A Hybrid Compiler-Architecture Technique to Manage Off-Chip Traffic for Multicore Chips

Bushra Ahsan
Supervisor: Mohamed Zahran
Department of Electrical Engineering
The City University of New York
New York, NY 10031
{bahsan, mzahran}@acm.org

ABSTRACT

Off bandwidth requirement is an important issue for modern computer systems. With the increase in number of on-chip cores, the traffic generated by these cores is also increasing. This increasing traffic is putting more pressure on off-chip resources, such as chip pins, busses, and memory ports, resulting in processor stalls and delays which can lead to performance loss. More cores are now competing for the off-chip bandwidth and their performance is limited by its availability. *This* ***bandwidth wall*** *is becoming critical for today's manycore systems and memory intensive applications, and will exacerbate in the future as system designs continue towards more and more cores on-chip, and as software's memory footprint increases. Most of the previous work done to mitigate this problem revolves around increasing on-chip memory to reduce off-chip accesses, or around optimizations of on-chip memory hierarchy to reduce traffic coming from memory towards the chip. In our work, we target to reduce the traffic going from the chip towards the memory. We propose a hybrid "Compiler-Architecture" technique that employs both compile-time and run-time information to dynamically manage the traffic generated during execution. Since the off-chip traffic is generated by the on-chip cache-hierarchy, specially the Last Level On-chip Cache (LLC), we make all our modifications to the LLC. Our simulations show a reduction of off-chip traffic with very little impact on performance.*

1. PROBLEM DEFINITION AND MOTIVATION

There is a large gap between processor speed and memory speed [1]. Although processor speed has continued to increase over the last few generations, the memory access speed has not. The memory is still hundreds of times slower than the processor, which has led to the processor-memory bottleneck. Every access to the memory is very costly to the processor in terms of latency. No matter how fast the modern processors become, their performance is limited by the memory, that is, off-chip memory access. Recent research is aiming towards single-chip supercomputers. Intel and Tilera have already launched test chips of 48 and 64 cores respectively.

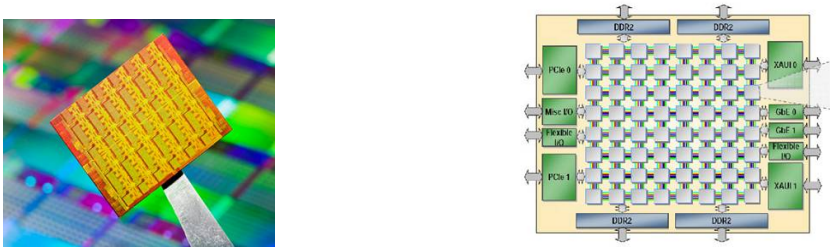

Figure 1: the current 48 core design from Intel and 64 core design from Tilera[1]

With the current advent of multicore design, the many cores on chip compete for the available bandwidth to access the memory. The off-chip bandwidth depends on the available resources like pins and buses and is thus limited. Although modern designs have alternatives such as sockets, HyperTransport and on-chip DRAMs to increase the bandwidth resources, ultimately the bandwidth is still limited and competing for it can result in delays leading to performance loss. Thus, proper management of this bandwidth amongst the on-chip cores is required to eliminate any possible stalls. A limited amount of traffic can be handled by the off-chip resources at any time. Figure 2 shows an example of available bandwidth. Any traffic above this threshold is bad news for the processor as it will result in stalls and delay the execution.

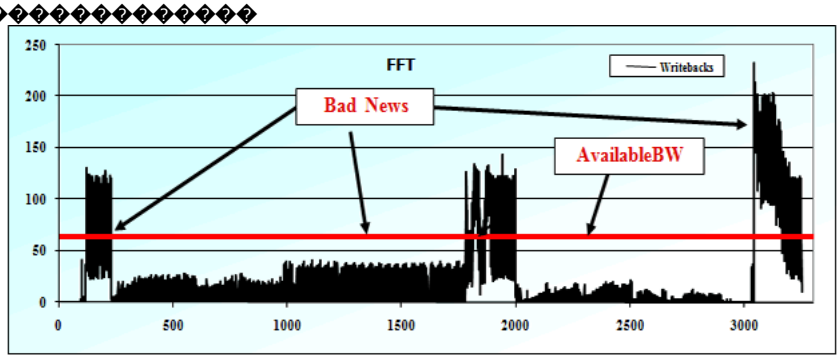

Figure 2: Traffic during entire execution counted at regular intervals. Any traffic above the available bandwidth level cannot be handled

As the number of cores on chip increases, the traffic generated from these cores is also increasing. The traffic is divided into traffic from chip towards memory and traffic from memory towards the chip as shown in Figure 3. A lot of work has been done to reduce the traffic from memory towards the chip. We concentrate on reducing the traffic going from chip towards the memory. This traffic is generated from the Last Level On-Chip Cache (LLC) whenever a block is evicted and is written to the memory. When and which block to choose as victim depends on the ��Replacement Policy�� of the LLC. The most commonly used Replacement Policy is the ��Least Recently Used�� (LRU)[2] which, as the name implies, evicts the least recently used block to make way for new blocks coming into the cache. Since the Replacement Policy of the LLC directly affects the traffic generated, we target to modify it in order to make it more bandwidth friendly. Modifications to the LRU will trade-off some cache performance but we will show that this effect is minimal in multicore design.

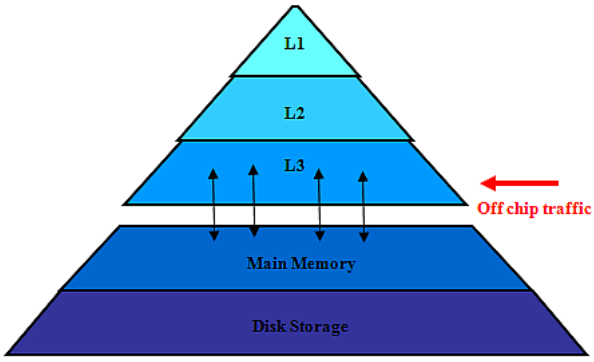

Figure 3: Memory Hierarchy of a Modern Processor. Traffic between Last Level On-chip Cache and the memory is the off-chip traffic. We target to reduce off-chip traffic from LLC to memory

In order to put our contribution in context, we present previous work done in Section 2. In Section 3 we show why modifications to the current Replacement Policy are possible without any negative effect on the performance. In Section 4 we describe our proposed scheme followed by the results in Section 5. Finally we conclude in Section 6.

---

2. BACKGROUND AND RELATED WORK

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. This includes managing available bandwidth by fair queuing [2, 3]. Also improving cache performance through adaptive caches have been proposed [4, 5, 6].

- The second path is to use compiler optimization to make software application more bandwidth friendly. Many studies [7, 8, 9] have focused on compiler analysis and optimization to improve cache performance and thus reduce traffic. All these proposed techniques try to reduce cache misses by improving data locality. The compiler does so by either placing the data efficiently in memory or change the memory access order to improve the temporal and spatial locality.

- The third path taken by researchers is to use compression for the data sent off-chip [10]. 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.

- The last path is to hide the latency resulting from off-chip bottleneck through multithreading. Multithreading is a method to hide latency [11, 12, 13, 14]. When a thread is stuck waiting for data to arrive from off-chip, another thread starts using the pipeline resources to make progress. Multithreading can increase throughput. However, it falls short when the different threads need to access the memory simultaneously or there is any kind of dependency between the threads.

Apart from this, work has been done to include hardware support to increase the available bandwidth. This includes networks on chip [15, 16] that have replaced the buses and have increased the amount of available bandwidth. Furthermore, unification of the processor and the DRAM into a single chip, the IRAM has been proposed [17]. In such case, the memory will be able to operate at processor speed, increasing the bandwidth 100-fold.

Although work has been done to mitigate the problem, bandwidth management continues to be a bottleneck and memory wall has yet to be scaled.

---

3. TOWARDS A NEW SOLUTION

To formulate a solution for proper bandwidth management, we target to make the Replacement Policy of LLC more **bandwidth aware**. This will trade off some cache performance. Can we afford to lose cache performance for better bandwidth management? Before we answer this question we show the results of three experiments that serve as a motivation behind our work. All experiments are done with a 1MB, 8-way set associative last level cache.

3.1 LRU for shared caches

LRU has been designed for single core architectures and although it works very well for caches of single core, its performance for shared caches amongst multicores would be different. To investigate this we perform an experiment in which we count the number of hits for each block in a set of a cache in the shared LLC. SPLASH-2 [18] benchmarks are used to show the pattern of hits in the cache. Figure 4 shows the results for one of the SPLASH-2 benchmark called Fft. The results show that LRU works well for one core since most of the hits are located at the top of the stack that is at the Most Recently Used (MRU) position. However, as we increase the number of cores, the hits get shifted from the MRU position. For 8 cores the hits become scattered in the entire cache. This is mainly due to two reasons. Firstly, most of the temporal and spatial locality of application cache blocks has been exploited at L1 cache, and hence there is lesser temporal locality at L2 cache and beyond. Secondly, the shared LLC is accessed by multiple cores. *Hence, the shared cache is not the best stack representation of any one of the cores due to interference amongst blocks from different cores.* This interference increases as the number of cores increases.

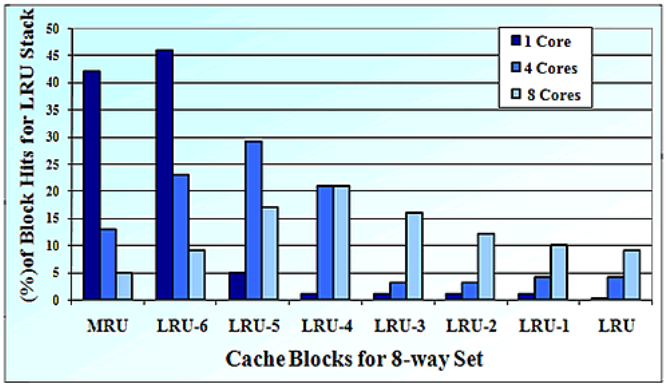

Figure 4: Comparison of Hit Rates for Each Block of an 8-Way Associative Cache for Different Number of Cores for Fft

3.2 LRU is potential for high traffic

The above experiment shows that LRU replacement policy is losing its efficiency for multicore architectures. Another experiment is conducted to see its effectiveness in terms of bandwidth. In regular LRU, the victim is always the least recently used block regardless of whether it is dirty or clean. The ratio of the number of times LRU is dirty to total cache accesses is compared. Figure 5 shows that in most SPLASH-2 benchmarks, more than 50% of the time, the LRU is dirty. Evicting a dirty block results in writeback and hence off-chip accesses. *Thus, there is a lot of potential traffic if these dirty LRU blocks are always the victim.*

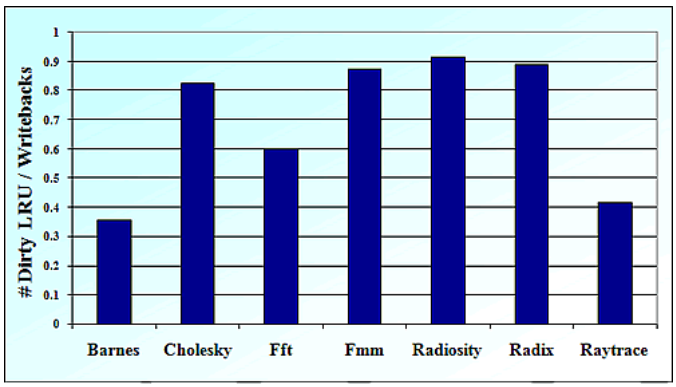

Figure 5: Ratio of Dirty LRU blocks to the TotalNumber of Accesses in LLC

So why always choose LRU to be the victim? Can another block in cache be chosen as victim? Will choosing a non-LRU affect performance?

3.3 Is LRU the way to go?

The above questions are answered by the third experiment in which the victim chosen for eviction is always a non-LRU block. We compare the performance in such a case to performance of the LRU scheme. Performance is measured in terms of execution cycles. Table 1 shows the execution cycles of always victimizing non-LRUs starting from LRU-1, the block just above the LRU, to LRU-7 which is the Most Recently Used block in an 8-way associative cache. The execution cycles are normalized to that of LRU. *From the table it is clear that the performance of evicting a non-LRU is very close to the performance of a regular LRU*. This is because of the earlier observation of interference in a shared cache. The worst performance occurs in Radiosity where execution cycles get doubled to that of LRU. However, this occurs only when we are always victimizing the MRU block. This shows that Radiosity is LRU friendly. On average, the performance of victimizing

Table 1: Normalized Values of Number of Cycles for Victimizing a Non-LRU Block

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	1.0	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

non-LRUs is close to victimizing LRUs. The above observation can be exploited to victimize non-LRUs if it is beneficial for bandwidth.

---

4. PROPOSED TECHNIQUE

We divide the explanation of our technique into two parts. Section 4.1 explains the scheme that modifies the LRU, called the Dirty-Aware LRU. Section 4.2 explains the profiler support added to the hardware scheme that leads to the hybrid scheme of bandwidth management at run time.

4.1 Dirty-Aware LRU (DA-LRU)

We start with a simple technique that works as follows. Suppose we have an N way associative cache. The blocks are placed from position 1 to N, with the most recently used block at position 1 and the least at position N. In a regular LRU, whenever a new block has to be placed in a cache set, the block at position N of the set (the LRU block) is evicted. Since we showed several reasons that always evicting LRU block might not be a good idea in case of shared caches, we present a new technique that exploits the shortcomings of LRU replacement policy for the sake of bandwidth management. In our proposed technique Dirty-Aware LRU (DA-LRU), we choose to victimize a clean block between LRU and LRU-M. This saves bandwidth by retaining the dirty blocks in the cache as long as possible. If, however, all the blocks from LRU to LRU-M are dirty, the LRU block of the set is evicted. For example if M=3, the technique would look for the first non-dirty block from three blocks starting from LRU and victimize it. Therefore, the main design parameter of this scheme is **M**.

M is the number of blocks we check in the cache set, starting with the LRU, to find a possible clean block.

This parameter designates how many blocks we examine in the set to determine the victim. M=1 is traditional LRU, since it means victimize only one block, that is the LRU. M=N means the entire cache set is checked to find a possible clean victim. So for an 8-way set-associative cache, the maximum M is 8. To find the optimum value of M for the LLC, we add compiler support to this technique as explained in the next sub-section.

4.2 Hybrid Compiler-Architecture Scheme

Initially the compiler counts the number of writebacks at regular intervals by profiling the application on one core. Based on this, a Write Frequency Vector is formed. The WFV is used to generate the values of thresholds (Min and Max). The thresholds divide the writebacks into regions of high and low traffic. During run time, the hardware keeps track of the number of writebacks in each interval. Depending on whether the writebacks are high or low, the replacement policy is chosen. Each step is explained in detail.

Profiling:
The compiler uses only one core to do profiling and to collect application behavior and generates a Write Frequency Vector.

Write Frequency Vector:
During profiling, the compiler builds a write frequency vector (WFV). WFV, as shown in Figure 6, is a histogram representation of the number of writebacks at several intervals and contains *Sets*. Each set in a WFV represents a range of writebacks. The compiler keeps count of the number of writebacks at each interval of length X. Then, depending on what range the writebacks are in that interval, the *COUNT* of the set representing that range is increased by one. As an example consider each set to represent a range of 10 writebacks. Then set A will represent range of writebacks from 0 to 9, B will represent 10 to 19 and so on. $COUNT_A$ will represent number of intervals that have writebacks within the range of 0 to 9. Suppose in an interval the number of writebacks is 56 then $COUNT_F$ is incremented because set F represents range of writebacks from 50 to 60.

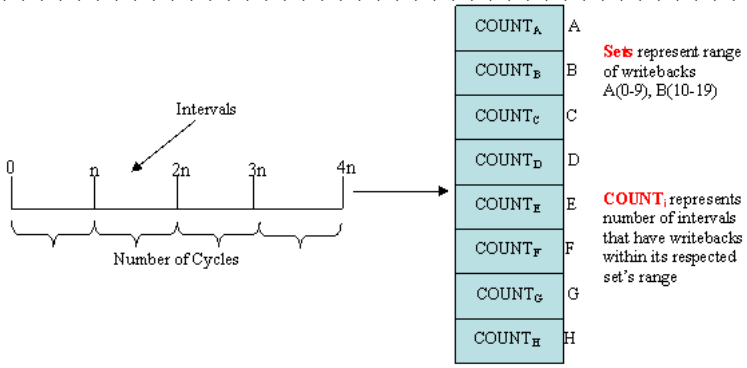

Figure 6: Generation of WFV by the profiler running the application on one core. The writebacks are counted every interval as shown on the left which helps generate the Write Frequency Vector (WFV) shown on the right

Defining Thresholds:
The WFV is used to generate thresholds. Thresholds represent two values called *Min* and *Max* that are used at run-time. These two thresholds are given to the hardware to divide the number of writebacks into regions of high and low traffic. With the WFV, we propose two algorithms; frequency-based algorithm, and weighted average algorithm.

Frequency-Based Algorithm:
Given a WFV,
Pick the two indices for the highest and lowest numbers above the noise level

- Min = highest number of the interval represented by the lowest index
- Max = lowest number of the interval represented by the highest index

For example, if the two indices were 7 for the lowest and 20 for the highest, Min will be 69 and Max will be 199.

Weighted Average Algorithm:
In this algorithm

- Min = zero
- Max = ﷺ (COUNTi * midi) / SUM

ﷺ
where

COUNTi is the content of element i of the WFV vector
midi is the mid range of element i (for example if we are talking about element 6, it spans range 50 to 59 with mid of 55)
SUM is the sum of all non-noisy entries of all WFV.
Max can be thought of as the weighted average of the indices based on their entries.

Regions:
Every X cycles, the hardware keeps a count of the number of writebacks from LLC in that interval. The values of Min and Max divide the off-chip traffic pattern into regions of high and low traffic as shown in Figure 7. For Frequency-Based Algorithm we have three regions: below Min, between Min and Max and above Max. For Weighted Average Algorithm we have only two regions: below Max and above Max since Min is always zero.

- Region 1: less than Min (low traffic)
- Region 2: between Min and Max (medium traffic)
- Region 3: greater than Max (high traffic)

Replacement Policy Selection for Each Region:
If the number of writebacks in any interval is below Min, LLC will use traditional LRU for victim selection, because it means that we have low numbers of replacements of dirty blocks and we do not want to risk losing performance. If the number of writebacks is between Min and Max, we use DA-LRU with M=associativity/2. Finally, a number above Max means we have a high number of writebacks, so we use DA-LRU with M=associativity. This is shown in Figure 7.
ﷺ

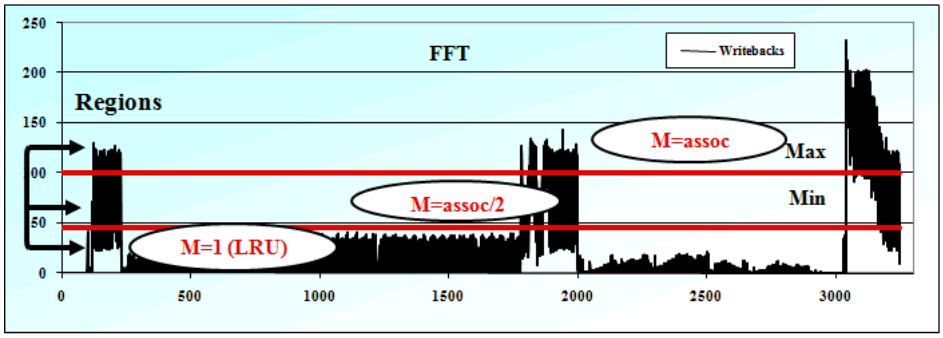

Figure 7: Selecting replacement policy according to regions of high and low traffic ﷺ

---

5. RESULTS

5.1 Experimental Setup

We modified SESC simulator [19] to implement our proposed schemes. SESC is a cycle-accurate microprocessor architectural simulator that models different processor architectures, such as single processors, chip multi-processors and processors-in-memory. Table 2 shows the main parameters for the system used. These parameters are similar to many state-of-the-art processors. We run the seven of the SPLASH-2 benchmarks to test our scheme. We have chosen the benchmarks with the highest number of writebacks. However, we use a relatively small LLC size to have more pressure on the last-level cache due to the small working set size of our benchmarks.

ﷺTable 2: Baseline computer system configuration

Cores	4
L1 Cache	32 KB, 64 B block size, 1 cycle access latency; LRU replacement, 2-way
L2 Cache	32 KB, 64 B block size, 1 cycle access latency; LRU replacement, 2-way
Memory	500 cycles access latency
Branch	Hybrid
CPU	Out of order, issue width of 4

5.2 Simulation Results

We compare each technique by looking at the following parameters.
Number of Write Backs: This is the traffic from LLC to the memory, and is our measure of success.
Number of Read Misses: This is the traffic from the memory to LLC. A write miss is considered a read miss followed by a write hit.
Number of Cycles: The total number of cycles taken by the program till completion.

We are trying to reduce the number of writebacks (our main measure of success), while not increasing the number of read misses (negative side effect), and with minimal impact to the total number of cycles (measure of performance).

Writebacks: Figure 8 shows the normalized number of writebacks. We compare Hybrid Technique with eager-writeback (a compromise between write-through and writeback), DIP (Dynamic Insertion Policy), the two proposed techniques using compiler support (frequency-based and weighted-average) and DA-LRU with M=4 (value of M kept constant at 4 during entire execution).

The best three schemes for 4 cores are the weighted-average, M=4, and DIP. However, the weighted-average becomes better as we increase the number of cores to 8 (while keeping LLC to 1MB), which is a sign of good scalability. The frequency based method is doing well too, but the LRU part of it (when number of writebacks is below Min) holds it a step behind the weighted-average.

Read Misses: Similar behavior can be said on the number of misses shown in Figure 9. As the number of cores increases weighted-average hybrid technique method and static DA-LRU with M=4 are doing much better.

Performance: The proposed techniques are not hurting performance as indicated by Figure 10. Raytrace is the only benchmarks that suffered some performance loss. This is because the interference at LLC among the threads is constructive (blocks shared by more than one core accounts for 99% of the blocks at LLC) which makes the program very sensitive to block replacement, as was indicated earlier in Table 1.

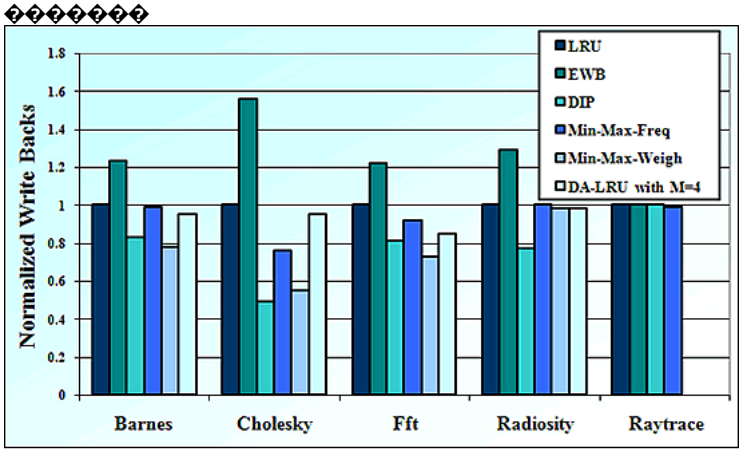

Figure 8: Normalized Number of Writeback

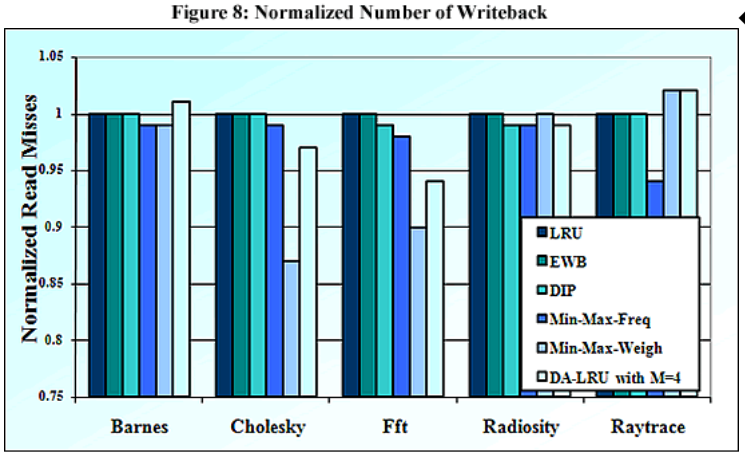

Figure 9: Normalized Number of Read Misses

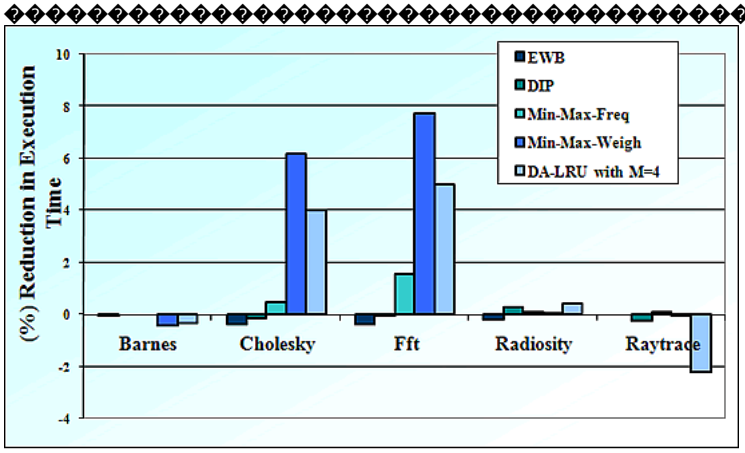

Figure 10: Speedup

---

6. CONCLUSIONS

In this project we made the following contributions:

1) We argue that off-chip traffic management will be the next bottleneck in the multicore era and we must derive methods to deal with it. As number of cores on chip increases, the bandwidth problem is likely to become more severe.

2) We propose a hybrid technique to deal with this problem.

3) The technique exploit the fact that the current replacement policy i-e, LRU is not always the best for shared caches and hence we can violate the LRU policy in favor of less off-chip bandwidth.

4) The hardware part of the technique is called Dirty-Aware LRU that alters the currently used replacement policy in a way to reduce off-chip traffic by victimizing clean blocks instead of dirty.

5) Reducing off-chip traffic is highly application dependent and hence the technique used for LLC should be alterable according to the traffic produced at run time. To achieve this we present a hybrid technique that manages bandwidth by employing both hardware and software to adapt to the program behavior dynamically.

6) Based on the traffic pattern and information collected through profiling, the best replacement policy is chosen for the Last Level Cache.

7) We can trade some cache performance with decrease in off-chip bandwidth, which can lead to an overall system performance enhancement due to decrease in off-chip contention on buses and memory.

8) Compiler and hardware can work together to assist in bandwidth management for multicores by choosing the best replacement policy for LLC based on the traffic generated at run time.

---

7. FUTURE WORK

Our future plans include the following:

- Testing the versatility of our techniques with the newer PARSEC benchmark suite for the multithreaded workload.
- Testing all techniques in a multiprogramming environment.
- Propose techniques to manage on-chip bandwidth among caches.

---

8. REFERENCES

[1] Mahapatra, N. R. and Venkatrao, B.: The processor memory bottleneck: problems and solutions. Proc. Crossroads 5, 3, 1999

[2] Kyle J. Nesbit, Nidhi Aggarwal, James Laudon, and James E. Smith. Fair queuing memory systems. In MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, pages 208-222, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2732-9.

[3] Thomas Moscibroda and Onur Mutlu. Memory performance attacks: denial of memory service in multi-core systems. In SS'07: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium, pages 1-18, Berkeley, CA, USA, 2007. USENIX Association. ISBN 111-333-5555-77-9.

[4] 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.

[5] A. S. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In Proc. 17th International Symposium on Computer Architecture, 2002.

[6] J-H Lee and S-D Kim. Application-adaptive intelligent cache memory system. ACM Transactions on Embedded Computing Systems, 1(1), 2002.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] D. J. Lilja. When all else fails, guess: The use of speculative multithreading for high-performance computing. Technical report, 2000.

[12] H. Akkary and M. A. Driscoll. A dynamic multithreading processor. In Proc. 31st Int'l Symposium on Microarchitecture, 1998.

[13] J-Y. Tsai et.al. Integrating parallelizing compilation technology and processor architecture for cost-effective concurrent multithreading. Journal of Information Science and Engineering, 14, March 1998.

[14] 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.

[15] W. Dally and B. Towles. Route packets, not wires: On-chip interconnection networks. In Proc. of 38th Conf. on Design Automation, 2001.

[16] L. Benini and G. DeMicheli. Networks on chips: A new soc paradigm. In IEEE Computer, January 2002.

[17] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A case for intelligent ram. IEEE Micro, 17(2):34{44, slash 1997.

[18] 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, pages 24-36, June 1995.

[19] P. M. Ortego and P. Sack. SESC: SuperESCalar Simulator. Dec 2004.

---

[1] Sources used: http://www.intel.com/ and http://www.tilera.com/

[2] In current Systems Psuedo LRU is used instead of regular LRU because of its hardware simplicity. However, the performance of LRU is better than Pseudo LRU so we compare our schemes with LRU.