially for SPEC INT benchmarks. This is because in these two schemes both L1 and L2 caches tend to follow the application behavior. This means that the blocks forming the current working set of the application tend not to be replaced, therefore decreasing the traffic going to the memory. However, because both il1 and dl1 are updating ul2, there must be some interference, leading to few conflict misses. This is the main reason for the traffic coming from memory. The highest traffic results from the usage of gmru. This scheme tends not to replace the LRU block at L2, which has the side effect of increasing the misses and hence the traffic coming from memory. The only exception is `cc1` where gmru has the least traffic. This is because `cc1` has high temporal locality. Hence, not evicting the MRU block at L1, which is the main function of gmru, tends to perform quite well for that benchmark.

One of the side effects of local replacement policies is to increase replacements at L1 cache. Figure 5 shows the replacement rate at dl1. The global policies are in general doing better than the local policies, that is, they tend to have lower replacement rate. The lowest replacement rate comes from gboth and gmru. The gfiltered and gsameset mechanisms have similar replacement rate as the local schemes.

Figures 6 and 7 show the number of misses per 1000 instructions for both L1 data cache and L2 shared cache. The gboth scheme has the lowest MPKI for L1 cache, which gives an indication that constantly updating L2 with L1 accesses leads to better cache behavior. The effect is not very apparent for SPEC FP because they have lower number of memory references than SPEC INT. Overall, the global schemes, with the exception of gmru, have lower MPKI than their local counterparts. However, the difference is not very high between the local ones and the global ones, especially schemes other than gboth.

Finally, the effect of cache hierarchy performance on the overall system is shown in Figure 8. The global schemes have an average of 5% lower CPI than the local schemes. This is mainly due to the fact that in the 1.5Billion instructions we simulated, the scenarios mentioned in Section 2 occurred only in less than 5% of the references, as indicated in Table 3. Gmru is doing the worst in all the benchmarks except `cc1`. This suggests that gmru cannot be used as a standalone scheme, but can be used in hybrid scheme with another policy.

From Table 3 we can see that when the scenarios occur, the global replacement can deal with them. The table shows the blocks invalidated at L1, both instruction and data, due to the replacement of an LRU block from L2. This happens more often for instruction cache than data cache. This is mainly due to the higher temporal locality for instructions than data. The two rightmost columns of the table show the same measurements but when using the gboth scheme. As we can see, they have been greatly reduced, even eliminated, in many cases for the data cache.

Benchmark	il1 (LRU)	dl1 (LRU)	il1 (gboth)	dl1 (gboth)
bzip2	70425	4330	29129	0
gcc	1434276	1433	1395972	2991
gzip	45077	9	27690	0
mcf	253420	21056	69236	0
parser	359669	7333	199566	0
perl	26322	6704	79	0
twolf	681397	15080	44430	0
vortex	67900	126357	31061	92154
vpr	783999	30421	65362	0
ammp	135629	828	59271	0
applu	200331	18360	12744	0
apsi	246450	1516	152995	0
art	121706	54791	33598	0
equake	64847	3511	3064	0
mesa	15967	2413	1723	12
swim	1050	184	6	0
wupwise	25090	1906	35	0

Table 3: Number of Invalidations for SPEC

4.3 Can We Do Better?

As we have seen in the previous section, the overall performance of the global policy for the SPEC benchmarks was less than impressive. This is due to the low (less than 5%) frequency of occurrence of the scenarios mentioned in Section 2. However, when these scenarios do occur, the global policy deals with them.

The question of whether the global policy is worth it depends on many factors. First, if the application has large cache footprint then the scenarios will occur more frequently. The working-set size of any SPEC 2000 benchmark is not large enough to make use of the global policy [5].

We started using SPEC 2006 but the results are not yet ready. We also started using DIS stressmark suite [6] and OLDEN benchmark suite. The OLDEN and DIS show similar behavior as the SPEC in terms of fre-

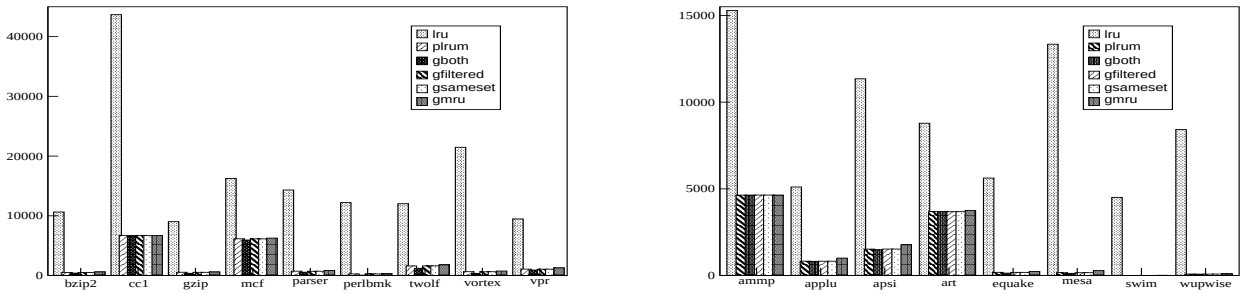

Figure 3: Total Traffic Between L1 Data Cache and Shared L2 Cache (in MBytes)

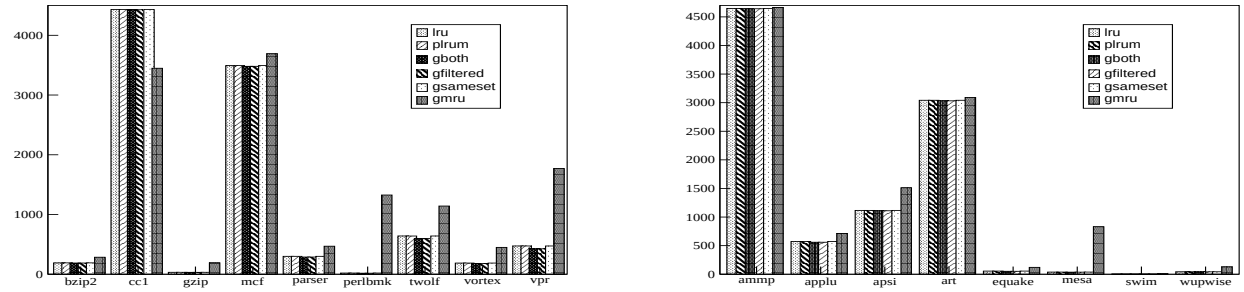

Figure 4: Total Traffic Between Shared L2 Cache and Memory (in MBytes)

quency of occurrence of the harmful scenarios. Table 4 shows the number of invalidations with local and global replacements. The global replacement has almost eliminated the invalidations due to replacement at L2.

Benchmark	il1 (LRU)	dl1 (LRU)	il1 (gboth)	dl1 (gboth)
field	1486	144	283	0
gups	0	0	0	0
matrix	2324865	15931	695382	0
neighbor	39666	334	28412	0
pointer	483	0	300	0
transitive	11260	149835	489	0
update	5318	384	322	0
bh	25266	2024	890	0
bisort	1547	52	209	0
health	625349	0	142129	0
mst	20363	284	690	0
perimeter	41955	1189	932	0
power	711	15	261	0
treeadd	33158	606	761	0
tsp	56996	2590	18969	0

Table 4: Number of Invalidations for DIS and OLDEN

5 Related Work

Most of the work done in cache replacement policy is local in nature. Some of the proposed policies adapt to the application behavior, but within a single cache. For instance, Qureshi *et.al* propose retaining some fraction of the working set in the cache so that some fraction of the working set contribute to the cache hits [7]. They do this by bringing the incoming block in the LRU position instead of MRU, thus reducing cache thrashing. Another adaptive replacement policy is presented in [8], where the cache switches between two different replacement policies based on the behavior. Several techniques are proposed to try to close the gap between LRU and OPT [9].

Cache misses are not of equal importance, and it depends on the amount of memory level parallelism (MLP) [10]. An MLP-aware cache replacement is presented in [11].

In order to study different replacement policies, a series of long simulations must be done. However, there are some proposals on using analytical models for cache replacement [12, 13].

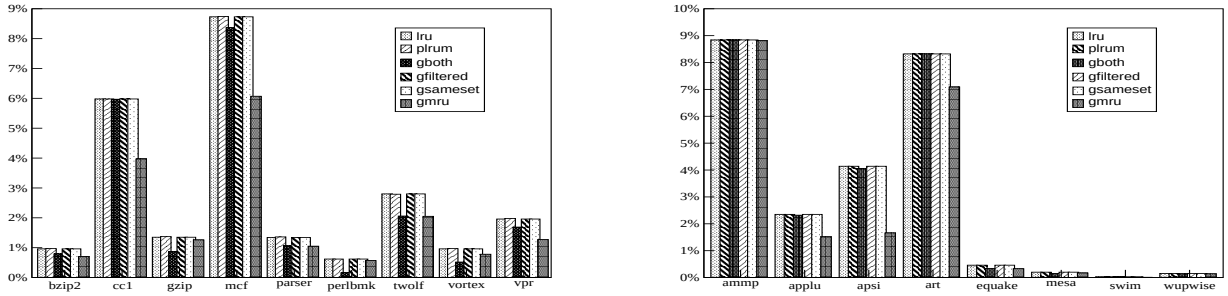

Figure 5: Replacement Rate at L1 Data Cache

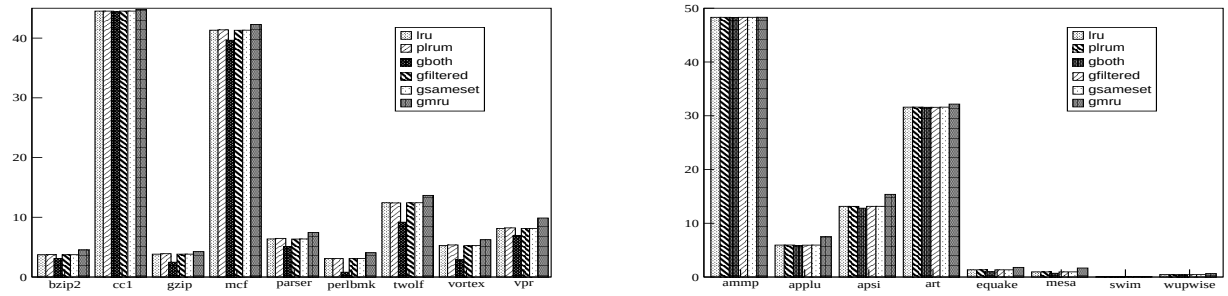

Figure 6: Misses Per Kilo Instructions (MPKI) for L1 Data Cache

6 Conclusion and Future Work

In this paper we discuss the effectiveness of local cache replacement policies. We show that local replacement may not be the most effective way to get the best performance from a cache hierarchy. We present several global replacement policies and some preliminary data about their performance. Although the global policies have greatly decreased the harmful scenarios, this did not materialize into much higher overall system performance. This is due to the low frequency of occurrence of the harmful scenarios within the 1.5 billion instructions we simulated from the SPEC benchmark.

Therefore we can say that the global replacement policy is worth using only for programs that have high percentage of L1 invalidations. But it is important to note that the global policy never does worse than the local, for instance gboth is always the best.

We plan to continue our investigations of the global replacement policies. First we would like to test our schemes with other benchmarks that stress the memory system more than the SPEC CPU 2000. Second we will conduct several experiments with different cache configurations to see the sensitivity of our schemes. Finally we plan to extend our scheme to work in multi-core environment, where the replacement policy is done globally among all cores.

References

[1] D. Burger and T. Austin, "The simplescalar toolset, version 2.0," Tech. Rep. 1342, University of Wisconsin, June 1997.

[2] B. Sinharoy, R. N. Kalla, J. M. T. andR. J. Eickemeyer, and J. B. Joyner, "Power5 system microarchitecture," *IBM Journal or Research and Development*, vol. 49, no. 4/5, 2005.

[3] S. Sair and M. Charney, "Memory behavior of the spec2000 benchmark suite," Tech. Rep. RC-21852, IBM T. J. Watson Research Center, October 2000.

[4] H. Al-Zoubi, A. Milenkovic, and M. Milenkovic, "Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite," in *Proc. 42nd ACM Southeast Conference*, 2004.

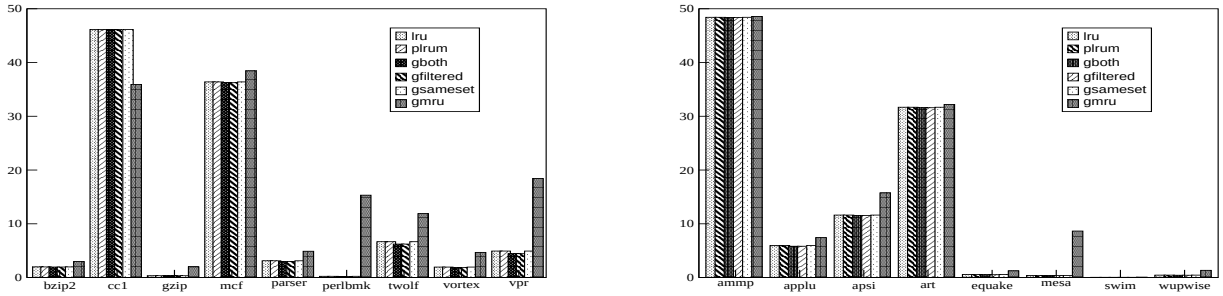

Figure 7: MPKI for Shared L2 Cache

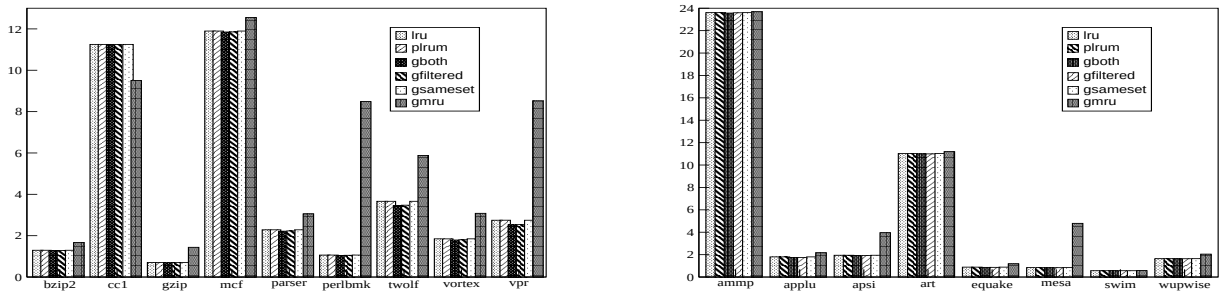

Figure 8: CPI for Different Replacement Policies

[5] A. Jaleel, "Web copy: Memory characterization of workloads using instrumentation-driven simulation, http://www.glue.umd.edu/ ajaleel/workload/," 2007.

[6] B. R. Gaeke, P. Husbands, X. S. Li, L. Oliker, K. A. Yelick, and R. Biswas, "Memory-intensive benchmarks: Iram vs. cache-based machines," in *Int'l Parallel and Distributed Processing Symposium*, Apr 2002.

[7] M. Qureshi, A. Jaleel, Y. Patt, S. S. Jr., and J. Emer, "Adaptive insertion policies for high performance caching," in *Proc. 34th Int'l Symposium on Computer Architecture*, 2007.

[8] R. Subramanian, Y. Smaragdakis, and G. Loh, "Adaptive caches: Effective shaping of cache behavior to workloads," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*, pp. 385–396, 2006.

[9] W. Wong and J.-L. Baer, "Modified lru policies for improving second level cache behavior," in *Sixth International Symposium on High-Performance Computer Architecture (HPCA-6)*, 2000.

[10] T. Puzak, A. Hartstein, P. Emma, and V. Srinivasan, "Measuring the cost of a cache miss," in *Workshop on Modeling, Benchmarking and Simulation (MoBS)*, 2006.

[11] M. Qureshi, D. Lynch, O. Mutlu, and Y. Patt, "A case for mlp-aware cache replacement," in *Proc. 33rd Int'l Symposium on Computer Architecture*, 2006.

[12] F. Guo and Y. Solihin, "An analytical model for cache replacement policy performance," in *SIGMETRICS '06/Performance '06: Proceedings of the joint international conference on Measurement and modeling of computer systems*, pp. 228–239, 2006.

[13] J. Reineke, D. Grund, C. Berg, and R. Wilhelm, "Predictability of cache replacement policies," Reports of SFB/TR 14 AVACS 9, SFB/TR 14 AVACS, September 2006. ISSN: 1860-9821.