Off-Chip Bandwidth for Multicore Processors: Managing The Next Big Wall

by

Bushra Ahsan

A dissertation submitted to the Graduate Faculty in Electrical Engineering in partial fulfillment of the requirenments for the degree of Doctor of Philosophy,
The City University of New York
2010

This manuscript has been read and accepted for the Graduate Faculty in Engineering in satisfaction of the dissertation requirenment for the degree of Doctor of Philosophy

Professor Mohamed Zahran

Co-chair of Examining Committee

Professor Tarek N. Saadawi

Co-chair of Examining Committee

Professor Mumtaz Kassir

Executive Officer

(Date)

Supervisory Committee:
Professor Thao Nguyen
Professor YingLi Tian
Professor Ramesh Karri

THE CITY UNIVERSITY OF NEW YORK

"If I have seen further it is only by standing on the shoulders of giants"

Isaac Newton

Abstract

Off-chip Bandwidth for Multicore Processors: Managing The Next Big Wall

by Bushra Ahsan

Adviser: Professor Mohamed Zahran

As we approach billion transistors on chip, the number of on-chip cores is skyrocketing. With the number of on-chip cores increasing, the traffic generated from these cores is also increasing. Recent studies have shown that this surge of traffic in multicores is bad news for supercomputing design. This is due to off-chip contention amongst applications running on multiple cores.

Traffic in a multicore system is divided into on-chip traffic (traffic amongst cores) and off-chip traffic (traffic from chip to memory). The off-chip traffic is mainly generated by on-chip cache hierarchy and is divided into traffic towards memory, due to writebacks, and from memory, due to read misses. There is a huge body of research on managing cache hierarchies, improving their performance and hence reducing the number of cache misses. Bandwidth requirement has always been of secondary importance. In the multicore and many-core era, this is no longer the case. The cache hierarchy designer must take into account both cache performance and traffic generated by the cache in order not to put pressure on the available bandwidth. If off-chip bandwidth is not managed, a 16 core machine will not give much performance benefit over a dual core machine.

In most processor architectures, the cache hierarchy consists of several private caches per core, followed by a shared Last-Level Cache (LLC). This LLC is the last wall before hitting off-chip and is the cause of off-chip bandwidth traffic i.e the writebacks. LLC, therefore, is a highly important factor in off-chip traffic generation. We manage the LLC in order to attain overall off-chip bandwidth management in a multicore system. In this thesis various methods to improve bandwidth by reducing traffic towards memory are proposed. We present hardware and hybrid techniques of varying complexities that work in rhyme to manage bandwidth for multicores . All techniques proposed to save bandwidth require very little overhead and reduce off-chip traffic considerably while not effecting overall performance. By bandwidth management we come closer to the ultimate goal of supercomputer on chip.

For my Daddy...

Acknowledgements

All thanks to Allah Almighty who has always been kind to me. Without his help none of this would have been possible.

First and foremost, I would like to express my heartfelt gratitude to my advisor, Dr. Mohamed M. Zahran, without whom this thesis would have only been a dream. I am indebted for his constant directions, encouragement, guidance and the tremendous support and opportunities he provided throughout my doctoral studies. He spent countless hours proofreading my research papers, discussing my research, evaluating innovative topics for me to explore, and providing me with countless ways to improve upon my dissertation. I immensely enjoyed and learned from every meeting I had with him, where, besides research, we discussed a variety of things in life. He totally transformed my way of thinking. He was a complete advisor who led me not only in the academic area, but also in life. He was extremely patient and lifted my hopes when I was going through the toughest time and was at my worst. He made this day possible and I feel myself extremely lucky to have him as an advisor.

With Dr. Zahran as my advisor, I got not only a mentor but also a life time friend. He made the time spent as a graduate student both fulfilling and fun. I can never thank Dr. Zahran enough.

Secondly, today where I am is due to my father. He has taught me everything I know. By dedicating this thesis to him, I have a gotten a chance to thank him for his love and guidance, something sadly I could not do in his lifetime. Here is hoping I made him proud.

I would like to thank my mother who's kindness cannot be matched. She has waited patiently for me to finish and I am sure she will be really happy when she finally reads this. Thanks to my sisters, Anita, Shazia, Sonia and Lyla who have always loved me and stood by me. Specially Sonia, without whom, I would never have gotten a chance to do my PhD.

I would also like to thank Prof. Nguyen, Prof. Saadawi, Prof. Tian and Prof. Karri for being in my committee. Prof. Saadawi was one of the first people who

helped me when I was very new in the school and felt completely lost. It has been a pleasure to have Prof. Nguyen as a neighbor to my lab and I really appreciated his checking up on me and my work. His constant cheerfulness always lifted my spirits. I would also like to thank Prof. Moshary for believing in me as a graduate student.

One of the best friends I made during my time as a graduate student and without whom this day would not have been possible is Jerry Backer. He constantly endured my whining with patience, helped me when I needed help the most, lifted my spirits in times of depression, elevated my hopes in times of hopelessnes and kept my feet to the ground. Thanks to him for all the endless discussions on research, for making me believe I could do this and for his friendship.

Other people who I absolutely must mention are Samrat, Ravi and Natia. They always guided me in the right direction and wacthed out for me. Without them, everything they made easy for me would have been twice as distressing.

I have made countless friends from outside of the lab. I would like to thank all of them for making my grad life fun. Specially, Rizwan and Asif for enduring me when I was at the peak of whining. They motivated me throughout and have witnessed the entire transformation of a lost and shy graduate student to a very confident Doctor.

Thank you Abhinav for bringing color to the lab. You made the long hours spent in the lab seem like fun.

Lastly, I would like to thank my family and all the life long friends I have. They are the ones who have believed in me even when I have not.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"Well begun is half done."

Aristotle

The multicore and manycore era is creating many challenges for researchers. These challenges include reliability, power, temperature, speed etc. One of the biggest challenge that is becoming pivotal as the number of cores increase is off-chip Bandwidth.

1.1 Why Manage Bandwidth?

The scaling of number of transistor on core was predicted in the famous Moore's law [1] which stated that the number of transistors will double every 18-24 months. This has almost been true to date and has led to the multicore architectures becoming mainstream. Most processor vendors have multicore products, e.g. Intel

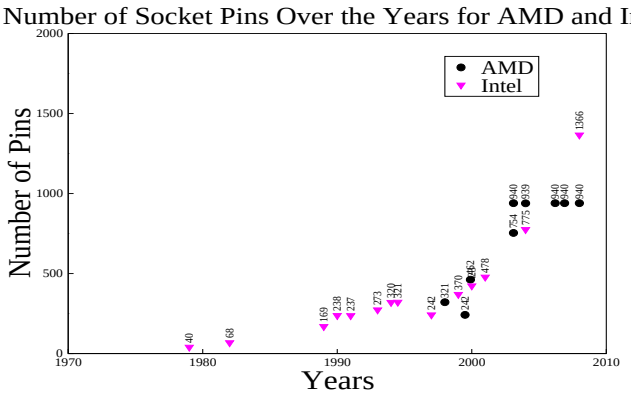

FIGURE 1.1: Scalability of Pins Over the Years for AMD and Intel

Quad and Dual Core Xeon [2], AMD Quad [3] and Dual Core Opteron, Sun Microsystems UltraSPARC T1 and T2 (8cores) [4], IBM Cell [5], etc with transistor technology scaling from 45nm to 32nm. The number of on-chip cores is expected to increase over the next few years[6]. There are various alternatives in designing cache hierarchy organization and memory access model for multicores. With the skyrocketing number of cores per chip making manycore architectures feasible within a few years, new challenges face computer architects. One of the most pivotal challenges to multicore and manycore architectures 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. The ports have not been as scalable as cores over the years as seen from Figure 1.1 and hence traffic pressure on them can severely affect overall performance.

The challenge 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.

1.2 Thesis Contributions

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. In this thesis, methods are proposed to manage the off-chip bandwidth while not hurting performance. The contributions of this thesis are threefold.

- First, several interesting observations of the Least Recently Used (LRU) replacement policy, it's weakness in the multicore environment and how these weaknesses can be exploited for bandwidth management are shown.
- Second, various techniques are presented for bandwidth management and to reduce off-chip traffic generated by the cache. A hardware technique, called Bandwidth-Aware LRU (BA-LRU) is presented. This technique is a modification of LRU and works to reduce off chip accesses without any support from software.
- Third, software support through profiling is added to BA-LRU which takes bandwidth management a step further. These techniques use compiler support to adapt to program behaviour at run time to reduce off-chip bandwidth.

The techniques are evaluated using the SPLASH-2 benchmark suite [7] to prove that we can trade off some cache performance for better bandwidth management to get overall better system performance.

1.3 Organization of the Thesis

The rest of the thesis is organized as follows. Chapter 2 presents the concepts of bandwidth for multicores. It explains the drawbacks of the current replacement policy in the multicore environment and how these drawbacks can be exploited to get a more bandwidth friendly policy.

Chapter 3 presents the previous work done in the field and lists the directions taken by researchers to enhance cache performance and optimize bandwidth.

Chapter 4 gives the first solution of bandwidth management which consists of modifying the current replacement policy for a more bandwidth aware one. This chapter presents a hardware solution with no support from profiling or software.

The Hardware-Software Hybrid scheme is discussed in Chapter 5. The chapter gives a detailed view of how software and profiling can assist with off-chip bandwidth management.

Chapter 6 explains the methodology and the simulation environment used in the experimental evaluation for the analysis of the new techniques. Then, a set of experiments are presented, followed by discussion to assess the potential of the new technique.

Finally, chapter 7 concludes the thesis by giving a summary of the work done as well as the future work needed to investigate the bandwidth management techniques even further.

Chapter 2

Problem Definition and Motivation

"After about 8 cores, there is no improvement. At 16 cores, it looks like 2"

James Peery (Director Sandia)

Researchers have realized the importance of bandwidth management in the multicore era. The full potential of multiple cores on chip cannot be exploited until we rethink the policies used for uniprocessor architecture before applying them to multicores. In a very interesting study [8], S. Moore showed that bandwidth is bad news for multicores and that if nothing is done to handle the bandwidth wall the performance will taper off with more than 16 cores. In this chapter we present two main points. First, we explain the bandwidth bottleneck and how it threatens to slow down the performance in multicores. Second, we show that the replacement policies of single core architectures are no longer safe to be used in multicores.

2.1 The Bandwidth Bottleneck

Memory bandwidth is the amount of data that can travel on the memory bus in a given period of time, and is usually expressed in MB/sec.

In order for multicores to deliver the promised performance benefits, it has to meet with the emerging challenges associated with their architecture. In most multicores the interconnects as well as the last level of cache are shared. The data travels on these shared interconnects and buses as it is written to or read from memory. An important issue of current multicore processors is the off-chip bandwidth sharing. This causes performance degradation due to contention amongst the applications running on these cores. Because of this, it's entirely possible for one of the memory-intensive applications to saturate the shared memory bus resulting in degraded performance for all the jobs running on that processor.

Off-chip Bandwidth wall is a situation in which bandwidth limitation becomes a bottleneck that limits multicore scaling.

As long as the memory bandwidth is shared between the cores, there will always exist the potential for bottlenecks. And as the number of cores per processor and the number of threaded applications increase, the performance of more and more applications will be limited by the processors memory bandwidth. Thus, memory bandwidth is the key reason that an eight-core machine will not have much performance and throughput benefit over a quad-core machine.

Previous work on bandwidth predicted the bandwidth wall to become performance and throughput bottleneck from as early as [9] for uniprocessors to more recently as in [10] for CMPs. Rogers et al. [11] show the bandwidth problem by calculating the total traffic generated in multicores as compared to single cores. They assume

that if number of transistors double every 18-24 months, according to proportional scaling number of cores should double as well. If we double the cache size, the traffic/core remains the same but total traffic (#cores * traffic/core) doubles. This shows that if off-chip bandwidth is limited, proportional scaling of cores is not attainable.

The problem for off-chip bandwidth is exacerbated for applications that have high number of memory accesses. There is an accelerating growth of world data. Applications today handle these data and hence have huge memory footprint and hence bandwidth is throttled when such applications are running on multiple cores. Applications like informatics (that require sifting through enormous databases of information), streaming voice and video, gaming, graphics, research e.t.c require constant off-chip and memory accesses. PARSEC is the latest benchmark suite that models the current applications. Bienia et al. in [12] show that PARSEC workloads have very demanding off-chip bandwidth requirement. Moreover these applications show a growing bandwidth demand per core as the number of cores increases. Thus these applications have high bandwidth requirements and will exceed current caches by far, bandwidth will be their most severe limitation of performance. Murphy et al in [13] examined the impact of memory latency and bandwidth on a set of traditional floating-point oriented and emerging integer oriented applications for the supercomputer environment. They demonstrate that there is some degree of bandwidth headroom in the construction of multicore supercomputers and that emerging applications are much more memory sensitive than traditional scientific computing codes.

Number of cores on chip are increasing, softwares are expanding, but pins are not scalable as already shown in Figure 1.1.

Running multiple memory hungry applications on many cores on chip creates a boost of off-chip traffic that cannot be handled with the limited pin scaling over the years. The bandwidth wall is born!

Thus, more cores does not mean better performance unless bandwidth is handled for them.

2.2 Rethinking LRU

Last Recently Used replacement policy has been considered safe to use for single core architectures. However, as we move towards multicore and manycore era the on-chip cache hierarchy is altering, arising the question of whether LRU is as effective for shared caches. Experiments are conducted to see the efficiency of the LRU replacement policy for shared last level cache. A 1MB L2, 8-way set associative cache is used for the motivational experiments, shared between four cores and the following three main observations are drawn.

2.2.1 Stack Hits

In our first motivational experiment, we count the number of hits at each block in the shared LLC. SPLASH-2 benchmarks[7] are used to show the pattern of hits in the cache.[1] Figure 2.1 to Figure 2.7 show the results for seven of SPLASH-2 benchmarks. 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)

[1]Benchmarks with highest number of off-chip accesses as they present the best scenario for our work

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 as shown from the figure. This is mainly due to two reasons. Firstly, the applications have exploited their temporal locality at L1 and hence there is lesser temporal locality at L2 or last level cache. 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 cores increase.

2.2.2 Dirty LRUs

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

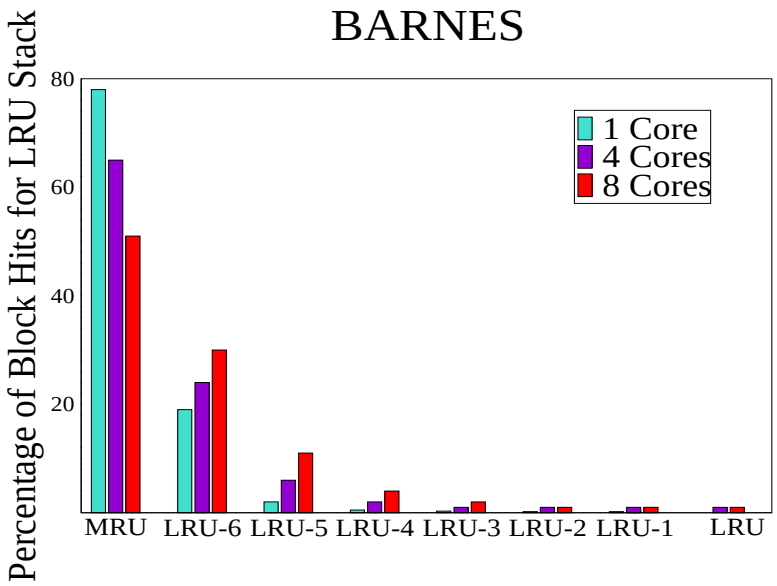

FIGURE 2.1: Comparison of Hit Rates for Each Block of an 8-Way Associative Cache for Different Number of Cores for Barnes

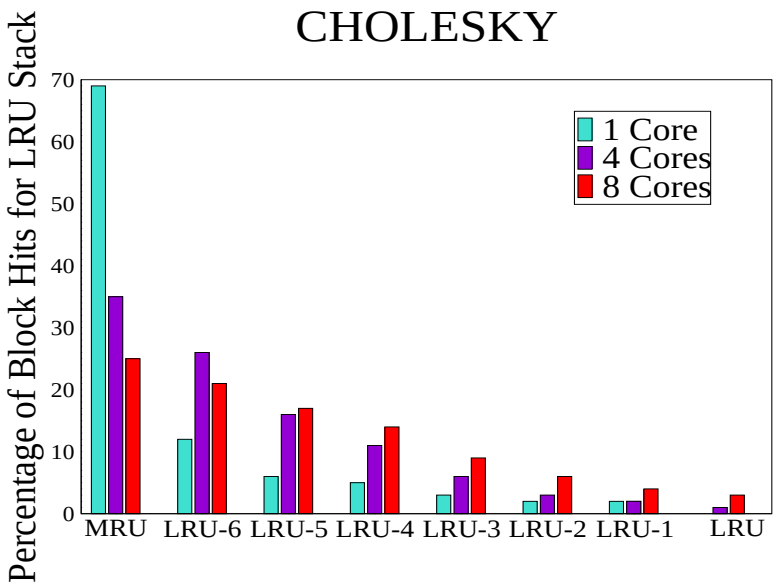

FIGURE 2.2: Comparison of Hit Rates for Each Block of an 8-Way Associative Cache for Different Number of Cores for Cholesky

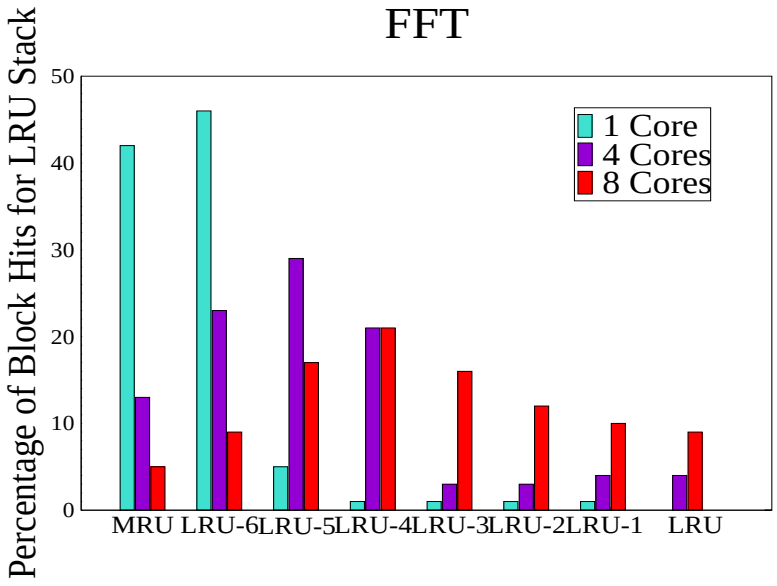

FIGURE 2.3: Comparison of Hit Rates for Each Block of an 8-Way Associative Cache for Different Number of Cores for Fft

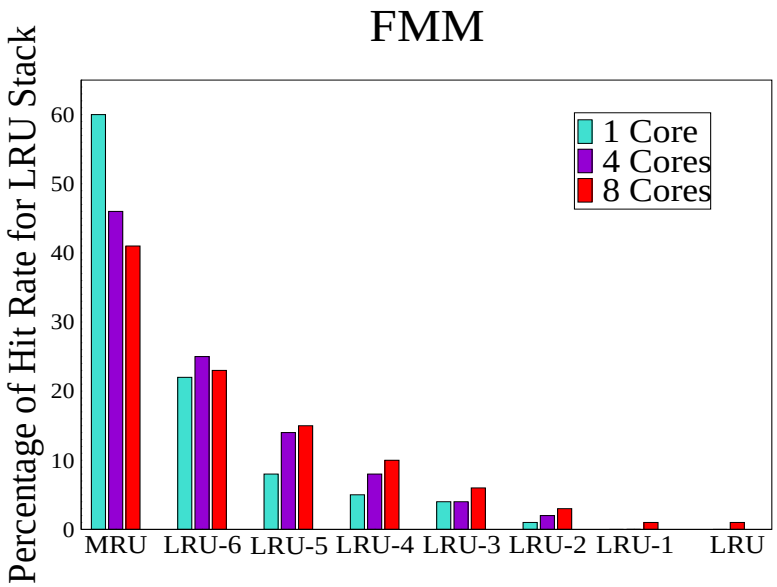

FIGURE 2.4: Comparison of Hit Rates for Each Block of an 8-Way Associative Cache for Different Number of Cores for Fmm

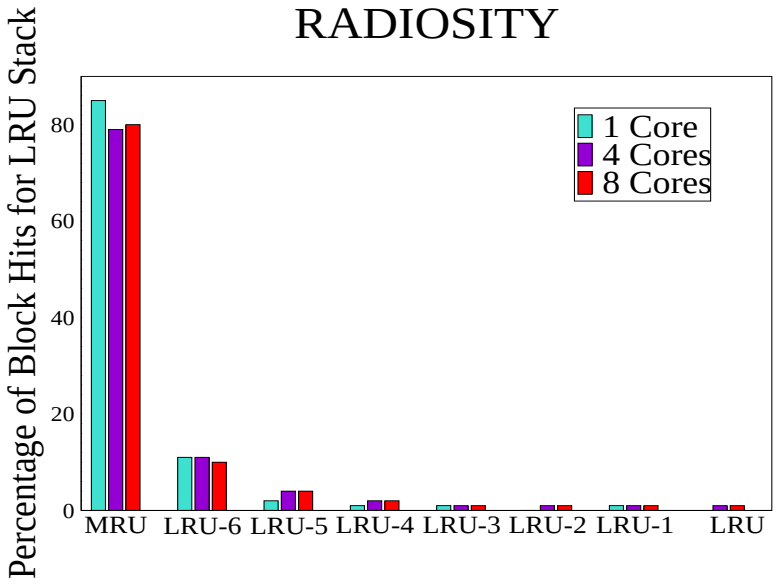

FIGURE 2.5: Comparison of Hit Rates for Each Block of an 8-Way Associative Cache for Different Number of Cores for Radiosity

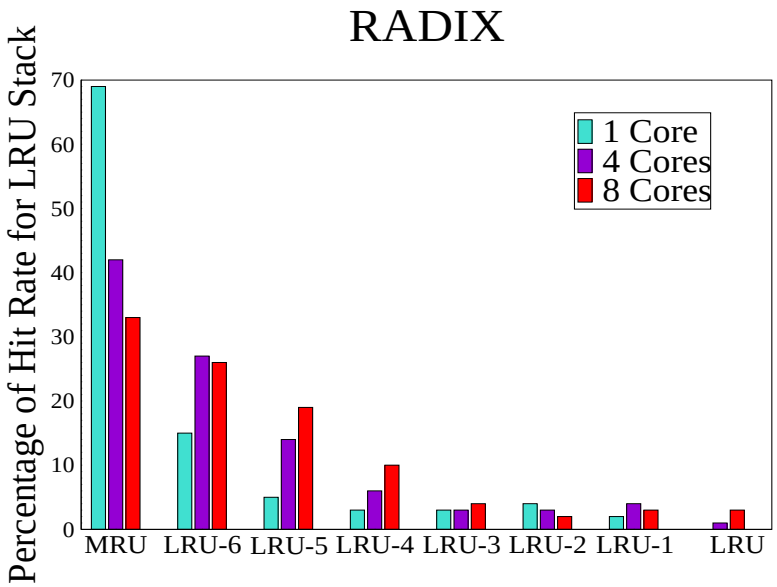

FIGURE 2.6: Comparison of Hit Rates for Each Block of an 8-Way Associative Cache for Different Number of Cores for Radix

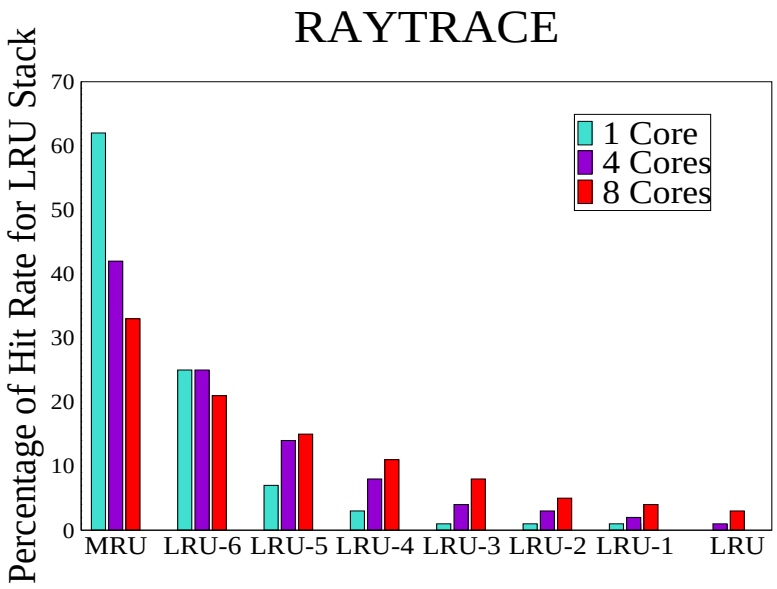

FIGURE 2.7: Comparison of Hit Rates for Each Block of an 8-Way Associative Cache for Different Number of Cores for Raytrace

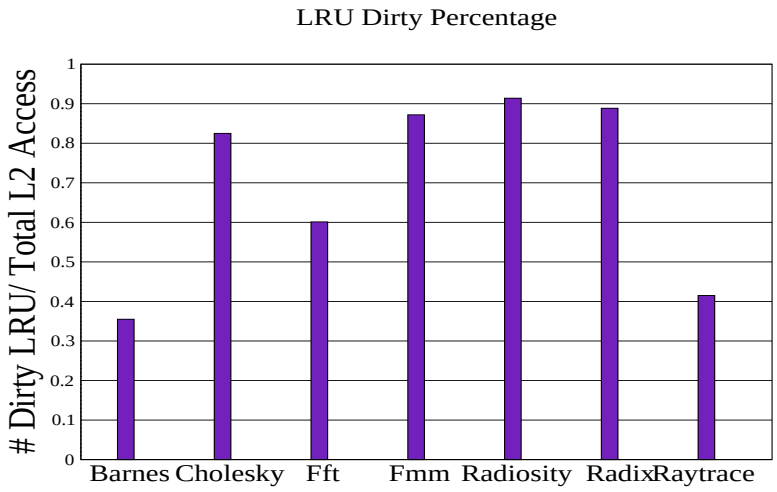

FIGURE 2.8: Ratio of Dirty LRU blocks to the Total Number of Accesses in LLC

times LRU is dirty to total cache accesses is compared. Figure 2.8 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 in an eviction. *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?*

2.2.3 Is LRU the Way To Go?

The above questions are answered in 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 2.1 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 non-LRUs is close to victimizing LRUs. The above observation can be exploited to victimize non-LRUs if it is beneficial for bandwidth.

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

2.3 Summary

The off-chip traffic produced by increasing number of cores on chip will require an ever increasing bandwidth. Research has to be done to manage bandwidth amongst cores in order to fully exploit the multicore architecture. Most of the off-chip traffic produced is from the Last Level Cache to memory. The replacement

policy of the LLC directly affects this traffic. The current commonly used replacement policy, that is the LRU, works well for uniprocessors. However, for shared caches, the LRU replacement policy shows some weaknesses and drawbacks.

- In a shared cache, neither are the hits at the top of the stack at the MRU position, nor does the LRU block represents the least important block. The interference causes hits to be scattered in the cache and hence the LRU stack is losing its credibility for shared caches.
- For many applications the LRU block is mostly dirty. Always evicting LRU would mean a write to the memory and hence potential off-chip traffic generation.
- Evicting a non-LRU block does not decrease performance.

If LRU stack is no longer important for shared caches, and LRU block is mostly dirty and hence evicting it causes traffic, why not victimize a clean non-LRU instead of a dirty LRU block and save writes to memory, specially when evicting non-LRUs does not mean performance loss.

We conclude that LRU can exploited to implement a more bandwidth aware replacement policy.

Chapter 3

Literature Survey

"On the shoulders of giants..."

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 optimization to make software application more bandwidth friendly.
- 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.

In the last decade, most of the research related to cache was taking two parallel paths. The first one is dealing with caches for uniprocessor systems, trying to enhance one or more aspect of the cache, such as the access time, accuracy, and power consumption.

3.1 Bandwidth Management by Cache Optimization for Uniprocessors

Cache enhancement techniques for Uniprocessors have taken several different paths. Two broad divisions of this work are explained below.

3.1.1 Adaptive Caches

Some work tries to capture the program behavior and adapt the cache based on this program's requirement [14, 15, 16]. Also, some attempts have been done in academia to enhance the memory system by varying one of its parameters dynamically [17, 18, 19, 20, 21]. Improving the hit rate was the main goal for a long time, for instance, column-associative cache [22], filter cache [18], and predictive sequential associative cache [21]. In [17], the authors found that in direct mapped cache, most of the address requests target a small subset of the cache lines, the rest being unused, or holes. They proposed to make use of these holes as internal victim cache, to store victim LRU blocks. They managed to obtain hit rate better than a 4-way set associative cache. In [23], the authors proposed a path balancing technique to help match the delays of cache and data paths. This has the effect of decreasing the access time. Another important aspect of any cache is its power consumption. Cache being one of the main power hungry structures on

chip, triggered a lot of research both academia and industry. In [20], the author proposed a cache with the ability to disable a subset of the ways in a set associative cache during periods of modest cache activity in order to save power. In [19], a cache memory is proposed where cache line is continuously adjusted by hardware based on observed application access. Replacement policy is another major design parameter in our quest to enhance cache performance.

3.1.2 Optimizing Replacement Policies

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 the 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.

To obtain a balance between the qualities of random replacement and LRU the PLRU (Pseudo LRU) scheme was proposed that employs the approximations of the LRU mechanism to speed up operations and reduce the complexity of implementation.

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)[24], 2Q (use two queues to quickly remove cold blocks), LIRS (Low Inter-reference Recency Set)[25], [26], CRFP [27] (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. 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.

Other work includes [28] which is cache injection which combines the good properties of data prefetching and data forwarding in order to reduce the miss ratio, which will decreases the overall traffic and indirectly reducing the bandwidth requirements of the system.

3.2 Bandwidth Management for Multiprocessors

The second path taken in cache hierarchy research is designing cache hierarchy for multiprocessor systems. The research in that area consists mainly of coherence protocol schemes [29, 30, 31, 32]. These two paths in memory system research were done in parallel. But with the introduction of CMPs [33, 34], the two paths start to converge [35, 36, 37], and the main theme was the high-bandwidth available on-chip, opposite to the limited off-chip bandwidth. The following methods are taken by researchers to mitigate the affect of off-chip bandwidth.

3.2.1 Network on Chip

Although high-bandwidth is available on-chip, the wire delay represents a big challenge because it does not scale with the transistor [36, 38]. This leads to the introduction of network on chip [39, 40, 41], and fair-sharing of caches [42]. Network on Chip is an approach to design the communication subsystem of System on Chip (SoC). NoC-based systems can accommodate multiple asynchronous clocking. NoC use networking theories and systematic networking methods and apply them to on-chip communication. This results in notable improvements over conventional bus systems. NoC has improved the scalability of SoCs, and shows higher power efficiency in complex SoCs compared to buses. In most modern architectures, buses have become obsolete and are replaced by NoCs.

3.2.2 Fair Queuing

Recently, researchers have recognized the fairness issues involved in the management of DRAM bandwidth. Allocating bandwidth shares to achieve the necessary

latency for the worst case is wasteful.

Nesbit et al. [43] propose a fair queuing scheme for providing fairness, but their technique can suffer from starvation They proposed a scheme which keeps track of the latencies threads would experience without contention. Moscribroda et al. [44] recognized the starvation problems in the work of Nesbit et al. and proposed a scheme which keeps track of the latencies threads would experience without contention.

The notion of fairness is well defined and well studied in the context of networking. Fair queuing systems are based on the concept of max-min fairness [45], [46].

A service discipline which follows the above definition of fairness divides the available bandwidth among the backlogged queues in proportion to their share. If a sharer is not backlogged, its share of bandwidth is distributed among backlogged sharers.

The scheduling algorithms which penalize sharers for the using idle bandwidth are generally regarded as unfair [47].

Most recent work on fair sharing has been done by Kim et al [48], [49] and [50].

The above techniques have achieved some success but have problems like starvation, complexity, and unpredictable DRAM access latency. Rafique et al. [51] propose a DRAM bandwidth management scheme with two key features allows OS to select the right combination of performance and strength of bandwidth share enforcement thereby avoiding unexpected long latencies or starvation of memory requests. Second, it provides a feedback-driven policy that adaptively tunes the bandwidth shares to achieve desired average latencies for memory accesses.

3.2.3 Memory Management and Scheduling

More work has been done that discusses the impending memory bandwidth limitations as CMP core scaling continues and the effect of the power law of cache miss rates on the relationship cache and bandwidth is discussed[50].

Iyer et al. [52] used a simple scheme for DRAM bandwidth allocation. [53] studied the impact of memory controller configuration on the latency-bandwidth curve.

There have been many proposals discussing memory access ordering [54], [55], [56] and [57].

Patterson et al. [58] propose the unification of the processor and the DRAM into a single chip, the IRAM. In such case, the memory will be able to operate at processor speed, increasing the bandwidth 100-fold.

3.2.4 Compiler Optimization

Another path taken by researcher for mitigating bandwidth bottleneck is to use compiler optimization. Many studies have focused on compiler analysis and optimization to improve cache performance [14, 59, 60] 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 [14, 60, 61], or change the memory access order to improve the temporal and spatial locality.

The live variable analysis is a very useful technique used for various static optimizations like register allocation and dead code elimination [62]. *However, we do not know of any work that passes the compiler generated dead value information to the hardware for dynamic optimization.* Some work has been done in dead

value detection for register values [63, 64]. [63] proposed a hardware technique for detecting dynamic instruction instances that generate unused results in registers. [64] performs static analysis to determine the dead register information, and uses this information at runtime to perform various optimization, such as decreasing the physical register file size, and eliminating register save and restore, at procedure calls and context switches. Using dead block prediction to enhance prefetching is proposed in [65].

3.2.5 Compression

Compressing off-chip traffic is the third path taken by researchers to make the best usage of the available bandwidth [66]. 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.

3.2.6 Multithreading

Multithreading is a method to hide latency [67, 68, 69, 70]. 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.

3.3 Summary

Previous work on bandwidth management has taken two broad paths. This chapter showed a brief description of each path. Bandwidth optimization in multicores requires off-chip traffic management which is divided into traffic to and from LLC. In the first path, the researchers improve the cache performance to reduce the traffic coming into the LLC from memory. This work is mostly for the uniprocessor architecture. The second path is targets the multicore architecture. The researchers use techniques like networks on chip, compiler support, compression and multithreading to reduce the pressure of traffic towards memory. All the above techniques have reached varied degrees of success. The target now is not only to enhance cache performance but also to mitigate the bandwidth bottleneck for multicore systems. Most of the work however is still targeted to reduce read misses or the traffic coming on chip with very little work done to reduce the traffic going off-chip. The field of memory systems for CMPs is not yet mature, and there is a long way to go. In none of the previous works the validity of LRU replacement in multithreaded applications is tested. Nor any work is done that targets solely at the bandwidth management in multicores which is the main topic of this thesis.

Chapter 4

Hardware Schemes

"Software comes from heaven when you have good hardware."

Ken Olsen

How can we make use of the fact about LRU stack in LLC to design an off-chip bandwidth friendly LLC? We propose several techniques of increasing complexity. The proposed techniques vary from simple design-time schemes, to dynamic schemes, to compiler support.

4.1 Static Technique

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 in Chapter 2

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 namely *Bandwidth-Aware LRU* (BA-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 like in the regular replacement policy. For example if M=3, the technique would look for the first non-dirty block from three blocks starting from LRU and victimize it. Thus this technique depends on the parameter M.

M is the number of blocks we check in the cache to find a possible clean block.

This parameter designates how many blocks we will 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 is checked to find a possible clean victim. So for an 8-way set-associative cache, the maximum M is 8. We test with values of M ranging from 1 to associativity.

4.1.1 Hardware Cost

The hardware cost of this technique consists of just an encoder. The input to that encoder is the dirty bits of each block in the accessed set and the output is the victim block number. So if M=n, the encoder has n inputs and $\log_2(n)$ outputs. Since the LRU stack hardware already exists, the only extra hardware we use is the encoder. For M=4 the encoder is fed with the dirty bits of the LRU block, LRU-1 block (i.e. the one just above the LRU block in the stack), LRU-2 block, and LRU-3 block. The output bits (A and B in the figure) designate one of the four blocks as the victim.

Figure 4.1 is an implementation of an encoder with M=4.

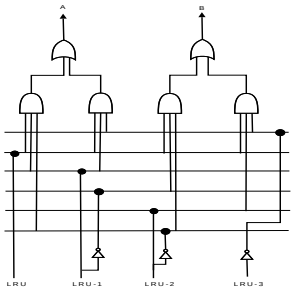

FIGURE 4.1: Encoder Implementation for M=4

4.2 Dynamic Technique

The main advantage of the static scheme is its simplicity. Once we decide on the design parameter, the implementation is fixed. However, determining this design parameter is usually not easy because it depends on runtime information that is not available at design time, so it relies on the designer experience and a lot of simulations. In the static design mentioned in the previous section, determining M is very challenging and is application dependent. We need a dynamic scheme where M changes dynamically depending on the application behavior. We propose two schemes for dynamically changing M; *Writeback-sensitive* and *LRU-sensitive*. Each of these schemes can either be Global (same M for the whole cache) or Local (M per set).

Writeback Sensitive Scheme: When M is high, we have a higher chance of reducing traffic, but also a higher chance of affecting performance. This means that M is governed by how much traffic (writebacks) is generated, and this traffic is generated only in case of a cache miss. So in writeback-sensitive, M is incremented when there is a writeback, to reduce off-chip traffic, and is decremented when there

is a cache miss with no writeback, to reduce performance loss. The new value of M will be used the next time a victim is to be chosen.

LRU-Sensitive Scheme: We check the LRU block every time the cache is accessed to see whether it became dirty. If it is, then we have potential traffic if we victimize the LRU block, so we increment M to reduce potential traffic. We decrement M whenever there is a cache miss to reduce potential performance loss. The technique works on the fact that if LRU is made dirty then there are more chances of a write, so we increase the value of M to check more blocks for a clean block. If however the application is LRU is friendly it will result in misses due to BA-LRU, each miss would reduce the value of M bringing it back to 1 (LRU). Thus this scheme tries to reduce "potential" bandwidth. It assumes implicitly that a dirty LRU will generate bandwidth. This assumption is not correct all the time this is why the former proposed scheme (writeback-sensitive) turns out to be better.

4.2.1 Hardware Cost

The hardware implementation here is more sophisticated than the static technique, but is still minimal. First, for M, we will use a saturating counter whose size equals the associativity of the cache. This means it will be at most 5 or 6 bits, which represents the largest associativity found in current state-of-the-art processors. In our experiments, we use a 3-bit counter corresponding to an 8-way set associative L2 cache. The encoder must accommodate all the blocks in the set, therefore must be the maximum associativity. Depending on the counter, which is M, some inputs (the highest blocks in the LRU stack) to the encoder will be set to one (i.e. dirty), hence are guaranteed not to be chosen by the encoder. For example, if M

is currently set to 5, the top three inputs (i.e. MRU, MRU+1, and MRU+2) are all set to one. This is a very straightforward implementation. However, if we want to save on power consumption, we can disable parts of the encoder depending on M.

4.3 Summary

This chapter makes the following contributions:

- The shortcomings of the LRU replacement policy in the multicore environment are exploited to propose a simple technique that reduces off-chip traffic. The technique presented is called *Bandwidth-Aware LRU*. It is a modification of the LRU replacement policy and requires only hardware support. In BA-LRU a clean non-LRU block is chosen as victim instead of a dirty LRU.
- A parameter M is defined which is the number of blocks checked in the cache to find a possible clean block.
- We present two variations of BA-LRU, the static and the dynamic technique.
- In static technique the value of M is constant throughout the program execution.
- In dynamic technique the value of M varies during run time. This gives a way to adjust this parameter according to program behavior. However, no software support is employed.
- We propose two different schemes to vary the parameter M at run time, the writeback-sensitive and the LRU-sensitive scheme.

- The hardware costs are compared and it is shown that all the techniques require minimal hardware overhead.

Chapter 5

Hybrid Schemes

"I'd rather have a search engine or a compiler on a deserted island than a game."

John Carmack

In this section we propose two algorithms that use compiler support to manage bandwidth by adapting to program behavior at run time. Each of the technique is explained in detail below.

5.1 Compiler Support

Each step in this technique is explained below. The technique starts by collecting application behavior through profiling.

5.1.1 Profiling

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

5.1.2 Write Frequency Vector

During profiling, the compiler builds a write frequency vector (WFV). WFV, as shown in Figure 5.1, is just a histogram representation of the number of write-backs at several intervals. During profiling, the compiler keeps a count of the number of writebacks at each interval of length X. Then, based on that number, an entry of the WFV is incremented by one. In our experiments, we made the compiler check the number of writebacks every 10,000 cycles. We have a WFV of 32 elements, where each element represent an interval of 10. For example, if the number of writebacks in an interval is 75, will increment the element number 7 in WFV. The above numbers are empirical based on the SPLASH-2 suite, but can be easily changed. After the profiling phase, the compiler has a WFV representing writebacks distribution for the application at hand.

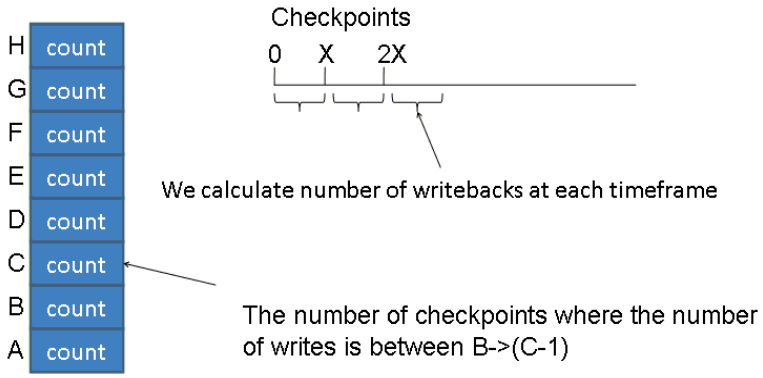

FIGURE 5.1: Write Frequency Vector (WFV)

5.1.3 Noise Removal

The following step removes noise from WFV. We define noise as any entries that are smaller than 5% of the largest value in WFV. This noise-removal step is necessary because we want our algorithms to be based on the frequent behavior, not some individual rare events.

5.1.4 Defining Thresholds

Thresholds represent two values called Min and Max that will be 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 noise-free WFV, we propose two algorithms; frequency-based algorithm, and weighted average algorithm.

5.1.4.1 Frequency-Based Algorithm

Given a noise-free 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.

5.1.4.2 Weighted Average Algorithm

In this algorithm Min is always zero and Max = $\sum_{i=0}^{32} \frac{WFV_i * mid_i}{SUM}$
where WFV_i is the content of element i of the WFV vector, mid_i 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), and 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. The hardware execution is similar to the frequency-based algorithm above.

5.1.5 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 5.2. 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)

5.1.6 Extrapolation from Single to Multicore

One last comment on the above thresholds is that they were computed through profiling with a single core, but will be used in a multicore environment. With

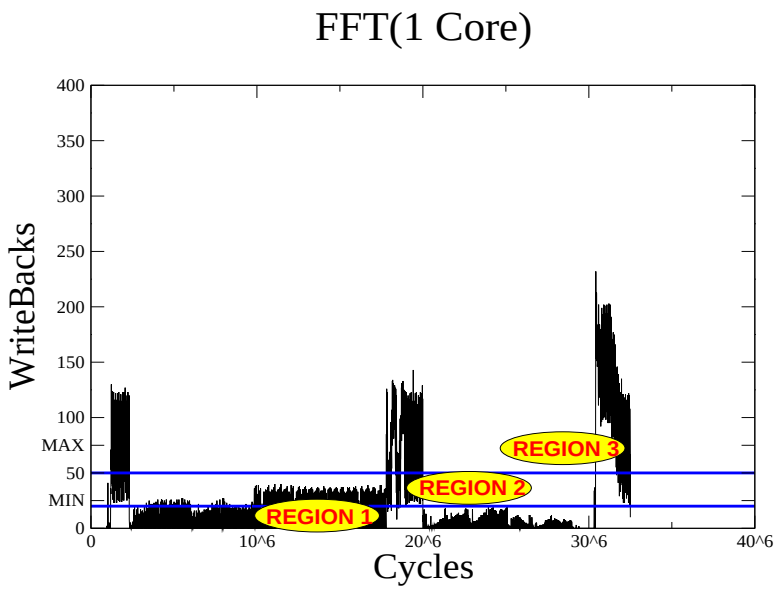

FIGURE 5.2: Off-Chip Traffic for Fft Benchmark in 1 core Showing Thresholds and Regions

multicore, and hence multithreading, the traffic is expected to be higher due to the increase in misses (coherence miss), coherence traffic, etc. So before using these thresholds, we multiply them by the total number of cores. Fft is shown as an example to show the traffic increase from one to four cores in Figure 5.3. Although the total execution time decreases, the overall off-chip traffic increases.

5.1.7 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 BA-LRU with M=associativity/2. Finally, a number above Max means we have a lot of writebacks, so we use BA-LRU with M=associativity. This is shown in Figure 5.4.

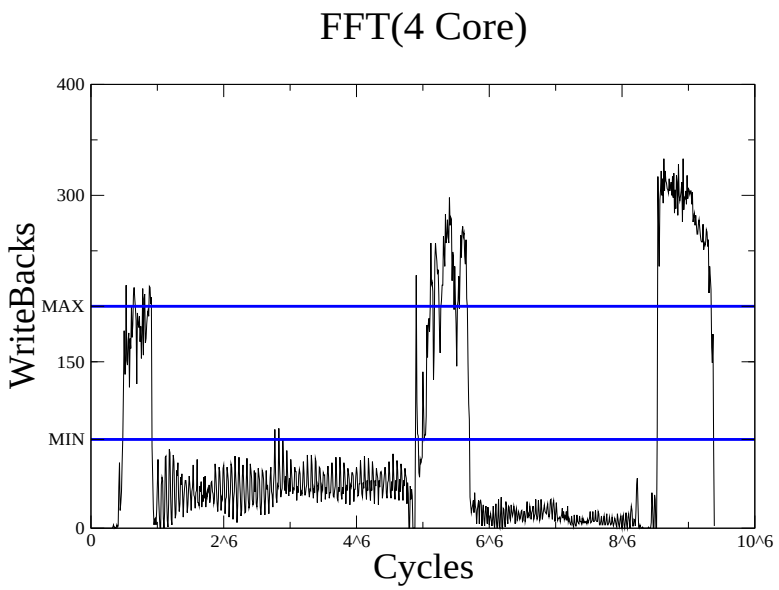

FIGURE 5.3: Extrapolation of Thresholds for Fft Benchmark Running on 4 Cores

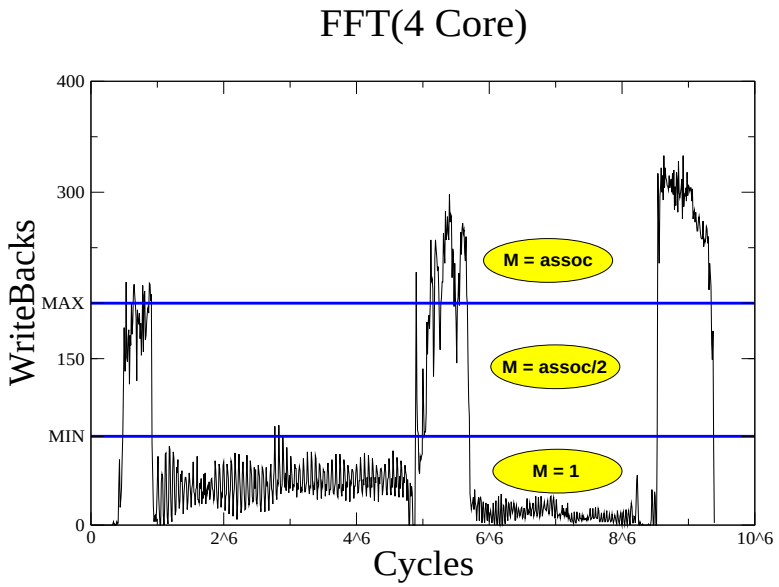

FIGURE 5.4: Replacement Policy Selection for Each Region

5.2 Compiler Support + Dynamic Techniques

The last set of techniques is to combine compiler support with dynamic adaptation. The hardware will start with the above thresholds, but will adjust them

dynamically based on the behavior. It will keep track of the number of writes every X cycles, as indicated above. The main difference is when an *event* occurs.

An event is defined as a situation where half or all the sets in the cache have dirty LRUs. Whenever this event occurs, the counter that keeps track of dirty LRUs is reset if all the sets have dirty LRUs, and Max is updated as follows. If the number of writes (numWRITES) at the current time frame (which is every X cycles) is larger than the current Max, then the $newMax = numWRITES - delta/4$, where delta is the difference of numWRITES and currentMax. Min is adjusted in a way to make the difference between Min and Max constant. The 4 does not represent the number of cores, but it means adding to numWRITES 25% of the difference, which decreases the effect of some bursty noisy behavior.

5.3 Summary

This chapter proposed techniques that use both hardware and software to reduce off-chip traffic.

- The software support is done through profiling of the application on one core. The frequency vector of the traffic pattern is obtained.
- From the frequency vector, the thresholds are chosen. The thresholds are minimum and max writebacks that divide the writeback graph over the entire run time into three regions of high, low and medium traffic.
- A value of M is chosen for each region.
- In compiler support the thresholds remain constant during the entire execution time.

- In compiler support + dynamic technique we alter the value of thresholds to adjust them according to program behavior.

Chapter 6

Experimental Results

"Seeing is believing."

In this chapter the simulation results of the techniques presented so far are shown and discussed in detail.

6.1 Evaluation Methodology

6.1.1 Experimental Setup

We modified SESC simulator [71] to implement our proposed schemes. The simulator is explained in detail in Appendix A. Table 6.1 shows the main parameters for the system used. These parameters are similar to many state-of-the-art processors. However, we use a relatively small LLC size to have more pressure on the cache due to the small working set size of our benchmarks. The inclusion property

is not enforced which is similar to most of the current processors. The L1 cache parameters are kept constant for all experiments.

6.1.2 Benchmarks

We use SPLASH-2 benchmark suite in our study. The SPLASH-2 suite have a total of eleven benchmarks. We choose seven of these benchmarks in our study because of their larger memory footprints. We run them till completion. A brief description of each benchmark along with the problem set used is given in Table 6.2.

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 [7] so we can understand how our techniques interact with these programs. Currently we have not yet been able to cross-compile PARSEC suite [12], which is a more recent

Table 6.1: Main Configuration

Number of Cores	4
Execution	Out of Order
Issue/Decode Width	4
Instruction Fetch Queue Size	8
Branch Mispredict Latency	2
Branch type	Hybrid
L1 I-Cache	32KB; 64B linesize; 2-way with LRU repl
L1 D-Cache	32KB; 64B linesize; 2-way with LRU repl
Baseline L2	1MB; 64B linesize; 8-way; shared
Main Memory	500 Cycles Static Latency

Table 6.2: SPLASH-2 Workloads and Inputs Used

Program	Application	Input Size
Barnes	Simulates interaction of a system of bodies by implementing Barnes-Hut	16K
Cholesky	Factorization on a sparse matrix	tk29.0
Fft	Fast Fourier Transform (Signal Processing)	64K
Fmm	Implements a parallel adaptive Fast Multipole Method to simulate the interaction of a system of bodies	16M
Radiosity	Computes the equilibrium distribution of light in a scene using the hierarchical diffuse radiosity method	room
Radix	Implements an integer radix sort	1M
Raytrace	Renders a three-dimensional scene onto a two-dimensional image plane using optimized ray tracing	car

benchmark suite from Princeton, on SESC.

6.2 Experimental Results

Our main goal in this thesis is to find a technique that reduces off-chip traffic going towards the memory with as little impact as possible to the performance. This means we want to reduce the number of writebacks while not affecting the total number of cycles needed to execute the multithreaded program.

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 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. So, 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.

6.2.1 Hardware Schemes

The hardware schemes are divided into Static and Dynamic techniques as shown in Section 4.1. The results for simulations based on these techniques are shown and compared with currently used schemes.

6.2.1.1 Static Technique

Our first set of experiments compares traditional LRU with BA-LRU for different values of M.

Writebacks: Figure 6.1 shows the number of writebacks normalized to LRU. All of the benchmarks show a decrease in the number of writebacks. This is because we keep dirty blocks in the cache for longer and overwrite them if they are not dirty which reduces the number of writebacks to the memory. The maximum reduction in writebacks is shown by `Raytrace` in which the writebacks are reduced to near 0. This means that most of the writebacks for that application were for local variables which are no longer needed or for register spilled. That is, they are dead values. `Raytrace and Cholesky` show a decrease of 36.8% and 93.9% respectively.

Read Misses: Although we have reduced the traffic going from LLC to off-chip, we want to be sure that we have not increased the traffic coming from off-chip to memory. Figure 6.2 shows the normalized number of read misses for different values of M. For most benchmarks the increase in misses is less than 15% except for `barnes` which has an increase in miss of 56.9%. The high increase in misses of `barnes` means that it is LRU friendly, and it has high temporal locality. For `Fft and Cholesky` we even see a decrease in the number of misses by 16% and 14% respectively. This means that LRU is not always the best replacement policy, and it is better sometime to *violate* LRU to gain reduction in off-chip bandwidth.

Performance: Figure 6.3 shows the total number of cycles normalized to the LRU scheme.

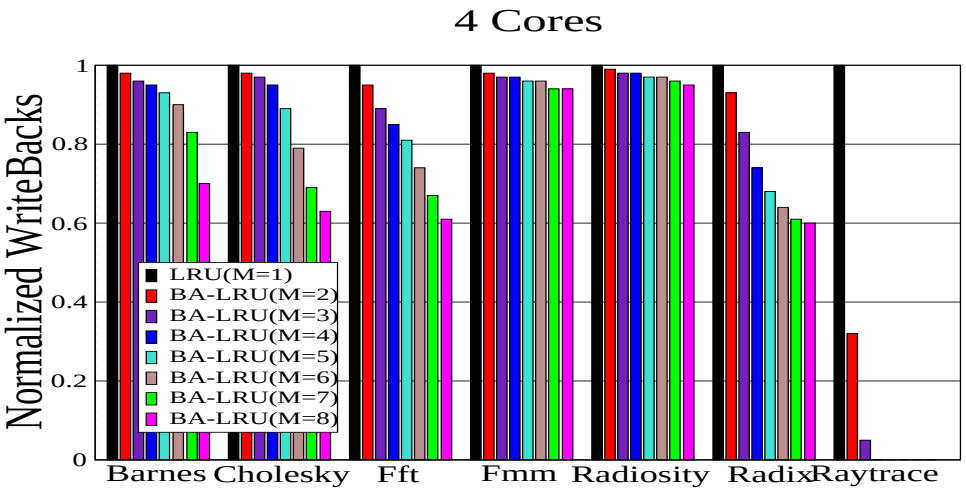

FIGURE 6.1: Normalized Number of Writebacks for BA-LRU

For three benchmarks (`Fft, Cholesky and Radix`) we see an increase in performance since the number of cycles have decreased. This is because we decreased delay caused by bus contention and memory port contention beside a reduced

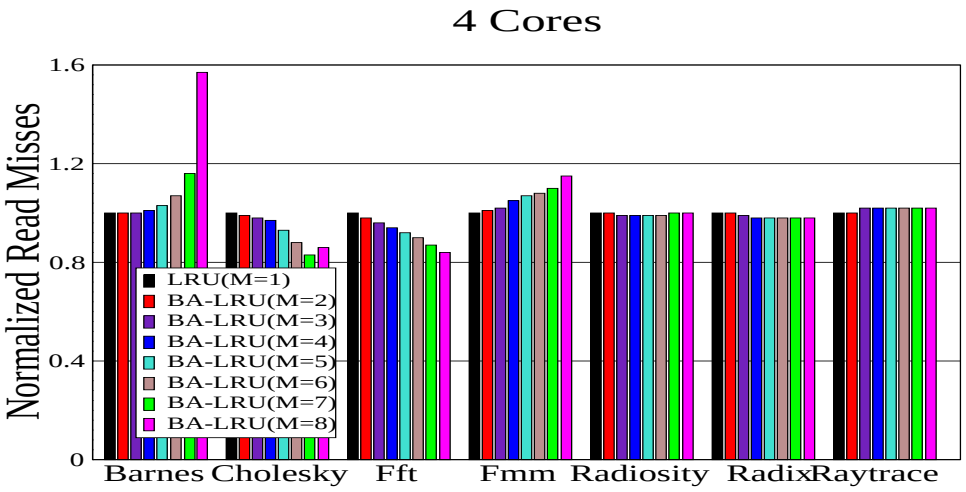

FIGURE 6.2: Normalized Number of Read Misses for BA-LRU

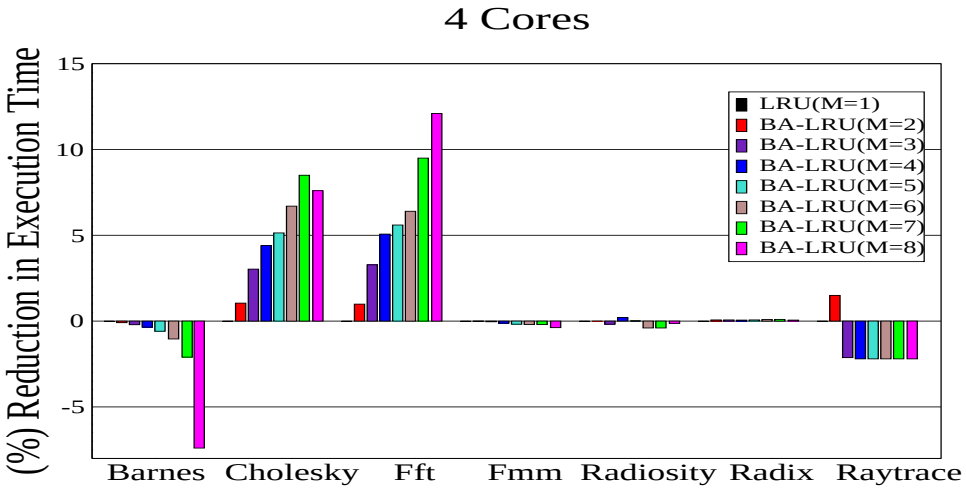

FIGURE 6.3: Speedup

miss penalty because the victim is overwritten not written-back. For the benchmarks that have an increase in misses we expect some performance loss. Only three benchmarks show a decrease in performance out of which two benchmarks have increase in number of cycles less than 1% except for barnes which is 7.5%. By using M=3 or M=4 we guarantee good performance. For the rest of thesis we use BA-LRU with M=4 as our main static technique to compare with other techniques. Also for the rest of the experiments, we will drop `Fmm and Radix` due to their very low number of writebacks.

6.2.1.2 Dynamic Techniques

The two presented Dynamic techniques in Section 4.2 are the `writeback-sensitive` and the `LRU sensitive` schemes. Both of these techniques can be either Global or Local.

Figure 6.4, 6.5 and 6.6 compare the writebacks, read misses and cycles of the four versions of the dynamic scheme respectively. The values are normalized to regular LRU. We see that the global scheme is better from a price/performance point of view. The gain we get from the local schemes does not justify the extra hardware. The writeback-sensitive is better than LRU-sensitive techniques for most of the benchmarks. So we will be using writeback-sensitive with global M as our dynamic technique (we will call it dynamic M). To get more insights into the mechanism of the dynamic scheme, we kept track of the value of M in the global dynamic scheme throughout all the simulations. Figure 6.7 shows an example of one of the benchmarks `Fft` as an example. We can see two important characteristics from this graph. We see two phases of program execution, and each phase there is a cyclic behavior. This corroborates with previous findings about program phases and cyclic behavior [72]. This is an indication that our criteria of when to increment and decrement M is successful in capturing the behavior and phases of a program execution.

6.2.2 Hybrid Schemes

This section presents the results on the two variations of the Hybrid Schemes. The compiler now assists the hardware to manage bandwidth at according to the

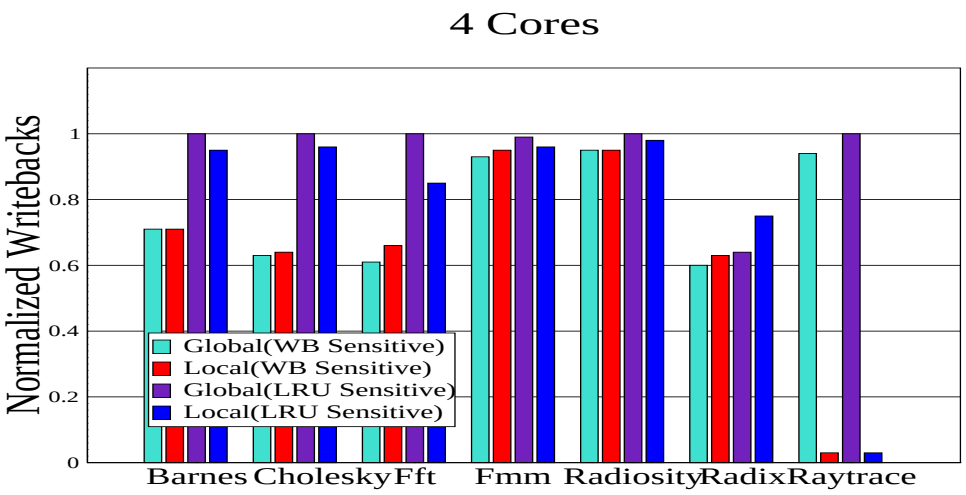

FIGURE 6.4: Normalized Number of Writebacks for Dynamic Schemes

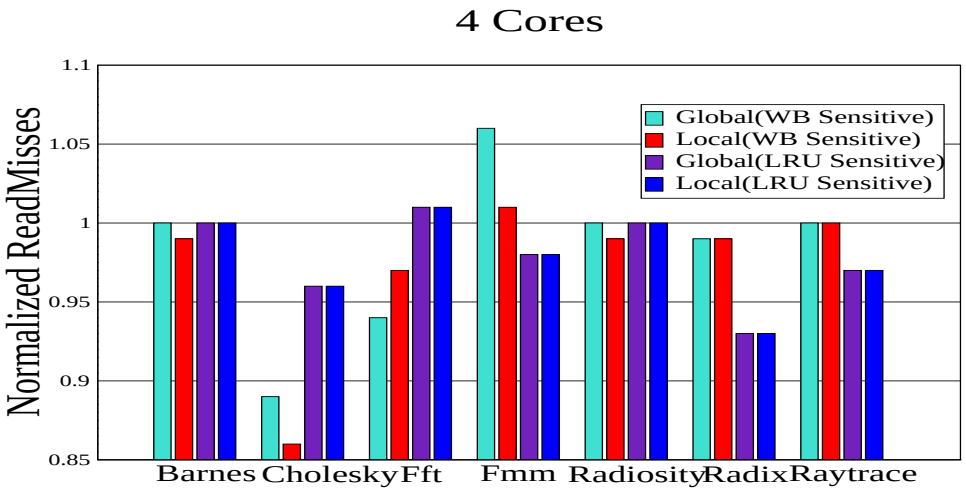

FIGURE 6.5: Normalized Number of Read Misses for Dyamic Schemes

traffic pattern at run time.

6.2.2.1 Compiler Support

Writebacks: Figure 6.8 shows the normalized number of writebacks. We compare with eager-writeback [73](a compromise between write-through and write-back), DIP [74](Dynamic Insertion Policy), the two proposed techniques using compiler support (frequency-based and weighted-average) and BA-LRU with M=4.

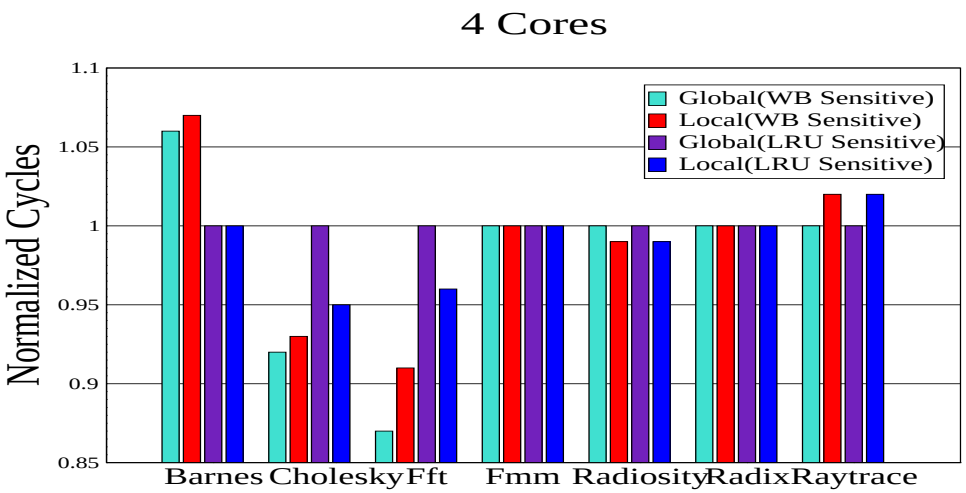

FIGURE 6.6: Performance for Dynamic Schemes

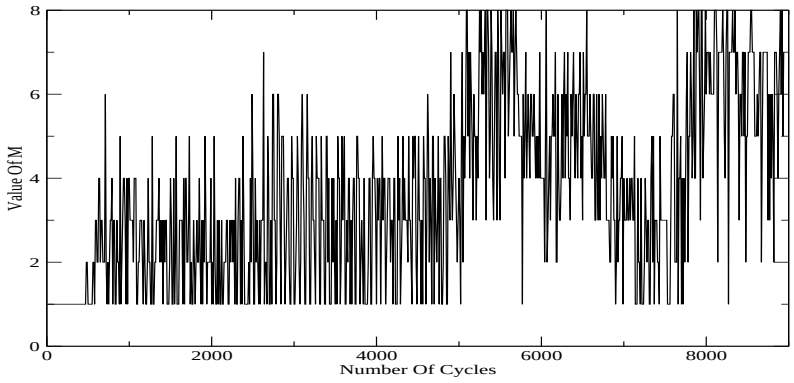

FIGURE 6.7: Changing Values of M in The Dynamic Scheme of Global Writeback-Sensitive

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 6.9. As the number of cores increases, compiler based weighted-average

method and static BA-LRU with M=4 are doing much better.

Performance: The proposed techniques are not hurting performance as indicated by Figure 6.10. `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 2.1.

6.2.2.2 Compiler Support + Dynamic Techniques

Writebacks: Figure 6.11 compares the total number of writebacks normalized to LRU. The weighted-average and dynamic M schemes are doing the best on average. This means that they are the most accurate capturing blocks behavior. The LRU component of the frequency-based scheme is affecting its ability to save on off-chip bandwidth.

Read Misses: Figure 6.12 compare the same techniques in terms of read misses. With the exception of `Barnes`, dynamic M and weighted-average methods are doing the best on average, together with DIP. To understand the behavior of `barnes` it is important to see the number of dirty blocks it has. We explain this in Section 6.2.3.

Performance: We see from the results that `barnes` a negative effect on performance. Figure 6.14 shows the percentage of dirty blocks to the total number of blocks at LLC for `barnes` over time. There are phases where this program has

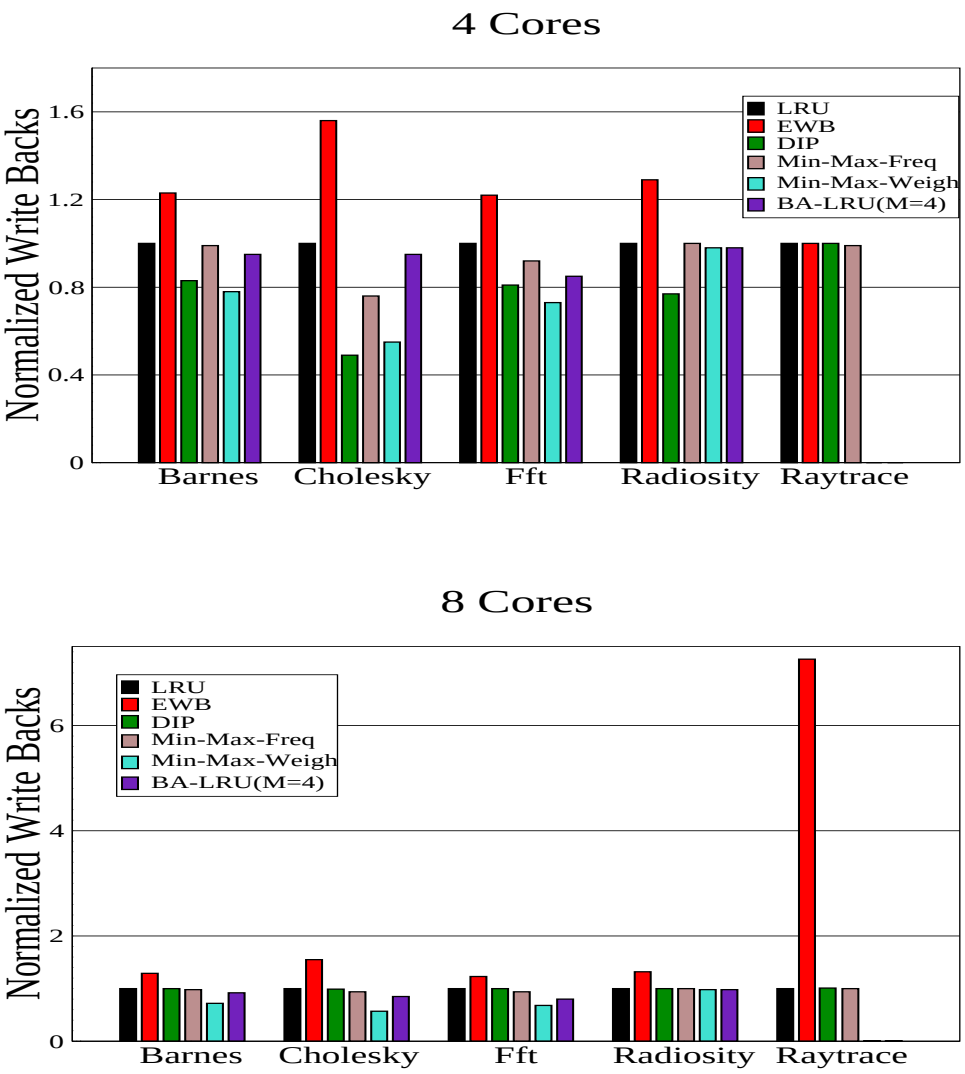

FIGURE 6.8: Normalized Number of Writebacks For 4 Cores (Top) and 8 Cores (Bottom) (Compiler Support and Static Schemes)

80% of its blocks dirty. This cause M to be very high which increases the likelihood of discarding an important block because high values of M affects `barnes` performance as indicated by Table 2.1 earlier. This also explains the speedup (and slowdown) shown in Figure 6.13.

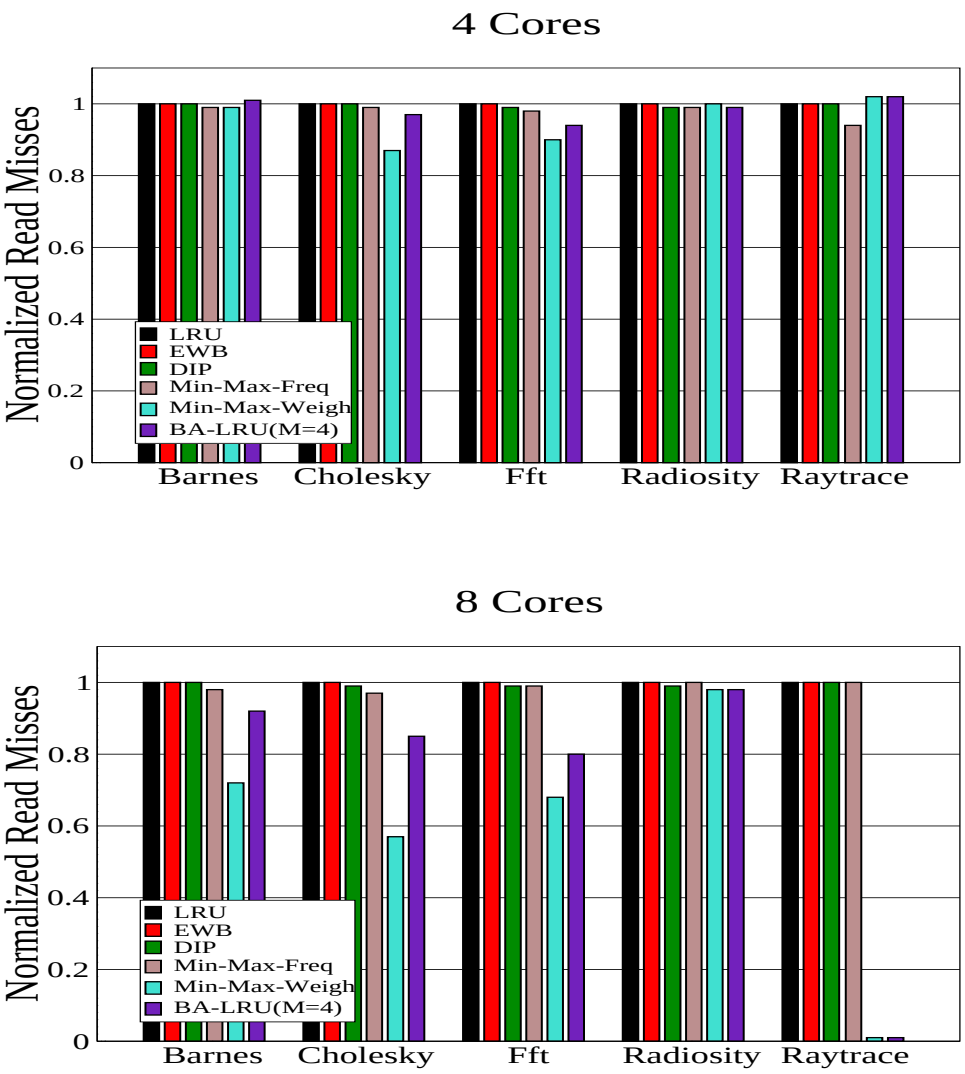

FIGURE 6.9: Normalized Number of Read Misses For 4 Cores (Top) and 8 Cores (Bottom)(Compiler Support and Static Schemes)

6.2.3 Some Insights

It is important to understand the behavior of the benchmarks in order to understand the effect of the techniques on them. All the techniques work to manage bandwidth by keeping dirty blocks in the cache for longer. We calculate the percentage of dirty blocks to the total number of blocks for each benchmark. The results for this experiment are shown in Figures 6.14 to Figure 6.18.

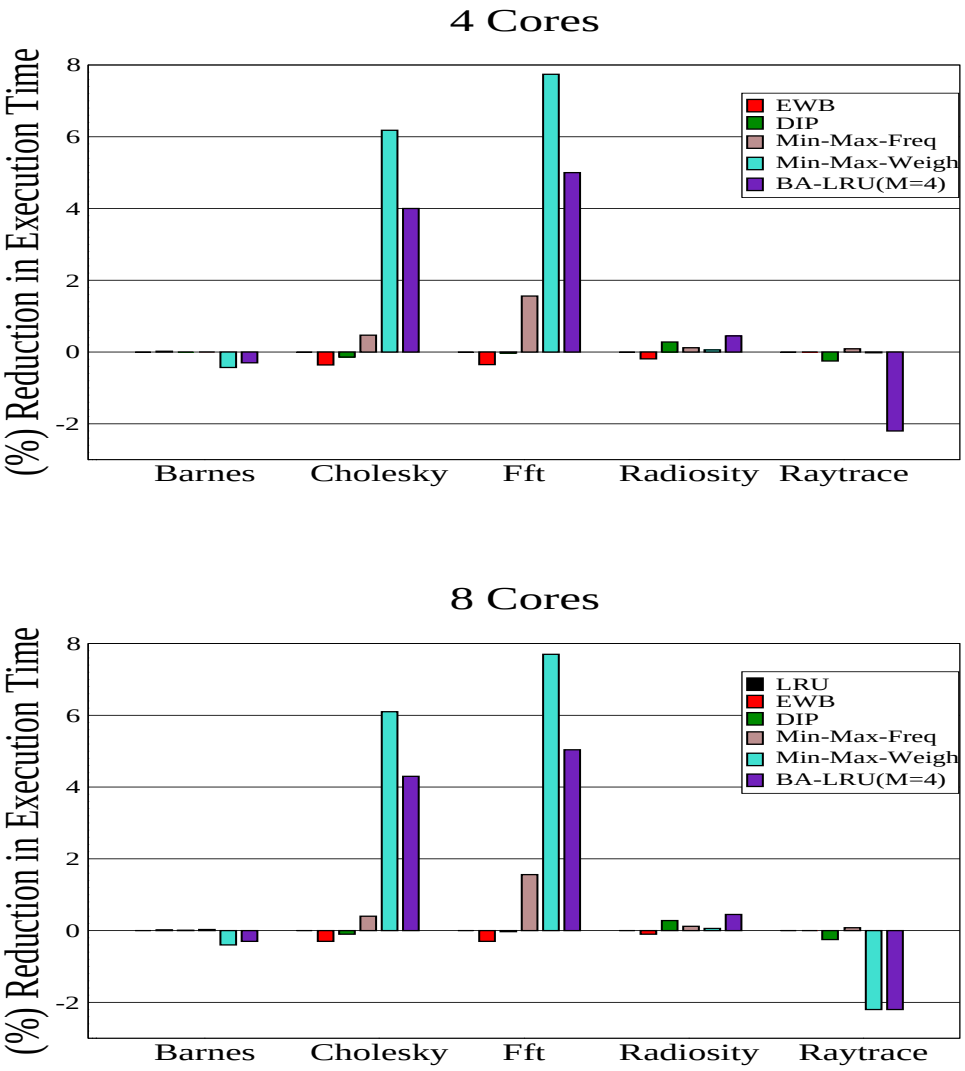

FIGURE 6.10: Speedup For 4 Cores (Top) and 8 Cores (Bottom)(Compiler Support and Static Schemes)

Figures 6.14 to 6.18 show the percentage of dirty blocks to the total number of blocks at LLC for the benchmarks over time. We explain the behavior of each of these benchmarks in response to the various techniques.

Barnes: There are phases where `Barnes` has 80% of its blocks dirty. Because of this high number of dirty blocks, higher values of M produce the most reduction in writebacks. Larger M would mean a better chance of finding a clean block. However, a clean block might be closer to MRU position and hence it can create read

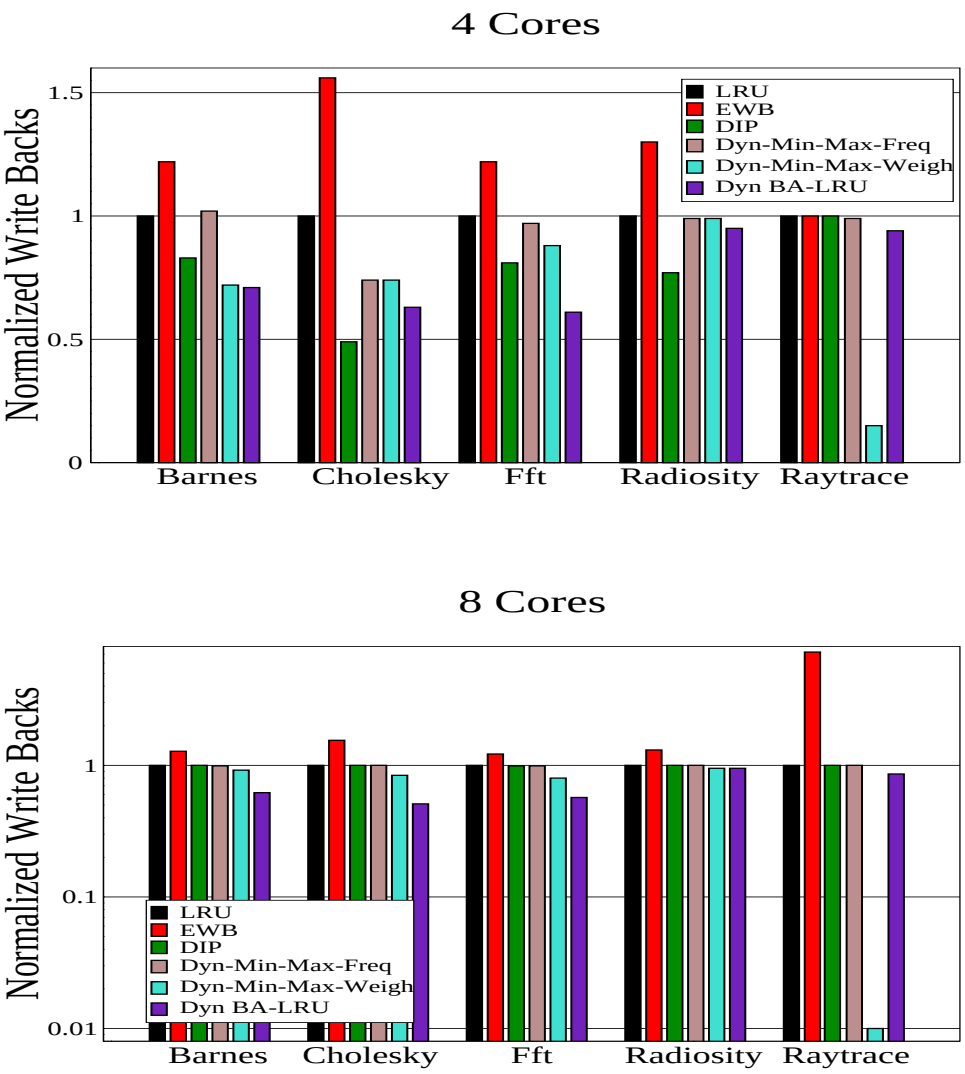

FIGURE 6.11: Normalized Number of Writebacks For 4 Cores (Top) and 8 Cores (Bottom)(Compiler Support and Dynamic Schemes)

misses and performance loss. The balance between writebacks and read misses is maintained for M=4. Dynamic techniques also show a negative effect on read misses because M adjusts to a higher value.

Cholesky: Dirty blocks for `Cholesky` are mostly between 65% to 100% during the entire execution. Since alot of fluctuation is seen in the number of dirty blocks over time, the dynamic techniques work better than the static since they try to capture program behavior.

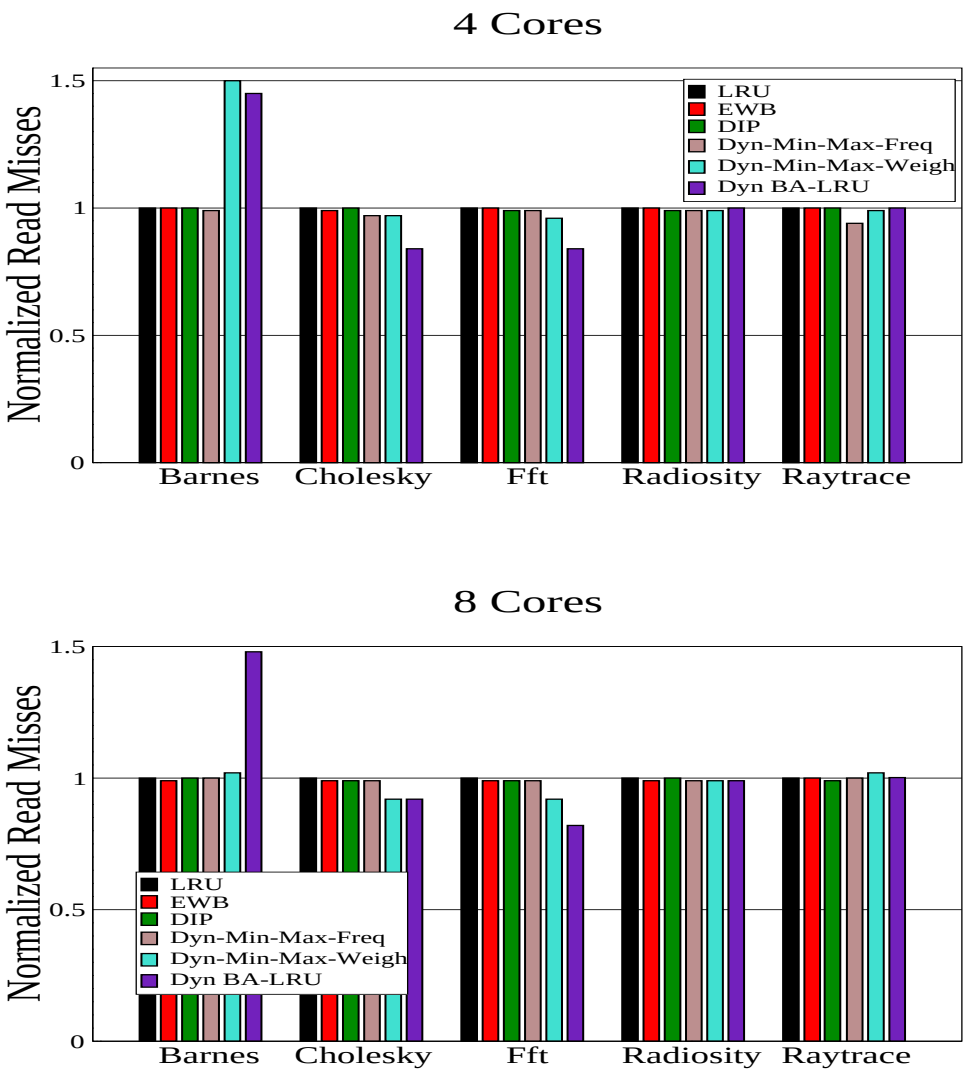

FIGURE 6.12: Normalized Number of Read Misses For 4 Cores (Top) and 8 Cores (Bottom)(Compiler Support and Dynamic Schemes)

Fft: Similarly for `Fft`, the percentage of dirty blocks varies during the entire program execution. During the first half of the execution time, less than 50% of the total blocks are dirty. In the second half, more than 50% of the blocks are dirty. Thus for `Fft`, the dynamic techniques work better than static as they adjust the value of M according to program behavior at run time.

Radiosity: In `Radiosity`, almost all the blocks are dirty all the time. Thus, the dynamic scheme will show no improvement over the static as there is not much

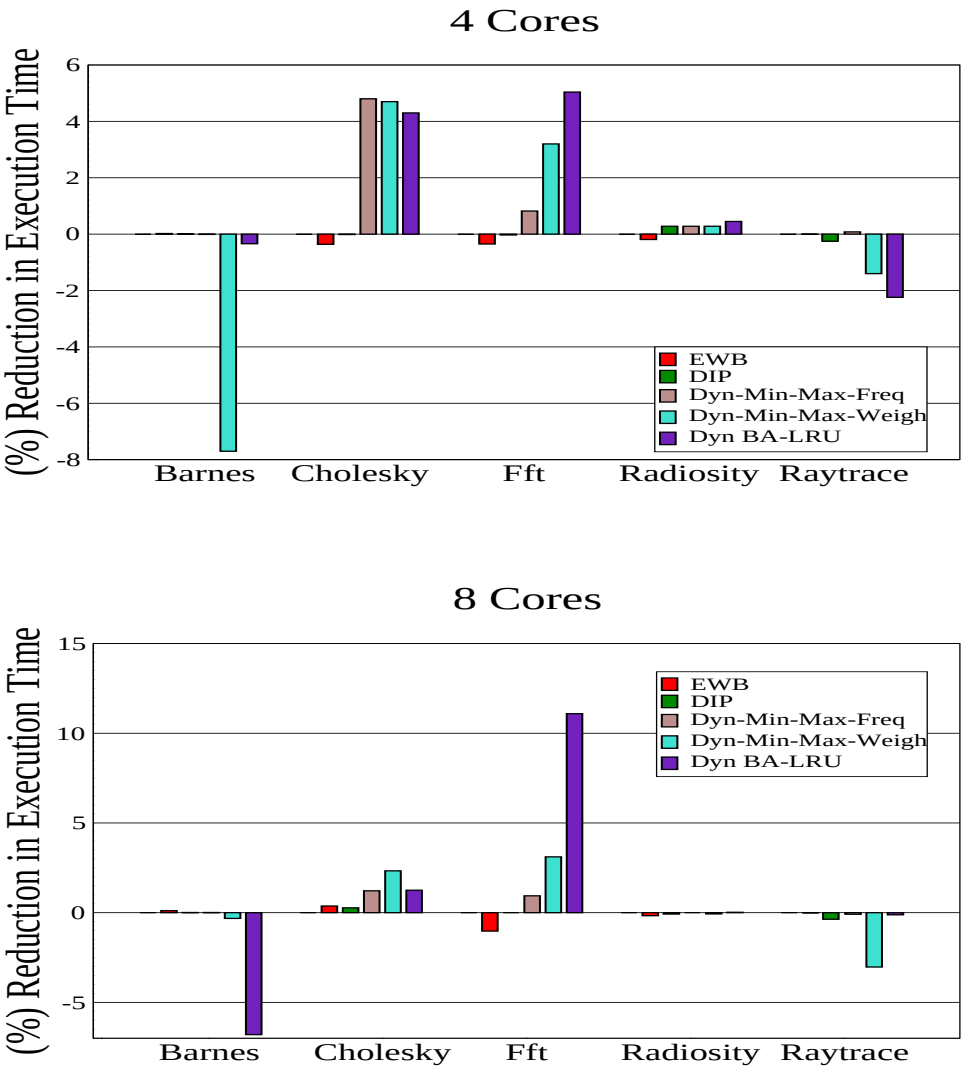

FIGURE 6.13: Speedup For 4 Cores (Top) and 8 Cores (Bottom)(Compiler Support and Dynamic Schemes)

chance of finding a clean block. In the absence of a clean block, the LRU is victimized. Hence the performance of Radiosity stays close to that of regular LRU.

Raytrace: In `Raytrace`, there are almost no dirty blocks during the entire execution. Thus, the static technique almost always find a clean block and the writebacks are reduced to zero. Dynamic technique however, negatively affects performance.

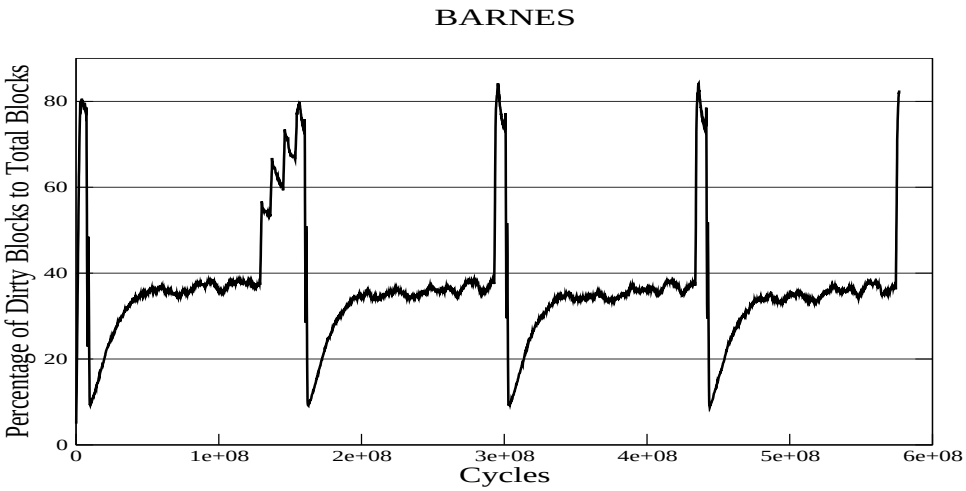

FIGURE 6.14: Percentage of Dirty Blocks to Total Blocks for Barnes

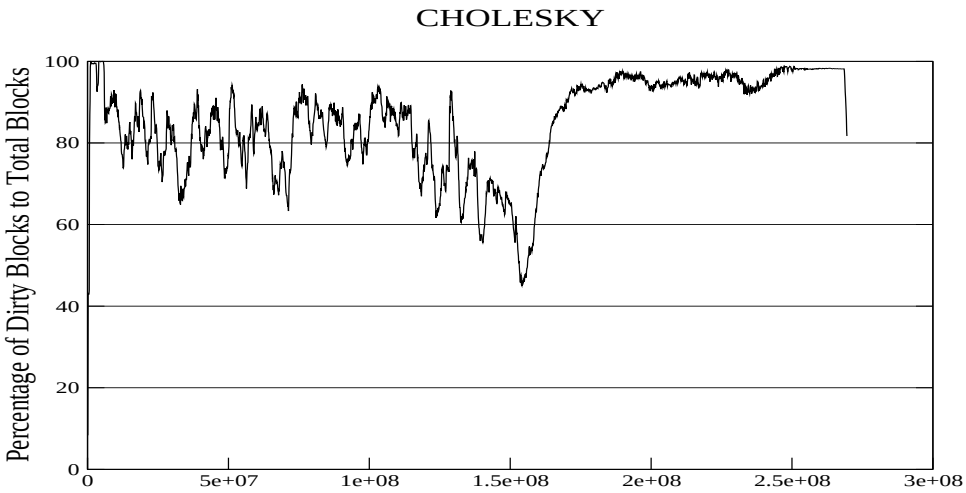

FIGURE 6.15: Percentage of Dirty Blocks to Total Blocks for Cholesky

We have repeated all the experiments with LLC cache of size 2MB and the results are very similar to the ones mentioned here. We cannot experiment with more than 2MB of size for LLC because the size of the benchmark's working set will fit in bigger sizes.

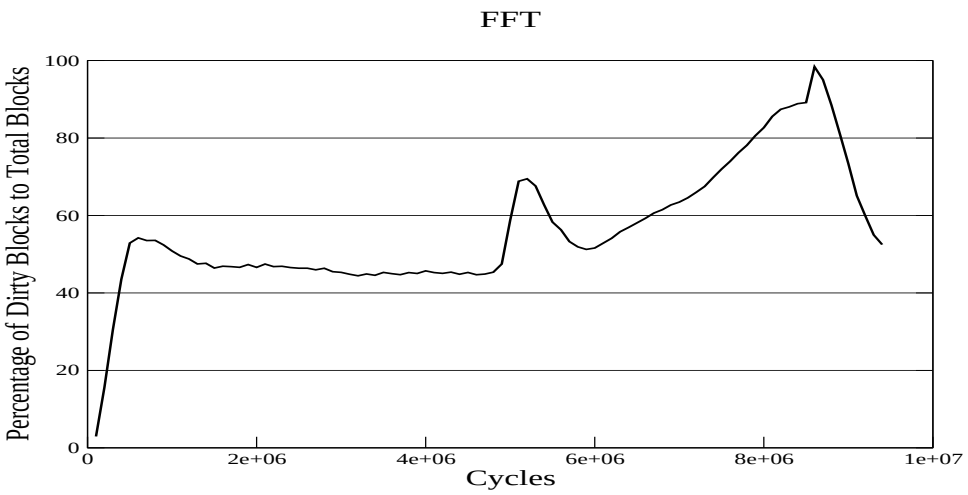

FIGURE 6.16: Percentage of Dirty Blocks to Total Blocks for Fft

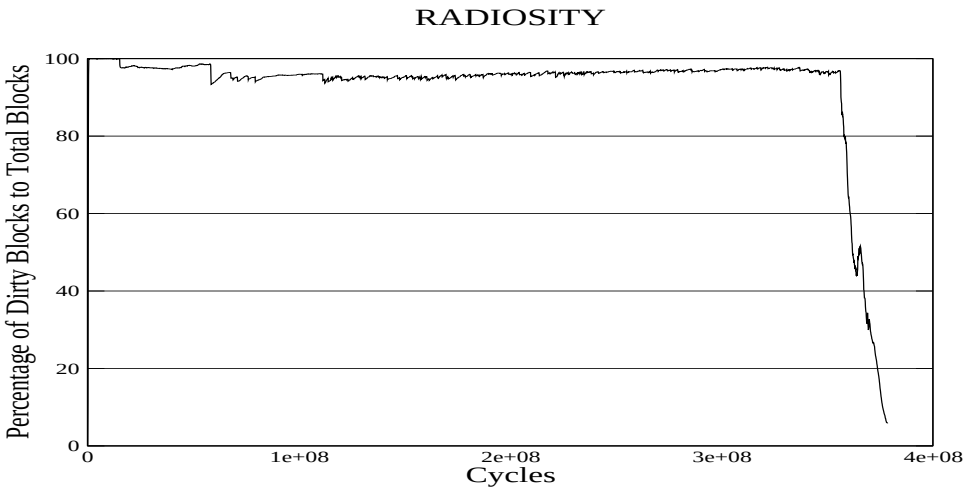

FIGURE 6.17: Percentage of Dirty Blocks to Total Blocks for Radiosity

6.3 Summary

This chapter showed the results of various techniques presented in Chapter 4 and Chapter 5. The following observations were drawn:

- Static techniques reduce the off-chip traffic i-e the writebacks considerably.

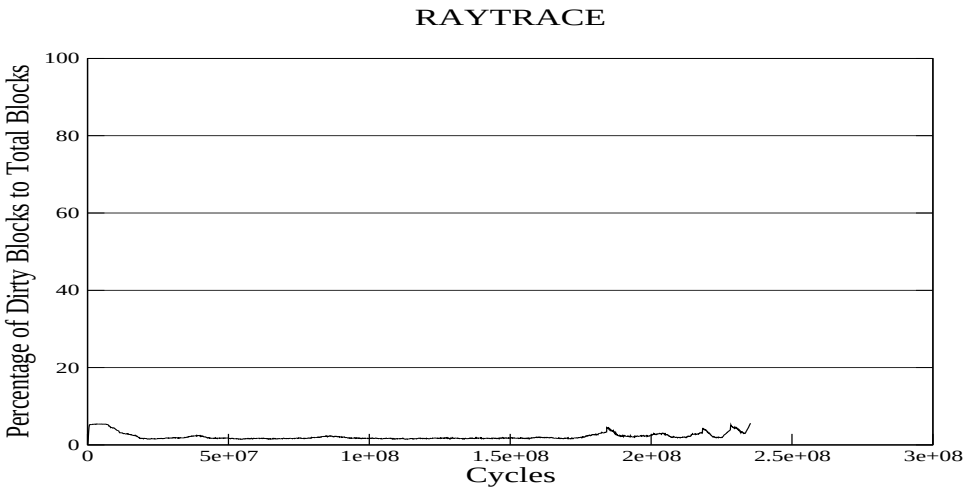

FIGURE 6.18: Percentage of Dirty Blocks to Total Blocks for Raytrace

- Static techniques are highly application dependent and might or might not work well for some benchmarks. The best results are shown for applications that are not LRU friendly and have a large number of writebacks.
- To solve this issue, the dynamic techniques are presented that adapt to the program behavior at run time.
- Dynamic techniques have the advantage of employing bandwidth management only when the application can benefit from it.
- On average the *Min-Max weighted-average technique* shows the best results.
- Our techniques show an average decrease in writebacks of 39%.
- A decrease in read misses of 4% was achieved on average .
- The techniques exploit LRU replacement policy but do not show any negative effect on performance.
- The techniques show better results as the number of cores increase. This shows a good sign of scalability of the techniques.

Chapter 7

Conclusion and Future Directions

"All's well that ends well."

Shakespear

This chapter gives a summary of the work presented in this thesis as well as future directions for bandwidth management in multicores. The bottleneck of bandwidth management has just been identified and further research has to be done to overcome this wall in order to exploit the full potential of many cores on chip.

7.1 Summary

In this thesis we made the following contributions:

- We argued 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.

- We propose hardware and hybrid techniques to deal with this problem, with several variations of each technique.
- The techniques exploit the fact that the current replacement policy i-e, LRU is not always the best and hence we can violate the LRU policy in favor of less off-chip bandwidth.
- Our first proposed technique is a simple hardware technique called Bandwidth-Aware LRU that alters the currently used replacement policy in a way to reduce off-chip traffic. No software or compiler support is needed for this technique. The simulations show a decrease in off-chip traffic with no negative effect on performance. Several variations of BA-LRU are shown.
- Reducing off-chip traffic is highly application dependent and hence the techniques should be alterable according to the traffic 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. Based on the traffic pattern and information collected through profiling, the best replacement policy is chosen for the Last Level Cache.
- We can trade some cache performance loss 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.

7.2 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.
- Tenchiques to manage on-chip bandwidth among caches.

Appendix A

Simulator Details

A.1 Simulator Background

SESC (SuperESCalar Simulator) is a microprocessor architectural simulator that models different processor architectures such as single processors and chip multiprocessors. It models a full out-of-order pipeline with branch prediction, caches, buses, and every other component of a modern processor necessary for accurate simulation. All of our experiments were done on SESC simulator.

SESC is an event-driven simulator. Many functions in the core of the simulator are called every processor cycle. But many others are called only as needed, using events.

SESC is written in C++ and is developed by Jose Renau at the University of California at Santa Cruz and i-acoma group at the UIUC.

For our research we use one of the configuration files distributed with the simulator and alter it according to our work. The file cmp.conf is shown. It configures the entire architecture of a 4 core multiprocessor.

A.2 Technical Details

Some of the technical details of the SESC simulator are explained.

A.2.1 Pipelining

In SESC, the pipeline is modeled through the generic processor object type (GProcessor) that coordinates interactions between the different pipeline stages. The loop of the simulator goes through the following steps:

Clock: Function advances each stage in the pipeline one clock cycle(advanceClock()).
Issue: Calling a function to fetch instructions into the instruction queue(fetch()).
Execute: There are two clusters that schedule and execute instructions, one for integer and one for floating-point instructions(executePC()).
Retire: a function is called to retire already-executed instructions from the reorder buffer(retire()). The class modeling of the pipeline is given in Figure A.1.

In our work we mostly deal with the memory hierarchy of the processor.

A.2.2 Memory Hierarchy

SESC models different cache:

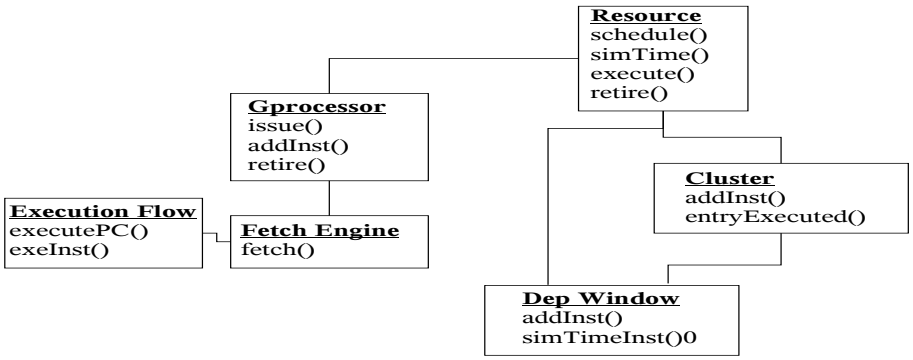

FIGURE A.1: The class interactions that model the pipeline

- Sizes
- Hit and Miss latencies
- Replacement policies
- Cache-line sizes
- Associativities

The Cache implementation in SESC is quite complex and uses many event-driven callbacks. This models all the latencies and transactions involved in caches. The most important parts of the Cache implementation are as follows:

All types of Caches and Buses inherit from a common class, MemObj.

Each GProcessor has a MemorySystem object. The MemorySystem object creates the hierarchy of caches and serves as an adapter between the GProcessor and the highest-level Caches.

GProcessor creates a MemoryRequest object when it needs to interact with the MemorySystem. There are DMemoryRequest objects for data and IMemoryRequests for instructions. Fields in DMemoryRequests can indicate if an access is a read or a write. MemoryRequests are created through the singleton DMemRequest::create() or IMemRequest::create() functions. These functions take as parameters the DInst object the request is associated with, the MemorySystem object this request is going through, and whether the operation is read or write.

A.3 Configuration File

```
### contributed by Hou Rui ###


procsPerNode = 4
L1cacheLineSize = 64
L2cacheLineSize = 64
issue = 4 # processor issue width
cpucore[0:$(procsPerNode)-1] = 'issueX'


#################################
# SYSTEM #
#################################
```

```
NoMigration = true
tech = 0.10
pageSize = 4096
fetchPolicy = 'outorder'
issueWrongPath = true
technology = 'techParam'


#################################
# clock-panalyzer input #
#################################

#techParam#
clockTreeStyle = 1 # 1 for Htree or 2 for balHtree
tech = 70 # nm
frequency = 5e9 # Hz
skewBudget = 20 # in ps
areaOfChip = 200 # in mm2̂
loadInClockNode = 20 # in pF
optimalNumberOfBuffer = 3


#################################
# PROCESSORS' CONFIGURATION #
#################################

#issueX#
frequency = 3e9
areaFactor = ($(issue)*$(issue)+0.1)/16 # Area compared to Alpha264 EV6
```

inorder = false

fetchWidth = $(issue)

instQueueSize = 2*$(issue)

issueWidth = $(issue)

retireWidth = $(issue)+1

decodeDelay = 6

renameDelay = 3

wakeupDelay = 6 # 6+3+6+1+1=17 branch mispred. penalty

maxBranches = 16*$(issue)

bb4Cycle = 1

maxIRequests = 4

interClusterLat = 2

intraClusterLat = 1

cluster[0] = 'FXClusterIssueX'

cluster[1] = 'FPClusterIssueX'

stForwardDelay = 2

maxLoads = 10*$(issue)+16

maxStores = 10*$(issue)$ + 16

$regFileDelay = 3$

$robSize = 36*$(issue)+32

intRegs = 32+16*$(issue)

fpRegs = 32+12*$(issue)

bpred = 'BPredIssueX'

enableICache = true

dtlb = 'FXDTLB'

itlb = 'FXITLB'

dataSource = "DMemory DL1"

instrSource = ”IMemory IL1”

OSType = ’dummy’

integer functional units###

#FXClusterIssueX#

winSize = 12*$(Issue)+32 # number of entries in window

recycleAt = ’Execute’

schedNumPorts = 4

schedPortOccp = 1

wakeUpNumPorts= 4

wakeUpPortOccp= 1

wakeupDelay = 3

schedDelay = 1 # Minimum latency like a intraClusterLat

iStoreLat = 1

iStoreUnit = ’LDSTIssueX’

iLoadLat = 1

iLoadUnit = ’LDSTIssueX’

iALULat = 1

iALUUnit = ’ALUIssueX’

iBJLat = 1

iBJUnit = ’ALUIssueX’

iDivLat = 12

iDivUnit = ’ALUIssueX’

iMultLat = 4

iMultUnit = ’ALUIssueX’

#LDSTIssueX#

```
Num = $(issue)/3+1
Occ = 1

#ALUIssueX# Num = $(issue)/3+1
Occ = 1

### floating point functional units###
#FPClusterIssueX#
winSize = 8*$(issue)
recycleAt = 'Execute'
schedNumPorts = 4
schedPortOccp = 1
wakeUpNumPorts= 4
wakeUpPortOccp= 1
wakeupDelay = 3
schedDelay = 1 # Minimum latency like a intraClusterLat
fpALULat = 1
fpALUUnit = 'FPIssueX'
fpMultLat = 2
fpMultUnit = 'FPIssueX'
fpDivLat = 10
fpDivUnit = 'FPIssueX'

#FPIssueX#
Num = $(issue)/2+1
```

Occ = 1

branch prediction mechanism###

#BPredIssueX#

type = "hybrid"

BTACDelay = 0

l1size = 1

l2size = 16*1024

l2Bits = 1

historySize = 11

Metasize = 16*1024

MetaBits = 2

localSize = 16*1024

localBits = 2

btbSize = 2048

btbBsize = 1

btbAssoc = 2

btbReplPolicy = 'LRU'

btbHistory = 0

rasSize = 32

memory translation mechanism###

#FXDTLB#

size = 64*8

assoc = 4

bsize = 8

```
numPorts = 2
replPolicy = 'LRU'
deviceType = 'cache'


#FXITLB#
size = 64*8
assoc = 4
bsize = 8
numPorts = 2
replPolicy = 'LRU'
deviceType = 'cache'


#################################
# MEMORY SUBSYSTEM #
#################################

### instruction source###
#IMemory#
deviceType = 'icache'
size = 32*1024
assoc = 2
bsize = $(L1cacheLineSize)
writePolicy = 'WT'
replPolicy = 'LRU'
numPorts = 2
portOccp = 1
hitDelay = 1
```

missDelay = 1 # this number is added to the hitDelay

MSHR = "iMSHR"

lowerLevel = "L1L2Bus L1L2 shared"

#iMSHR#

type = 'single'

size = 32

bsize = $(L1cacheLineSize)

data source###

#DMemory#

deviceType = 'smpcache'

size = 32*1024

assoc = 2

bsize = $(L1cacheLineSize)

writePolicy = 'WB'

replPolicy = 'LRU'

protocol = 'MESI'

numPorts = 2 # one for L1, one for snooping

portOccp = 2

hitDelay = 2

missDelay = 2 # exclusive, i.e., not added to hitDelay

displNotify = false

MSHR = "DMSHR"

lowerLevel ="L1L2DBus L1L2D shared"

```
maxWrites = 8


#DMSHR#
type = 'single'
size = 64
bsize = $(L1cacheLineSize)


### bus between L1s and L2###
#L1L2DBus#
deviceType = 'systembus'
numPorts = 1
portOccp = 1 # assuming 256 bit bus
delay = 1
lowerLevel = "L2Cache L2"
BusEnergy = 0.03 # nJ
#lowerLevel = "MemoryBus MemoryBus"


#L1L2Bus#
deviceType = 'bus'
numPorts = 1
portOccp = 1 # assuming 256 bit bus
delay = 1
#lowerLevel = "MemoryBus MemoryBus"
lowerLevel = "L2Cache L2"
```

```
### shared L2###

#L2Cache#

deviceType = 'cache'

evalPeriod = 10000

BFWritesPeriod = 10000

minBandwidth = 0.000285851 #0.000285851 for fft

maxBandwidth = 0.000254963 #0.000254963 for fft

inclusive = false

size = 1024*1024

assoc = 8

bsize = $(L2cacheLineSize)

writePolicy = 'WB'

replPolicy = 'MLRU'

numPorts = 2 # one for L1, one for snooping

portOccp = 2

hitDelay = 9

missDelay = 11 # exclusive, i.e., not added to hitDelay

displNotify = false

MSHR = 'L2MSHR'

lowerLevel = "SystemBus SysBus "


#L2MSHR#

size = 64

type = 'single'

bsize = $(L2cacheLineSize)
```

```
#SystemBus#
deviceType = 'bus'
numPorts = 1
portOccp = 1
delay = 1
lowerLevel = "MemoryBus MemoryBus"


#MemoryBus#
deviceType = 'jjbus'
savePeriod = 2500
numPorts = 1
portOccp = $(L2cacheLineSize) / 4 # assuming 4 bytes/cycle bw
delay = 15
lowerLevel = "Memory Memory"


#Memory#
deviceType = 'niceCache'
size = 64
assoc = 1
bsize = 64
writePolicy = 'WB'
replPolicy = 'LRU'
numPorts = 1
portOccp = 1
hitDelay = 500 - 31 # 5.0GHz: 100ns is 500 cycles RTT - 16 busData
missDelay = 500 - 31 # - 15 memory bus =¿ 500 - 31
```

```
MSHR = NoMSHR
lowerLevel = 'voidDevice'


#NoMSHR#
type = 'none'
size = 128
bsize = 64


#voidDevice#
deviceType = 'void'
```

Bibliography

[1] Ethan Mollick. Establishing moore's law. *IEEE Annals of the History of Computing*, 28(3):62–75, 2006. ISSN 1058-6180.

[2] PawełGepner, David L. Fraser, and Michał F. Kowalik. Second generation quad-core intel xeon processors bring 45 nm technology and a new level of performance to hpc applications. In *ICCS '08: Proceedings of the 8th international conference on Computational Science, Part I*, pages 417–426, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-69383-3.

[3] P. Conway and B. Hughes. The amd opteron northbridge architecture. *Micro, IEEE*, 27(2):10–21, March-April 2007. ISSN 0272-1732.

[4] Thomas Ziaja and P. J. Tan. Efficient array characterization in the ultrasparc t2. In *VTS '09: Proceedings of the 2009 27th IEEE VLSI Test Symposium*, pages 3–8, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3598-2.

[5] Jim Kahle. The cell processor architecture. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, page 3, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2440-0.

[6] Krste Asanovic, Ras Bodik, Bryan C. Catanzaro, Joseph J. Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William L. Plishker, John Shalf, Samuel W. Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical report, Electrical Engineering and Computer Sciences, University of California at Berkeley, 2006. URL http://techreports.lib.berkeley.edu/accessPages/EECS-2006-183.html.

[7] 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.

[8] S.K. Moore. Multicore is bad news for supercomputers. *Spectrum, IEEE*, 45 (11):15–15, November 2008. ISSN 0018-9235.

[9] Doug Burger, James R. Goodman, and Alain Kägi. Memory bandwidth limitations of future microprocessors. In *ISCA '96: Proceedings of the 23rd annual international symposium on Computer architecture*, pages 78–89, New York, NY, USA, 1996. ACM. ISBN 0-89791-786-3.

[10] Jaehyuk Huh, Doug Burger, and Stephen W. Keckler. Exploring the design space of future cmps. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 199–210, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1363-8.

[11] Brian M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin. Scaling the bandwidth wall: challenges in and avenues for cmp scaling. *SIGARCH Comput. Archit. News*, 37(3):371–382, 2009. ISSN 0163-5964.

[12] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[13] Richard Murphy. On the effects of memory latency and bandwidth on supercomputer application performance. In *IISWC '07: Proceedings of the 2007 IEEE 10th International Symposium on Workload Characterization*, pages 35–43, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 978-1-4244-1561-8.

[14] 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.

[15] 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.

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

[17] J-K Peir, Y. Lee, and W Hsu. Capturing dynamic memory reference behavior with adaptive cache toplogy. In *Proc. International Conf. on Architecture Support for Programming Languages and Operating System (ASPLOS)*, 1998.

[18] J. Kin, M. Gupta, and W. H. Mangione-Smith. The filter cache: An energy efficient memory structure. In *Proc. 30th Annual International Symposium on Microarchitecture (MICRO-30)*, 1997.

[19] A. V. Veidenbaum, W. Tang, R. Gupta, A. Nicolau, and X. Ji. Adapting cache line size to application behavior. In *Proc. of the 1999 international conference on Supercomputing*, 1999.

[20] D. H. Albonesi. Selective cache ways: On-demand cache resource allocation. *Journal of Instruction-Level Parallelism*, 2002.

[21] B. Calder, D. Grunwald, and J. Emer. Predictive sequential associative cache. In *Proc. 2nd International Symposium on High Performance Computer Architecture*, 1996.

[22] A. Agarwal and S. D. Pudar. Column-associative cache: A technique for reducing the miss rate of direct-mapped caches. In *Proc. 20th International Symposium on Computer Architecture*, pages 179–190, 1993.

[23] J-K Peir, W Hsu, H. Young, and S. Ong. Improving cache performance with balanced tag and data paths. In *Proc. International Conf. on Architecture Support for Programming Languages and Operating System (ASPLOS)*, 1996.

[24] 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.*, pages 297–306, 1993.

[25] 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.

[26] 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*, pages 267–272, 2004.

[27] 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*, pages 72–79, 2008.

[28] Hsien-Hsin S. Lee, Gary S. Tyson, and Matthew K. Farrens. Eager writeback - a technique for improving bandwidth utilization. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 11–21, New York, NY, USA, 2000. ACM. ISBN 1-58113-196-8.

[29] James Archibald and Jean-Loup Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, May 1986.

[30] Anant Agarwal and et al. An evaluation of directory schemes for cache coherence. In *25 Years ISCA: Retrospectives and Reprints*, 1988.

[31] S. J. Eggers and R. H. Katz. Evaluating the performance of four snooping cache coherence protocols. In *Proceeding of the 22nd Annual Int'l Symposium on Computer Architecture*, pages 2–15, 1989.

[32] Milo Tomasevic and Veljko Milutinovic. *The Cache Coherence Problem in Shared-Memory Multiprocessors: Hardware Solutions*. IEEE Computer Society Press, 1993.

[33] L. Hammond, B. Nayfeh, and K. Olukotun. A single-chip multiprocessor. *IEEE Computer*, 1997.

[34] Basem A. Nayfeh. *The Case for a Single-Chip Multiprocessor*. PhD thesis, Stanford University, 1998.

[35] M. Zahran and M. Franklin. A feasibility study of hierarchical multithreading. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2002.

[36] B. Beckmann and D. Wood. Managing wire delay in large chip-multiprocessor caches. In *Proc. 37th Int'l Annual Symp. on Microarchitecture (Micro-37)*, December 2004.

[37] J. Chang and G. Sohi. Cooperative caching for chip multiprocessors. In *Proc. 20th Int'l Symposium on Computer Architecture (ISCA)*, June 2006.

[38] A. Agarwal and et al. Evaluation the raw microprocessor: An exposed-wire-delay architecture for ilp and streams. In *Proc. 32nd Int'l Symposium on Computer Architecture (ISCA)*, 2004.

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

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

[41] G. De Micheli T. T. Ye. Physical planning for on-chip multiprocessor networks and switch fabrics. In *14th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP'03)*, 2003.

[42] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 111–122, 2004.

[43] 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.

[44] 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.

[45] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *SIGCOMM '89: Symposium proceedings on Communications architectures & protocols*, pages 1–12, New York, NY, USA, 1989. ACM Press. ISBN 0897913329. URL http://dx.doi.org/10.1145/75246.75248.

[46] Pawan Goyal, Harrick M. Vin, and Haichen Cheng. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. *IEEE/ACM Trans. Netw.*, 5(5):690–704, 1997. ISSN 1063-6692.

[47] Abhay K. Parekh and Robert G. Gallager. A generalized processor sharing approach to flow control in integrated services networks—the single node case. In *IEEE INFOCOM '92: Proceedings of the eleventh annual joint conference of the IEEE computer and communications societies on One world through communications (Vol. 2)*, pages 915–924, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press. ISBN 0-7803-0602-3.

[48] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 111–122, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2229-7.

[49] Nauman Rafique, Won-Taek Lim, and Mithuna Thottethodi. Architectural support for operating system-driven cmp cache management. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 2–12, New York, NY, USA, 2006. ACM. ISBN 1-59593-264-X.

[50] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *J. Supercomput.*, 28(1):7–26, 2004. ISSN 0920-8542.

[51] Nauman Rafique, Won-Taek Lim, and Mithuna Thottethodi. Effective management of dram bandwidth in multicore processors. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 245–258, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2944-5.

[52] Ravi Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell, Yan Solihin, Lisa Hsu, and Steve Reinhardt. Qos policies and architecture for cache/memory in cmp platforms. In *SIGMETRICS '07: Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 25–36, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-639-4.

[53] Chitra Natarajan, Bruce Christenson, and Fayé Briggs. A study of performance impact of memory controller features in multi-processor server environment. In *WMPI '04: Proceedings of the 3rd workshop on Memory performance issues*, pages 80–87, New York, NY, USA, 2004. ACM. ISBN 1-59593-040-X.

[54] Ibrahim Hur and Calvin Lin. Adaptive history-based memory schedulers. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 343–354, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2126-6.

[55] Sally A. McKee, Assaji Aluwihare, Benjamin H. Clark, Robert H. Klenke, Trevor C. Landon, Christopher W. Oliver, Maximo H. Salinas, Adam E. Szymkowiak, Kenneth L. Wright, Wm. A. Wulf, and James H. Aylor. Design and evaluation of dynamic access ordering hardware. In *ICS '96: Proceedings of the 10th international conference on Supercomputing*, pages 125–132, New York, NY, USA, 1996. ACM. ISBN 0-89791-803-7.

[56] Scott Rixner. Memory controller optimizations for web servers. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 355–366, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2126-6.

[57] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory access scheduling. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 128–138, New York, NY, USA, 2000. ACM. ISBN 1-58113-232-8.

[58] 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. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.17.2471.

[59] 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.

[60] 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.

[61] T. Chilimbi, M. Hill, and J.Larus. Cache conscious structure layout. In *Proc. of Int'l Conference on Programming Language Design and Implementation (PLDI)*, 1999.

[62] R. Sethi A. Aho and J. Ullman. *Compilers: Principles, Techniques and Tools.* Addition-Wesley, 1986.

[63] J. A. Butts and G. Sohi. Dynamic dead-instruction detection and elimination. In *Proc. Int'l Conf. on Architecture Support for Programming Languages and Operating System (ASPLOS)*, 2002.

[64] M. M. Martin, A. Roth, and C. N. Fischer. Exploiting dead value information. In *Proc. of International Symposium on Microarchitecture (MICRO-30)*, 1997.

[65] A-C Lai, C. Fide, and B. Falsafi. Dead-block prediction and dead-block correlating prefetchers. In *Proc. Int'l Symposium on Computer Architecture (ISCA) 24*, 2001.

[66] 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.

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

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

[69] 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.

[70] 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.

[71] José Renau. SESC. `http://sesc.sourceforge.net/index.html`, 2002.

[72] T. Sherwood and G. B. Calder. Time varying behavior of programs. Technical report, University of California at San Diego, 1999.

[73] H.S. Lee, G.S. Tyson, and M.K. Farrens. Eager writeback – a technique for improving bandwidth utilization. In *Proc. IEEE/ACM 33nd International Symposium on Microarchitecture*, pages 11–21, December 2000.

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