Managing Off-Chip Bandwidth: A Case for Bandwidth-Friendly Replacement Policy

Bushra Ahsan
Electrical Engineering Department
City University of New York
bahsan@gc.cuny.edu

Mohamed Zahran
Electrical Engineering Department
City University of New York
mzahran@ieee.org

ABSTRACT

With the expected increase in the number of on-chip cores the demand for off-chip bus, memory ports, and chip pins increases. This makes off-chip bandwidth a very scarce resource and can severely hurt performance. Off-chip bandwidth is mainly generated by the on-chip cache hierarchy (cache misses and cache writebacks), which depends on the replacement policy. There is a huge body of research on enhancing the cache replacement policy to reduce the number of misses of a cache. Bandwidth requirement has always been of secondary importance. In the multicore and manycore era this is no longer the case. Replacement policy must take into account both the cache performance and the traffic generated by the cache, which is the topic of this paper.

In this paper we study several dynamic replacement policies with the aim to reduce the traffic generated from last level cache to main memory while not increasing the number of misses. We show that we can do better than a traditional LRU policy with very little hardware overhead.

1. INTRODUCTION

One of the major constraints to multicore and manycore architecture performance is the tremendous increase of bandwidth requirements, especially off-chip bandwidth. Software applications are becoming more sophisticated with large memory footprint. This means there will be an increase in cache memory misses and more accesses to off-chip memory. This in turn will put a lot of pressure on memory ports, memory bus, socket-pins, and so on, and can severely affect overall performance. The fundamental question is how to deal with the tremendous increase in off-chip bandwidth requirements for multicore and manycore chips in a way that sustains high performance.

Most of the work in academia/industry focuses on increasing the number of cores, the interconnection network, reducing power consumption, or memory system design. However, off-chip bandwidth has always been thought as a technological problem not an architectural problem, unlike on-chip bandwidth requirement.

Benchmark	Ratio
Barnes	3533239/9943689= 0.355
Cholesky	6240219/7561266 = 0.825
fft	58261/96910 = 0.6011
fmm	450362/516111 = 0.872
Radiosity	1653286/1808682 = 0.9140
Radix	205613/231387 = 0.8886
Raytrace	203368/4899164 =0.415

Table 1: Ratio of Dirty LRU to Total L2 Cache Access

Any cache miss is a potential for a lot of traffic, not only for bringing the missed block but also writing back victimized dirty blocks. Table 1 shows the percentage from the number of cache accesses when the LRU block of the accessed set is found dirty for some SPLASH-2 benchmark suite on a multicore chip of four cores. From the table we can see that on average whenever the L2 cache is accessed, 69.5% of the time the LRU block of the accessed set is dirty. This means a high chance of generating a writeback, which leads to off-chip traffic.

The contribution of this paper is threefold. First, it shows the importance of off-chip traffic as a main design parameter that must be tackled as early in the design process as possible. Second, it proposes several dynamic techniques for reducing off-chip traffic with little or no performance loss. Third, it evaluates the proposed techniques using the SPLASH-2 multithreaded benchmark suite and shows that the traditional LRU may not be the best way to go for such programs which are expected to be the mainstream in the multicore era.

2. A QUICK LITERATURE SURVEY

Off-chip bandwidth is a bottleneck of performance and can be a limiting factor for the number of the on-chip cores. To mitigate this bottleneck, computer architecture researchers have taken four different paths. The first is to enhance the performance of on-chip cache hierarchy. This leads to a decrease in the number of cache misses and hence a reduction in off-chip traffic. The second path is to use compiler optimizations to make software application more bandwidth

friendly [1, 2, 3]. The third path taken by researchers is to use compression for the data sent off-chip. The last path is to hide the latency resulting from off-chip bottleneck through multithreading [4, 5, 6, 7].

Cache replacement policies greatly 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 and writebacks, which is usually directly related to performance. Over the decades, there has been a large body of work on basic replacement policies.

The most commonly used replacement policy is the Least Recently Used (LRU) replacement policy. The LRU mechanism uses the application's memory access patterns to estimate cache line that has been least recently used and that should be replaced by the cache controller. Although the LRU replacement is relatively efficient, it requires a number of bits to track when each block is accessed, and relatively a complex logic. Another problem with the LRU is that each time the cache hit or miss occurs the block comparison and LRU stack shift operations require time and power. Also the LRU does not exploit frequency of the block usage and does not work well with working sets larger than the available cache size.

To reduce the cost and complexity of the LRU replacement policy, other simpler policies like random policy, can be used, but at the expense of performance. Random replacement policy chooses its victim randomly from all the cache lines in the set. Round Robin (or FIFO) replacement heuristic simply replaces the cache lines in a sequential order, replacing the oldest block in the set. Each cache memory set is accompanied with a circular counter which points to the next cache block to be replaced; the counter is updated on every cache miss.

The LRU was further improved upon by proposing schemes such as LRFU (Combining LRU and LFU), LRU-k (replacement decision based on the time of the Kth-to-last reference)[8], 2Q (use two queues to quickly remove cold blocks), LIRS (Low Inter-reference Recency Set)[9], [10], CRFP [11] (a self-tuning replacement policy that can switch between different cache replacement policies adaptively and dynamically in response to the access pattern changes) etc. These techniques require to be tuned to the optimal replacement and the tunable parameters depend on the workload and cache size. Dynamic insertion policy (DIP) [12] is a replacement policy that depends on the fact that many blocks are brought into the cache and used only once or very few times, and therefore do not deserve to come to the MRU position. DIP tends to place an incoming block into the LRU position until it gets accessed a second time. The authors reported a better performance than LRU, but no study has been done on its effect on bandwidth. There is still a large gap between the LRU and the optimal replacement policy (OPT, a replacement algorithm selects the block in a set that will be accessed farthest away into the future). This requires further research on replacement policies in order to bridge this gap.

Compressing off-chip traffic is a path taken by researchers to make the best usage of the available bandwidth [13]. This method reduces the amount of data sent on the bus. However it suffers from two main drawbacks. The first is the extra hardware required for the compression and decompression. The second is the extra penalty involved in these operations.

Victim cache [14] has been introduced as a way to decrease conflict misses. It can reduce bandwidth requirement if the block to be evicted is dirty. It differs from the scheme presented in this paper. First, victim cache needs more hardware, a small fully associative cache, and more sophisticated mechanisms to control its interaction with the cache. Second, it has more effect on on-chip bandwidth. For off-chip bandwidth it must be associated with last-level-cache which is usually of larger size and can causes contention at the victim cache. The scheme we present in this paper can be used together with victim cache in any system.

All the above techniques have reached varied degrees of success. However none of them tried to rethink the validity of LRU replacement in multithreaded applications and use it to reduce off-chip bandwidth, which is what we are trying to do in this paper.

3. PROPOSED TECHNIQUES

Moore's law allows us to pack many cores on-chip. These cores need to be fed with data and instructions. In order to do so, they need to access off-chip memory. This includes accessing chip pins, accessing the bus, and finally accessing the memory. In each one of these steps there are *walls*.

In this section we introduce several dynamic schemes where the victim block chose to be replaced is not necessarily the LRU. The victim is chosen based on several criteria that balances bandwidth and performance. The main idea of the proposed schemes is that LRU is not always the best replacement policy. And therefore, we do not always need to victimize the LRU block, especially when it is dirty.

3.1 Is LRU The Way to Go?

LRU replacement policy always victimizes the LRU block. This is acceptable if the application at hand is LRU friendly. That is, the LRU block has high chance of not being used in the near future. This may be true for a single application running on a single core. And even in that case, some applications are not LRU friendly, which is why there is always a flow of papers on how to enhance replacement policies.

When we talk about multithreaded applications, the problem of non-LRU friendly is more obvious. There is interference in the shared cache from the different threads. These threads have their own *important* blocks, and hence their own LRU stack. When they all share a cache (for example a shared L2 in a four core design like the one we use for this paper), the LRU stack of that cache is not really very useful, as the results of this paper will show. Therefore we do need to victimize the LRU block. As we have seen from Table 1 this may lead to a lot of unnecessary traffic.

Table 2 is an evidence that violating LRU strategy does not always lead to severe performance loss. The Table shows the total number of cycles of the whole execution of SPLASH-2 benchmarks for several non-LRU schemes normalized to the

LRU scheme. LRU-n means we always victimize the M^{th} element from the LRU stack. For an 8-way set associative, which is the one we use in this paper, LRU-7 is the MRU block. As we see from the table, even when we always victimize the MRU block the performance loss can be as little as 6%.

Our next *motivational* experiment is to try to victimize a non-dirty block irrelevant of performance. Table 3 shows a comparison between LRU and another scheme that chooses a non-dirty block starting from the LRU position. So it looks at all the LRU stack and chooses the closest non-dirty block to LRU. The non-dirty scheme is better than LRU in most of the benchmarks. However, some programs like `raytrace` and `Barnes` suffers some performance loss. Some other programs, `radix` and `radiosity` have unchanged performance but a decrease in number of writebacks. This means that a purely static way in determining the victim is not the best way to go. A dynamic scheme is needed here.

3.2 Proposed Dynamic Scheme

Our dynamic techniques are based on the non-dirty LRU presented above. But instead of looking for a victim at all LRU stack we search in the bottom M blocks. When M is high, we have higher chance of reducing traffic but higher chance of affecting performance. The proposed techniques change M dynamically in order to reduce traffic while not affecting performance. The new value of M will be used the next time a victim is to be picked. This technique is associated with shared L2 cache only, and by cache miss and writebacks we mean in L2 cache only. All the techniques presented in this paper are targeting only the L2 cache shared by four cores, but it can be applied to any last level cache.

We have four different dynamic schemes.
WriteBack-Based Global: M is incremented when there is a cache miss with writeback, and is decremented when there is a miss without writeback.
WriteBack-Based Local: Same as above but there is an M for each cache set.
LRU-Based Global: M is incremented when the LRU block becomes dirty, and is decremented when there is a cache miss.
LRU-Based Local: Same as LRU-based global but with M for each set.

4. EXPERIMENTAL SETUP

We have modified SESC simulator [15] to implement the proposed techniques. SESC is a cycle-accurate simulator that implements a multipcorcessor and multicore architecture. Table 4 shows the cache hierarchy parameters. These parameters are similar to many state-of-the-art processors. Inclusion property is not enforced which is similar to most of the current processors. We use seven programs from the SPLASH-2 benchmark suite [16]. The reason we have chosen a multithreaded benchmark suite over a multiprogramming environment is to put our techniques in test when there are coherence and communication overheads. We have chosen SPLASH-2 because it has been used for over a decade now and its behavior has been studied deeply so we can understand how our techniques interact with these programs. Currently we have not yet been able to cross-compile PARSEC suite [17], which is a more recent benchmark suite from Princeton, on SESC.

Component	Parameter
Processor Model	Out of Order
L1 I-Cache Size/Associativity	32KB/2-way
L1 D-Cache Size/Associativity	32KB/2-way
L1 D-Cache Write Policy	Write Back
L1 Cache Block Size	64B
L1 I-Cache Replacement Policy	LRU
L1 D-Cache Replacement Policy	LRU
L2 Cache Size/Associativity	1MB/8-way
L2 Cache Block Size	64B
L2 D-Cache Write Policy	Write Back

Table 4: Cache Hierarchy Parameters

SPLASH-2 consists of 12 applications and kernels. We have picked up the seven programs with the highest number of writebacks, as they represent the most interesting scenarios for the studies presented in this paper. We run the seven programs till completion in all the experiments.

5. EXPERIMENTS AND DISCUSSION

Our main goal in this paper is to find a technique that reduces off-chip traffic going toward the memory. This means we want to reduce the number of writebacks while not affecting the total number of cycles needed to execute the multithreaded program.

Figures 1, 2, and 3 summarize the results. These results show that the dynamic schemes, both global (i.e. single counter for the whole cache) and local (i.e. a counter per cache) are better than the traditional LRU in all aspects: performance, L2 misses, and number of writebacks, but with different degrees of success. Policies depending on writebacks are in general performing better than policies depending on LRU. Policies depending on whether LRU becomes dirty are more conservative as they try to decrease *potential* traffic. On the other hand, policies depending on writebacks are dealing with *actual* traffic.

Figure 1 shows the total number of cycles of the whole execution of each benchmark, normalized to the traditional LRU. We notice that the dynamic scheme did not hurt the overall performance in any benchmark. `Barnes`, `fmm`, and `radix` did not benefit from the dynamic schemes. This is due to several factors. First, these three benchmarks are among the highest in number of writebacks as indicated above from Table 2. This means that M has a tendency to increase. Second, these benchmarks are LRU friendly, as indicated by Table 1. This means that the higher the M the lower the expected performance. However, the other factors of decreasing M in case of misses, offset this phenomena but results in no performance gain.

Figure 2 shows the total number of writebacks normalized to LRU. Here all the schemes are doing a good job in decreasing off-chip traffic. Although in some cases local schemes are not justifying the extra hardware over the global ones.

In order to show that our scheme did not affect the perfor-

Scheme/Bench	Barnes	Cholesky	Fft	Fmm	Radiosity	Radix	Raytrace
LRU	1	1	1	1	1	1	1
LRU-1	1	1.03	1.01	1	1.01	1	1.02
LRU-2	1	1.06	1	1	1.01	1.04	1.06
LRU-3	1	1.15	1.01	1	1.03	1.07	1.12
LRU-4	1.02	1.23	1.01	1.03	1.03	1.09	1.21
LRU-5	1.04	1.37	1.02	1.06	1.03	1.12	1.36
LRU-6	1.13	1.55	1.04	1.12	1.05	1.18	1.63
LRU-7	1.46	1.69	1.06	1.21	1.09	1.25	2.05

Table 2: Normalized Values of Number of Cycles for Victimizing a Non-LRU Block

Scheme	Benchmark	Cycles	Bus Accesses	L2 WriteBacks	L2 ReadMisses
LRU	FFT	9387058	198469	65759	132695
	Radix	16160296	17392	1184	16208
	Raytrace	235260001	683761	5856	677104
	Radiosity	378950664	976546	465226	511207
	Fmm	70273861	22543	3457	19084
	Barnes	576681542	1164635	399366	763696
	Cholesky	269020265	2584732	1141150	1439349
non-dirty policy	FFT	8246494	152307	39891	111364
	Radix	16151120	16535	707	15818
	Raytrace	240549924	733882	0	733789
	Radiosity	377554002	957059	443185	511428
	Fmm	70549168	25327	3235	21917
	Barnes	619991031	1548476	280623	1198674
	Cholesky	249562396	2109534	720558	1236621

Table 3: Comparison of Traditional LRU scheme with non-Dirty LRU Policy

mance of the cache itself, Figure 3 shows the total number of read misses. Write misses are considered read misses followed by write hits. Here also most of the techniques are performing better than LRU in all benchmarks except `fmm`.

Finally, Figure 4 shows the average traffic, measured as the number of bus accesses, over the whole execution, normalized to LRU average traffic. The importance of such measurement is that it links price (in terms of bus accesses due to read or write misses) to performance (in terms of total number of cycles) and is a good indication of bandwidth requirement. The WB sensitive techniques, local and global, have better price performance for most of the benchmarks. But for some others, like `fmm`, `radix`, and `raytrace`, the LRU sensitive techniques are better. The main conclusion is that the dynamic techniques are better than traditional LRU for these benchmarks.

6. CONCLUSIONS

In this paper we tackled the problem of off-chip bandwidth. We argued that this problem will be the next bottleneck in the multicore era. We proposed several dynamic techniques to deal with this problem, with several variations in each. The techniques exploit the fact that LRU is not always the best choice and hence we can violate the LRU policy in favor of less off-chip bandwidth. We concluded that off-chip bandwidth can be decreased with minimal effect on performance and with very little hardware for the SPLASH-2 benchmarks. Techniques based on writebacks are performing better than techniques based on dirty LRU. Moreover, global techniques are better than local ones in terms of price/performance.

Acknowledgments

This work is supported by National Science Foundation (NSF) grant CCF-0750951.

7. REFERENCES

[1] B. Calder, C. Krintz, S. John, and T. Austin, "Cache-conscious data placement," in *Proc. Int'l Conf. on Architecture Support for Programming Languages and Operating System (ASPLOS)*, 1998.

[2] S. Carr, K. McKinley, and C. W. Tseng, "Compiler optimizations for improving data locality," in *Proc. Int'l Conf. on Architecture Support for Programming Languages and Operating System (ASPLOS)*, 1994.

[3] T. Chilimbi, B. Davidson, and J. Larus, "Cache conscious structure definition," in *Proc. of Int'l Conference on Programming Language Design and Implementation (PLDI)*, 1999.

[4] D. J. Lilja, "When all else fails, guess: The use of speculative multithreading for high-performance computing," tech. rep., 2000.

[5] H. Akkary and M. A. Driscoll, "A dynamic multithreading processor," in *Proc. 31st Int'l Symposium on Microarchitecture*, 1998.

[6] J.-Y. T. et.al, "Integrating parallelizing compilation technology and processor architecture for cost-effective

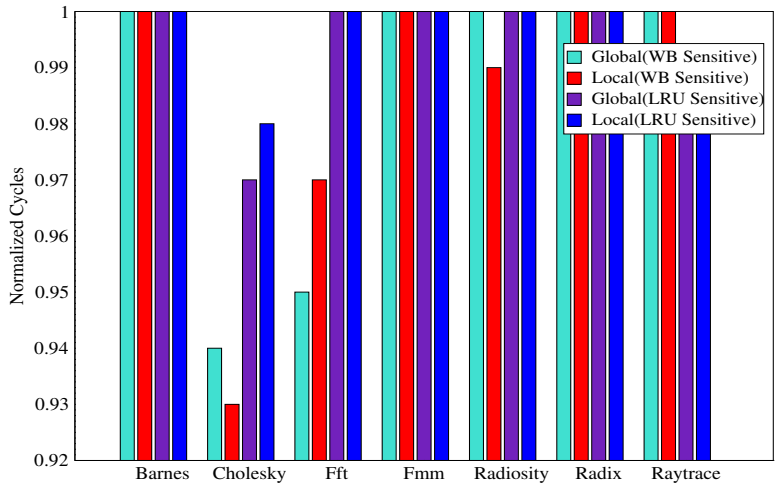

Figure 1: Performance Evaluation

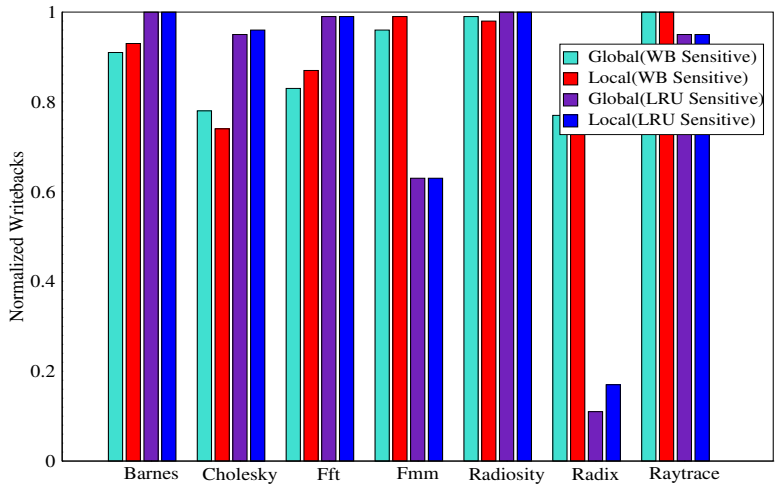

Figure 2: Comparison of WriteBacks

concurrent multithreading," *Journal of Information Science and Engineering*, vol. 14, March 1998.

[7] D. M. Tullsen, S. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proc. 22th Int'l Symposium on Computer Architecture*, 1995.

[8] E. J. O'neil, P. E. O'neil, G. Weikum, and E. Zurich, "The lru-k page replacement algorithm for database disk buffering," in *SIGMOD Rec.*, pp. 297–306, 1993.

[9] S. Jiang and X. Zhang, "Lirs: An efficient low inter-reference recency set replacement policy to improve buffer cache performance," in *In Proc. ACM SIGMETRICS Conf*, 2002.

[10] H. Al-Zoubi, A. Milenkovic, and M. Milenkovic, "Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite," in *ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference*, pp. 267–272, 2004.

[11] Z. Li, D. Liu, and H. Bi, "Crfp: A novel adaptive replacement policy combined the lru and lfu policies," in *CITWORKSHOPS '08: Proceedings of the 2008 IEEE 8th International Conference on Computer and Information Technology Workshops*, pp. 72–79, 2008.

[12] M. Qureshi, A. Jaleel, Y. Patt, S. S. Jr., and J. Emer, "Adaptive insertion policies for high performance caching," in *Proc. 34th International Symposium on Computer Architecture (ISCA)*, pp. 381–391, Jun. 2007.

[13] A. R. Alameldeen and D. A. Wood, "Adaptive cache compression for high-performance processors," in *31st Int'l Symposium on Computer Architecture (ISCA)*, June 2004.

[14] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffer," in *Proc. 17th International Symposium on Computer*

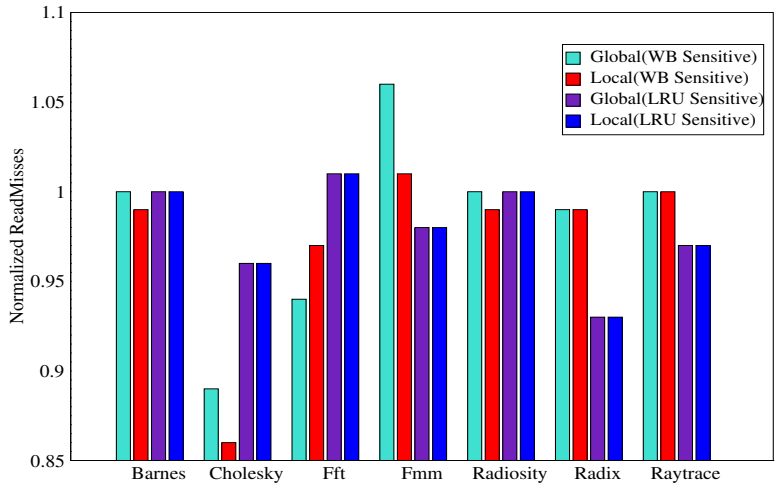

Figure 3: Comparison of Read Misses

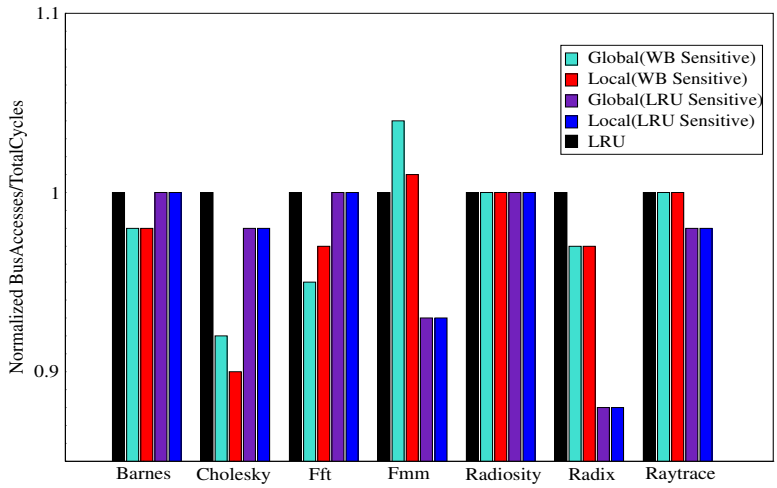

Figure 4: Normalized Bus Access per Cycle

Architecture, pp. 364–373, May 1990.

[15] J. Renau, "SESC." `http://sesc.sourceforge.net/index.html`, 2002.

[16] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *Proc. 22nd IEEE/ACM International Symposium on Computer Architecture*, pp. 24–36, June 1995.

[17] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.