A MULTIPLE QUEUE REPLACEMENT EXPLOITING FREQUENCY

FOR LOW LEVEL CACHES

A DISSERTATION
in
Electrical Engineering
and
Computer Science

Presented to the Faculty of the University
of Missouri–Kansas City in partial fulfillment of
the requirements for the degree

DOCTOR OF PHILOSOPHY

by
BING XUE

M. S., Florida International University, 2005
B. S., Beijing Jiaotong University, China, 1998
B. S., Shenyang University, China, 1996

Kansas City, Missouri
2011

A MULTIPLE QUEUE REPLACEMENT EXPLOITING FREQUENCY

FOR LOW LEVEL CACHES

Bing Xue, Candidate for the Doctor of Philosophy Degree

University of Missouri–Kansas City, 2011

ABSTRACT

Cache replacements play a vital role in cache misses and miss penalty reductions. We illustrate Cache replacements including Least Recently Used (LRU), Least Frequently Used (LFU), and combinations of both, by discussing how these replacements were used to exploit the recency and frequency information. We compare a series of studies by which we and other researchers have proposed the effectiveness of these replacements. This dissertation discusses the results from these earlier studies, proposes a Frequency based Multiple Queue (FMQ) replacement that is designed to improve potential performance to our Frequency based Single Queue (FSQ) replacement, and, in the process, sheds some light on those unsolved problems of the FSQ replacement from our earlier studies. The FMQ exploits frequency for low level caches (LLC) with low hardware overhead and minimum design changes. Cache

lines in each cache set are arranged in multiple queues for replacement decisions. An incoming line randomly enters one of the queues from the bottom instead of placing the incoming line at the top of the queue. On a cache miss, one queue is randomly selected from the multiple queues and the victim line is chosen from either the bottom or the top of the queue. On a cache hit, the hit cache line exchanges its position with the cache line above in the queue, and moves its way to the top of the queue. The frequency information is stored through the location of the cache lines in the queue. On average, the FMQ achieves a 12% Miss Per Kilo-Instruction (MPKI) reduction over a conventional 1MB 16-way unified L2 cache. FMQ simplifies the circuit design compared to the traditional LRU. The storage requirement for the FMQ is reduced to 48%.

APPROVAL PAGE

The faculty listed below, appointed by the Dean of the School of Computing and Engineering, have examined a dissertation titled “A Multiple Queue Replacement Exploiting Frequency for Low Level Caches” presented by Bing Xue, candidate for the Doctor of Philosophy degree, and hereby certify that in their opinion it is worthy of acceptance.

Supervisory Committee

Chuanjun Zhang, Ph.D., Committee Chair
Department of Computer Science Electrical Engineering

Ghulam Chaudhry, Ph.D.
Department of Computer Science Electrical Engineering

Xiaojun Shen, Ph.D.
Department of Computer Science Electrical Engineering

Cory Beard, Ph.D.
Department of Computer Science Electrical Engineering

Baek-Young Choi, Ph.D.
Department of Computer Science Electrical Engineering

CONTENT

LIST OF ILLUSTRATIONS

LIST OF TABLES

ACKNOWLEDGEMENTS

As my Ph.D. study is ending, I extend my gratitude to the people who have been helping me go through the most important and difficult period of my life.

First, I would like to express my deepest gratitude to my advisor, Dr. Zhang. Dr. Zhang is an excellent mentor in teaching me the ways to analyze complex problems, parse experimental results and present academic ideas. My professional and personal life has greatly benefited from Dr. Zhang's encouragement and support. I would never have been able to finish my dissertation without the guidance and help of Dr. Zhang.

I would also like to thank all the members of my advisory committee, including Dr. Ghulam Chaudhry, Dr. Xiaojun Shen, Dr. Cory Beard, Dr. Baek-Young Choi, for their willingness to accept my invitation to serve on my committee, and to guide my research for the past several years. They have generously devoted their time to my project and provided insightful suggestions during the process of my research and dissertation writing. Dr. Xiaojun Shen helped me to develop my background in algorithms.

Finally, I would like to thank my family. My wife Ying is always supporting and encouraging me with her best wishes. I thank my two wonderful children Iris and Skyler for their patience with me.

CHAPTER 1

INTRODUCTION

1.1 Problem Definition

As the advancements in semiconductor fabrication technology continue to outpace the improvement in memory performance, application programs increasingly spend more time waiting for memory [103]. The performance of processors grows 60% every year, while the performance of DRAM grows 7% every year. The performance gap between processors and memory grows over 50% every year. So the performance of computer systems has not gained as much from continued processor frequency increases. For example, an experimental study conducted on Itanium 1 showed that almost 50% of the execution time of the SPEC2K CPU [95] benchmark runs was caused by cache memory.

Cache replacement policies can be designed to reduce the total cache misses and the time spent on accessing the next level memory. The Least Recently Used (LRU) does not exploit frequency information of cache accesses. LRU may experience cache thrashing when workloads with differing working sets compete for a shared cache. Many variants of the LRU replacement, including (but not limited to) complicated cache replacements that subsume the LRU and LFU [60], modified LRU

[101], EELRU [83] and cost sensitive replacements [43], have been presented in academic conferences and publications. The LRU-based replacement policies throw out the cache block that has not been used for the longest period. In order to track which cache block was accessed and when to evict the least recently used cache block, these replacement algorithms must measure cache block access distance or tune parameters by extra structure. Some proposals use hybrid schemes that dynamically select the best one from two or more separate replacement policies [65] [77] [78]. Hybrid schemes need to keep track of the information from all competing replacement policies. So they require extra structures and consume more power. Although these replacements may work for general applications, there is still a significant performance gap to the Belay's Optimal Replacement (OPT) algorithm [8].

An ideal cache replacement policy has these features:

First, an ideal replacement policy should exploit both temporal locality and spatial locality. It can track and use both recency and frequency information of cache accesses.

Second, an ideal replacement policy can bring a better performance than LRU and LFU for different kinds of workloads. This policy should reduce the performance gap between traditional replacement policies and Belay's OPT algorithm as maximal as possible.

Finally, an ideal replacement policy should require minimum extra hardware cost and power consumption compared with other current hybrid schemes for low level caches.

1.2 Contributions of This Dissertation

We propose a Frequency based Multiple Queue (FMQ) replacement that exploits both the recency and frequency information of the cache lines. The FMQ replacement not only outperforms the LRU in cache misses reduction, but also significantly reduces storage requirements and circuit complexity for hardware implementation compared to the LRU. The major contributions of this dissertation are as follows:

1. Multiple Queue Division - The n cache lines in each cache set are divided into N (2<=N<=n/2, where n presents n-way cache) queues, where each queue implements independently the same insertion, promotion, and victim selection. For example, in the baseline 16-way cache, the 16 cache lines in the cache are divided into four separate queues.

2. Insertion Policy - An incoming line randomly enters one of the queues from the bottom instead of placing the incoming line at the top of the queue.

3. Promotion Policy - On a cache hit, the hit cache line exchanges its location with the cache line above in the queue and moves towards the top of the queue, while the LRU promotes the hit cache line to the MRU position. For status bit update patterns, FMQ

modified only two lines while the LRU may update all the lines' statuses when the cache hits on the LRU position.

4. Victim Policy - On a cache miss, one queue is randomly selected from the multiple queues and the victim line of the queue is kicked out from either the bottom or the top. But the incoming line is always inserted to the bottom no matter whether the victim line is selected from the bottom of the queue. All the lines in the queue move one position up in the queue to make space for the incoming line if the victim line is selected from the top of the queue.

5. Both the recency and frequency information are exploited in FMQ, since more frequently used lines will be stored on the top of the queues while less frequently used lines have to keep on the lower part of the queue; the most recently used block is moved upward and away from the bottom of queue; therefore, the FSQ replacement exploits both recency and frequency information.

6. Our method does not require any counters that are widely used in cache replacements that exploit frequency information. The frequency information is recorded through the location of the cache lines in the queue.

7. The proposed FMQ replacement improves traditional LRU performance with reduced design complexity. The proposed FMQ achieves an average 12% MPKI reduction over a conventional 1MB 16-way unified L2 cache. The storage

requirement for the FMQ is reduced to 48%, and the updates of the status bits are reduced to 92%.

1.3 Outline of This Dissertation

The rest of this dissertation is organized as follows: Chapter 2 covers background and the review over the previous related work in traditional cache optimizing techniques. It is followed by discussing the experimental methodology and results on two replacements – FSR and FMR that are in Chapter 3 and Chapter 4, respectively. Chapter 5 presents case studies of the SPEC2000 benchmarks. System performance and energy consumption are analyzed in Chapter 6; Chapter 7 describes the related work. Finally, the summary and discussion on future work are included in Chapter 8.

The gist of Chapter 3 and Chapter 4 appeared in our 23rd ACM International Conference on Supercomputing (23rd ICS) paper, while we extend the published work in several aspects:

(1) We categorize the SPEK2000 benchmarks into four groups according to their performance of FSQ and FMQ over the LRU. We provide case studies in each group and analyze the access behaviors of the SPEK2000 benchmarks.

(2) We implement FMQ and LRU replacements in hardware using the Cadence tools. Power, area, and circuit complexity reductions of FMQ over the baseline LRU are discussed.

(3) To investigate the influence of cache configurations on the FMQ, we conduct experiments in different cache configurations for the proposed FMQ. The performance evaluation is extended with the impact of cache capacity, cache line size, associativity, and the number of queues to the proposed FMQ replacement.

(4) A discussion of scalable FMQ and a possible implementation are included.

(5) There are 85 figures and 20 tables in this dissertation. The original paper published in the 23rd ICS has only 16 figures and 4 tables.

These expansions make the evaluation of the proposed FMQ replacement more complete.

CHAPTER 2

BACKGROUND AND PREVIOUS WORK

In this dissertation, a multiple queue replacement exploiting frequency for low level caches is discussed. Before unfolding these discussions, we do a review on traditional optimization techniques. We present our previous work in optimization techniques. Since one of the major contributions of this dissertation is the use of a novel replacement exploiting recency and frequency with low hardware complexity, we also review the use of existing research topics on different cache replacement policy.

2.1 Background and Terminology

We present the background about our research topic. The related terminology in memory hierarchy, temporal locality, and spatial locality is introduced in this subsection.

2.1 .1 Memory Hierarchy

Memory latency is increasing as semiconductor technology advances. To alleviate the memory latency's impact on system performance, memory system is divided up into a hierarchy [84].

Table 1: Cache Parameters in Commercial Processors

Processor Family	Processor Name	Capacity		Line size(Byte)	
		L1	L2	L1	L2
Pentium III	Pentium III (0.25 µm)	16KB/16KB	512 KB	32B/32B	32B
	Xeon (0.25 µm)		512 KB~2 MB		
	Celeron (0.18 µm)		128 KB		
	Pentium III (0.18 µm)		256 KB		
	Xeon (0.18 µm)		256 KB		
	Xeon (0.18 µm) A		1 or 2 MB		
	Celeron (0.13 µm)		256 KB		
	Pentium III (0.13 µm)		512 KB		
Intel Pentium IV	Celeron(0.18 µm)	8KB/8 or 16KB	128 KB	64B/64B	64B
	non-Celeron (0.18 µm)		256 KB		
	Celeron (0.13 µm)		128 KB		
	mobile Celeron (0.13 µm)		256 KB		
	non-Celeron (0.13 µm)		512 KB		
	Celeron (0.09 µm)		256 KB		
	non-Celeron (65 nm) (0.09 µm)		1024 KB		
	non-Celeron (65 nm) (0.09 µm)		2048 KB		
Intel Core	Celeron (low-cost) Core 2 (mobile, desktop, and extreme) Xeon and Xeon MP (server)	32KB/32KB	512 KB	64B/64B	64B
			1.0 MB		
			2.0 MB		
			3.0 MB		
			4.0 MB		
AMD K8	Sempron (Low-End) Athlon 64 (Mid-Range) Athlon 64 FX (High-End) Turion 64 (Mobile) Opteron (Server)	64KB/64KB	128KB~1M	64B/64B	64B
IBM Power PC 750	PowerPC 750CX PowerPC 750CXe PowerPC 750FX PowerPC 750GX PowerPC 750CL	32KB/32KB	256 or 512KB	32B/32B	64B
			1M		128B

Figure 1 shows memory system includes register, cache, main memory, and disk storage. From bottom to top, the access time grows faster while capacity size decreases. For example, the speed for register is faster than 1ns, but the capacity size is less than 1KB.

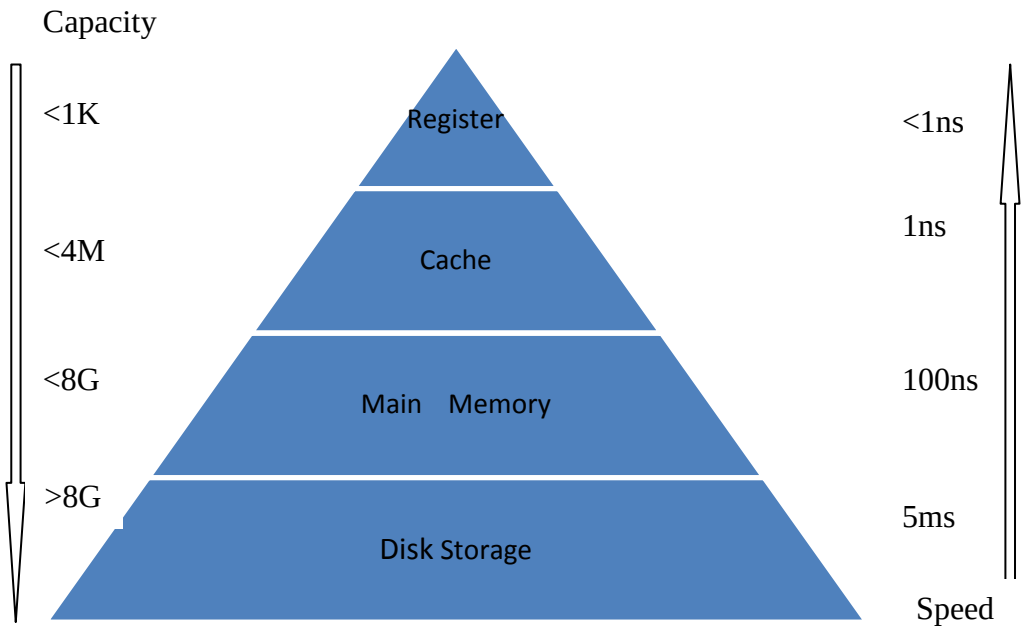


Figure 1: Memory hierarchy

Table 1 shows the capacity and line size for L1 cache and L2 cache in main processor family made in Intel, AMD, and IBM. From table 1, L1 Cache capacity is from 8KB to 64KB while L2 Cache capacity is 128KB to 4MB. The line size for most of processors is 32B or 64B. In a multi level cache hierarchy, first level cache (L1 cache) is for latency and last level cache (LLC) is for capacity. The lower levels of the memory hierarchy tend to be more access time, but larger capacity size. Therefore,

computer system will achieve a better performance if programs access memory while it is cached in the L1 cache. Larger lower levels cache size helps only to the extent memory bandwidth, but it cannot decrease the gap between processor and memory system in computer architecture. The memory wall causes the performance bottleneck in computer system.

2.1.2 Locality of Reference

A program spends 90% of its time in 10% of its program codes. Different behaviours in the codes include memory access patterns, data structure layout, and among others. There are two different types of locality: temporal locality and spatial locality.

Temporal locality: if at one point in time a special memory address is accessed, then it is possible that the same address will be accessed again in a short time.

Spatial locality: if a special memory address is accessed at a special time, then it is possible that nearby memory address will be accessed in a short time.

This is a locality example:

```
total = 0;
for (j = 0; j < m; j++)
        total += s[j];
return total;
```

Data

- Access array elements in succession
- Access sum each iteration

Instructions

- Access instructions in sequence

Substantial research has been conducted in optimizing cache management and exploiting principle of locality. Traditional optimizing techniques that reduce cache misses and effective miss penalties can be generally categorized into the following directions: (1) modified cache organizations; (2) memory level of parallelism; (3) data prefetching; and (4) improved cache replacement.

2.2 Modified Cache Organization

Modified index decoding schemes, including the prime modulo hashing [55], the skewed-associative cache [82], the balanced cache [105], the V-Way cache [76], and the IIC cache [37][38], reduce cache misses through modified cache lookup schemes [96]. For example, the IIC cache uses decoupled tag and data stores with a generational global replacement policy. The balanced cache reduces conflict misses for direct mapped caches using programmable decoders.

Split cache organizations [69], way-predicting set-associative cache [41], predictive sequential associative cache [13], global block priority [7] and cache with various buffers [50][104][109] exploit temporal or spatial localities exhibited in data streams with different internal cache organizations. For example, small fully

associative victim caches [50] have been successfully implemented in several processors in the industry.

Non-uniform cache architecture (NUCA) [23][56][63] migrate frequently used cache data into closer and faster cache locations. Configurable cache architectures [106][108] tune cache parameters to suit particular applications for improved performance and reduced energy dissipation. However, the adaption of the configurable cache is still restricted to limited cache memory parameters.

High associativity is necessary for some workloads. A CAM based Highly Associative Cache (CHAC) has the advantage of low miss rates. But access time and power consumption will increase as the cache size increases because cache hit signal is generated from all blocks. Circuits and wire routing have to be redesigned in order to avoid too much power consumption. A CHAC also depends on efficient replacement policy to achieve low miss rate. Prefetching future information or tracking more status bit by counters is required for these replacement policies. Therefore, a CAM-based Highly Associative Cache is both costly in hardware and exhibits poor scalability.

One of our previous works in modified cache organizations, two dimensional Cache (T-Cache) [110] was presented in ICCD 2008. T-Cache implements the CAM-HAC in macro-blocks to improve scalability. The blocks are divided into rows and columns to exploit new replacements. There are 128-row and 8-column of cache blocks in each macro-block. Random replacement policy is in used in rows to balance

cache sets usage. The random policy only needs the trivial hardware. LRU is used in columns to select the best victim from a row. The total number of counters is 1024×3 bits for the LRU. Figure 2 shows the total hardware complexity of T-cache for replacement policy approaches a traditional eight-way cache.

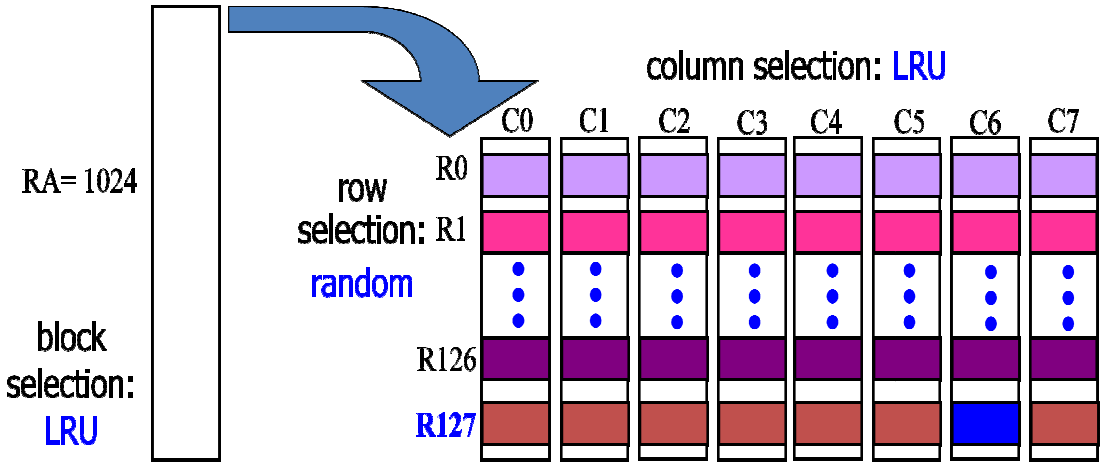
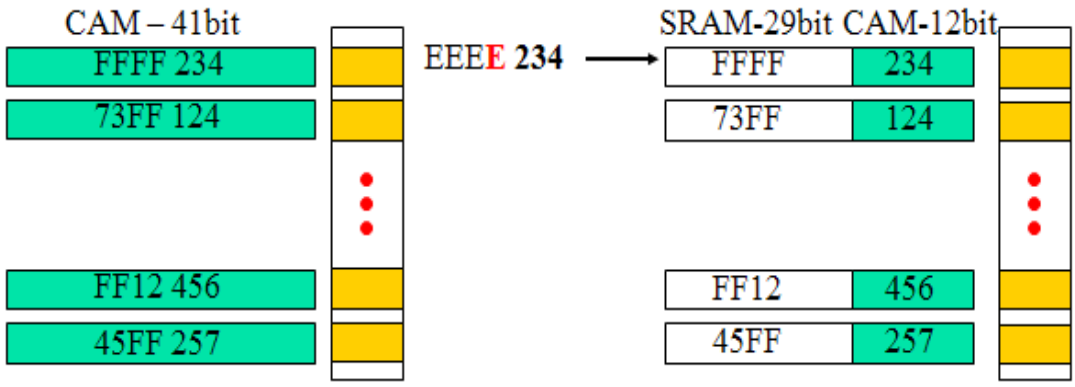


Figure 2: Hardware Complexity of T-Cache



(a) a traditional CHAC (b) The T-Cache

Figure 3: An example of the T-Cache

A longer CAM tag means a lower CAM tag matching, so the replacement policy can be fully exploited. An associativity of each macro-block in the T-cache is equivalent to 1024-way (128 rows×8 columns). The 12 bits of the TCache's tag uses CAM, while the remaining tag are implemented by using SRAM. The length of the CAM tag is decided through experimental results. For example, a desired address is EEEE234. There is no address matching the desired address in the tag store on a cache miss. However, maybe there is an address, FFFF234, in the tag store. If the tag address stored in the CAM tag is 234 and the SRAM tag is FFFF resulting in a CAM tag match (both addresses have 234). The T-Cache must select the cache block that holds address FFFF234 as the victim to maintain unique decoding as shown in Figure 3.

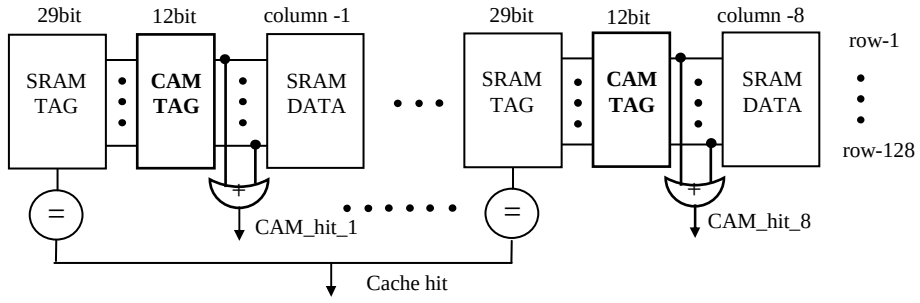


Figure 4: The organization of the proposed T-cache block cache

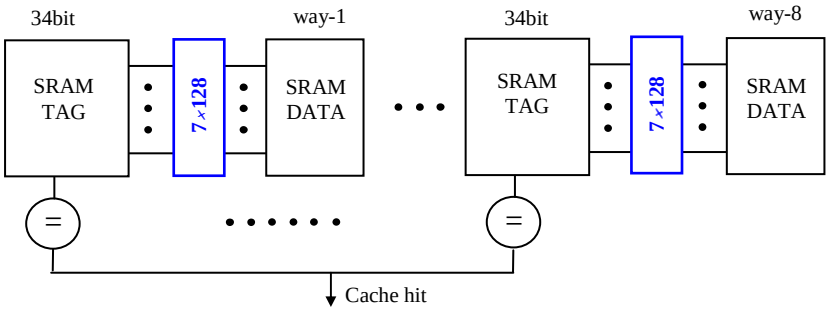
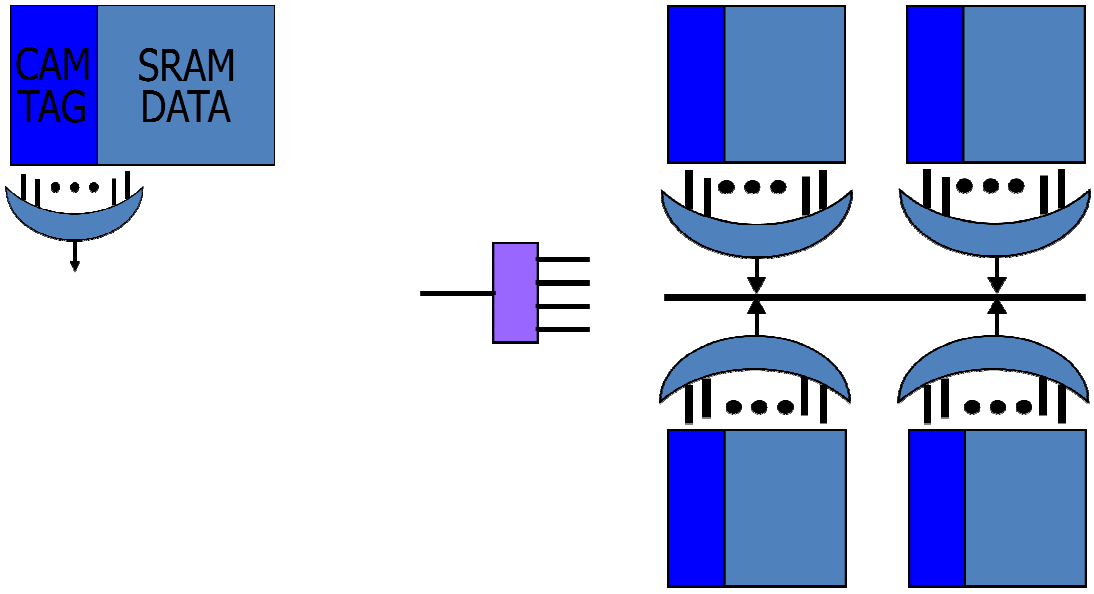


Figure 5: The organization of a traditional 8-way cache

Figure 4 shows the organization of a 128 KB macro-block T-Cache with 128 bytes line size. The address is assumed to be 48 bits. Comparison to the organization of a same sized 8-way cache shown in Figure 5, the T-Cache replaces the original 7×128 decoders with the 12-bit CAM based decoder. The function of the CAM decoder is as both the row decoders and partial tag comparison. Eight extra signals, from CAM_hit_1 to CAM_hit_8 are in the T-Cache. An assertion of the signal CAM_hit_n indicates a CAM tag hit in column n. These extra signals are used in cache miss handling to determine the victim.

We can use the proposed T-Cache macro-block to design two dimensional large caches. Figure 6(a) shows one T-Cache macro-block. Figure 6 (b) shows four T-Cache macro-blocks is used to implement a L2 cache of 512 KB. We use a macro-block decoder indexed by using high order bits of the address. For a 1MB cache

capacity, a 3×8 traditional decoder is used to locate the desired macro-block. The extra power consumption of the T-Cache due to CAM cells will not be increased as the cache capacity increases because only one macro-block is accessed during a cache visit.



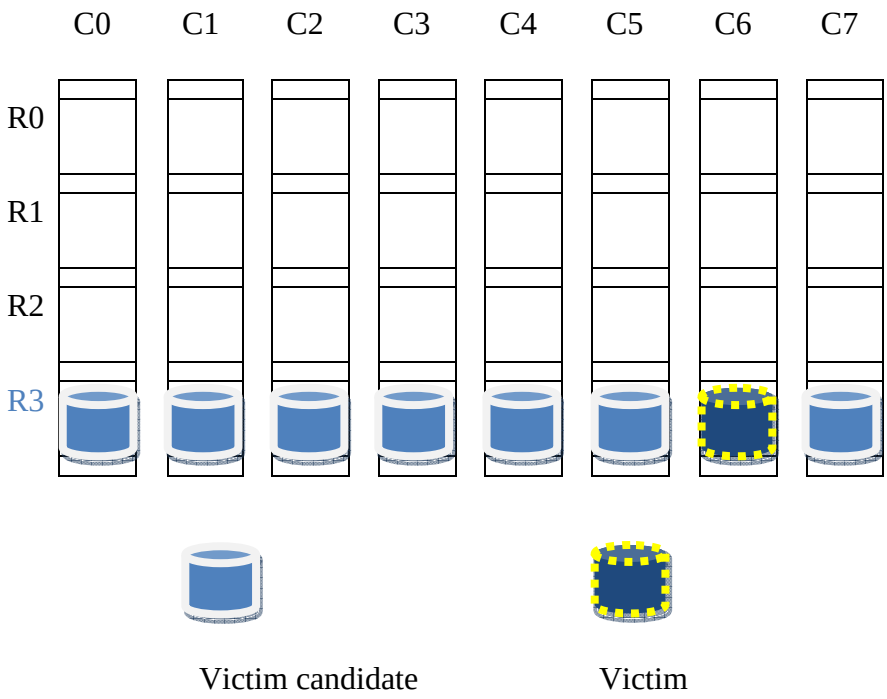
(a) One T-Cache macro-block (b) Four T-Cache macro-blocks

Figure 6: A 512 KB T-Cache cache

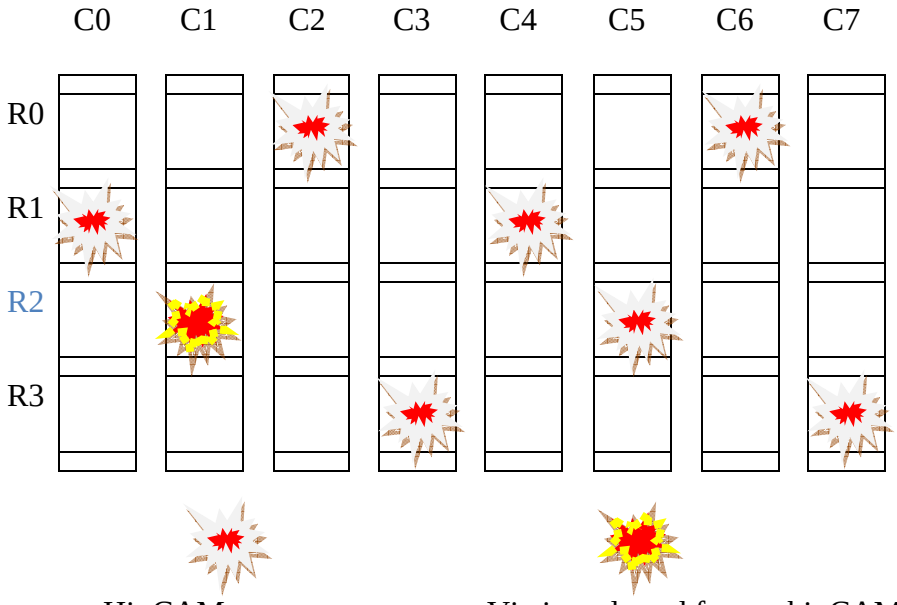
The operation of the T-Cache includes two situations. One situation is searching for the hit block for a cache hit; the other situation is searching for the victim block for a cache miss.

For the hit block, one macro-block, which is determined through the block decoder, is probed to search the desired block. All the columns are searched concurrently while the row replacement of the T-Cache guarantees that there is only one CAM tag hit block in each column.

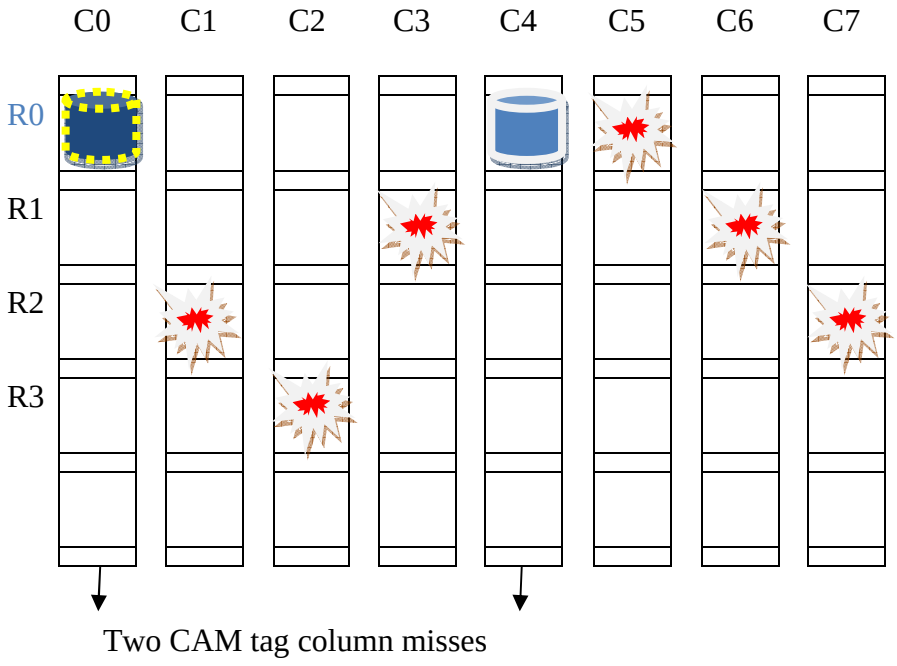
For the victim block, there are three different operations for a cache miss as shown Figure **7**. Figure **7** (a) shows the operation in which no CAM tag matches the desired address. A victim row is chosen based on the random replacement policy while the victim block will be chosen from the eight blocks in the selected row based on LRU policy.



(a) CAM tag hits



(b) CAM tag hits for all columns



(c) Part of CAM tag hit

Figure 7: Searching for a victim for the proposed two-dimension cache

Figure 7 (b) shows the operation in which each column has one CAM tag that matches the desired address. The victim block must be selected from these hit CAM blocks because each column can only have one CAM tag hit to keep the unique CAM tag decoding. One row from these CAM hit rows is selected based on the random replacement policy. The victim block is then selected from the CAM hit blocks in the chosen victim row based on LRU replacement. Figure **7** (c) shows the operation in which some columns have CAM tag hits while other columns do not have CAM tag hits. The victim will be chosen from those unmatched columns. The victim row R0 is selected based on the random replacement policy. Based on the LRU replacement policy, the victim is selected from the two unmatched columns, which are C0 and C4

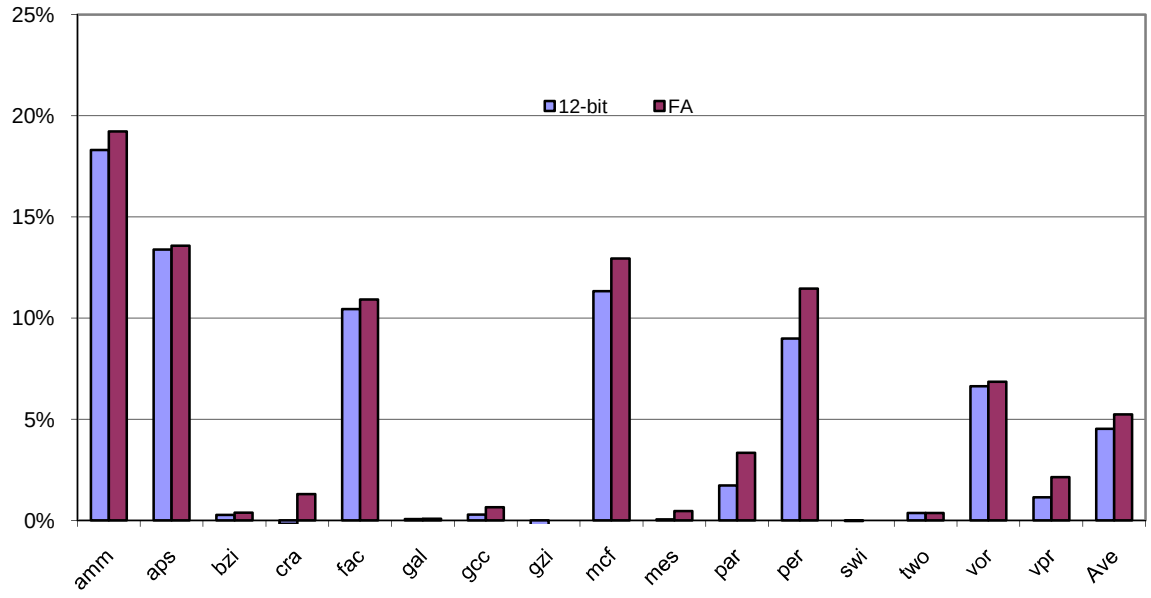


Figure 8: IPC improvement over the processor with an eight-way baseline cache.

in this example.

The proposed T-Cache uses CAM decoders and dividing the high associativity into rows and columns to reduce the hardware complexity. It reduces the miss rate of the baseline 8-way unified L2 cache by 16%. Figure 8 shows the Instructions Per Cycle (IPC) improvement in T-cache can be up to 18% and on average is 5%. On average, the T-Cache consumes 3.4% less energy than the baseline.

2.3 Memory Level of Parallelism

Previous some techniques increase memory level parallelisms (MLP) to allow an overlap of multiple misses to reduce the effective penalties. The number of memory of stall cycles due to memory accesses can be reduced by exploiting MLP method.

These MLP techniques include: microarchitecture optimization[20], run ahead execution [68], dividing the program execution into computation and memory access [86], continual flow pipelines [90], control-flow prediction to execute the memory-access ahead of the computation [81], speculative precomputation [25], slipstream processor [94], and caching scheduled groups[39]. These techniques may suffer from the limitations in control-flow predictability. Furthermore, run-ahead techniques and a large execution window do not increase memory level parallelisms when memory accesses are dependent. For example, one memory access must be complete before the subsequent dependent access can proceed in linked-data structures.

2.4 Data Prefetching Techniques

Cache prefetching brings cache blocks from lower level caches and/or memory before these blocks are requested. The time spending on prefetching can be overlapped with computation. The performance of some scientific applications can be significantly improved by using suitable cache prefetching techniques.

Conventional prefetching techniques to reduce cache misses, including hardware prefetching [13][49][67], modified cache organizations [92][70], software-based prefetching and data forwarding [18][59][64], and hardware/software cooperative prefetching [15][98], use heuristics to identify addressing patterns and predict memory addresses that are probably to be requested in the near future.

For on chip caches, cache blocks are brought from memory or low level caches to high level caches. The on chip caches are mostly organized with limited associativity. Most of the prefetching techniques for on chip caches, the requested block (maybe a cache hit or miss) and the prefetched block are stored in different cache sets, while the requested and prefetched pages are stored in the same set (A fully associative cache has only one set).

From the point of view of cache prefetching, prefetching exploits special program access patterns. For example, the One Block Look ahead (OBL) prefetching, one of the simplest sequential prefetching schemes, assumes that the program exhibits

spatial locality that when one cache block is requested, the next block is likely to be requested in the near future.

For long-latency memory access, conventional prefetching is of limited value for L2 caches. One problem is that addressing patterns must be highly predictable, which is not always the case in real applications. The other problem is that prefetching must occur far in advance to overcome long memory latencies. This makes prefetch addresses more difficult to predict. Finally, cache prefetching from memory will put more stress on the bandwidth since not all prefetchings is useful. Prefetching benefits are limited if memory bandwidth is a primary resource [91]. The increase of coherence traffic is difficult to avoid because of the increase of memory traffic. For the OBL prefetching, maybe the memory system has no enough time between the use of the N_{th} block and the next block $N_{th}+1$ to completely avoid a processor memory stall. Another problem of cache prefetching is the cache pollution problem when useless prefetched blocks replace useful ones.

The ideal situation is that we know the oracle information and prefetch them ahead of access time, making the memory latency totally hidden. Compiler based algorithms, including the cache-conscious data layout in memory [18][19] [20][21][22] and the cache-oblivious algorithm [29], optimize the data layout to reduce cache misses. Static optimizations techniques request involvement of compilers and cannot fully take advantage of the knowledge of a program. Kadayif et al. proposed the study of the interaction between prefetching with cache line turnoff

for power reductions [51]. Alameldeen et al. combine link compression technology with prefetching to reduce prefetching's bandwidth demand by extra compression tags [1]. Srinath et al. proposed feedback directed prefetching scheme to reduce potential for cache pollution and memory bandwidth contention by dynamically adjusting prefetcher distance and degree [91]. Prefetcher distance determines how far in advance to prefetch. Prefetcher degree determines the number of prefetches is sent out on a demand access. Prefetch accuracy can be measured by adding a pref-bit to each L2 tag store entry and two counters pref-total and used-total.

2.5 Improved Cache Replacement

Cache replacements play an important role in cache misses and miss penalty reductions. Cache replacements determine: (1) where to insert the incoming block; (2)

Table 2: Time Complexity in Replacement Policies

No.	Replacement Policy	Time Complexity
1	Least Recently Used (LRU)	O(1)
2	Most Recently Used (MRU)	O(1)
3	Least Frequently Used (LFU)	O(log(n))
4	Frequency Based Replacement (FBR)	O(1) to $O(\log_2(n))$
5	Least kth-to-last Reference (LRU-k)	O(log(n))
6	Least Frequently Recently Used (LFRU)	O(1) to O(log(n))
7	Two Queue (2Q)	O(1)

Table 3: Cache Asscociativity and Replacement in Processor Family

Processor Family	Processor Name	Asscociativity		Replacement Policy	
		L1	L2	L1	L2
Pentium III	Pentium III (0.25 µm)	4-way/4-way	4-Way	LRU/LRU	LRU
	Xeon (0.25 µm)		4-Way		
	Celeron (0.18 µm)		4-Way		
	Pentium III (0.18 µm)		8-Way		
	Xeon (0.18 µm)		8-Way		
	Xeon (0.18 µm) A		8-Way		
	Celeron (0.13 µm)		8-Way		
	Pentium III (0.13 µm)		8-Way		
Intel Pentium IV	Celeron(0.18 µm)	8-way/4-Way(8 KB) or 8-way(16 KB)	4-Way	PLRU/ PLRU	PLRU
	non-Celeron (0.18 µm)		8-Way		
	Celeron (0.13 µm)		2-Way		
	mobile Celeron (0.13 µm)		4-Way		
	non-Celeron (0.13 µm)		8-Way		
	Celeron (0.09 µm)		4-Way		
	non-Celeron (65 nm) (0.09 µm)		8-Way		
	non-Celeron (65 nm) (0.09 µm)		8-Way		
Intel Core	Celeron (low-cost) Core 2 (mobile, desktop, and extreme) Xeon and Xeon MP (server)	8-way/8-way	2-Way	LRU/LRU	LRU
			4-Way		
			8-Way		
			12-Way		
			16-Way		
AMD K8	Sempron (Low-End) Athlon 64 (Mid-Range) Athlon 64 FX (High-End) Turion 64 (Mobile) Opteron (Server)	2-way/2-way	16-way	LRU/LRU	PLRU
IBM Power PC 750	PowerPC 750CX PowerPC 750CXe PowerPC 750FX PowerPC 750GX PowerPC 750CL	8-way	2-way	PLRU	PLRU

how to update block access statuses on a cache hit; (3) which block to evict on a cache miss. Therefore, it involves the following three policies:

- Insertion policy for an incoming block
- Promotion policy on a cache hit
- Victim selection policy on a cache miss

Belady's well-known Min theoretical memory cache replacement is proposed in 1966 [8]. Traditional practical cache replacement policies [16][47][49][45] [71][88] [79] [83] include Least Recently Used (LRU), Least Frequently Used (LFU), Most Recently Used (MRU), Random Replacement (RR), and combinations of these replacements (Hybrid schemes). The LRU and LFU are two typical replacement policies. It is well known that many proposals implement their combinations by software methods [28] and hardware designs [54]. Recently cache replacement policies research progresses apply adaptive insertion policies [66][78], reuse distance prediction [29][53][75], dead block prediction [59], memory region based prediction [98], counter-based prediction [54], frequency-based prediction [80], memory reference behavior [36], re-reference interval prediction [42], and other prediction methods to these traditional cache replacements. Table 2 shows the time complexity of these replacement policies. The time complexity is O(1) for LRU, MRU and 2Q, while the time complexity is O(log(n)) for LFU and LRU-k. We introduce LRU,

LFU, and hybrid schemes in this subsection. We present our previous work in the tag-based replacement [112] to exploit the frequency information, too.

2.5.1 Least Recently Used (LRU)

The least recently used (LRU) replacement, pseudo LRU (PLRU) replacement, and their derivatives are widely studied in academic and industrial research. Table 3 shows cache asscociativity and replacement in main processor family. LRU is implemented by the Intel Core and the Intel Pentium III. PLRU is employed by the IBM PowerPC 750 and the Intel Pentium II-IV.

LRU does not exploit frequency information of cache accesses. For low associative cache, implementing the LRU is economical. But it is hard to implement a true LRU algorithm for high associative cache in hardware because it is expensive if hardware is always keeping track of what was used and when discards the least recently used item. LRU suffers the problem of hardware complexity that requests O (n^2) status bits when the reference matrix based method is used, where n is the cache associativity.

Other proposals improve the performance by using modified LRU [2][13][28] [66][71][73][93][101][115] and modified PLRU [32]. These LRU-based replacement algorithms measure cache block access distance or tune parameters by extra structure. The experimental results prove these modified LRU policies have better performance than LRU policy for some benchmarks, and worse performance in other benchmarks.

The gap between LRU and OPT replacement policy, up to 50%, new research to close the gap is necessary.

2.5.2 Least Frequently Used (LFU)

Frequency information is recorded in the LFU. Frequency means how many times cache line is being accessed in cache memory. The LFU policy [80] counts how often a cache line is needed but without considering recent history.

Those cache lines that are used least often are discarded first. It is logarithmic implementation complexity in cache size. The LFU policy does not adapt well when access patterns of the program change. E.g. some cached data may be heavily accessed only during a particular time in running this program. As a consequence, those data will have a high value for accessing count. It means these data may keep in the cache for a long time, even though they are no longer being used in the near future.

Several studies have exploited frequency information for improving cache replacement. These frequency-based replacement algorithms rely on counters, tables, high associativity, or the intervention of the operation system. They require more space overhead. Lee et al. [60] proposed a Least Frequently Recently Used (LFRU) algorithm for file management systems. The LFRU uses a weight function to control the contribution of recency and frequency to the replacement decision. The weight function is application and system dependent. A counter-based cache replacement

[54] for an on-chip cache is proposed to evict the dead blocks in a L2 cache quickly. Each cache block is augmented with a counter, and the cache block expires when the counter exceeds a threshold. The threshold is dynamically determined for each block and stored in a 40KB table.

We propose to implement the tag-based replacement [112] with a fixed 256 tags to exploit the recency and frequency information. In traditional block-based

Table 4: Number of different tags at an interval of 1M cache access

Name	Tag(max, ave, min)			Name	Tag(max, ave, min)		
	512KB(8192)	1MB (16384)	2MB(32768)		512KB(8192)	1MB(16384)	2MB(32768)
amm	844,617,485 406,305,220	424,346,294 356,300,187	213,183,162 202,190,158	mcf	1241,379,96 119,29,18	637,202,54 109,32,18	321,110,33 114,37,22
aps	1714, 465,72 659,216,20	863, 307,40 674,227,21	458,186,24 433,166,20	mes	284,189, 104 122,93,17	142,100, 59 79,66,50	76,53,34 54,45,37
art	104,99,98 32,24,17	55,53,52 31,28,25	30,29,29 30,26,24	mgr	882,685,511 23,22,19	447, 346, 257 23,21,19	227,176,130 21,21,20
fac	327,327,327 183,183,183	173,173,173 141,141,141	92,92,92 82,82,82	par	1162,1055,954 874,685,359	604, 545, 493 569,490,243	317, 285, 257 311,275,219
gal	325,317,312 195,119,37	174,167,164 109,81,38	98,92,90 93,71,33	luc	1310,1239,1132 28,27,25	665,627,575 28,27,26	343,321,297 28,27,26
gap	486,337,217 50,38,17	248,176,118 48,33,17	128, 94, 65 42,30,17	swi	2369,1491,1160 27,23,19	1190,750,585 27,23,19	601, 380, 297 131,36,19
gcc	272,160,64 177,108,71	181,108,48 161,123,50	112,71,33 119,106,34	vor	1263,1236, 1209 659,605,544	711,702, 687 609,581,557	377,375, 373 376,373,370
gzi	33, 31, 29 26,25,25	19,18,17 25,23,20	11,11,10 23,21,12	vpr	1029, 836, 643 393,318,256	591, 492, 348 317,277,233	316, 271, 186 234,201,164
six	84,82,82 69,52,40	55,54,53 55,54,54	36,36,36 37,37,37	wup	1277, 1126,1018 49,36,18	642, 567,513 46,34,18	324, 288,261 47,36,18

cache replacements, only the access information of blocks that are in cache is recorded. It is well known that tracking the tags instead of all the blocks has lower hardware complexity. Tracking more tags may keep the record of cache blocks that have been evicted from the cache, too.

Table 4 shows the maximum, average, and minimum number of different tags that have been accessed (first row of each benchmark) and that are still in cache (second row of each benchmark) of the SPEC2K CPU benchmarks at cache capacities of 512KB, 1MB, and 2MB measured at an interval of one million L2 cache access over 250 million instructions executed. We made the following observations from the table. First, the number of accessed tags during the 1M accesses is more than the number of blocks that are still in the cache. This means that some cache blocks have been evicted for most of the benchmarks during the 1M access interval. There are many identical tags across different cache sets because of spatial locality of programs. Second, as cache capacity increases, the number of tags that have been accessed and tags that are still in cache is decreasing for all the SPEC2K benchmarks. The length of tag is decreasing when cache capacity increases. Third, the total number of different tags is significantly less than the number of cache blocks for cache capacity of 512KB, 1MB, and 2MB for all the benchmarks.

In the tag-based replacement, the access information of each tag instead of each block is recorded. The tag-based replacement can exploit access frequency and

recency information of the tags instead of each block.

Figure 9 shows the hardware structure for a 16-way 1MB cache with the tag-based replacement. There are one 8 x 256 decoder, 256 5-bit counters, and 1-bit counter for the proposed tag-based replacement. The decoder activates the status bits by the low order 8 bits of the tags. The 256 5-bit counters are used to record the access frequency of each tag. The 1-bit counter is used to record the recency of each tag.

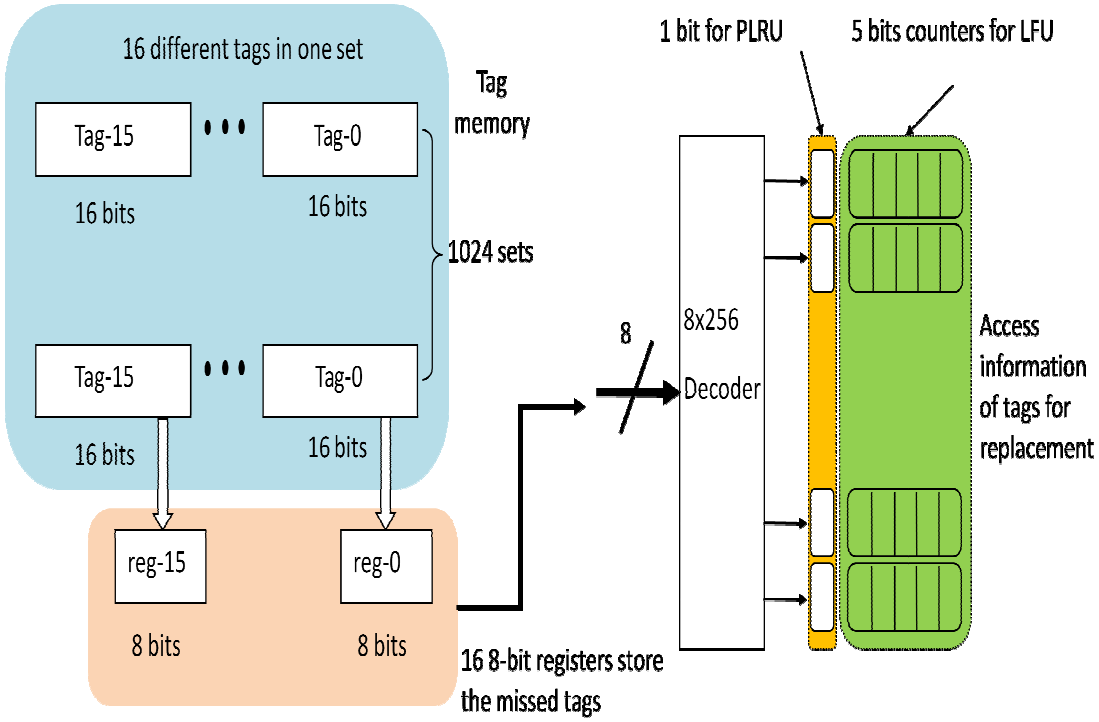


Figure 9: The hardware structure of the proposed tag-based replacement for a 16-way 1MB cache with a line size of 64B

Figure 10 shows the miss rate reduction of the tag based replacement at cache sizes of 512KB, 1MB and 2MB over the block-based LRU. We see the tag-based replacement with a fixed 256 tags reduces the average miss rate of the baseline 1MB L2 cache by 15% over LRU with lower hardware complexity. The 6bit, 8bit, 10bit bars represent that 64, 256, 1024 tags are recorded in the tag-based replacements. The average value of miss rate can be reduced for the tag-based replacement when the bits increase from 6 (tracking 64 tags) to 10 (tracking 1024 tags). On average, the miss rate in 8 bits is almost the same to tracking 1024 tags for the baseline 1M L2 cache.

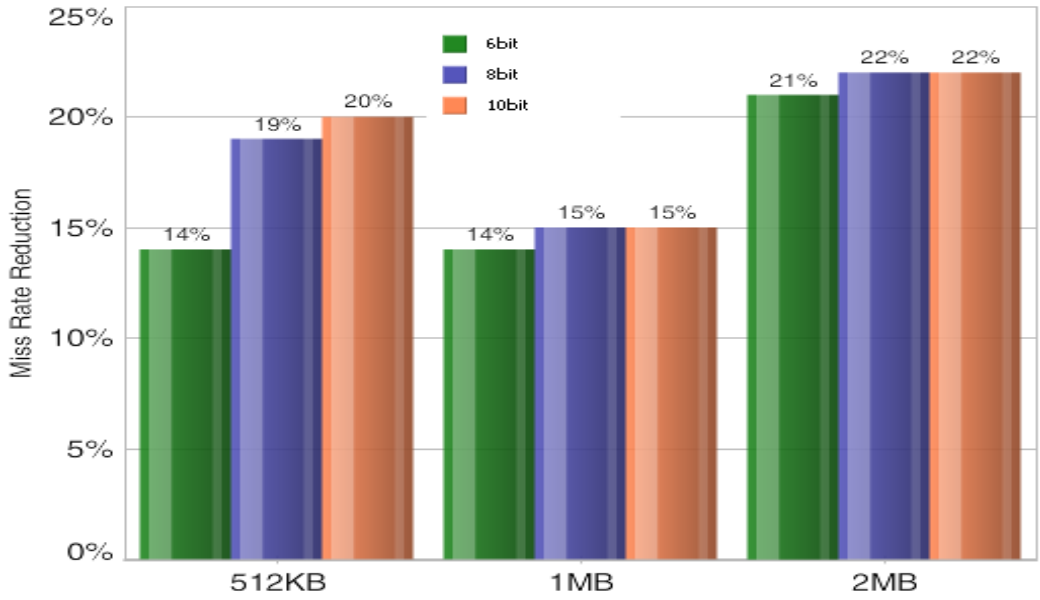


Figure 10: The miss rate reduction of the tag based replacement at cache sizes of 512KB, 1MB and 2MB over the block-based LRU

Figure 11 and Figure 12 show the miss rate of the tag based replacement at cache sizes of 512KB and 2M, respectively. According to experimental results as shown in Figure 11 and Figure 12, our tag-based replacement select 8 bit (tracking 512 tags). One reason is tracking 256 different tags can achieve lower miss rate and better performance than tracking 64 different tags. In the other hand, increasing the tags number to 1024 would not significantly reduce the miss rate.

Figure 13 shows the processor performance improvement of our tag-based replacement is up to 40% (benchmark gap) and on average 4.5% compared to the block-based replacements.

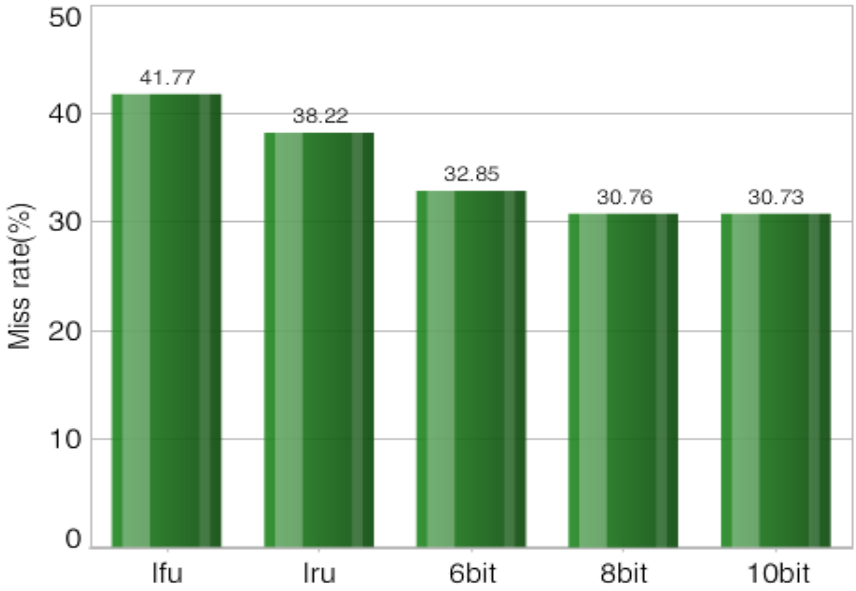


Figure 11: The miss rate of the tag based replacement at cache sizes of 512KB

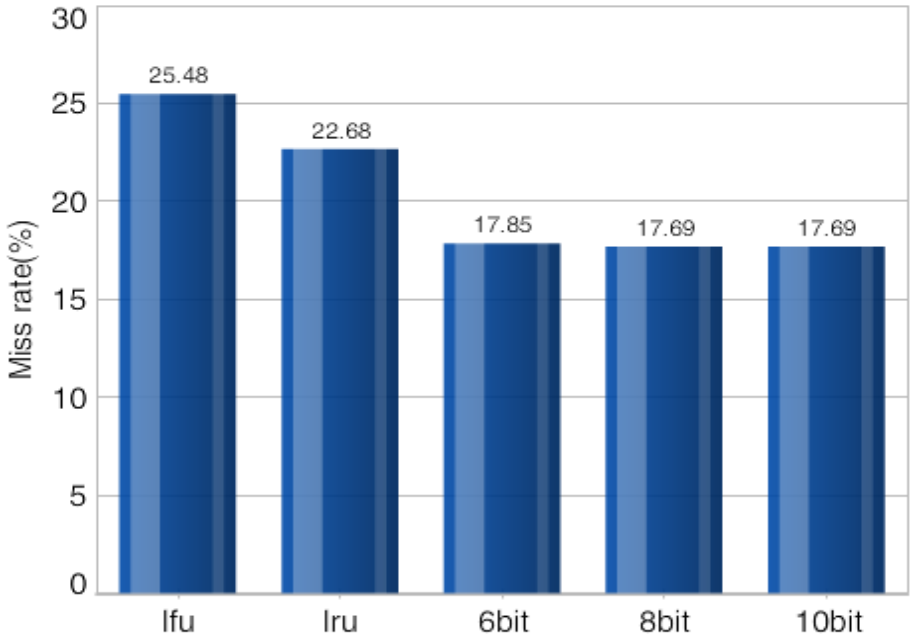


Figure 12: The miss rate of the tag based replacement at cache sizes of 2MB

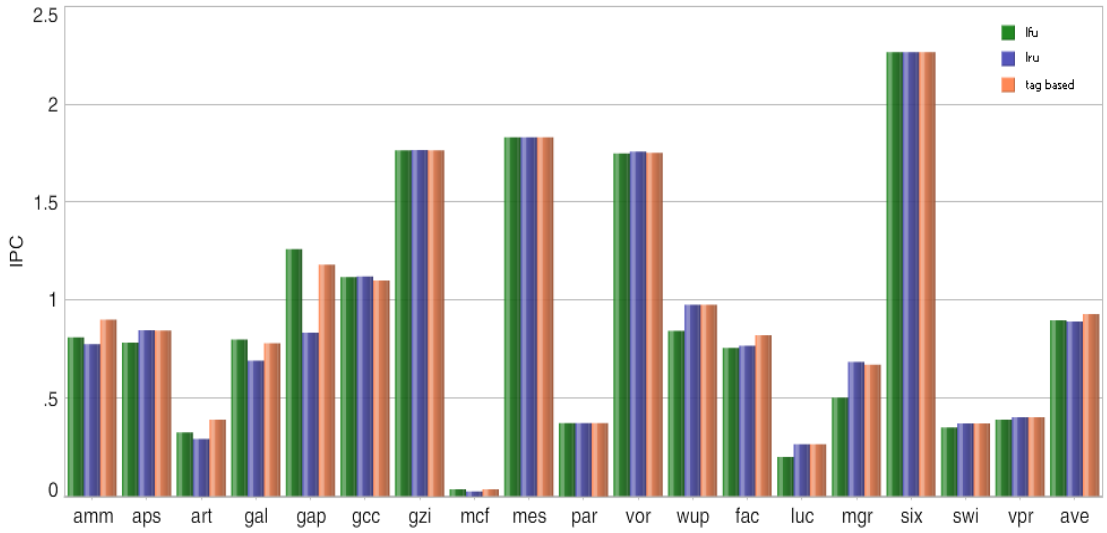


Figure 13: The IPC using the tag-based replacement compared to the block-based LRU and LFU

2.5.3 Optimal Replacement (OPT)

BELADY's Optimal Replacement (OPT) [8] is defined that "the optimal candidate for replacement is the one that is accessed farthest in the future". OPT can discard cache block that will not be used for the longest time in the future. The victim block should be the block with longest reuse distance.

OPT replacement cannot be implemented in hardware because it requires future knowledge and provides a great upper-bound. We may model it with known trace. It is used in simulations to compare the performance of different replacement policies.

The earlier eviction algorithms and dead block prediction algorithms [59] tried to emulate this OPT behavior. Last touch prediction [61] evaluates the predictability of the victim under the OPT but without considering efficient implementation issues. The EELRU adaptive replacement algorithm [83] approximates the OPT based on the LRU stack model of program behavior for page replacement.

2.5.4 Hybrid Schemes

Hybrid schemes [35][47][69][77][113][114] include adaptive policies and non-adaptive policies. Adaptive policies replacements that dynamically select and use the best replacement policy from a pool of multiple replacements to suit particular applications include dynamic tracking of misses [113] for page replacement and multiple replacements for the second buffer cache [114]. Both static and dynamic

cache bypassing schemes [35][47][69][77] attempt to bypass data with little likelihood of in-cache reuse based on particular load instructions, addresses being accessed, or dynamic referencing behaviors. Qureshi et al. presented a dynamic insertion policy (DIP) using set dueling monitors (SDMs). The dedicated sets are used to estimate the performance of a pre-defined replacement policy. DIP uses a single PSEL counter. Non-adaptive policies, for example, the shepherd cache [52] mimics the replacement decisions of the OPT by logically dividing the L2 into two components, a shepherd cache and a main cache with an emulation of optimal replacement. The shepherd cache plays the dual role of caching lines and guiding the replacement decisions in the main cache. However, the limited number of cache ways in the shepherd cache restricts the future information it can provide for the main cache.

The performances of these techniques are mostly program dependent. For example, in the dynamic insertion policy [78] for cache replacement, four out of the 16 benchmarks exhibit performance improvement over 10%, while five out of the 16 benchmarks exhibit performance degradation and others exhibit moderate performance improvements.

2.5.5 Behavior Difference between OPT and the LRU

Substantial studies have shown that cache misses between using the LRU and the OPT are significant, especially in highly associative caches. This motivates us to

analyze the behavior of the OPT algorithm and to discover both recency and frequency information that we can use to emulate the OPT.

We describe program characteristics through analyzing the memory access behaviors of the OPT. We ran the 15 memory-intensive SPEC2K CPU [95] benchmarks using the SimpleScalar tool set [11] to collect address traces of the benchmarks but only show results of several programs for easy illustration. The benchmarks are pre-compiled for the Alpha ISA [99]. For each benchmark, we use a representative sample of instructions obtained from SimPoint [74] using reference inputs. For each sample, we run 2 billion instructions. The processor is configured as an eight-issue with an instruction window of 128 entries, separate 2-way L1 caches with a capacity of 32 KB and a line size of 64 byte, and an 8-way 1MB L2 cache with a line size of 128 byte.

We fed address traces to the OPT algorithm and implemented the LRU in the OPT to generate the LRU status of each block, while the victim block was determined through the OPT. Figure 14 shows the LRU statuses of the OPT victim blocks. The LRU statuses are represented as LRU-0, LRU-1, …, and LRU-7 in this figure, where the LRU-7 is the least recently used block and the LRU-0 is the mostly recently used (MRU) block.

We made the following observations from Figure 14. First, The LRU-7 blocks exhibit a very small portion in the OPT victims, which explains the significant performance gap between the LRU and the OPT. Second, the OPT mostly replaces the MRU block (LRU-0) instead of the LRU-7 block to make room for the requested data. The OPT proactively evicts the block that will not be used in the near future, since the OPT knows the future information. Therefore, the mostly recently used block will be evicted if this is the last use of the block. Third, the behavior of the OPT algorithm is non-uniform in that over 80% of the victims are the MRU for the benchmark *mgrid* while around 30% of the MRU is the victim for the benchmark

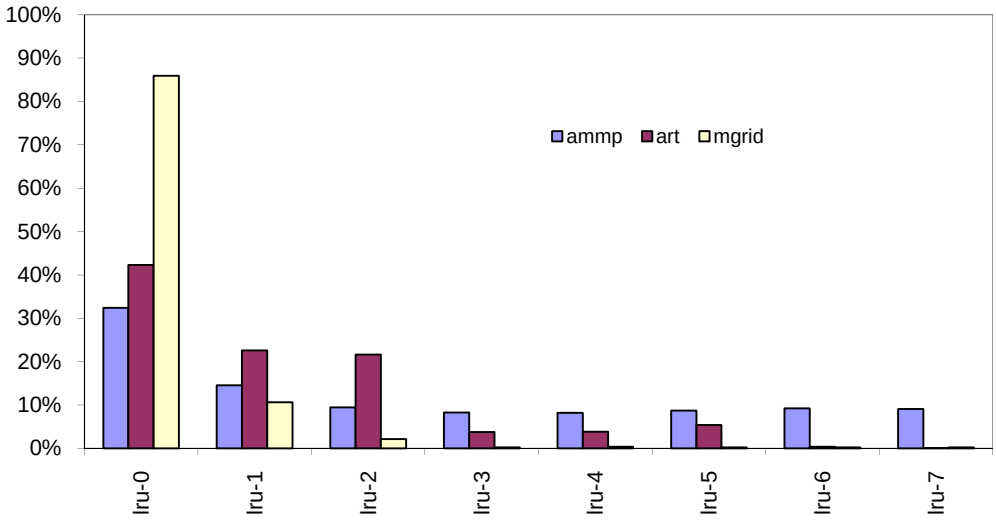


Figure 14: The distribution in percentage of LRU status of the victim when OPT is used.

ammp. Finally, the LRU status of the optimal victim block spreads almost all over from the MRU to the LRU block. This means that the optimal victim cannot be solely determined through the LRU status, which motivates us to exploit frequency to determine the optimal victim.

2.6 Summary

This chapter introduced traditional cache optimizing techniques. These techniques include modified cache organization, memory level of parallelism, data prefetching techniques, improved cache replacement, and their combinations.

Table 5: Difference in LRU, LFU and Hybrid Replacements

	Exploit Recency	Exploit Frequency	Drawback
LRU	Yes	No	Cache thrashing
LFU	No	Yes	Not adapt well to changing access patterns
Hybrid	Yes	Yes	Require extra structures and consume more power

In improved cache replacement policies, we discuss some widely used replacement policies including LRU, LFU, and Hybrid replacements. We also present our previous published work in T-cache and the tag-based replacement policy. In hybrid replacements, adaptive policies select the best replacement by dynamically measuring the miss rate of different replacement policies. Table 5 shows the

differences in these replacement policies. Although hybrid replacements exploit both recency and frequency, additional hardware cost is an issue for it.

CHAPTER 3

FREQUENCY BASED SINGLE QUEUE (FSQ)

3.1 Introduction

Some previous proposals [80] [54] try to use frequency information to improve cache replacement policy in order to achieve better performance. However, these techniques request extra counters or tables to exploit both frequency and recency. We proposed Frequency based Single Queue (FSQ) replacement that we arrange the cache lines of one set in a queue for the replacement decision. Table 6 shows three rules for the FSQ replacement on a cache miss or a cache hit. Every rule includes the following three policies: insertion policy, promotion policy, and victim selection policy.

Table 6: The FSQ Replacement Policy

Rule No.	Set-hit flag	Victim Selection	Insertion	Promotion	Exploit Recency	Exploit Frequency
1	H->M	Bottom	Bottom	Unchanged	Yes	
2	M->M	Top	Bottom	Rollover	Yes	Yes
3	H->H; M->H			if hit on the top, then unchanged; else exchange	Yes	Yes

Insertion policy: an incoming block is inserted at the bottom of the queue when a cache miss occurs. The traditional LRU inserts an incoming block to the MRU

position. The FSQ replacement inserts the incoming block into the bottom of the queue since it has only one access and all the blocks in the queue have more accesses than the incoming block. Therefore, the frequency information of a cache block is considered in the FSQ replacement.

Promotion policy: a block exchanges its position with its adjacent block above in the queue when a cache hit occurs. If the hit block is already at the top of the queue, then no position exchange is performed. The exchange position promotion policy moves frequently used blocks to the top of the queue but less frequently used blocks to the bottom of the queue. Since the most recently used block is moved upward to the top of the queue and away from the bottom of queue, the FSQ replacement exploits both recency and frequency information of programs. The FSQ replacement is different from the LFU in that the FSQ replacement does not record the absolute but relative frequency of the blocks.

Victim selection policy: the victim block may be chosen from the bottom or top block of the queue, which is controlled by a set-hit flag. The set-hit flag is set to “H” when the last access to this set is a hit or to “M” if the last access to this set is a miss. When this set-hit is “M”, then the top block is chosen as the victim and the bottom block is chosen as the victim when the bit is “H”. When the top block is chosen as the victim, all the remaining blocks in the queue will be moved one step

upward to the top, which we name “rollover”. Rollover is used to move new blocks into the cache when working set changes.

3.2 The Operation of the FSQ

In this subsection, we show the operation of the FSQ replacement when address sequence is EBFGB. The miss or hit status is indicated by the set-hit flag. Assuming the last access to the set is a cache hit, it is shown in Figure 15(a).

Rule No.	Set-hit flag	Victim Selection	Insertion	Promotion	Exploit Recency	Exploit Frequency
1	**H->M**	**Bottom**	**Bottom**	**Unchanged**	**Yes**	
2	M->M	Top	Bottom	Rollover	Yes	Yes
3	H->H; M->H			if hit on the top, then unchanged; else exchange	Yes	Yes

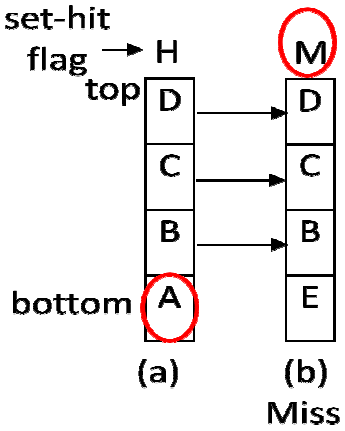


Figure 15: Operation of the FSQ replacement (incoming block is E)

The first incoming block is the block E. The requested block E incurs a cache miss as shown in Figure 15. According to rule 1, the block A at the bottom of the

queue is chosen as the victim, which is the same as the LRU replacement. The incoming block E is inserted into the bottom of the queue, which is different from the LRU who inserts the block at the MRU position. The other blocks in the queue keep unchanged.

The incoming block is the block B. According to rule 3, Figure 16 shows the promotion procedure when the block B hits and the block B and C exchange their positions in the queue. The block B moves up one position, while block C moves down one position.

Rule No.	Set-hit flag	Victim Selection	Insertion	Promotion	Exploit Recency	Exploit Frequency
1	H->M	Bottom	Bottom	Unchanged	Yes	
2	M->M	Top	Bottom	Rollover	Yes	Yes
3	H->H; **M->H**			if hit on the top, then unchanged; **else exchange**	Yes	Yes

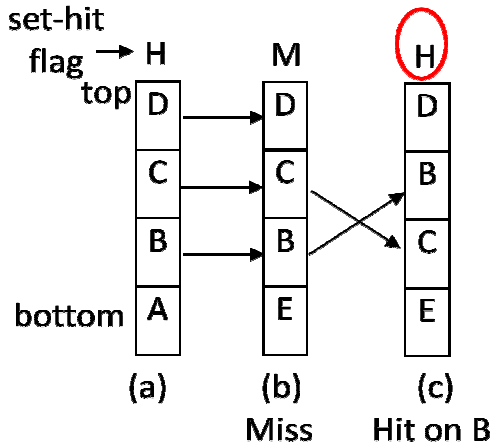


Figure 16: Operation of the FSQ replacement (incoming block is B)

Figure 17 shows the situation where the last access to the set is a cache hit. The requested block F incurs a cache miss. According to rule 1, block E is chosen as the victim and the remaining blocks D, B, and C keep unchanged, while the incoming block F is still inserted at the bottom.

Rule No.	Set-hit flag	Victim Selection	Insertion	Promotion	Exploit Recency	Exploit Frequency
1	**H->M**	**Bottom**	**Bottom**	**Unchanged**	**Yes**	
2	M->M	Top	Bottom	Rollover	Yes	Yes
3	H->H; M->H			if hit on the top, then unchanged; else exchange	Yes	Yes

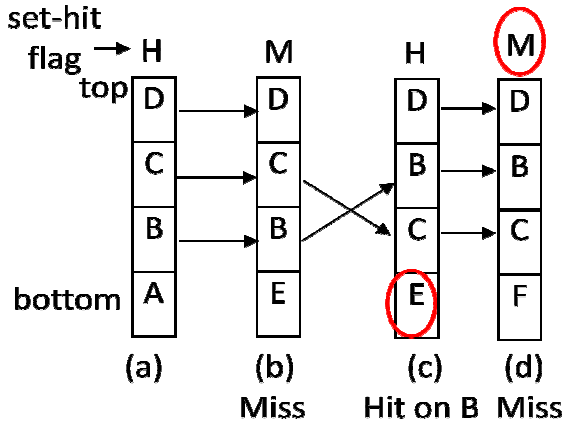


Figure 17: Operation of the FSQ replacement (incoming block is F)

Figure 18 shows the situation where the last access to the set is a cache miss. The requested block G incurs a cache miss. According to rule 2, block D is chosen as the victim and the remaining blocks. The block B, C and F are moved one step

upward, while the incoming block G is still inserted at the bottom.

Rule No.	Set-hit flag	Victim Selection	Insertion	Promotion	Exploit Recency	Exploit Frequency
1	H->M	Bottom	Bottom	Unchanged	Yes	
2	**M->M**	**Top**	**Bottom**	**Rollover**	**Yes**	**Yes**
3	H->H; M->H			if hit on the top, then unchanged; else exchange	Yes	Yes

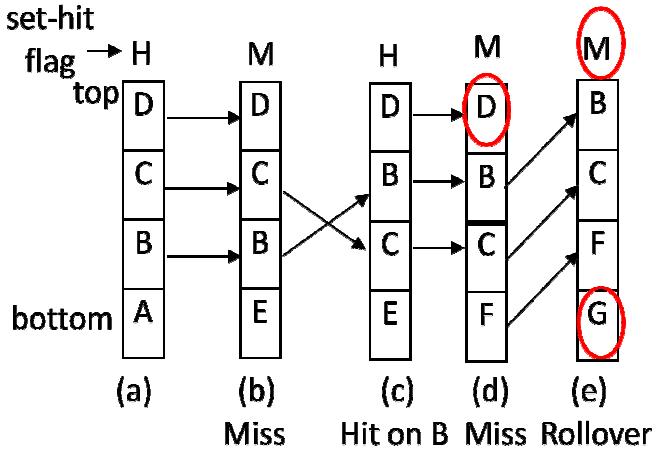


Figure 18: Operation of the FSQ replacement (incoming block is G)

The last incoming block is the block B. According to rule 3, Figure 19 shows the promotion when the access hits on the top block B and the position of all the blocks remain unchanged but the set hit flag is set to “H”. The block B has the most hits. Table 7 shows recency and frequency of this trace (address sequence: EBFGB) for FSQ.

Rule No.	Set-hit flag	Victim Selection	Insertion	Promotion	Exploit Recency	Exploit Frequency
1	H->M	Bottom	Bottom	Unchanged	Yes	
2	M->M	Top	Bottom	Rollover	Yes	Yes
3	H->H; **M->H**			**if hit on the top, then unchanged;** else exchange	Yes	Yes

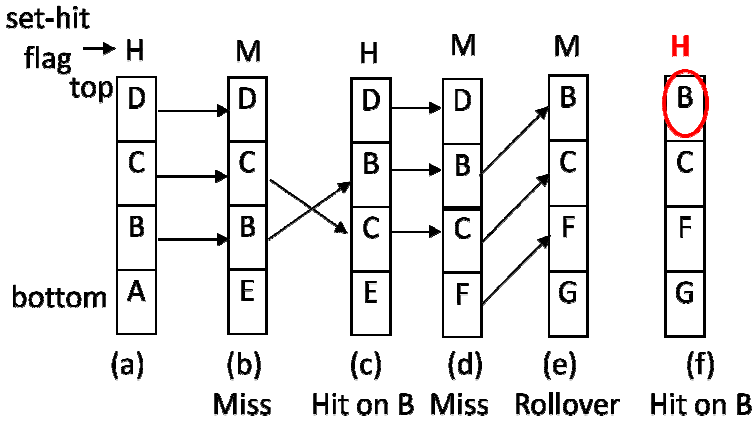


Figure 19: Operation of the FSQ replacement (incoming block is B)

Table 7: Analysis of Trace for FSQ

Position	1	2	3	4	5	6	7	8	9
Incoming Block	A	B	C	D	E	B	F	G	B
Distance		0				4			3
Frequency		1				2			3

3.3 Experimental Methodology

3.3.1 System Configuration

In our experiments, we use SimpleScalar tool set [11] to measure the cache misses and processor performance for evaluating our FSQ replacement. The miss rate and IPC are the primary metric [45][87][85]. Table 8 shows the parameters of the first level (L1) instruction and data caches. The L1 cache parameters were kept constant for all experiments. The baseline the second level (L2) cache is 1MB 16-way set associative with LRU replacement. All L1 cache parameters in the baseline use a 64B line size, while the L2 cache uses a 128B line size.

Table 8: Cache Configuration in FSQ

Parameter	Setting
L1 I-Cache	16kB; 64B line size; 2-way with LRU replacement
L1 D-Cache	16kB; 64B line size; 2-way with LRU replacement
Baseline L2 Cache	1MB; 128B line size; 16-way with LRU replacement
Main Memory	200 cycles unlimited size

3.3.2 Benchmarks

We do not enforce inclusion in our memory model. We run memory intensive SPEC2K benchmarks [95]. The benchmarks are pre-compiled for the Alpha ISA [99]. For each benchmark, we used a representative sample of 250 Million instructions obtained from SimPoint [12] using reference inputs with cache warm-up to eliminate

the impact of cold start misses. Table 9 shows the number of fast forward instructions and the Misses Per Thousand Instructions (MPKI) of the SPEC2K benchmarks.

Table 9: Benchmark Summary (B=Billion)

Name	Type	FFWD(B)	MPKI
equa	FP	0	0.06
sixt	FP	4.6	0.26
face	FP	94.2	2.17
luca	FP	38	11.41
art	FP	14	41.99
ammp	FP	30.4	2.53
apsi	FP	182.7	5.13
vort	INT	7.8	0.38
bzip	INT	143.4	0.04
galg	FP	32.5	5.83
mcf	INT	14.2	81.86
gap	INT	8.3	1.39
swim	FP	20.2	11.97
twol	INT	0.2	0.40
mesa	FP	49.1	0.33

3.4 Results

Figure 20, Figure 21, Figure 22, and Figure 23 show the MPKI reductions of the proposed FSQ replacement over the baseline 1M 16-way, 8-way, 4-way, and 32-

way LRU. The results of the FSQ without rollover operation replacement are also shown. The benchmarks are shown by their first four letters to save space.

From Figure 20, we made the following observations. First, the FSQ replacement outperforms the LRU for eight benchmarks, *equake, sixtrack, facerec*, *art*, *galgel*, *mcf, gap*, and *mesa*, but increases the MPKI for other five benchmarks, *lucas, ammp, apsi, vortex,* and *swim*. On average, the FSQ replacement reduces the MPKI by 2% over the LRU. Second, we observed that the rollover is important to avoid the significant MPKI increase for benchmarks *lucas, ammp,* and *apsi*.

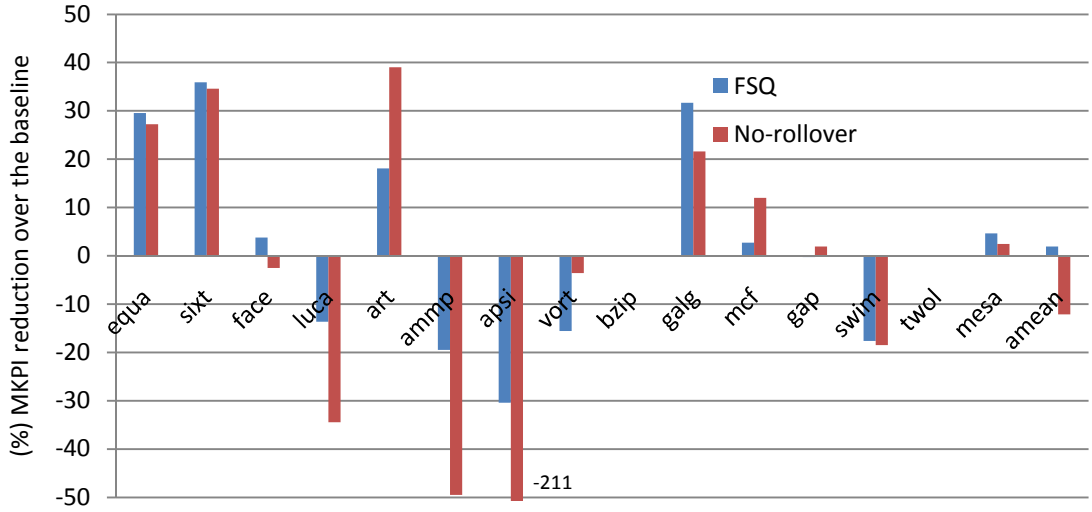


Figure 20: The percentage MPKI reduction of the FSQ and the FSQ replacement without rollover operation over the baseline 16-way LRU

From Figure 21, we made the following observations. First, the FSQ replacement outperforms the LRU for nine benchmarks, *equake, sixtrack, facerec, art, ammp, galgel, mcf, gap,* and *mesa*, but increases the MPKI for other four benchmarks, *lucas, apsi, vortex,* and *swim*. On average, the FSQ replacement reduces the MPKI by 2.7% over the LRU. Second, we observed that the rollover is important to avoid the significant MPKI increase for benchmarks *lucas, ammp,* and *apsi*.

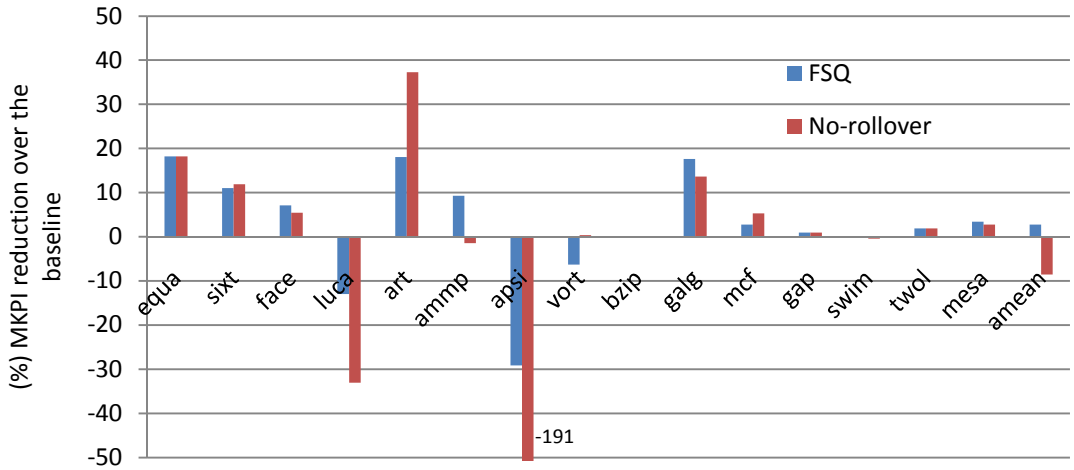


Figure 21: The percentage MPKI reduction of the FSQ and the FSQ replacement without rollover operation over the baseline 8-way LRU

From Figure 22, we made the following observations. First, the FSQ replacement outperforms the LRU for eight benchmarks, *equake, sixtrack, facerec, art, ammp, galgel, mcf, and twolf*, but increases the MPKI for other three benchmarks, *lucas, apsi,* and *vortex*. On average, the FSQ replacement reduces the MPKI by 4.8%

over the 4-way LRU. Second, we observed that the rollover is important to avoid the significant MPKI increase for benchmarks *lucas* and *apsi*.

From Figure 23, we made the following observations. First, the FSQ replacement outperforms the LRU for eight benchmarks, *equake, sixtrack, facerec, art, galgel, mcf, twolf,* and *mesa*, but increases the MPKI for other six benchmarks, *lucas, ammp, apsi, gap, vortex* and *swim*. On average, the FSQ replacement reduces the MPKI by 0.8% over the LRU. Second, we observed that the rollover is important to avoid the significant MPKI increase for benchmarks *lucas, ammp,* and *apsi*. But benchmarks *mcf, gap, vortex* and *art* cannot get benefit from the rollover operation in 32-way cache.

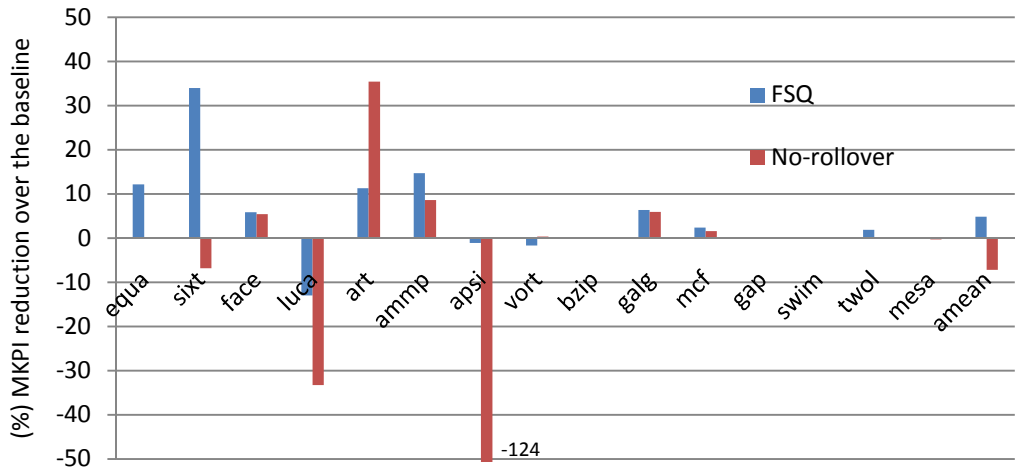


Figure 22: The percentage MPKI reduction of the FSQ and the FSQ replacement without rollover operation over the baseline 4-way LRU

We summarize these observations from Figure 20 to Figure 23. On average, the FSQ replacement reduces the MPKI over the all traditional (4-way, 8-way, 16-way, and 32-way) LRU. The FSQ reduces the MPKI up to 4.8% over the traditional 4-way LRU. In addition, the rollover operation can avoid significantly MPKI increase for most benchmarks because the rollover operation is very important for working set changes.

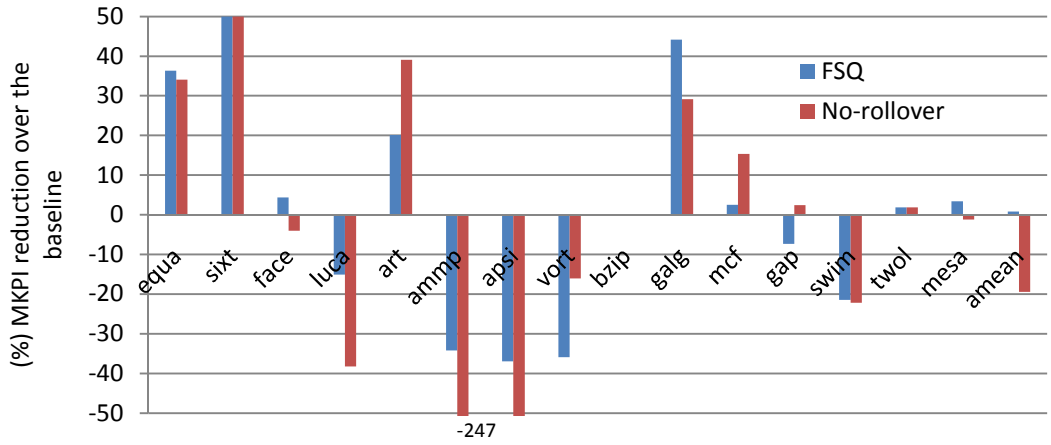


Figure 23: The percentage MPKI reduction of the FSQ and the FSQ replacement without rollover operation over the baseline 32-way LRU

3.5 Illustrative Example

The address traces can be categorized into three types: LRU-averse trace and LRU-friendly trace, and working set change trace. We prove the benefits of the FSQ replacement by three examples.

3.5.1 An example of LRU-averse Trace

First, we input a LRU-averse trace of x, y, z, D, E, x, F, y, z, G, D. The distance between any two same addresses is longer than the length of FSQ queue. For example, the distance between the same address x is 5. The distance between the same address y is 6.

Figure 24(a) show no cache hits occurs for the LRU. The LRU experience the worst situation in this example. On the other hand, Figure 24(b) show three cache hits if we employ the FSQ replacement. The FSQ exploits the frequency information of this trace. Table 10 shows recency and frequency information of this LRU-averse Trace. The access frequency for

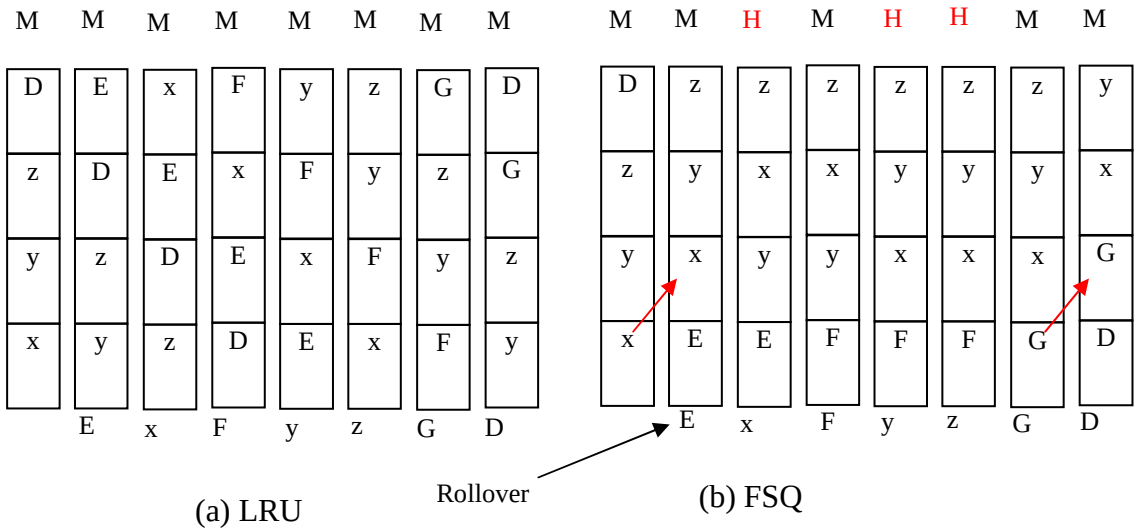


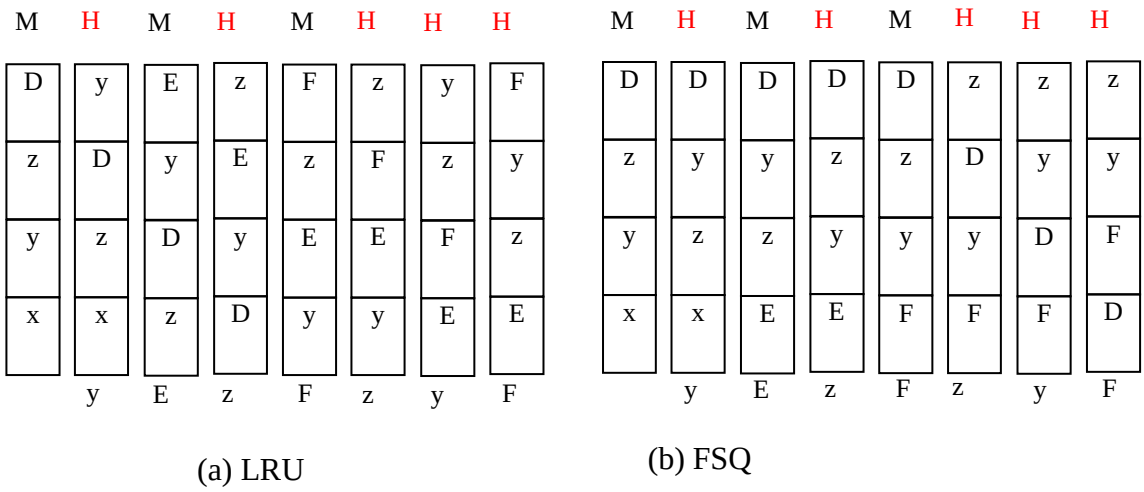
Figure 24: Comparison of the FSQ and LRU on a LRU-averse trace.

block x, y, and z is twice. Block x and y are placed in the top of the queue, while block z is kicked out of the queue after two rollovers occur, since block z has been on top of the queue most of the time.

Table 10: Analysis of LRU-averse Trace

Position	1	2	3	4	5	6	7	8	9	10	11
Incoming Block	x	y	z	D	E	x	F	y	z	G	D
Distance						5		6	6		7
Frequency					1	2	1	2	2	1	2

3.5.2 An Example of LRU- friendly Trace



(a) LRU

(b) FSQ

Figure 25: Comparison of the FSQ and LRU replacements on a LRU-friendly trace.

We input a LRU-friendly address trace of x, y, z, D, y, E, z, F, z, y, and F. Figure 25 show both the LRU and the FSQ replacements have the same hits rates. The hit number is five. The FSQ keeps the most frequently accesses block z at the top of the queue while the block z is kept at the second to the bottom of the LRU stack. Table 11 shows recency and frequency information of this LRU-friendly Trace. For example, the distance between the same address F is 3. They are shorter than the length of FSQ queue.

Table 11: Analysis of LRU-friendly Trace

Position	1	2	3	4	5	6	7	8	9	10	11
Incoming Block	x	y	z	D	y	E	z	F	z	y	F
Distance					3		4		2	5	3
Frequency	1			1		1			3	3	2

3.5.3 An example of the Working Set Change Trace

FSQ replacement adapts to changes in the working set by using the rollover operation. An address sequence of E, F, G, H, z, y, x, w, y, w, x represents a working set change as shown in Figure 26. Figure 26(a) shows LRU kicks out the old working set (line E, F, G, and H) through the bottom of LRU stack. Figure 26(b) shows the FSQ replacement moves the same old working set out of the queue through the top of the queue. The order of the evicted line is E, F, G, and H in the LRU, while the FSQ reverses the order. Both LRU and FSQ replacement have the same frequency of cache hit for this trace. Table 12 shows recency and frequency information of this working set change trace.

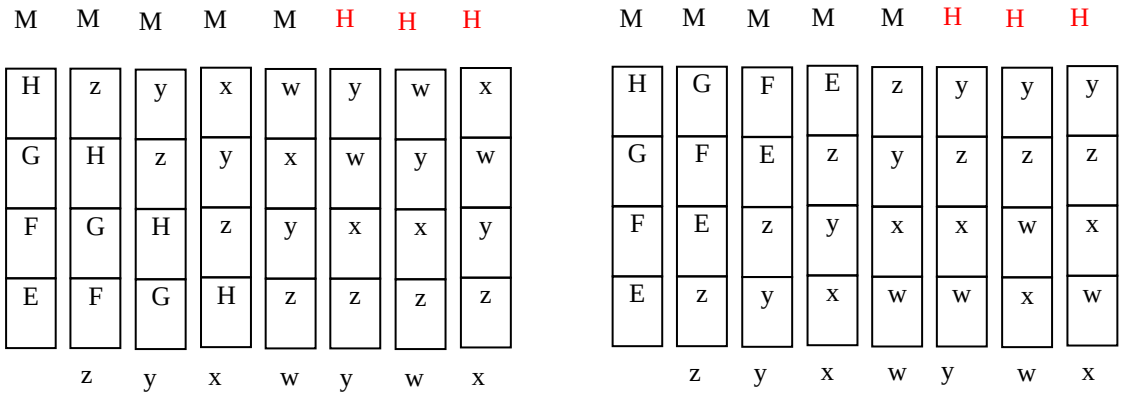


Figure 26: Adaptation to the working set change of LRU and the FSQ replacements.

Table 12: Analysis of Working Set Change Trace

Position	1	2	3	4	5	6	7	8	9	10	11
Incoming Block	E	F	G	H	z	y	x	w	y	w	x
Distance									3	2	4
Frequency									2	2	2

3.6 Problems of the FSQ Replacement

We discuss four unsolved problems in FSQ replacement and investigate how to solve these issues in this subsection.

3.6.1 Competition on Entering the Queue

The insertion policy makes the FSQ replacement too aggressive in evicting blocks that are not used continuously. A LRU-friendly address sequence A, B, C, D, m, C, n, C, m, C, and n shows this problem in Figure 27. The blocks m and n can enter the LRU stack and experience cache hits afterwards. In FSQ replacement, however, the blocks m and n are evicting each other before they have a chance to be promoted in the queue. This problem is caused by the fact that the FSQ queue has only one entry point.

One method to solve this problem is to dynamically record the access pattern and determine the rollover operation adaptively; however, that will need extra hardware. The other method is to use multiple queues instead of one queue. Multiple queues provide multiple entries for the incoming cache lines, so that two consecutive

incoming lines may be inserted into two different queues and the problem shown in the Figure 27 can be solved. Table 13 shows recency and frequency information of trace in this example.

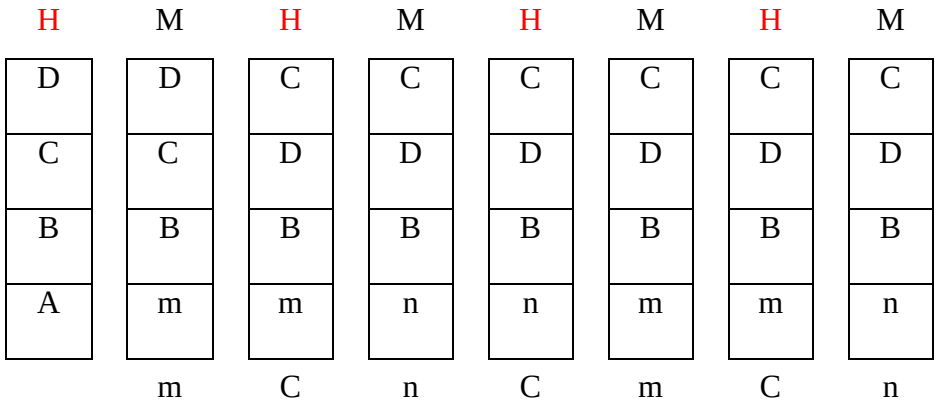


Figure 27: Competition on Entering the Queue

Table 13: Trace of Competition on Entering the Queue

Position	1	2	3	4	5	6	7	8	9	10	11
Incoming Block	A	B	C	D	m	C	n	C	m	C	n
Distance						3		2	4	2	4
Frequency						2		3	2	4	2

3.6.2 The Graduation of the Top Cache Line

To reach the top of the queue, the hit number of a cache line must be equal to or higher than the length of the whole queue. The cache capacity may not be used efficiently by the FSQ replacement due to the following reasons.

First, the cache lines on top of the queue are left unused if the most frequently used cache line exhibits a usage frequency lower than the length of the whole queue. Thus no cache lines can reach the top of the queue. Therefore, the length of the queue should be less than usage frequencies of the lines.

Second, some frequently used cache lines can reach the top, however, when these lines are not used anymore in the near future, the remaining lines may not be used as frequently as those lines and cannot replace those lines that are on top of the queue.

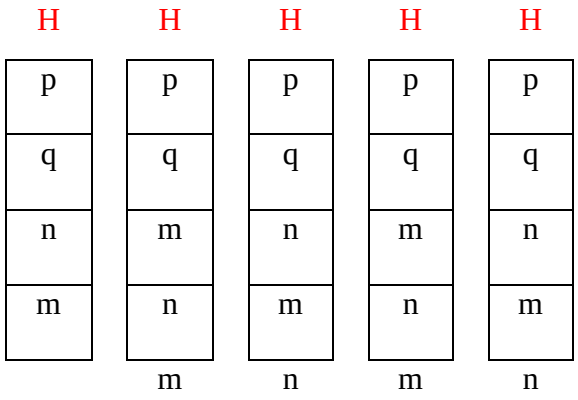


Figure 28: Graduation of the Top Cache Line

Third, no any cache lines can be promoted to top of the queue if two or more frequently used lines are accessed alternatively. For example, an address sequence of m, n, m, n is shown in Figure 28, where the line m is right below the line n in the queue. The two competing lines exchange positions in the queue and cannot make their way to the top of the queue, even if the cache lines p and q have lower access

frequencies than both lines m and n. This is because the queue can only record the relative access frequencies of the cache lines.

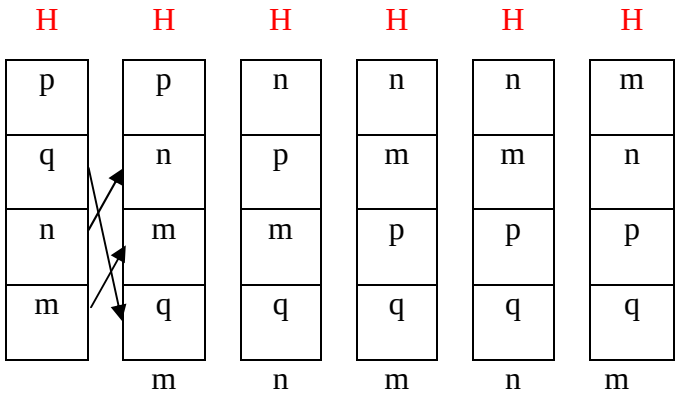


Figure 29: Graduation of the Top Cache Line (Solution 1)

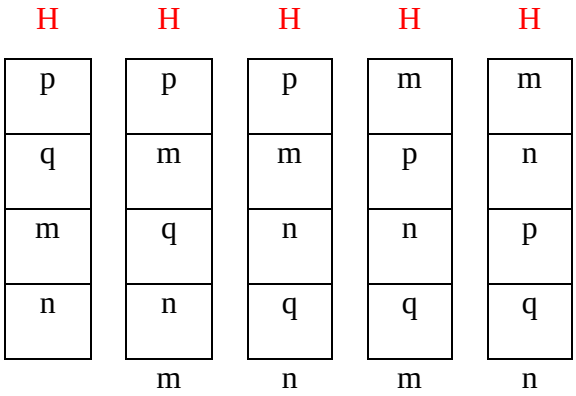


Figure 30: Graduation of the Top Cache Line (Solution 2)

The first method of solving this problem exchanges three lines' position as shown in Figure 29. In this example, line m and n move up one position but line q move down two positions. The other method of solving this problem makes the line m above the line n as shown in Figure 30, and both lines can be promoted to the top after

four cache hits. To break up the competition of these two alternatively accessed address patterns, the replacement can allocate them into different queues, which also need multiple queue techniques. The multiple queue method is easier than the first method.

3.6.3 The Pathological Problem of the FSQ

Figure 31 shows the FSQ replacement suffer from a strict cyclic access pattern: A, B, C, D, m, n, o, p, q, m, n, o, p, q. The FSQ replacement may experience no hit at all as shown in Figure 31. After these lines enter the queue, since there will be no cache hit to the set, the rollover operation moves these lines out of the queue without keeping any of them in the cache.

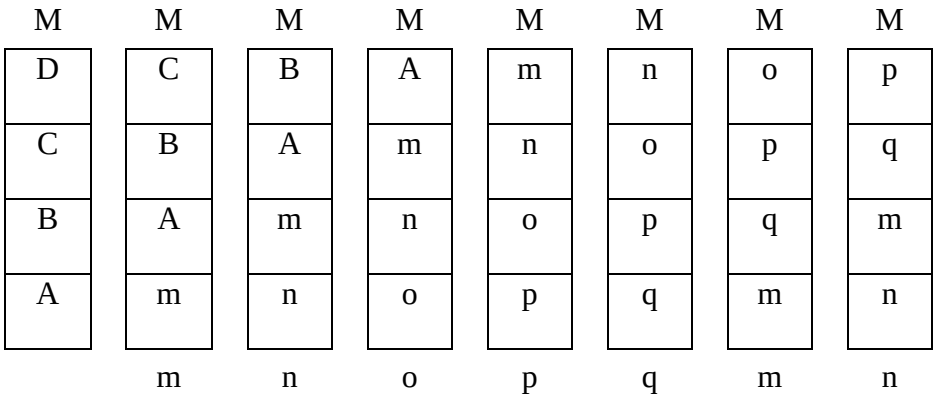


Figure 31: A Strictly Cyclic Pattern

To resolve this issue, the key technique is to break up the cyclic access pattern so that part of the cyclic accessed lines can be retained in the cache. Table 14 shows recency and frequency information of this strictly cyclic pattern.

Table 14: Analysis of a Strictly Cyclic Pattern

Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Incoming Block	A	B	C	D	m	n	o	p	q	m	n	o	p	q
Distance										5	5	5	5	5
Frequency										2	2	2	2	2

3.6.4 Similar Hardware Complexity to the LRU

The FSQ replacement requests a similar hardware complexity to traditional LRU. The bottom block corresponds to LRU block while the top block corresponds to the MRU block in LRU. The replacement also needs to keep track of the status of all the blocks for promotion. For example, the status of all the n cache blocks must be updated when an incoming block is inserted into the MRU position in an n-way LRU cache design. We may use multiple queues to reduce hardware complexity.

3.7 Summary

The FSQ replacement exploits both frequency and recency information for L2 cache, and reduces the MPKI by 2% over traditional 16-way cache with LRU. Our experiments prove the rollover operation is very important to improve performance for most of the benchmarks. The FSQ keeps the frequency information by the rollover operation. But FSQ replacement suffers the following four problems:

- competition on entering the queue
- The graduation of the top cache Line
- The pathological problem of the FSQ replacement
- Similar hardware complexity to the LRU

CHAPTER 4

FREQUENCY BASED MULTIPLE QUEUE (FMQ)

To resolve these problems of the FSQ replacement, we propose a multiple queue scheme to divide the n cache lines in one set into N (2<=N<=n/2, where n presents n-way cache) queues, where each queue implements the FSQ replacement independently. We name the FMQ replacement. The FMQ replacement resolves all the problems of the FSQ replacement with reduced hardware costs and improved performance.

4.1 Modeling

In the FMQ scheme, the proposed FSQ replacement is used independently in each queue. The incoming line is inserted into the selected queue. Then only the status bits of the selected queue are updated while the other N-1 queues remain unchanged.

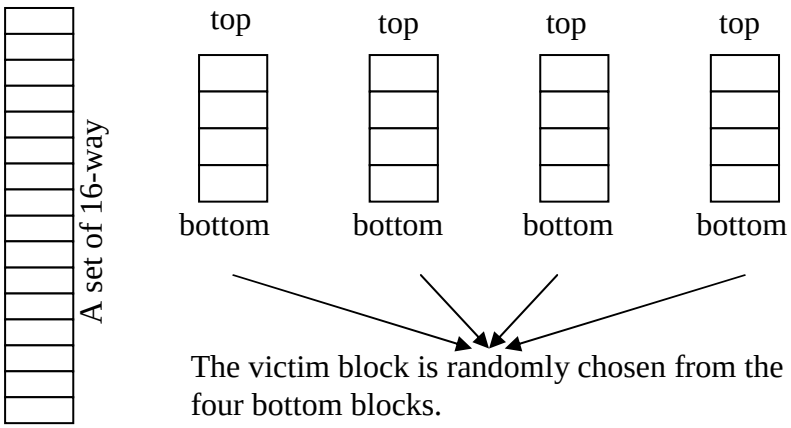


Figure 32: FMQ-4 scheme: A cache set is divided into four queues

Figure 32 shows the proposed FMQ-4 scheme. A 16-way set is divided into four queues. Each queue has four blocks. The victim block is randomly chosen from the victim candidate blocks in these four queues.

Figure 33 shows the proposed FMQ-2 scheme. A 16-way set is divided into 2 queues. Each queue has eight blocks. The victim block is randomly chosen from the victim candidate blocks in these two queues.

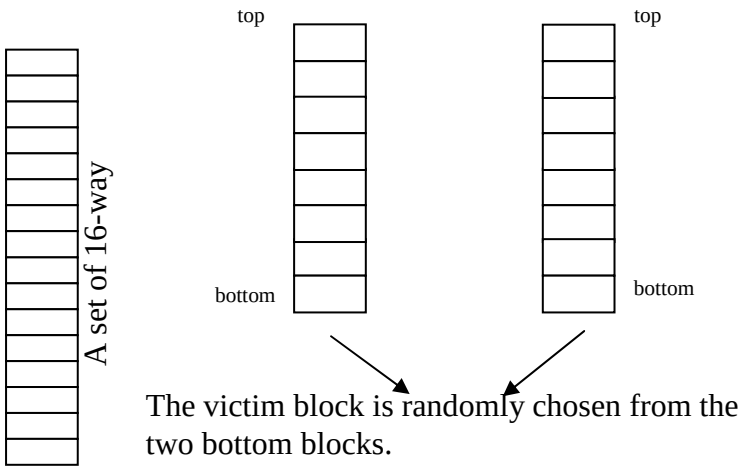


Figure 33: FMQ-2 scheme: a cache set is divided into two queues.

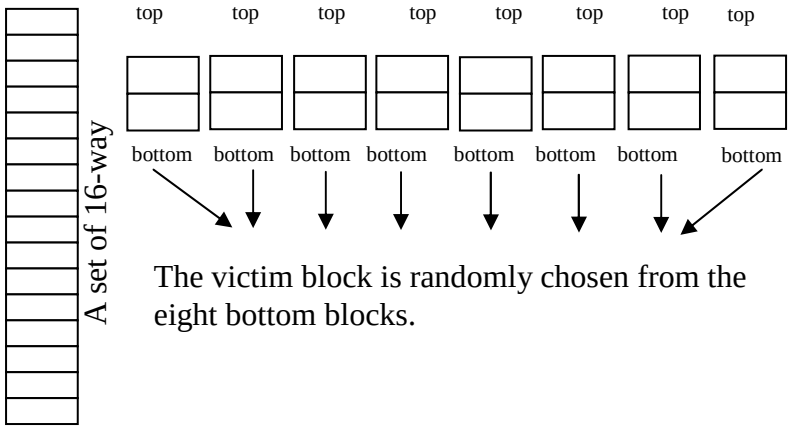


Figure 34: FMQ-8 scheme: A cache set is divided into eight queues

Figure 34 shows the proposed FMQ-8 scheme. A 16-way set is divided into 8 queues. Each queue has two blocks. The victim block is randomly chosen from the victim candidate blocks in these eight queues.

The problem that the FSQ is too competitive upon entering the queue is resolved through inserting arriving lines into separate queues. Therefore, the incoming cache lines will have a chance to be promoted into the queue, since a new incoming line may be inserted into another queue without evicting the last inserted cache line.

The problem of top line graduation is resolved since the length of the queue is significantly reduced. For a 16-line queue, a cache line must be accessed more than 16 times to reach the top. While in the FMQ scheme, a four accessed cache line may get to the top. The possibility that a line cannot be removed from the top of the queue is greatly reduced.

The cyclic access pattern problem is solved since those cyclic accessed lines are allocated to multiple queues and the victim is randomly chosen from these queues. The lines from this cyclic pattern will have a chance to hit, then the rollover operation to this set is stopped and part of the cyclic lines will be retained in the queue.

The circuit complexity is reduced significantly. The storage requirement of the FMQ-4 is equivalent to four times the four-way set associative cache in addition to the set-hit bit per set, which is significantly lower than a 16-way associative cache that uses the LRU replacement. Cache must update the status of all the 16 cache blocks when an incoming block is inserted into the MRU position in a 16-way LRU

cache design. In FMQ-4 scheme, at most 4 blocks in the proposed scheme will be updated.

4.2 Experimental Methodology

Table 15: Baseline System Configuration

Parameter	Setting
L1 I-Cache	16kB; 64B line size; 2-way with LRU replacement
L1 D-Cache	16kB; 64B line size; 2-way with LRU replacement
Baseline L2	1MB; 128B line size; 16-way with LRU replacement
Main Memory	200 cycles unlimited size
Machine width	4 instructions/cycle; 4 functional units
Inst. Window size	32 instructions
Branch Predictor	Hybrid 64K, miss penalty 10 cycles

We measure the cache misses and processor performance by SimpleScalar [11]. Table 15 shows the parameters of the L1 and L2 caches in the system configuration. Our experiments mainly include these parts: First, we compute the miss rate of LRU and FMQ by changing the number of queues of FMQ and configuring 1MB L2 cache as 8-way, 16-way, and 32-way set associativity. Second,

we compute the miss rate of LRU and FMQ by changing the capacity of L2 cache from 1MB to 8MB. Third, we compute the miss rate of LRU and FMQ by changing the cache line size and set associativity. Last, we compute the power consumption of the LRU and proposed FMQ.

4.3 Results

Figure 35 shows the MPKI reductions of the FMQ (FMQ-1, FMQ-2, FMQ- 4, and FMQ-8), DIP, and OPT over traditional LRU, *N* is the number of queues in one cache set of the FMQ replacement. FSQ is the same with FMQ-1. From Figure 35, we made the following observations (the benchmarks' names are marked by the first four letters in this graph).

First, FMQ-1 in the figure outperforms the LRU for seven benchmarks, *equake, sixtrack, facerec, art, mcf, twolf,* and *mesa*, but degrade the MPKI for the other five benchmarks, *lucas, ammp, apsi, vortex,* and *swim*.

Second, the number of queues of the FMQ that shows the highest MPKI reduction is benchmark dependent. To achieve the highest MPKI reduction, a scheme that can dynamically determine the number of queues is desirable, but the hardware design complexity will be the same with LRU, since FMQ-1 replacement is the best for some benchmarks.

Next, to achieve MPKI reductions with reduced hardware design complexity, the FMQ with 4 queues outperforms other FMQ schemes with an average MPKI

reduction of 12% over the LRU. A four group FMQ has the design complexity of 4-way LRU plus the set-hit bit, which is 512 bits (equals to the number of sets) in our baseline.

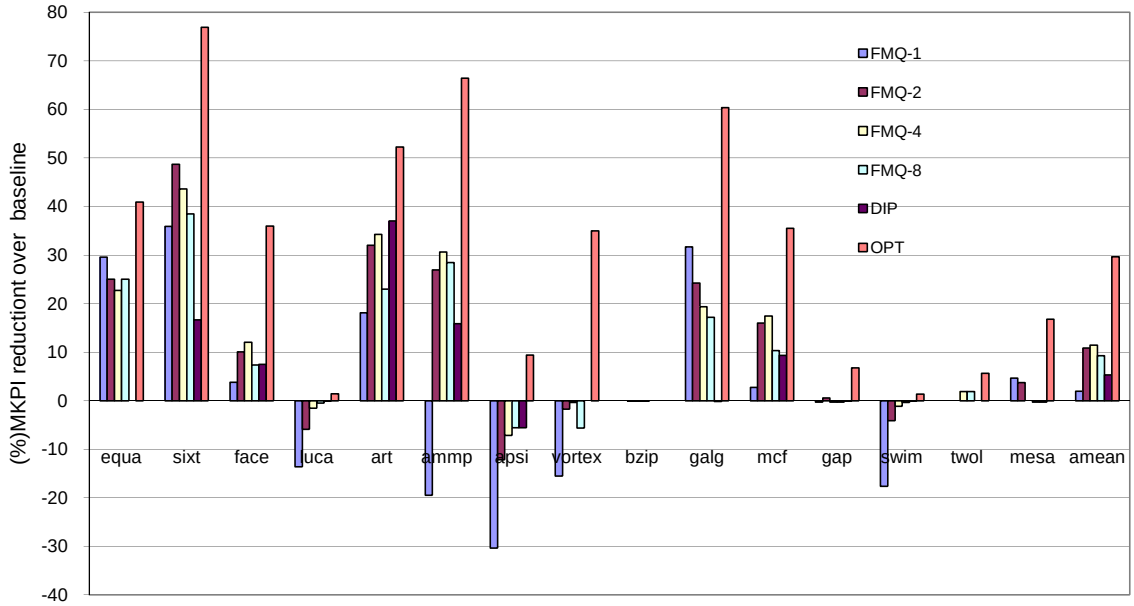


Figure 35: The MPKI (%) reduction of the FMQ (FMQ-N), DIP and OPT over the baseline LRU

Last, the OPT results are also shown and the OPT outperforms the LRU on average 30%. The proposed FMQ close 40% of the gap between the OPT and the LRU with a reduced storage requirement and circuits complexity. The performance improvement by using the FMQ is up to 44%. (The FMQ will not change the access cycle of the L2 cache). The processor is configured as a four issue out order processor with a memory access.

We provide case studies on the access behaviors of the benchmarks for the FSQ and FMQ replacement in chapter 5. The experimental results in performance evaluation are presented in Chapter6.

CHAPTER 5

CASE STUDIES OF THE BENCHMARKS

The purpose of this chapter is two-fold. First, we show that we can reduce the cache misses using FMQ and FSQ. Second, we show how the proposed FMQ reduces the cache misses, and which access behaviors were likely to improve performance. It is different from previous research in performance evaluation of cache replacement policies for the benchmarks [3][10][34][40]. From a fresh perspective in access behaviors, recency information, and frequency information, we describe a study where we used the FMQ and FSQ to tune the cache performance of five programs from the SPEC200 benchmark suite: *ammp*, *art*, mcf, *galgel*, and *swim.*

The performance of the benchmarks can be generally categorized into the following groups:

Group 1: FSQ↑ > FMQ↑

Both the FSQ and FMQ outperform the LRU. The MKPI reduction of the FMQ over the baseline LRU consistently reduces as the number of queues increases.

Group 2: FSQ↑ < FMQ↑

Both the FSQ and the FMQ outperform the baseline LRU and the FMQ outperforms the FSQ replacement

Group 3: FSQ ↓< FMQ↑

The FSQ replacement degrades the MPKI while the FMQ improves the MPKI over the baseline LRU.

Group 4: FSQ ↓< FMQ→ or FSQ ↓< FMQ↓ or FSQ →≈ FMQ→

The FMQ achieves similar MPKI as the baseline LRU, while the FSQ replacement degrades the MPKI. Or both the FSQ and the FMQ achieve similar MPKI as the baseline LRU.

5.1 The Benchmark galgel

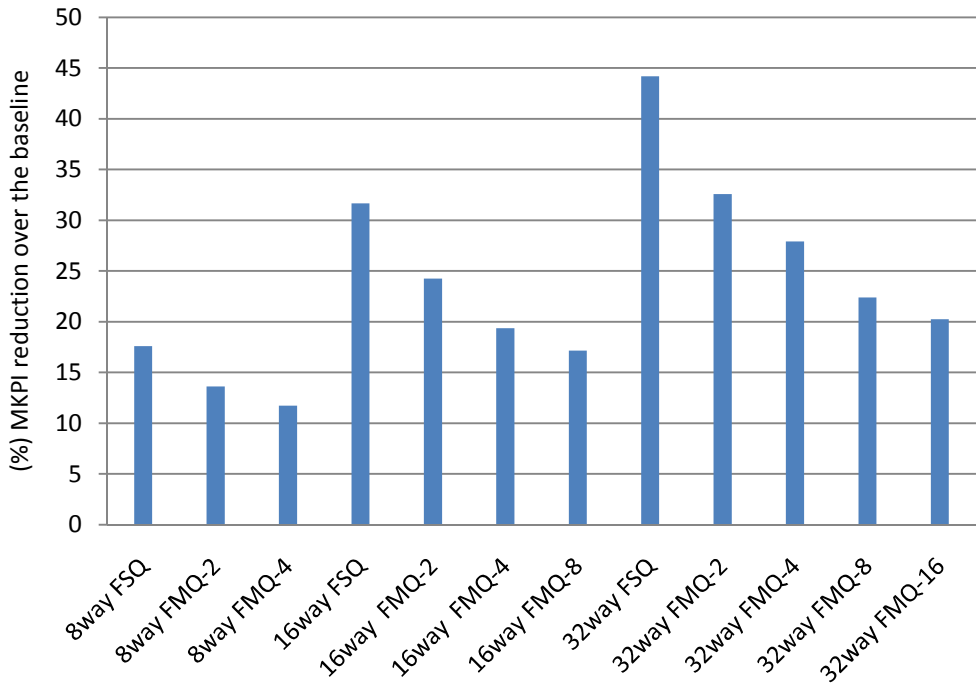


Figure 36: The percentage MPKI reduction of the FSQ and the FMQ replacement over LRU in 1M cache (Benchmark *galgel*)

In group 1, we analyze the Benchmark galgel. The benchmark galgel is a Fortran 90 program for computational fluid dynamics. Figure 36 illustrates both FSQ and FMQ are better than LRU for the benchmark *galgel*. The FSQ has the best performance among these three replacements in 1M L2 cache. The MKPI reduction of the FMQ over the baseline LRU consistently reduces as the number of queues increases.

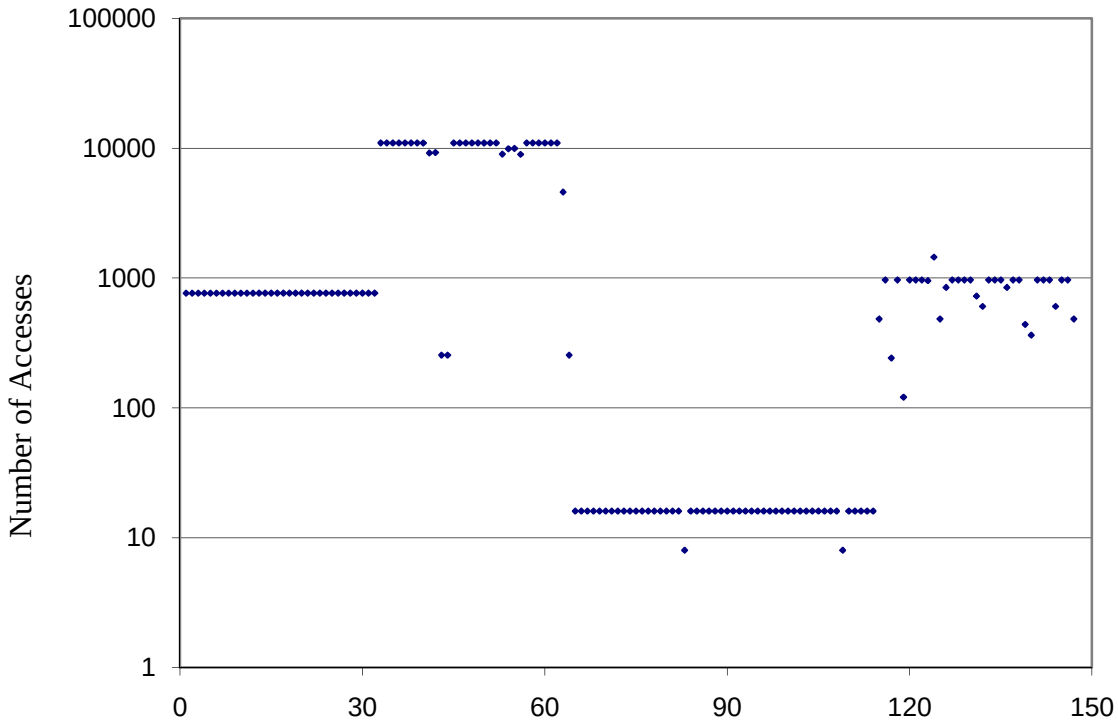


Figure 37: The address access frequencies of the benchmark *galgel*

To understand the behavior of the FMQ on the benchmark *galgel*, we collect the address access trace and show them in three figures. Figure 38 presents a sample of 2000 consecutive accesses to the cache set. The horizontal axis and the vertical axis represent the access sequence of the addresses and the address value, respectively. Some of the addresses are heavily accessed while others are less frequently accessed. The address access frequencies are shown in Figure 37. There are about 150 different addresses accessed in this cache set. The number of frequently used addresses (FUA) is 28. The frequencies of FUA are more than ten thousand. The remainder addresses are accessed less frequently and their access frequencies are between 10 and 1000. Figure 39 shows a zoom in snapshot of the Figure 38.

First, we explain why the FSQ replacement is better than the FMQ. From Figure 39, the access behavior of the benchmark *galgel* follows the pattern of A, A, A, A, A, A, A, B, A, B, A, B, A, B, A, C, C, B, A, B, A, ……, C, B, B, B, D and D. From this access sequence, the lines A, B, and C can easily enter the queue and will not suffer the first problem (competition on entering the queue) of the FSQ. The access frequencies of the heavily accessed cache lines (larger than 10000) are significantly higher than the 16-way cache, therefore the FSQ will not suffer the second problem of the FSQ (top line graduation). There are no cyclic access patterns. Therefore, the FSQ is better than the FMQ.

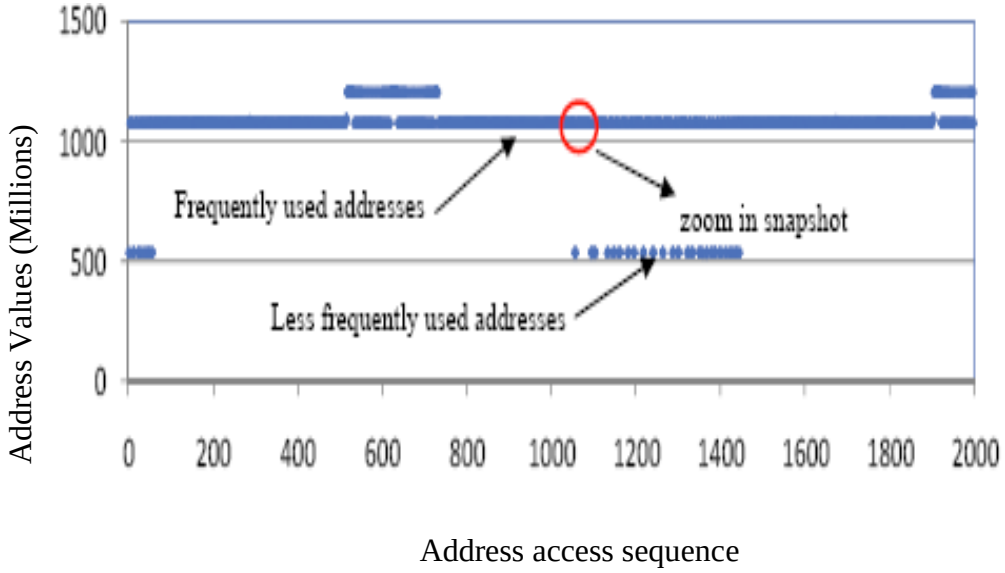


Figure 38: The address access pattern of the benchmark *galgel* in one cache set

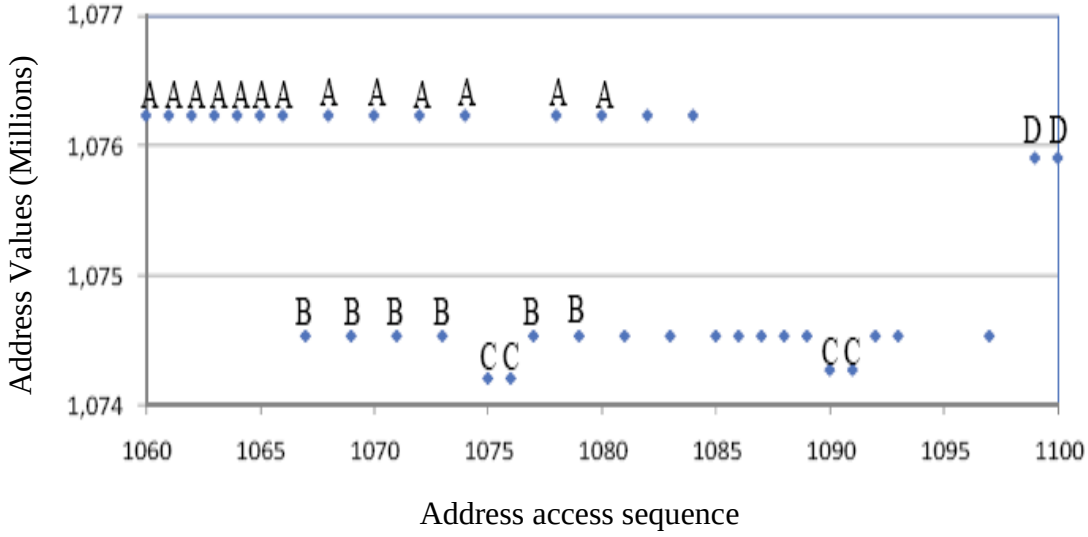


Figure 39: A zoom in snapshot of the address access pattern of the benchmark *galgel*

Second, the MPKI reduction of the FMQ consistently decreases when the number of queues increases. The victim of the FMQ is chosen from multiple queues and each queue has a shorter length when the number of queues increases. The FMQ may choose a frequently used line as a victim and make the FMQ perform poorly as the number of queues increases.

Last, both FSQ and FMQ replacements perform better than the LRU since there are frequently used lines that are mixed with less frequently lines as shown in Figure 38. These less frequently used lines will evict those frequently used lines and degrade the performance of the LRU. From this example, we can prove that recording frequency information in a cache replacement is important.

5.2 The Benchmark art and mcf

In group 2, we analyze the Benchmark *art* and *mcf*. The benchmark art is a C program for image recognition/neural networks application. Figure 40 shows FSQ↑ < FMQ↑ for the benchmark art. The FMQ has the best performance (up to 40% in 32way FMQ-4). Figure 41 shows a sample of 1000 consecutive accesses to the cache set. There are two cyclic access patterns and other irregular addresses as shown in the figure. Figure 42 shows a zoom in snapshot of the cyclic pattern-2. The cyclic pattern-2 includes 20 different addresses and will cause cache thrashing in the baseline of a 16-way cache.

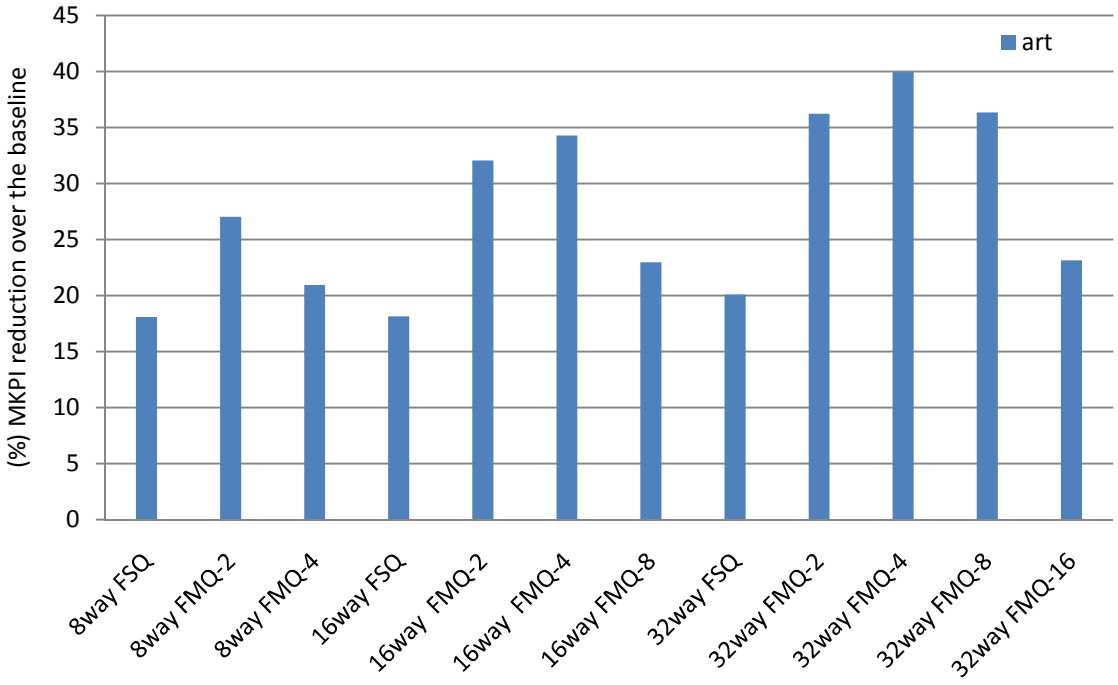


Figure 40: The percentage MPKI reduction of the FSQ and the FMQ replacement over LRU in 1M cache (Benchmark *art*)

The FMQ outperforms the FSQ replacement since the two cyclic access patterns are interspersed with each other. Using one queue will cause the two cyclic access patterns to evict each other. Part of the two cyclic access patterns can be kept in the cache set when multiple queues are used since these two cyclic access addresses are randomly allocated into different queues; therefore, the cyclic access patterns seen

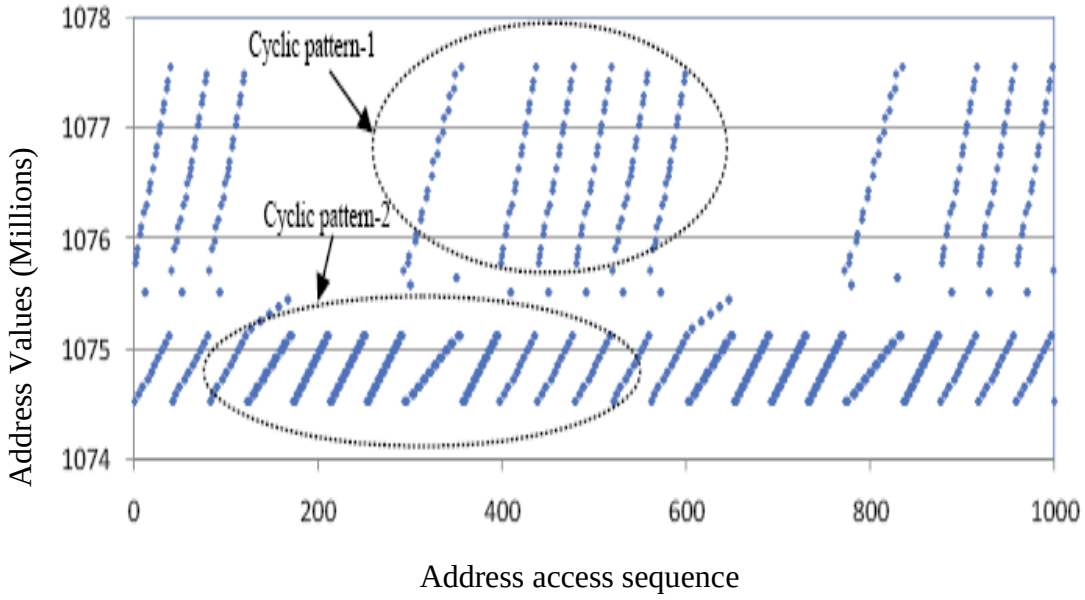


Figure 41: The address access pattern of the benchmark *art* in one cache set

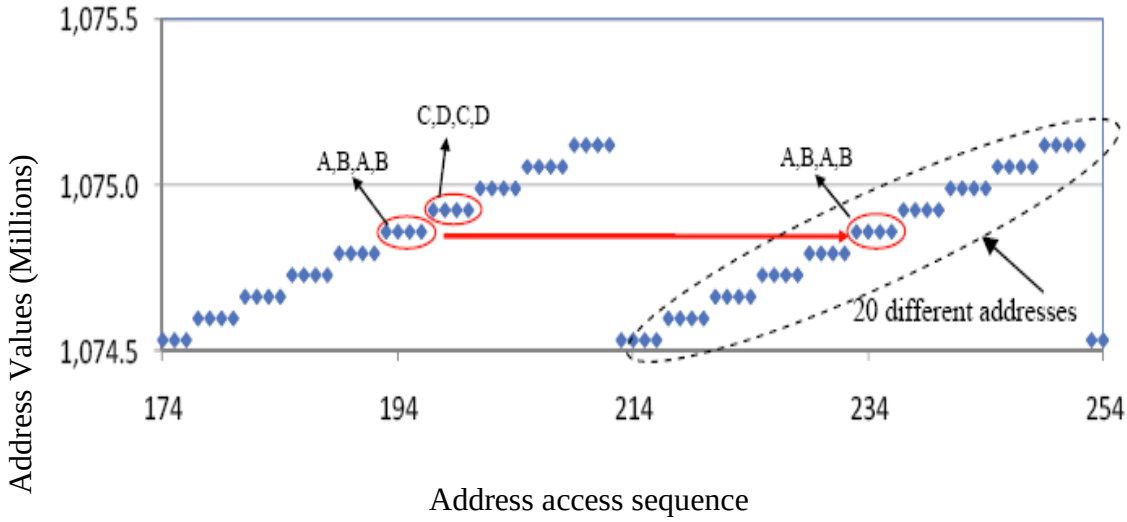


Figure 42: A zoom in snapshot of the address access pattern of the benchmark *art*

by each particular queue is broken up. This will allow the FMQ to avoid the cache thrashing problem caused by the cyclic access patterns.

The FMQ-4 achieves the highest MPKI reduction. The MPKI reduction is reduced in the FMQ-8. The frequency information cannot be fully exploited because there are only two elements in every queue.

The benchmark *mcf* is a C program for combinatorial optimization. Figure 43 shows the address access patterns of benchmark *mcf* in one cache set. We show the addresses access behaviors of one cache set since LRU and the proposed FQM are all local replacements and are performed at each cache set independently. There are both frequently and less frequently used addresses. On average, the frequently used addresses account for 80% of the total accesses. Specifically, there are 12 frequently used addresses and more than 200 less frequently used addresses in this set. The accesses of those frequently used addresses are mixed with less frequently used addresses over time.

Figure 44 shows a zoom in snapshot of one hundred accesses of addresses in the benchmark *mcf*. From this figure, we can see that those frequently used addresses will be mostly evicted by those less frequently used addresses. FMQ can account for the frequency information so that those frequently used addresses are kept, so the miss rate can be reduced.

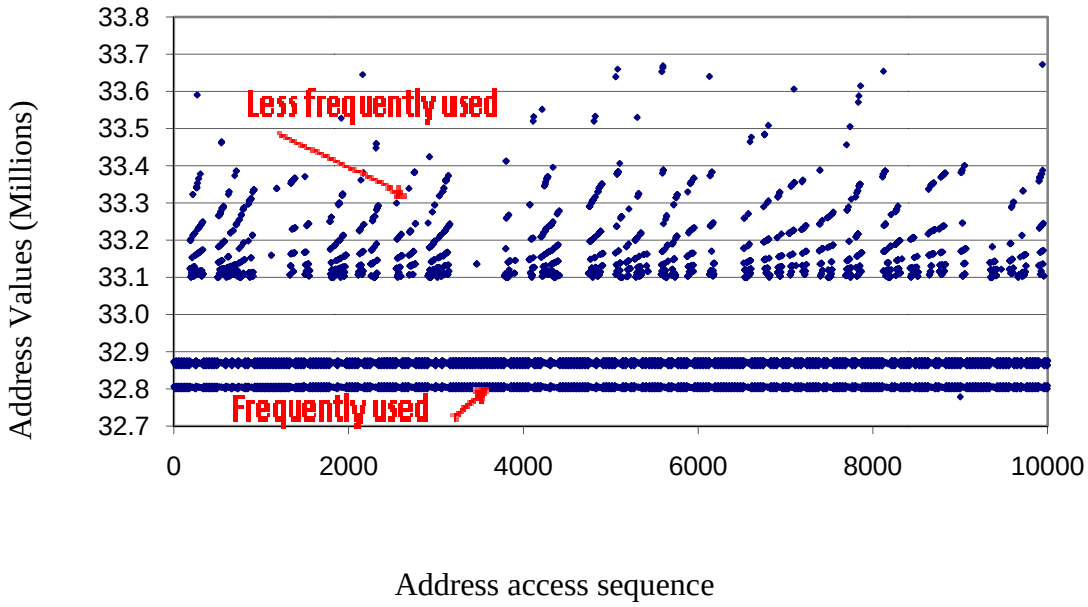


Figure 43: The address access pattern of the benchmark *mcf* in one cache set

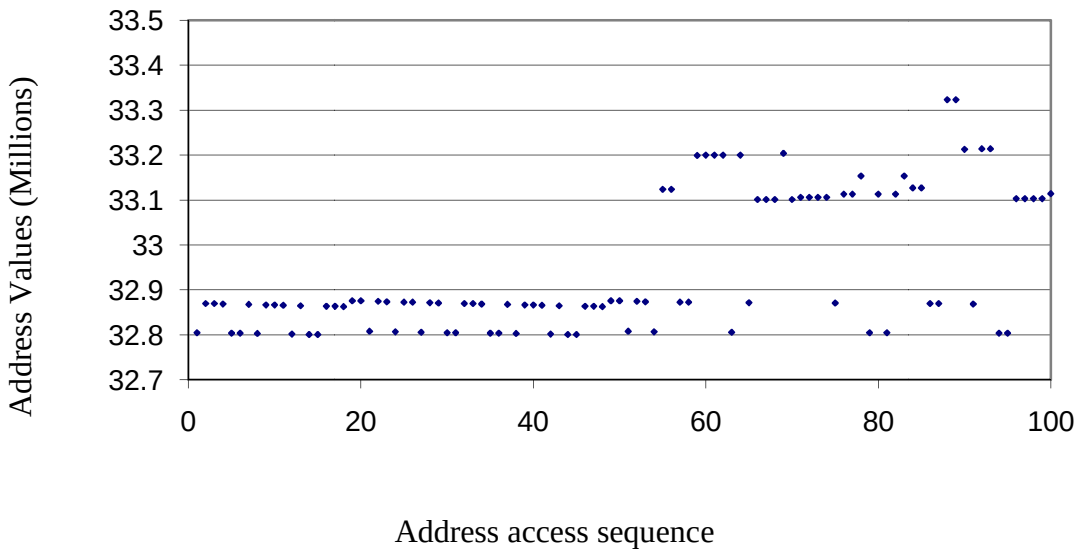


Figure 44: A zoom in snapshot of the address access pattern of the benchmark *mcf*

Figure 45 shows FSQ↑ < FMQ↑ for the benchmark *mcf*, as a result the FMQ has the best performance among these three replacements.

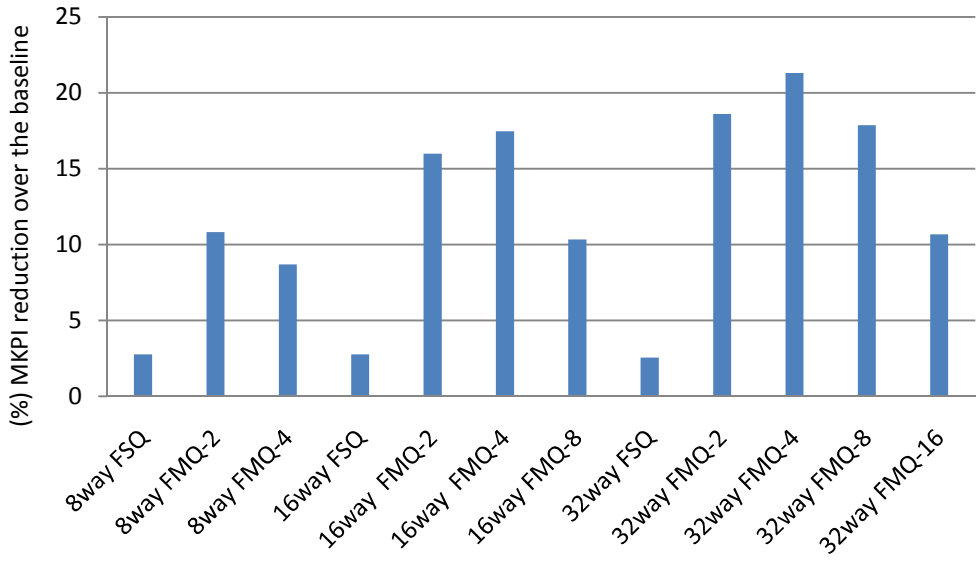


Figure 45: The percentage MPKI reduction of the FSQ and the FMQ replacement over LRU in 1M cache (Benchmark *mcf*)

The benchmark *facerec* is a Fortran 90 program for image processing and face recognition. The benchmark *facerec* exhibits similar access patterns with benchmark art. Figure 46 shows both the FSQ and the FMQ outperform the baseline LRU and the FMQ outperforms the FSQ replacement for the benchmark *facerec* (except 8way FMQ-4).

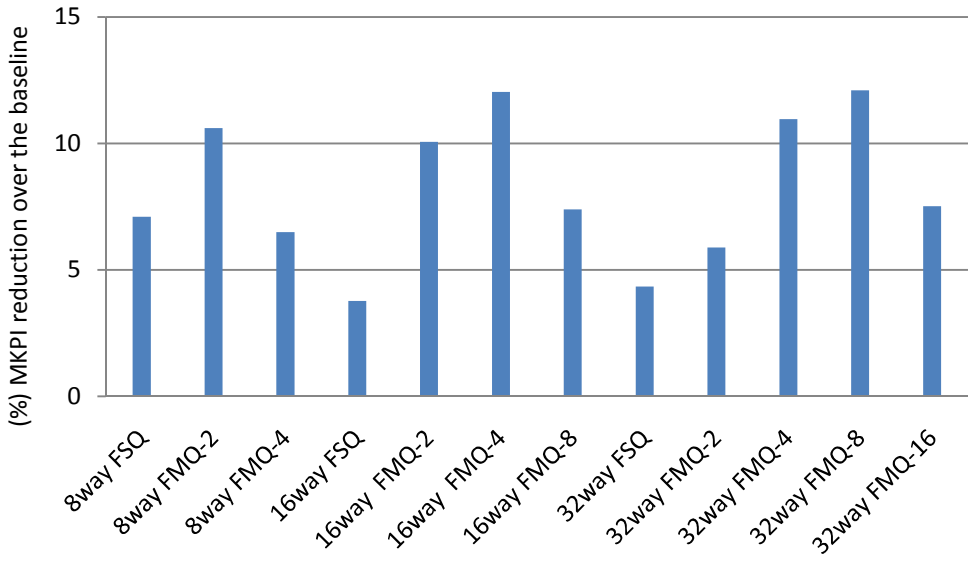


Figure 46: The percentage MPKI reduction of the FSQ and the FMQ replacement over LRU in 1M cache (Benchmark *facerec*)

5.3 The Benchmark ammp

In group 3, we analyze the benchmark *ammp*. The benchmark *ammp* is a C program for computational chemistry. Figure 47 shows FSQ ↓< FMQ↑ for the benchmark ammp. The performance of LRU is better than FSQ.

Figure 48 shows the address access pattern of the benchmark ammp. There are significant frequently accessed addresses mixed with less frequently used ones. Figure 49 is a zoom in snapshot of the more frequently accessed addresses. The addresses accessed follow the following pattern: A, B, C, D, E, A, B, C, D, E, A, C, D, E. This access sequence explains that the FSQ replacement degrades the MPKI over the baseline LRU since the lines cannot reach the top of the queue (More detail about the

graduation of the top cache line is described in section 3.6.2). These lines hit alternatively in Figure 49. Allocating the cache lines to multiple queues randomly can break up this access pattern seen by each individual queue and the cache lines can reach the top of the queue. The FMQ-4 replacement achieves the highest MPKI reduction. The FMQ can filter out those less frequently used cache lines and avoid the potential cache pollutions caused by those less frequently used lines. Those frequently used lines are stored in the queue.

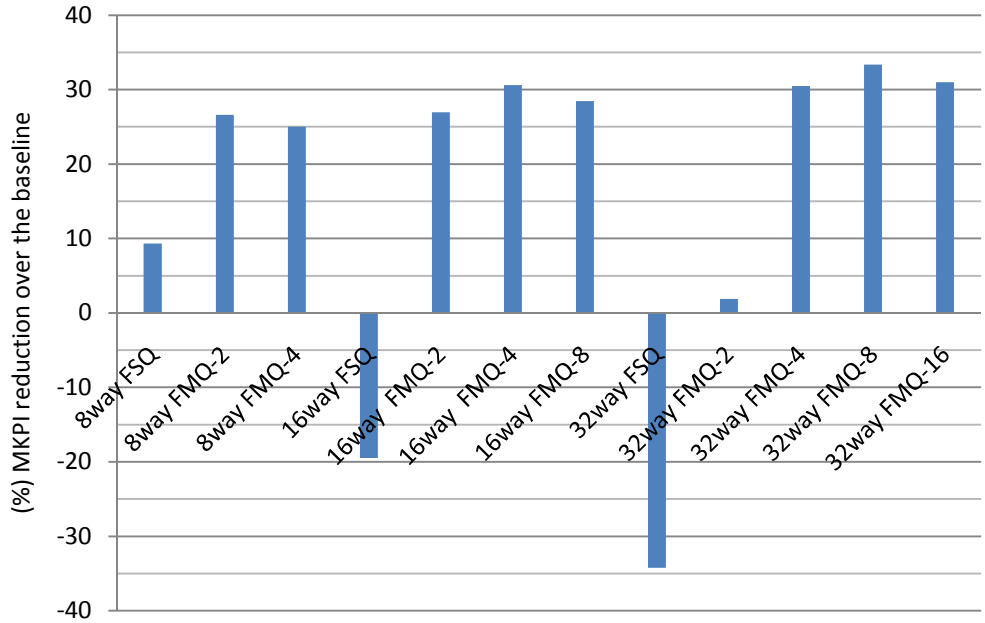


Figure 47: The percentage MPKI reduction of the FSQ and the FMQ replacement over LRU in 1M cache (Benchmark *ammp*)

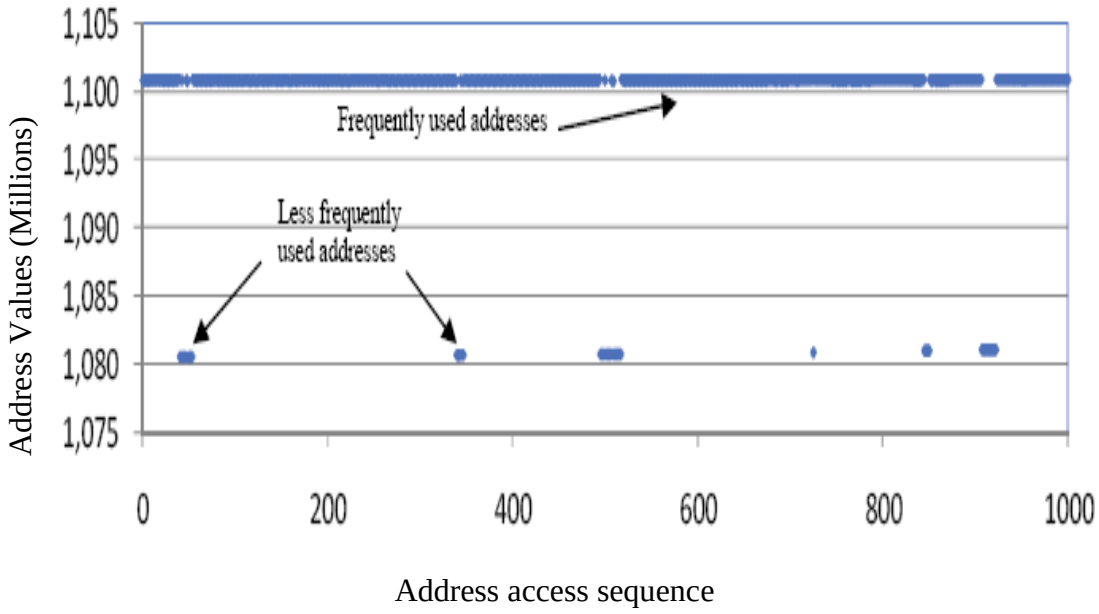


Figure 48: The address access pattern of the benchmark *ammp* in one cache set

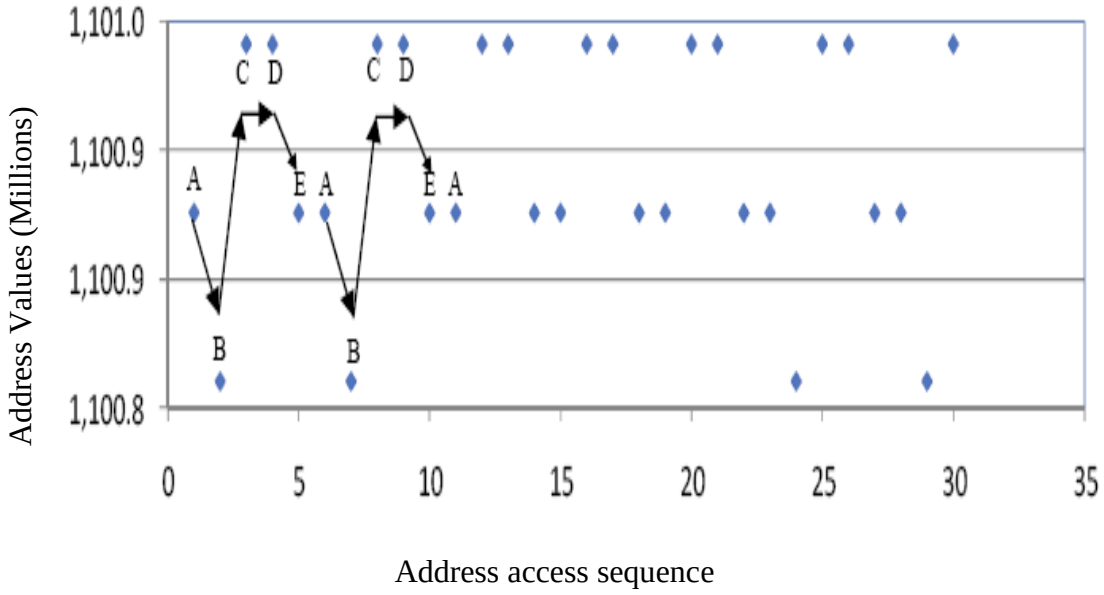


Figure 49: A zoom in snapshot of the address access pattern of the benchmark *ammp*

5.4 The Benchmark swim

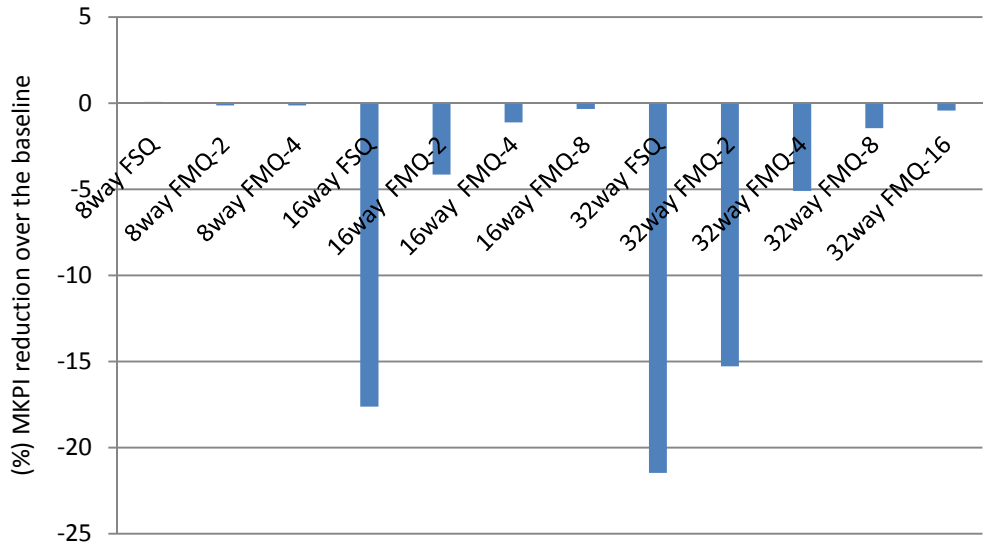


Figure 50: The percentage MPKI reduction of the FSQ and the FMQ replacement over LRU in 1M cache (Benchmark *swim*)

In group 4, we analyze the benchmark swim. The benchmark swim is Fortran 77 program for shallow water modelling. Figure 50 shows FSQ ↓< FMQ→ for the benchmark swim. There are no cyclic access patterns and frequently used addresses in the benchmark *swim* shown in Figure 51. Figure 52 is a zoom in snapshot of the more frequently accessed addresses.

Figure 53 shows the address access frequency of the benchmark *swim*. Most of the addresses are just used four times only. Therefore, the FSQ cannot exploit the whole cache capacity since the access frequencies of the addresses are too low and the

addresses cannot reach the top of the queue. Furthermore, since the lines compete for the single entry of the FSQ queue, the FSQ replacement is worse than the LRU.

Other benchmarks, *lucas*, *vortex*, *bzip2*, *gap*, *twolf*, *apsi* and *mesa* exhibit similar access patterns. There are neither cyclic access patterns nor frequently used addresses in these benchmarks. So the FMQ of these benchmarks achieves the similar MKPI compared to the baseline LRU. Figure 54, Figure 55, Figure 56, Figure 57, Figure 58, Figure 59, and Figure 60 show the results of the FSQ and FMQ in these benchmarks.

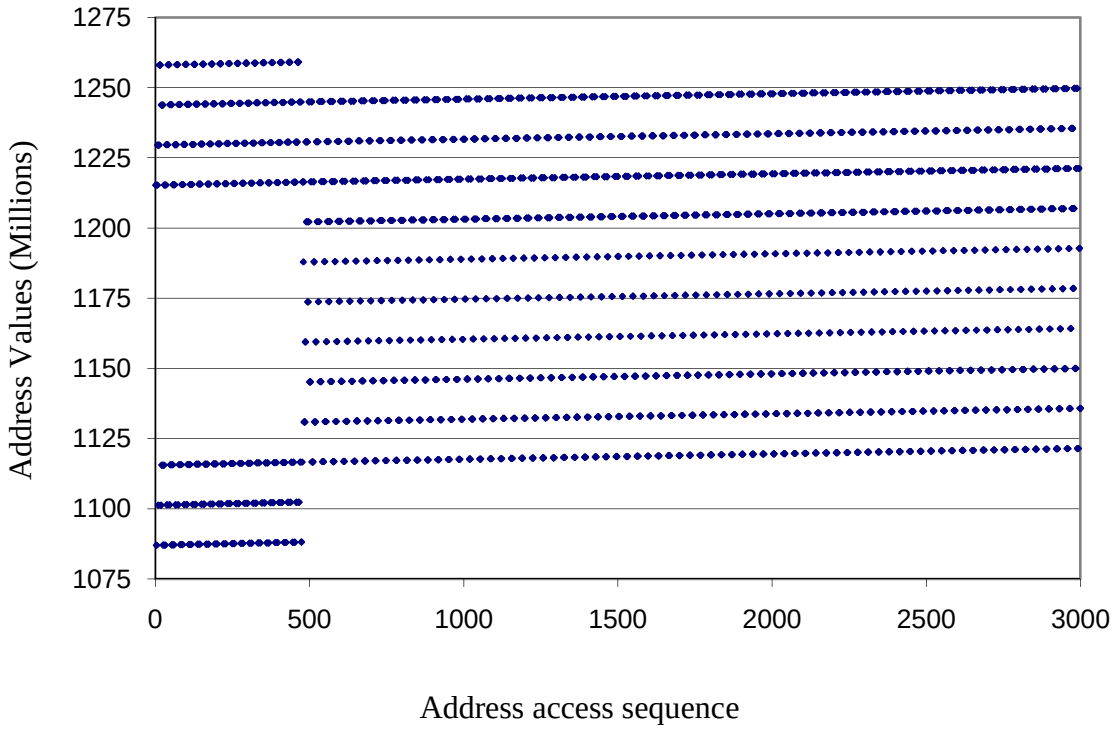


Figure 51: The address access pattern of the benchmark swim in one cache set

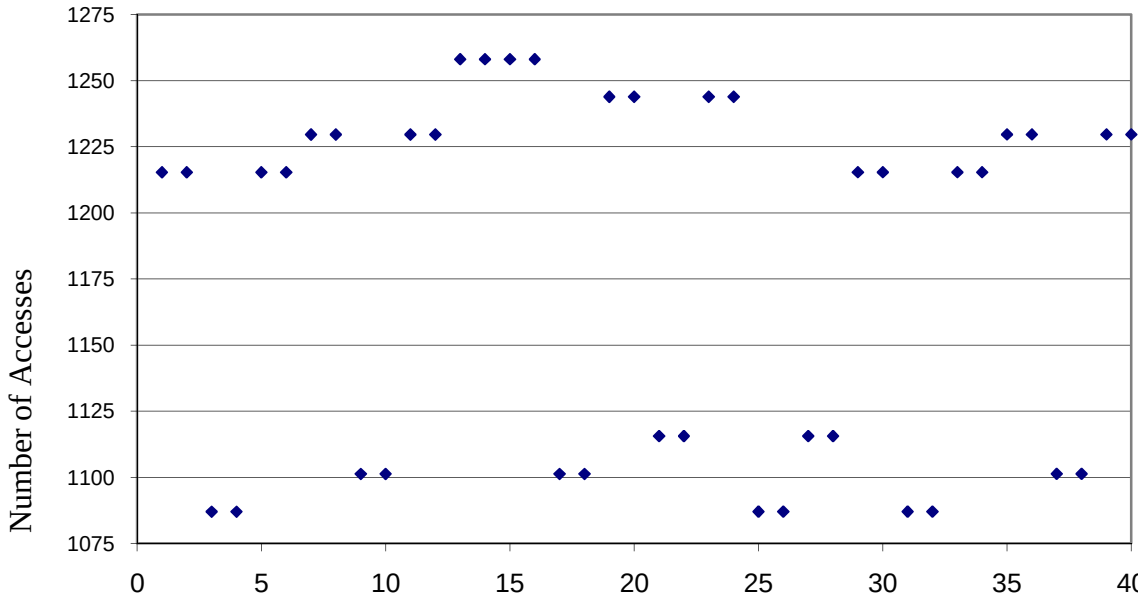


Figure 52: A zoom in snapshot of the address access pattern of the benchmark swim

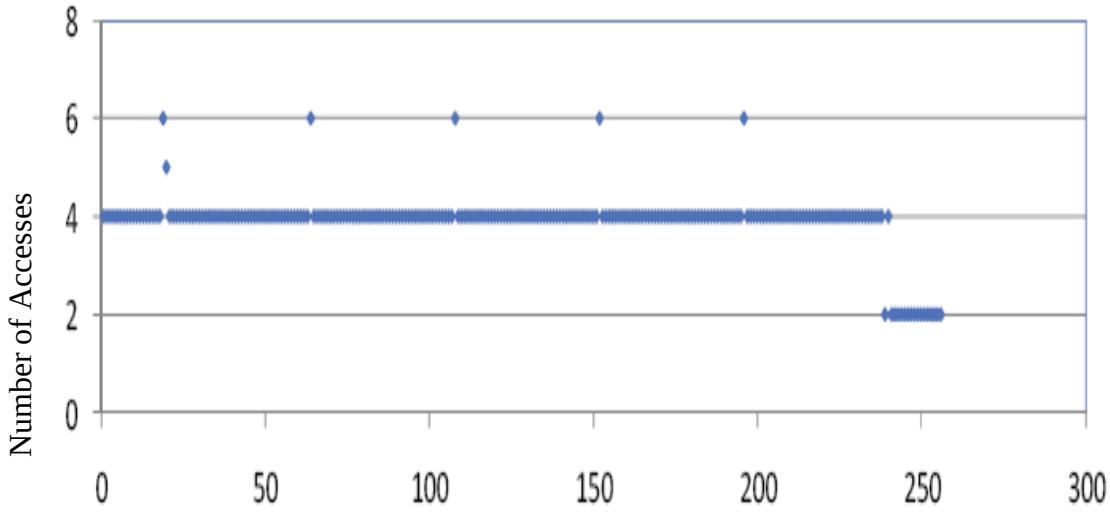


Figure 53: The address access frequencies of the benchmark *swim*

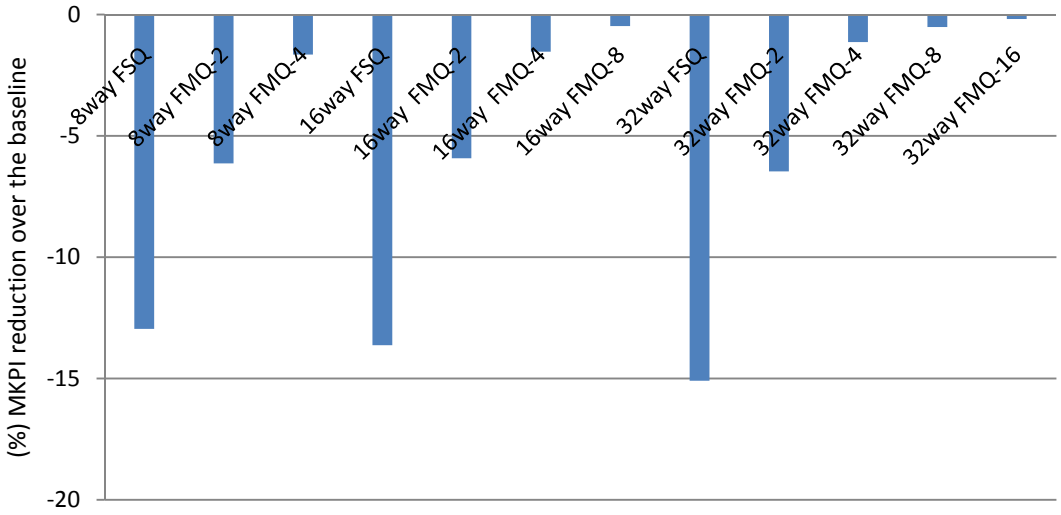


Figure 54: The percentage MPKI reduction of the FSQ and the FMQ replacement over LRU in 1M cache (Benchmark *lucas*)

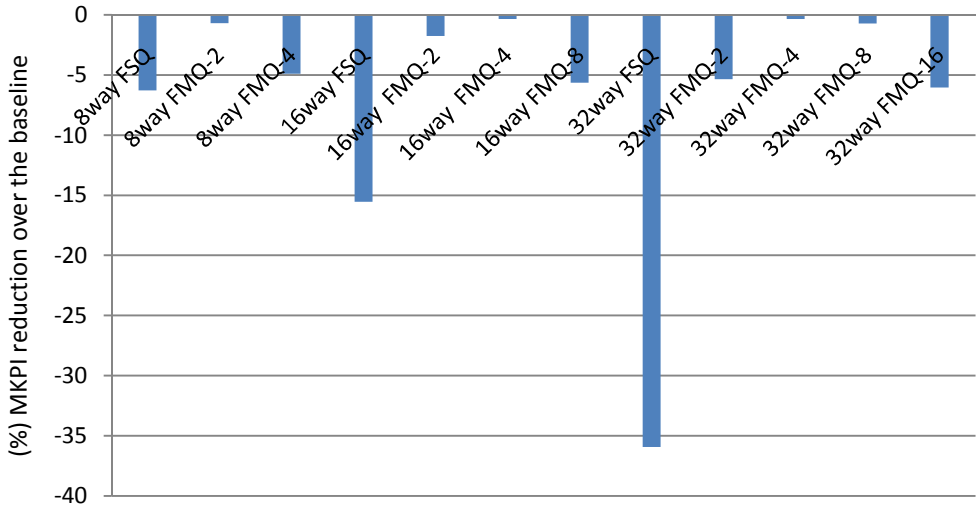


Figure 55: The percentage MPKI reduction of the FSQ and the FMQ replacement over LRU in 1M cache (Benchmark *vortex*)

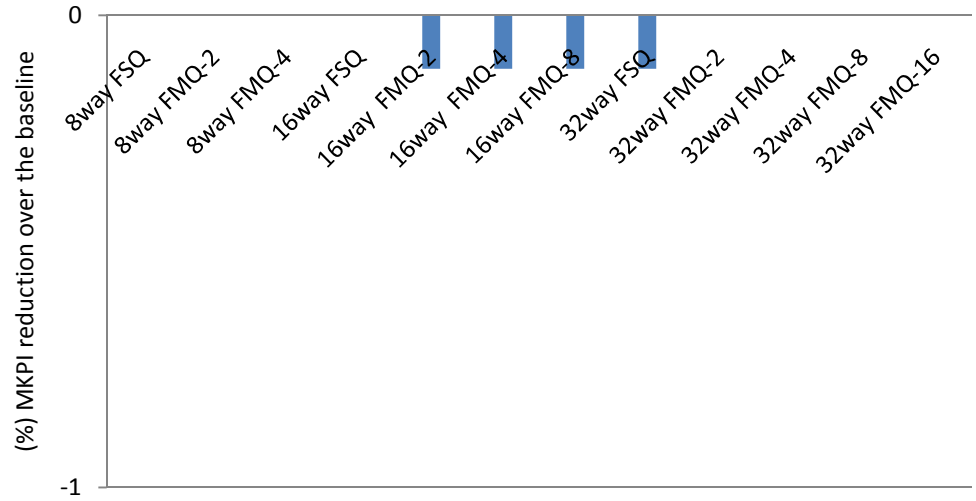


Figure 56: The percentage MPKI reduction of the FSQ and the FMQ replacement over LRU in 1M cache (Benchmark *bzip2*)

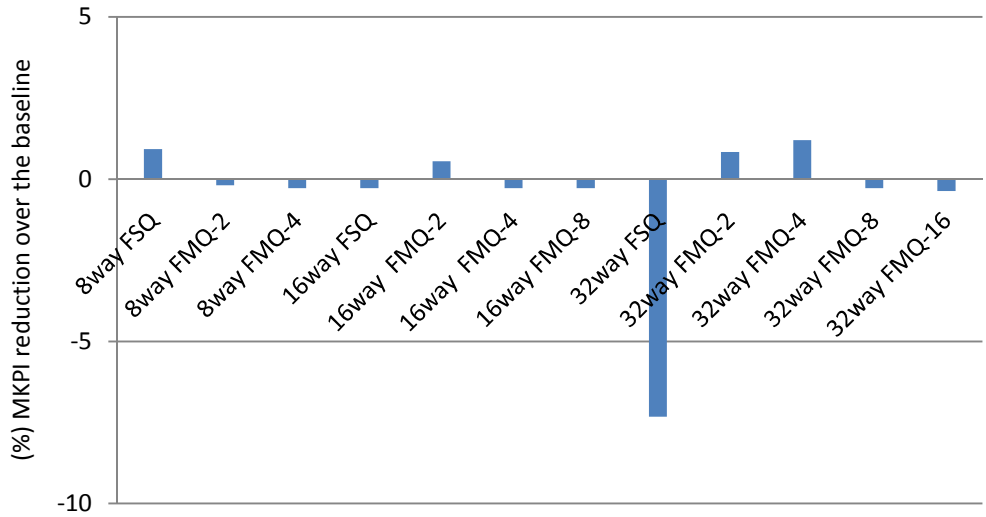


Figure 57: The percentage MPKI reduction of the FSQ and the FMQ replacement over LRU in 1M cache (Benchmark *gap*)

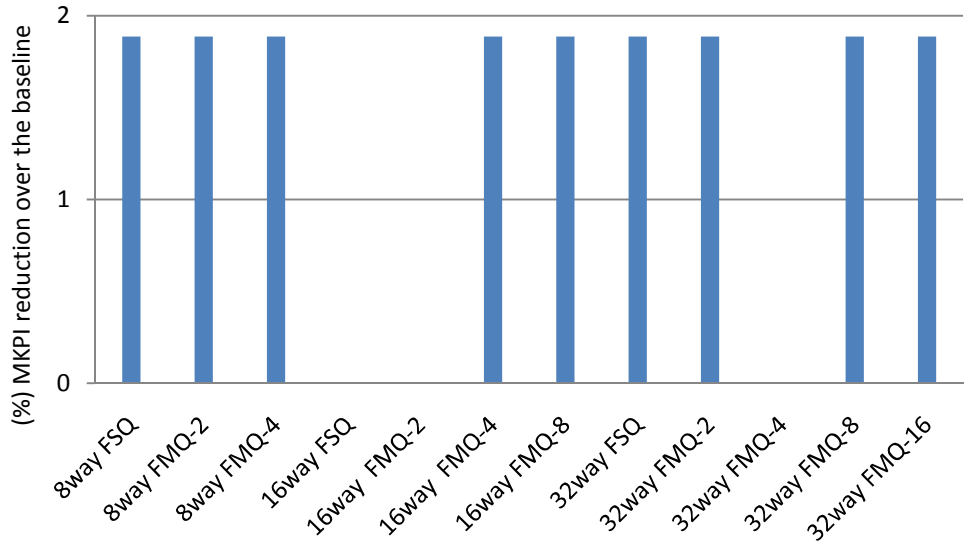


Figure 58: The percentage MPKI reduction of the FSQ and the FMQ replacement over LRU in 1M cache (Benchmark *twolf*)

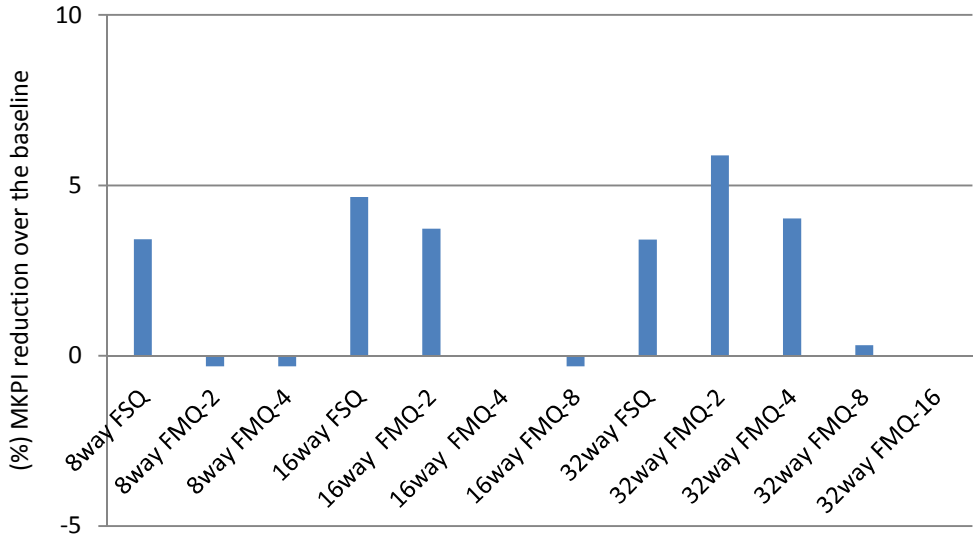


Figure 59: The percentage MPKI reduction of the FSQ and the FMQ replacement over LRU in 1M cache (Benchmark *mesa*)

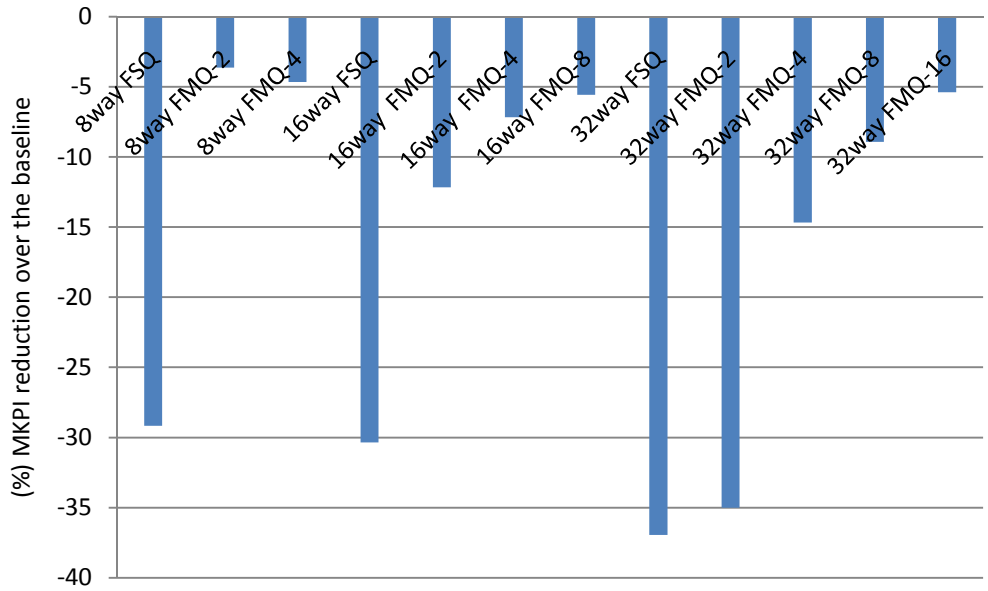


Figure 60: The percentage MPKI reduction of the FSQ and the FMQ replacement over LRU in 1M cache (Benchmark *apsi*)

5.5 Summary

To investigate the relation between cache replacement policy and the access behaviors of program, we categorize the SPEK2K benchmarks into four groups in this chapter. The impact of the access behaviors of the SPEK2K benchmarks is analyzed in each group. Cyclic access patterns and frequency information affect the performance of the FMQ. Table 16 summarizes the feature of the access behaviors in these four groups.

Table 16: Access Behaviors in SPEC2K Benchmark

Group	Performance	Benchmark	Access Behaviors
1	FSQ↑ > FMQ↑	*galgel*	No cyclic access patterns; frequently used lines are mixed with less frequently lines
2	FSQ↑ < FMQ↑	*art, mcf, facerec*	Cyclic access patterns are interweaved; frequently used lines are mixed with less frequently lines
3	FSQ ↓< FMQ↑	*ammp*	Significant frequently used lines are mixed with less frequently lines; regular pattern
4	FSQ ↓< FMQ→ or FSQ ↓< FMQ↓ or FSQ → ≈ FMQ→	*swim, lucas, vortex, bzip2, gap, twolf, mesa, apsi*	No cyclic access patterns and frequently used addresses

CHAPTER 6

PERFORMANCE ANALYSIS

This chapter examines the impact of cache performance according to four major parameters: cache capacity, line size (block size), set associativity, and the number of queues. The different cache parameters in the proposed FMQ replacement that we examined are:

Cache capacity: 1MB, 2MB, 4MB, 8MB

Associativity: 8-way, 16-way, 32-way

Line size: 64 bytes, 128 bytes

Queues: 1, 2, 4, 8

We implement the FMQ and LRU replacements in hardware using the Cadence layout tools. We will also analyze power, area, and circuit complexity reductions of the FMQ over the baseline LRU.

6.1 Impact of Cache Capacity

We analyze below the impact of the cache capacity from 1MB to 8MB. Figure 61 shows the MPKI of the baseline LRU and the FMQ for four cache sizes: 1MB, 2MB, 4MB, and 8MB with the constant associativity of 16-way. The MPKI values are shown relative to the baseline 1MB cache using the LRU replacement. The bar labeled with *amean* represents the arithmetic mean of the MPKI measured over all the

fifteen benchmarks. The FMQ reduces the MPKI more than doubling the size of the baseline 1MB cache for the benchmark *mcf*. The FMQ continues to reduce the MPKI for benchmarks that benefit from increased capacity. The working set of some benchmarks, e.g. *sixtract* and *twolf*, fits in a 2MB cache. Neither the LRU nor the FMQ reduces the MPKI when the working set of these benchmarks fits in the cache. Overall, the FMQ reduces the average MPKI over the LRU for cache sizes up to 4MB because the working set for most of benchmarks is less than 4MB.

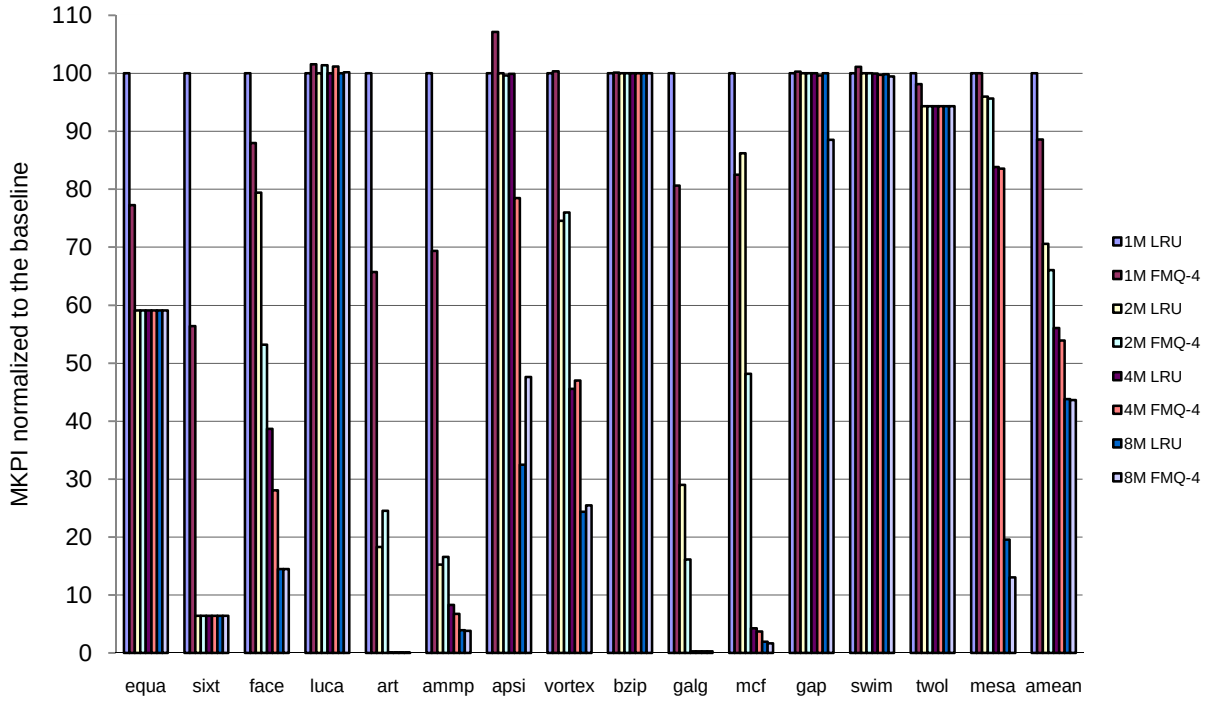


Figure 61: Comparison of the LRU and the FMQ for 1M, 2M, 4M and 8M cache.

6.2 Impact of Cache Line Size and Associativity

First, we measure the MPKI in cache lines of 64B and 128B. Figure 62 compares the MPKI of the FMQ and the LRU at an L2 cache line size of 64B. On average, the MPKI is reduced by 13%, which is only 1% higher than the baseline with a line size of 128B. Eleven of the 15 benchmarks, e.g., *art*, show increased MKPI improvement when the line size is 64B, while the other four benchmarks show decreased MPKI improvement, e.g. *apsi, luca, swim and vort*. The largest MPKI reduction of the FMQ is 50.6% for the benchmark *art*.

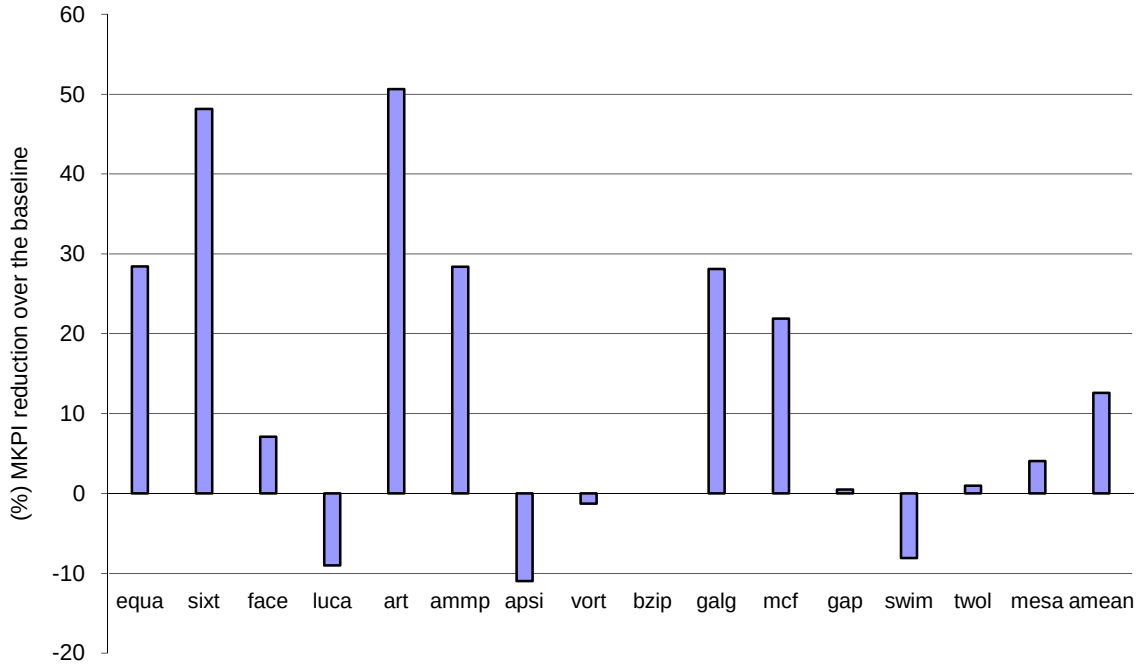


Figure 62: MPKI reduction of the FMQ over the LRU at an L2 cache line size of 64B

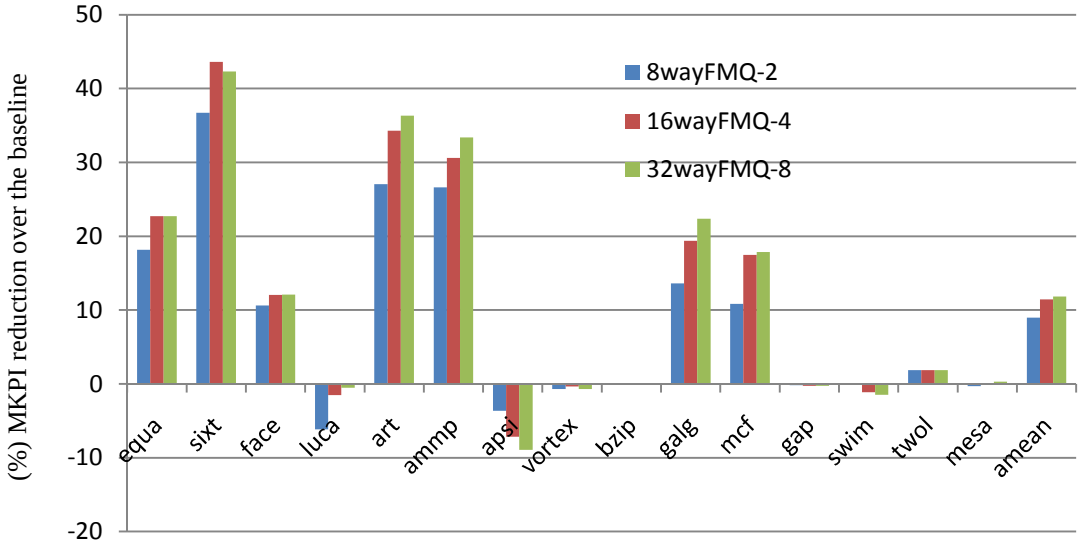


Figure 63: The MPKI reduction of the FMQ over the LRU for 8-way, 16-way, and 32-way 1MB cache

Next, we analyze the impact of 8-way, 16-way, and 32-way associativity in cache capacity of 1MB, 2MB, 4MB, and 8MB to the proposed FMQ replacement. We keep the length of FMQ queue at the constant of four. For example, the length of 16-way FMQ-4 = 16-way/4 queues= 4.

Figure 63 shows the MPKI reduction of the FMQ over the LRU at 1MB cache associativity of 8-way, 16-way, and 32-way. On average, increased associativity brings increased MPKI improvement. An exception is benchmark *apsi* that shows degraded MPKI when associativity is increased.

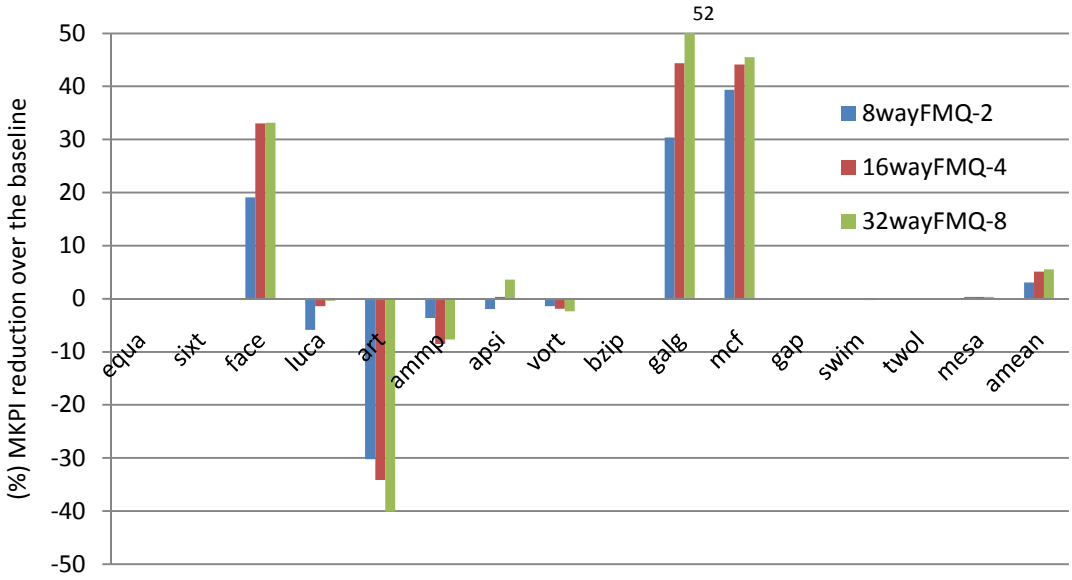


Figure 64: The MPKI reduction of the FMQ over the LRU for 8-way, 16-way, and 32-way 2MB cache

Figure 64 shows the MPKI reduction of the FMQ over the LRU at 2MB cache associativity of 8-way, 16-way, and 32-way. On average, increased associativity brings increased MPKI improvement. But benchmark *art* shows degraded MPKI when associativity is increased.

Figure 65 shows the MPKI reduction of the FMQ over the LRU at 4MB cache associativity of 8-way, 16-way, and 32-way. On average, the best MPKI improvement is at an associativity of 16-way.

Figure 66 shows the MPKI reduction of the FMQ over the LRU at 8MB cache associativity of 8-way, 16-way, and 32-way. On average, the best MPKI improvement is at an associativity of 8-way. Increased associativity in 8MB cache will not bring increased MPKI.

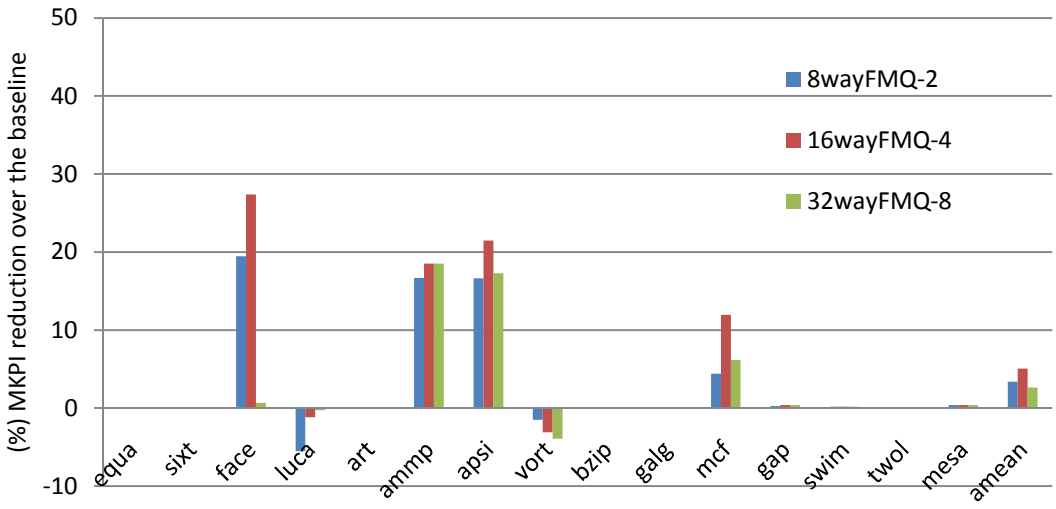


Figure 65: The MPKI reduction of the FMQ over the LRU for 8-way, 16-way, and 32-way 4MB cache

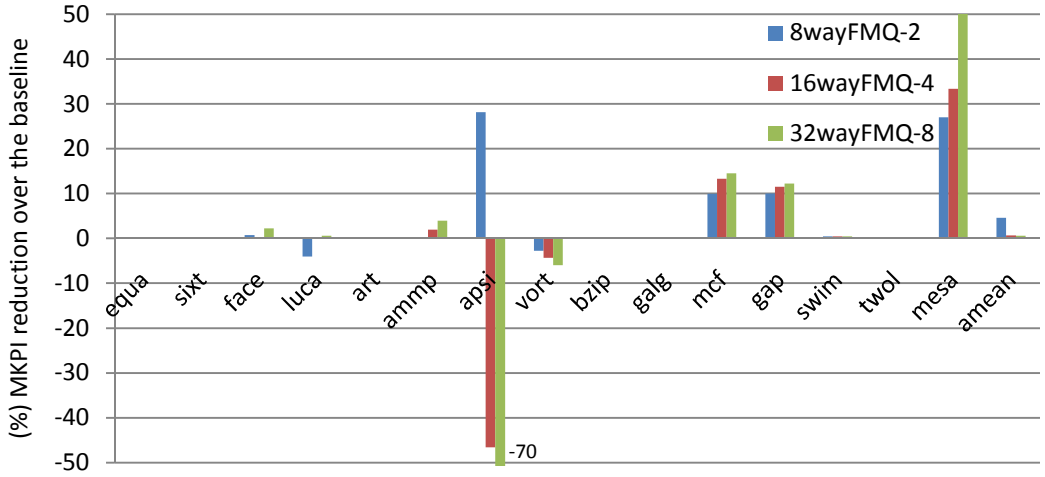


Figure 66: The MPKI reduction of the FMQ over the LRU for 8-way, 16-way, and 32-way 8MB cache

6.3 Impact on System Performance

We present the performance improvement in terms of Instructions Per Cycle (IPC) of a processor that is equipped with a L2 cache using the FMQ replacement. We use SimpleScalar to evaluate the effect of the FMQ on the overall processor performance. Table 17 shows the baseline cache configuration. Both L1 instruction cache and L1 data cache are 2-way 16KB with LRU. The line size in L1 cache is 64B. The 16-way associativity is configured in 1M L2 cache as a baseline. Figure 67 shows the performance measured in instructions per cycle (IPC) improvement by using the FMQ. The IPC improvement is on average 7%. Note that the FMQ replacement

reduces the storage and circuit complexity and will not increase the access cycle of the L2 cache.

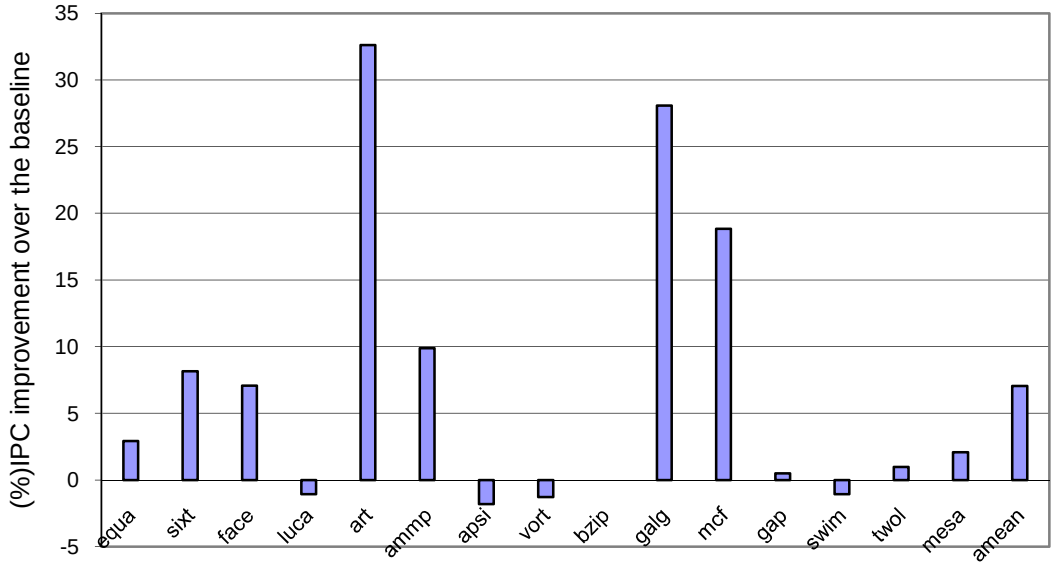


Figure 67: IPC Improvement with FMQ over the baseline

Table 17:Baseline cache configuration

Parameter	Setting
Branch Predictor	Hybrid 64K, miss penalty 10 cycles
L1 instruction cache	16KB, 64B line size, 2-way with LRU
L1 data cache	16KB, 64B line size, 2-way with LRU
L2 unified cache	1MB, 128B line size, 16-way, 6 cycle hit
Main memory	200 cycles unlimited size

6.4 Power Analysis

6.4.1 Circuit Designs

To evaluate the power consumption of the proposed FMQ, we implement the LRU and FMQ using the Cadence layout tools. Low associative LRU needs n×(n-1)/2 [84] status bits for replacement. For example, a 4-way LRU replacement needs 6 bits for one set, which is lower than 8 bits for one set in a counter based 4-way LRU replacement implementation. However, a 16-way naïve counter based LRU needs 64 bits (that are much lower than using Smith's method: n×(n-1)/2 =16×(15)/2 = 120 bits) for 16-lines. Therefore, we implement a 16-way cache with LRU using a counter based LRU. The counter based LRU implementation is evaluated by Ghasemzadeh [31].

Figure 68 and Figure 69 shows the status bit circuits of a 4-way cache for the LRU and FMQ replacements, respectively. We use the design of a 4-way LRU to show the difference between the LRU replacement and the FMQ replacement. We show the number of status bit updates in the LRU for a 4-way cache as shown in Figure 75. For the baseline, we use a 16-way LRU, which can be extended from the 4-way LRU design.

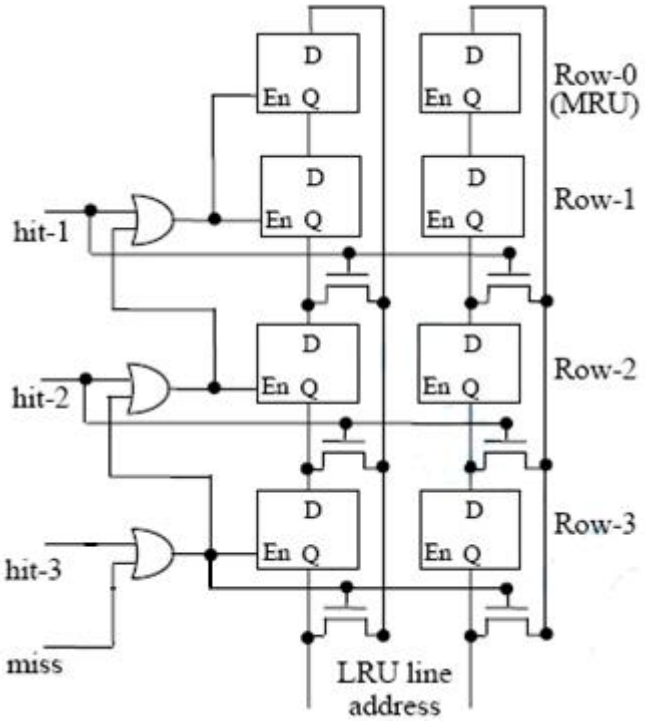


Figure 68: The LRU status bits circuits for a 4 –way cache (The lines to En on the flip-flop on the left are meant to continue to the adjacent flip-flop on the right.)

Figure 68 shows the circuit design of a 4-way LRU replacement. The status bits of each cache line are stored in two flip-flops (four flip-flops for 16-way cache). The four rows include the row-0 (the MRU cache line), row-1, row-2, and row-3 (the LRU line). There are four inputs to the circuit, including three hit signals and one miss signal. Since a cache hit on the MRU cache line will not change the status bits, we need only 3 hit signals to the three cache lines that are not the MRU line. A cache hit on a cache line that is not the MRU line incurs status bit updates. For example, the hit-2 signal is high when the hit occurs on the row-2. The status bits of the row-2 are

read out and sent to the MRU cache line through the two transfer lines. The status bits of the original MRU line and row-1 are shifted to the next line in the LRU stack. On a cache miss, the miss signal enables all the flip-flops and the status bits to rotate along the four-way status bits.

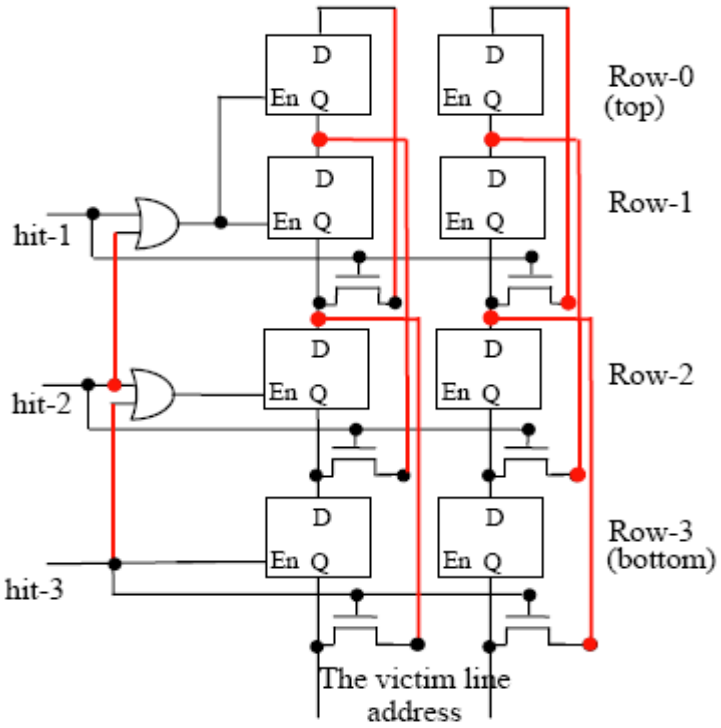


Figure 69: The FMQ status bits circuits for a 4 –way cache

Figure 69 shows the circuit design of a 4-way FMQ replacement. Compared to the LRU circuit, no extra gates or transistors are required. The modifications we made are the connections among the status bit flip-flops. On a cache hit (e.g., a cache hit on the row-2), the status bits of the row-2 are exchanged with the status bits of row-1,

which is implemented through connecting the outputs of the row-2 to the inputs of the row-1 through the NMOS transistors. Similarly, the outputs of the row-3 and row-1 are connected to the row-2 and row-0, respectively.

The FMQ replacement needs the rollover operation to move the contents of the flip-flops of the row-3, row-2, and row-1 to the row-2, row-1, and row-0, respectively. It is implemented by generating three control signals of hit-1, hit-2, and hit-3, consecutively. Figure 70 shows the rollover control circuit. The three-bit shift counter provides the control signals of the hit-1, hit-2, and hit-3 signals. The shift counter is preloaded with 1, 0, 0 and the outputs are controlled by the rollover signal. This three-bit counter is inside the cache controller and can be used by all the cache sets; therefore, the extra area is trivial to the whole cache controller.

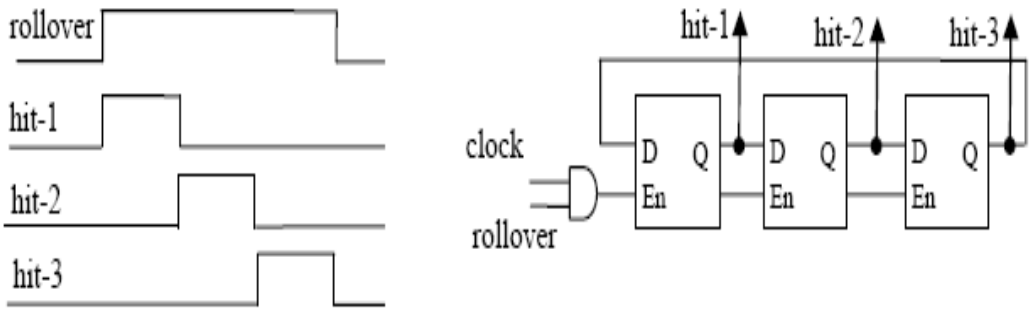


Figure 70: The rollover control circuit

6.4.2 Status Bit Update Patterns of the Benchmarks

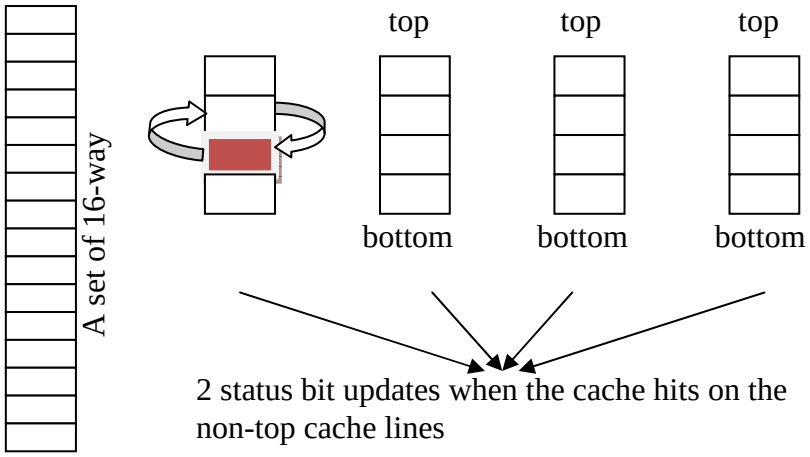


Figure 71: The number of status bit updates when the cache hits on the non-top cache lines in the FMQ

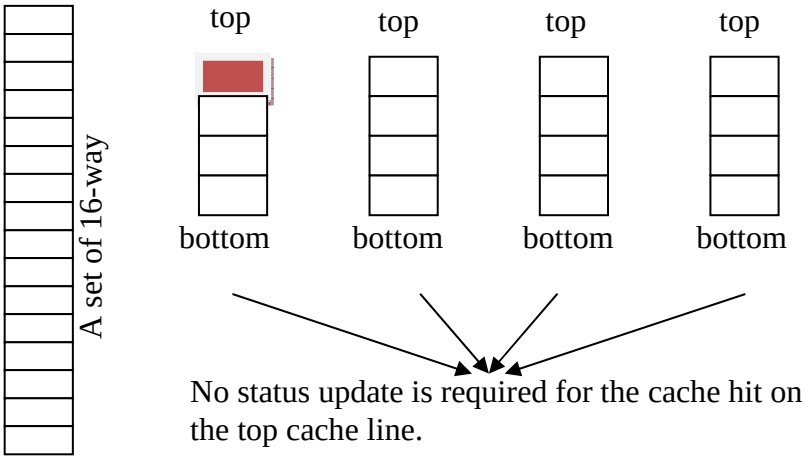


Figure 72: The number of status bit updates when the cache hits on the top cache lines in the FMQ

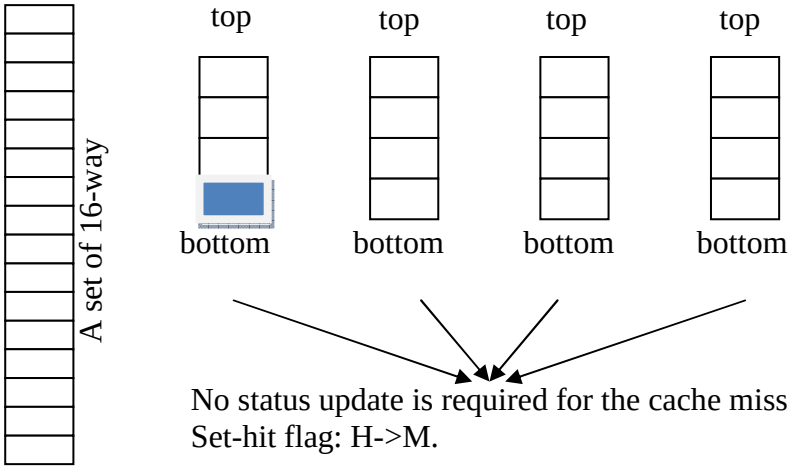


Figure 73: The number of status bit updates when the cache miss in the FMQ (Set-hit flag: H->M)

The power consumption of the status bit updates for both the LRU and FMQ depends on the hit locations of the cache lines in the LRU stack and the FMQ. Figure 75 shows there are 4 status bit updates when the cache hits on the LRU cache line in a 4-way LRU cache, while no update is required when the cache hits on the MRU line. Figure 71 and Figure 72 show the number of status bit updates under cache hits for the FMQ in a 16-way cache. There are two situations: (1) 2 status bit updates when the cache hits on the non-top cache lines; (2) no status update is required for the cache hit on the top cache line.

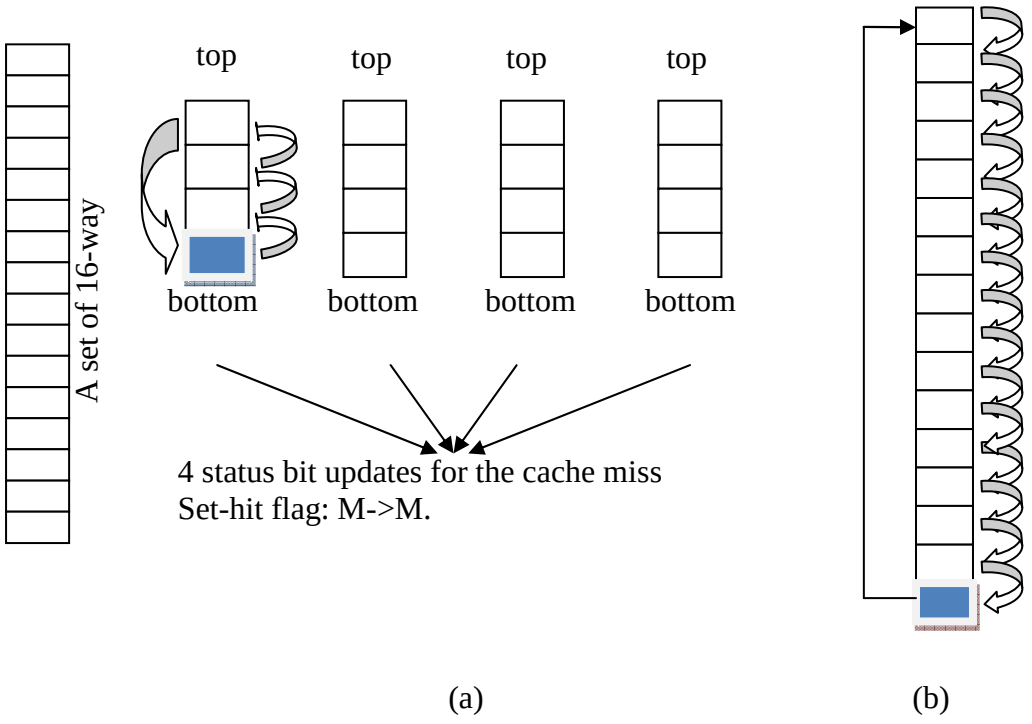
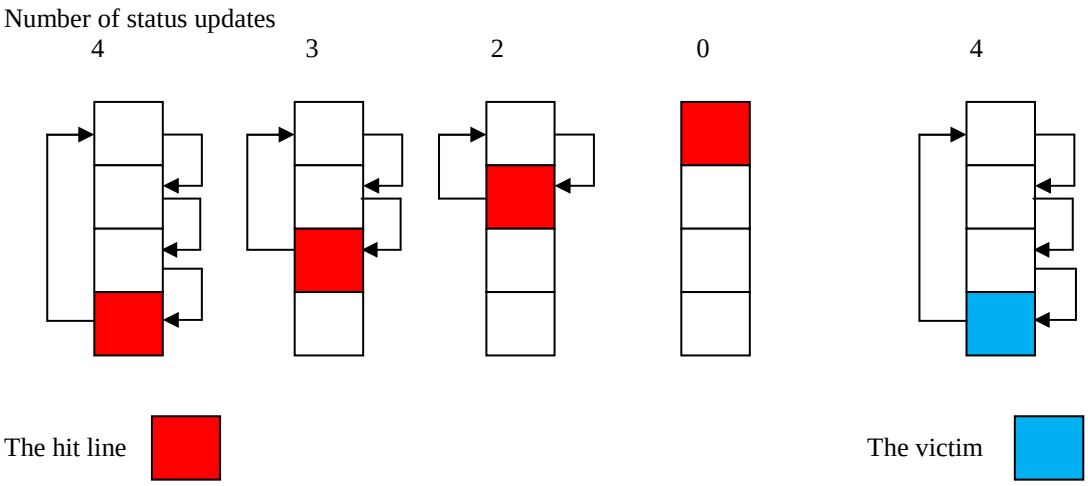


Figure 74: The number of status bit updates when the cache miss in the FMQ (Set-hit flag: M->M) and LRU in a 16-way cache

Figure 73 and Figure 74 (a) show the number of status bit updates under cache miss for the FMQ and LRU in a 16-way cache. Figure 73 shows no status update is required for the cache miss on the bottom cache line and cache hit happens on last access. Figure 74 (a) show all the status bits need updates when a cache miss occurs and last set-hit flag is M. Similarly, Figure 74 (b) shows a cache miss need 16 status bit updates for a 16-way LRU cache.



(a) Hit for LRU (b) Miss for LRU

Figure 75: The number of status bit updates in the LRU for a 4-way cache

To compute the power consumption due to status bit updates, we collect the hit and miss information of the programs. Figure 76 shows the percentage of cache hits on the cache lines at locations from LRU-0 (MRU cache line) to LRU-15 (LRU cache line) for the baseline 16-way LRU. The percentage of cache hits in LRU-0 is the largest in all the 15 benchmarks from LRU-0 (the MRU cache line) to LRU-15 (the LRU cache line). The highest hit rate in LRU-0 is 91.4% in the benchmark *equake*. The lowest hit rate in LRU-0 is 44.2% in the benchmark *lucas*. For twelve of the 15 benchmarks, the percentage of cache hits in LRU-1 is the second largest. For ten of the 15 benchmarks, the cache hit rate is less than 0.8% from LRU-3 to LRU-15.

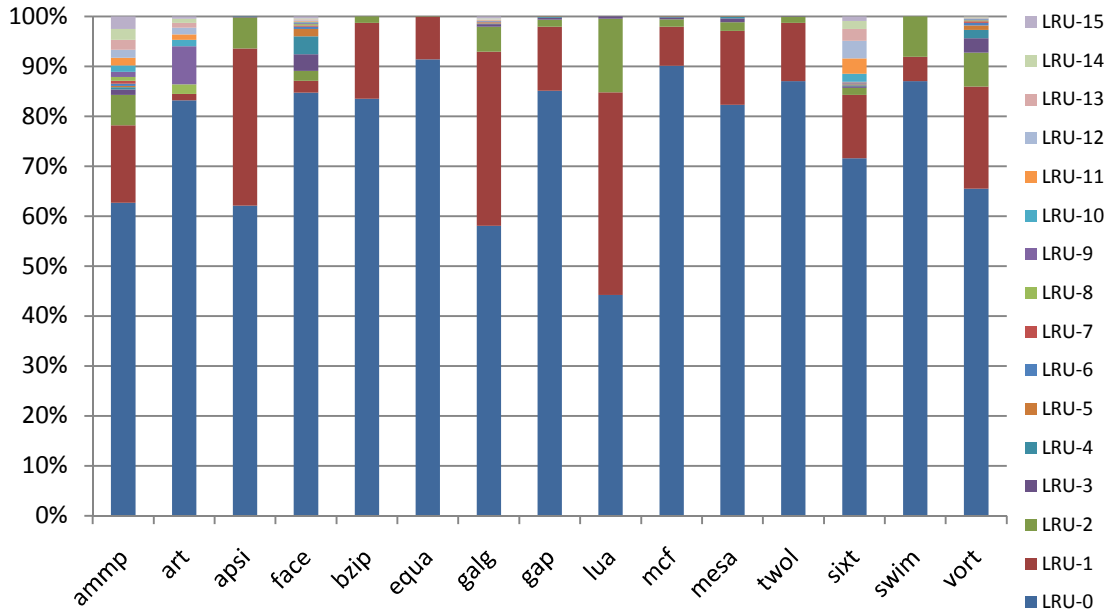


Figure 76: Percentage of the cache hit locations in the LRU stack for the baseline 16-way LRU

Figure 77 shows the percentage of the cache miss (MS) in the LRU stack for the baseline 16-way LRU. The highest miss rate in LRU is 62.0% in the benchmark *mcf*. The lowest miss rate in LRU is 0.4% in the benchmark *equake*. For ten of the 15 benchmarks, cache miss rate in LRU is less than 20%.

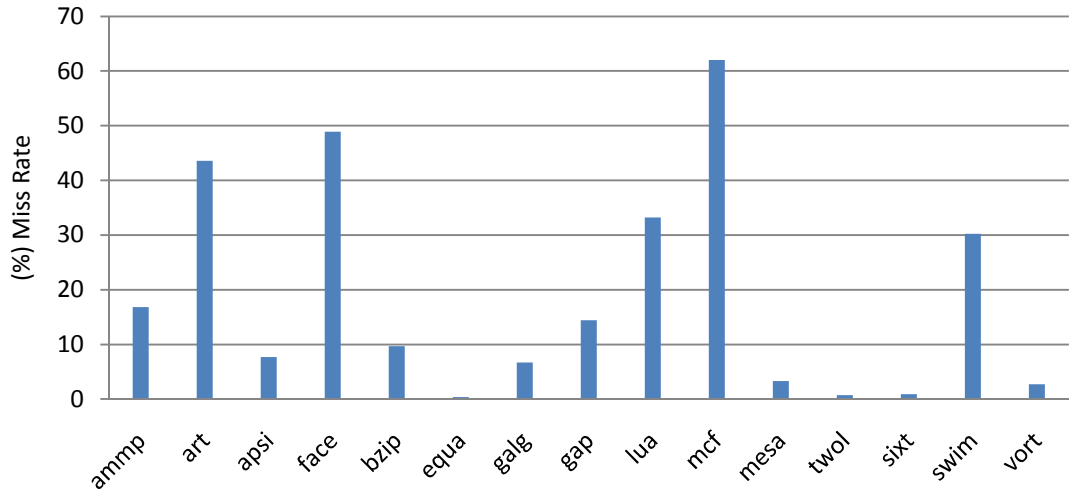


Figure 77: Percentage of the cache miss rate (MS) in the LRU stack for the baseline 16-way LRU

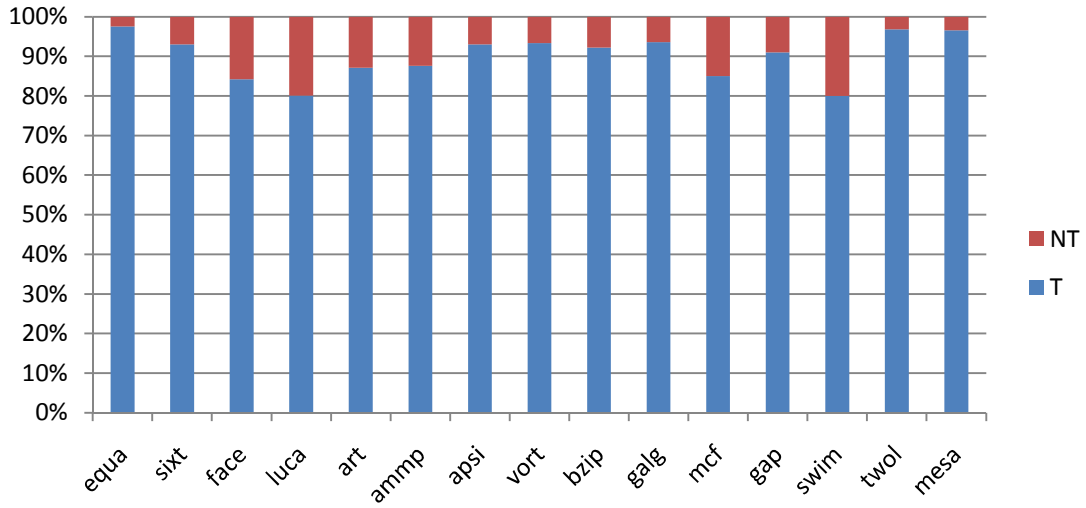


Figure 78: Percentage of the cache hit locations in the 4-elements FMQ

Figure 78 shows the percentage of cache hits on top (T) and non-top (NT) location for the 4-element of FMQ replacement (there are four queues in the FMQ). The hit locations are from the top cache line (T) to non-top cache line (NT) for the FMQ. For all the 15 benchmarks, the percentage of cache hits on top (T) is more than 80% and up to 96.8%. On average, the percentage of cache hits on top (T) is 90.1% in the FMQ.

Figure 79 shows the percentage of rollover operation (RO) and cache miss rate (MS) in the FMQ. The MS is the miss rate of the baseline in the FMQ. The RO is the percentage of cache misses that incur rollover operation when the FMQ is used. We collect the percentage of rollover operation (RO) accounts for the total misses, since the number of status bits updates for rollover operation is 4 updates while there are no status bit updates for the cache misses when no rollover operations for the FMQ replacement. The highest RO is 38.2% in *facerec*. For eleven of the 15 benchmarks, RO is less than 0.3% in the FMQ. The highest MS is 44.9% in *mcf*. On average, the MS is 16% in the FMQ.

Figure 80 represents the percentage of status bit Updates Reductions (UR) of the FMQ over the baseline 16-way cache. For twelve of the 15 benchmarks, the reductions of the status bit updates are over 90%. On average, the UR is 92%. The highest UR is 96.3% in the benchmark *art.*

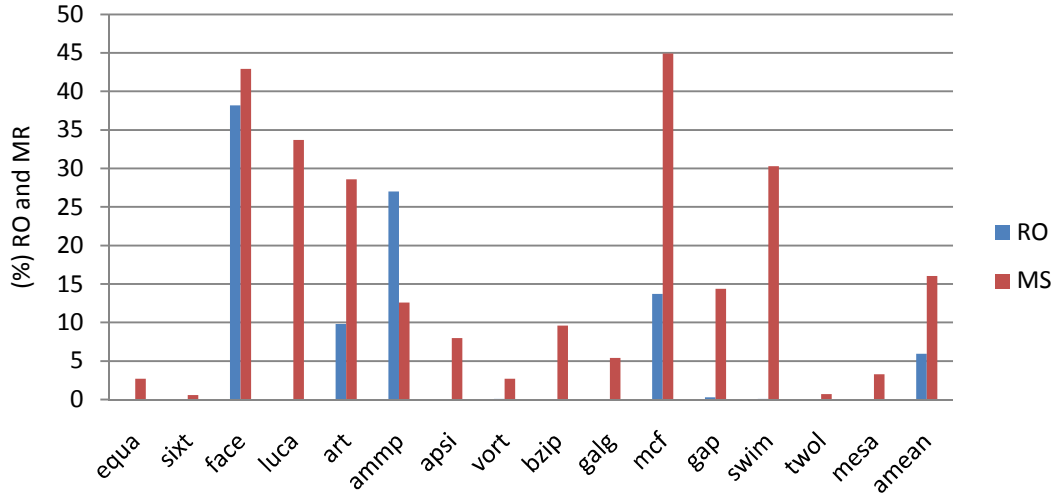


Figure 79: The percentage of rollover operation (RO) and miss rate (MR) in FMQ

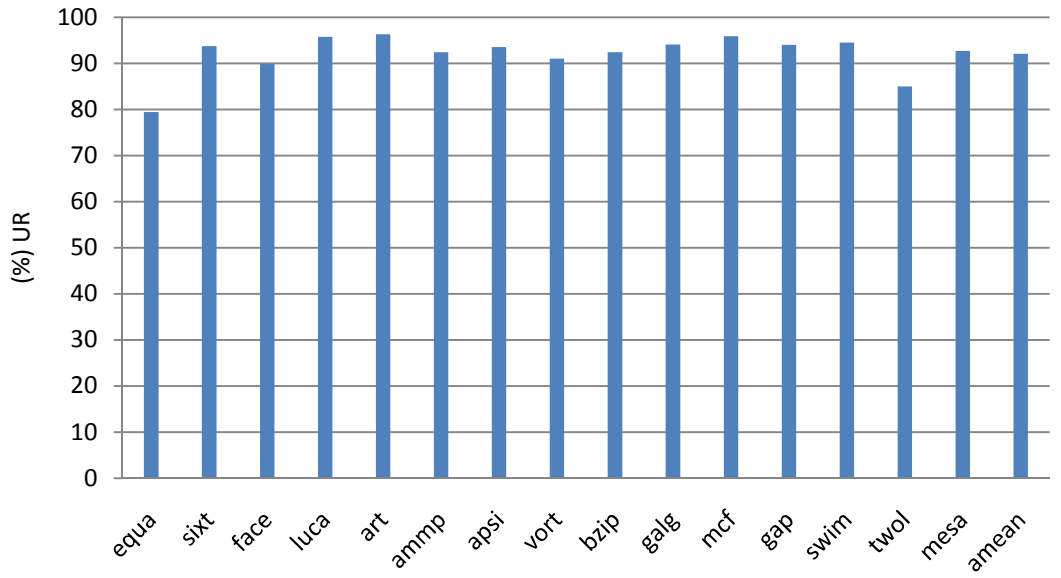


Figure 80: The percentage of status bit Updates Reductions (UR) of the FMQ over the baseline 16-way cache.

6.4.3 Power Evaluations

We use the following equation to compute the total power consumptions of the LRU and the FMQ:

Equation for power consumptions of the LRU:

$$Power_{LRU} = \sum_{i=1}^{15} p\_hit_i * num\_hit_i + p\_miss * num\_miss$$

Where the p_hit_i is the power consumption when the cache hit occurs at the position of i of the LRU stack; The mum_hit_i is the total number of the hits on the location i of the LRU stack; The *p_miss* is the power consumption of cache misses for the LRU stack, which needs update 16 cache lines status bits. The *num_miss* is the total number of cache misses of the LRU replacement.

Equation for power consumptions of the FMQ:

$$Power_{FMQ} = p\_NT\_hit * num\_NT\_hit + p\_miss\_rollover * num\_miss\_rollover$$

Where the *p_NT_hit* is the power consumption when cache hit occurs on the non-top cache lines of the FMQ replacement (No status bits are updated when the cache hits on the top position of the queue). The *num_NT_hit* is the total number of cache hits on the non-top cache lines in the queue. The *p_miss_rollover* is the power consumption of cache miss for the FMQ when rollover operation occurs, which needs to update the status bits of four cache lines. No status bits needs update when no rollover operation is required for the FMQ replacement. The *num_miss_rollover* is the total number of cache misses of the FMQ replacement when rollover operation occurs.

We implement the LRU and FMQ by using the Cadence layout tools (90 nm fabrication technology) to evaluate the power consumption. Figure 81 shows the power consumptions when the hit occurs at the various locations in the LRU stack and FMQ. The hit locations are from cache line location of 0 (the MRU line) to 15 (the LRU line) for the LRU and Non-top (NT) cache line for the FMQ.

The total power consumptions of the benchmarks are shown in Figure 82. The FMQ consistently consumes less power than the LRU. On average, the FMQ consumes 96% less energy than the LRU replacement for the benchmarks we simulated.

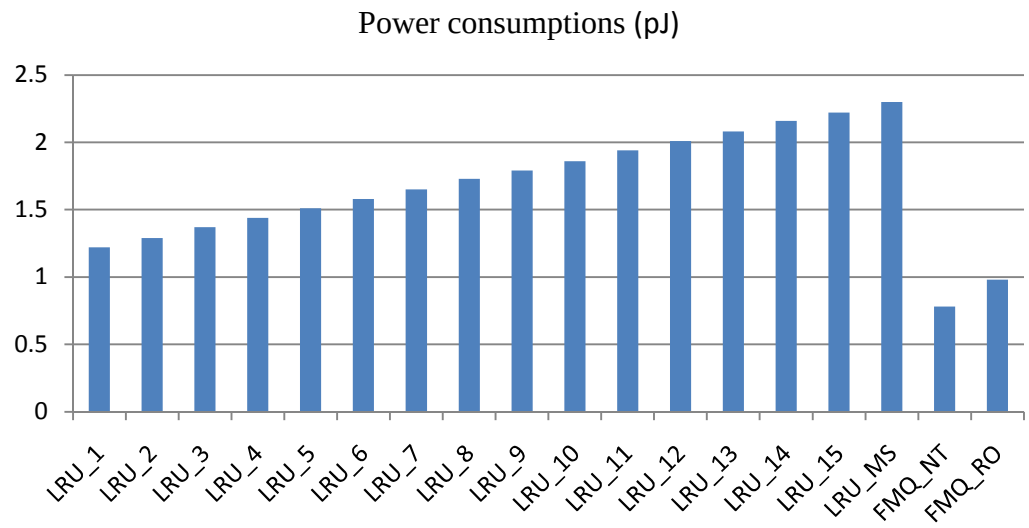


Figure 81: Power consumptions (pJ) of the status bit update for the baseline 16-way LRU and the FMQ

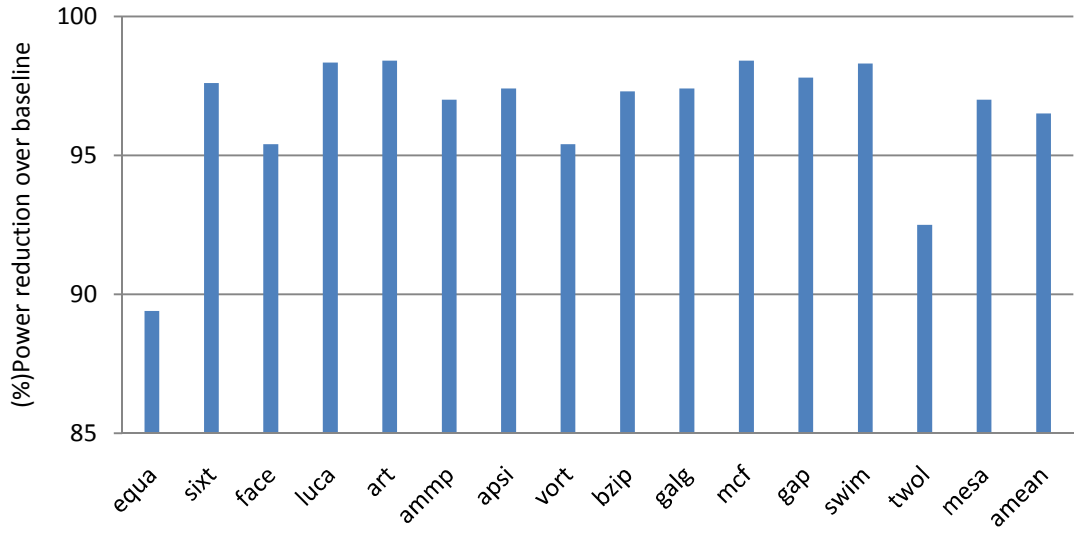


Figure 82: Percentage of power consumption reduction of the status bit updates of the FMQ over the baseline LRU

6.4.4 Hardware Budget

Table 18: Storage of the FMQ

Parameter	Storage	Total Storage (512 sets)
set type per set	8bits (per queue) × 4 (queue)	8bits (per queue) × 4 (queue) ×512 (sets)
set-hit	1bit	512 bits (64B)
Total		2.0 KB+64B = 2112 B

Table 18 shows how to use the number of status bits to evaluate the storage savings for the FMQ. The FMQ needs 8bits (per queue) × 4 (queue) ×512 (sets) =

2.0KB. The set-hit bit is 512 bits (64B), thus the total number of storage for replacement is 2.0KB + 64B = 2112B for the FMQ. While the baseline 16-way LRU needs 64bits (per set) × 512 (sets) = 4096B.

In addition, the circuit complexity is reduced since the status of all the 16 cache lines have to be updated when an incoming line is inserted into the MRU position, while at most four lines in the proposed FMQ needs updating. Table 19 summarizes the hardware requirements, including the storage requirements for status bits and circuit design complexities for the Sampling-Based Adaptive Replacement (SBAR) [77], DIP [78] and FMQ replacements. The hardware overhead of the DIP is two bytes over the LRU, but the FMQ reduces the hardware storage by 48% over the LRU and simplifies the circuit design. The SBAR needs extra structure for the dynamic selection and counters for the LFU. The total storage requirement for the SBAR is 11KB but the FMQ needs less than 2.1KB.

Table 19: Compare Storage of the SBAR, DIP and FMQ

Replacement Policy	Additional HW Cost	Storage and Circuits Complexity
SBAR(LRU+LFU)	heavy	4.0KB (LRU)+7.0KB (LFU) , 16-way LRU+LFU
DIP	light	4.0KB(LRU)+2B, 16-way LRU 16/32/64 dedicated sets, epsilon 1/32
FMQ	very light: a single bit set-hit flag per set	2KB(LRU)+64B, 4-way LRU

6.5 Summary

Cache capacity, line size, set associativity, and the number of queues affect the L2 cache performance. The FMQ continues to reduce the average MPKI as cache capacity is from 1MB to 4MB. On average, increased associativity brings increased MPKI improvement in cache capacity of 1MB and 2MB. The average MPKI in cache line size of 64B is reduced by 13%, which is 1% higher than the baseline with 128B cache line size. The IPC improvement can be on average 7% over the baseline cache. The circuit design in the FMQ is simplified compared to the traditional LRU. The storage requirement for the FMQ is reduced to 48%, too. The FMQ consistently consumes less energy than the LRU. The average power consumption of the FMQ is 96% less than the LRU.

CHAPTER 7

RELATED WORK

Substantial research on cache replacement has been conducted. Most of this work has been described in Chapter 2. In this chapter we summarized the work that is closely related to the FMQ and compared the FMQ with the DIP [78] and SBAR (LRU+LFU) [77].

Some studies have exploited frequency information for improving cache performance. The performance of the LFRU [60] largely depends on the weighting function, which decides the weight of the LRU and LFU. Further research is required to determine the best weighting function dynamically for a particular program. A frequency based cache replacement [28] incorporated with the LRU is proposed for a disk file system. Cache lines are divided into sectors based on the usage frequency. The victim is chosen from the “old sector”, which is composed of less frequently used lines, using the LRU. The frequency counters are periodically adjusted to factor out the impact of locality.

A software based generational replacement algorithm [37] is proposed to exploit the fully associative cache organization. Cache lines are categorized into priority groups. Regularly referenced lines are promoted to high priority groups and infrequently used lines are demoted to low priority groups. On a cache miss, a line from the lowest priority group is selected. The generational replacement uses extra

hardware that comes from the implementation of the fully associative cache through the indirect indexing. For example, for a 1M cache with a line size of 128B, the extra hardware for implementing the indirect index cache and the generational replacement is 68.75KB.

Some other techniques have been proposed for workloads that cause thrashing with the LRU. Jiang and Zhang [44] proposed a Low Inter-reference Recency Set (LIRS) replacement for buffer cache. Cache line access distance is measured by the number of other lines accessed between two consecutive references to the line is used for cache replacement. For example, the lines with the longest distance are the candidates for the victim. The LIRS effectively resolved the problem of the LRU that performs poorly on sequential scans and cyclic access patterns.

Hybrid replacement schemes [65][78] [92] dynamically determine and use the best replacement, which requires tracking separate replacement information for each of the competing policies. The dynamic selection between two policies requires extra structures needs more hardware storage and consumes more power.

Megiddo and Modha proposed an Adaptive Replacement Cache (ARC) [65] that maintains two lists: a recency list and a frequency list. The recency list records pages that were touched only once, whereas the frequency list contains pages that were touched at least twice. ARC dynamically tunes the number of pages devoted to each list and requires 64KB storage.

In the Sampling-Based Adaptive Replacement (SBAR) [77], for example, if the two policies are LRU and LFU, then each tag-entry in the baseline cache needs to be appended with counters (5 bits each) that must be updated on each access. This means SBAR requires 5-byte LFU counter per each line (10 KB) and 2KB of the ATD directory.

Qureshi et al. proposed DIP, an adaptive hybrid replacement that dynamically selects the best of multiple policies between the LRU and the Bimodal Insertion Policy (BIP) through set dueling. 32 sets use the LRU and 32 sets use the BIP, while the remaining sets will use the replacement that demonstrates the better performance. Bimodal Insertion Policy (BIP) offers thrash protection. The BIP inserts incoming lines predominantly (31 out of 32 misses) to the LRU position to resolve the cache thrashing problem and to the MRU position occasionally (1 out of every 32 misses) to adapt to the working set change. The DIP does not consider the frequency information. The FMQ can resolve the cache thrashing by inserting the incoming lines to the bottom of the queue and exploits frequency information through the promotion policy. The real contribution of DIP is not that the performance is significantly better than SBAR while DIP does not require too much hardware changes compared with SBAR. However, DIP cannot provide more benefit if the line size in L2 cache is larger than the line size in L1 cache. The reason is that the second half of the line installed in L2 cache may be requested soon after promoting the line from LRU position to MRU position. The locality for these accesses in this case is only spatial

locality and not temporal locality. DIP is designed for filtering temporal locality, and a larger line size will not filter temporal locality for the second half of the line. Table 20 summarizes the MPKI and insertion policy for these replacements.

Compared to these other cache replacements that exploit frequency information, FMQ does not use extra structures, counters, tables, high associativity, or the intervention of the operation system. Instead, FMQ represent the relative frequency of lines by the location of a line in the FSQ queue. Figure 83 shows the MPKI reductions of FMQ, DIP, and SBAR (LRU+LFU) over the LRU for the baseline 1M 16-way cache.

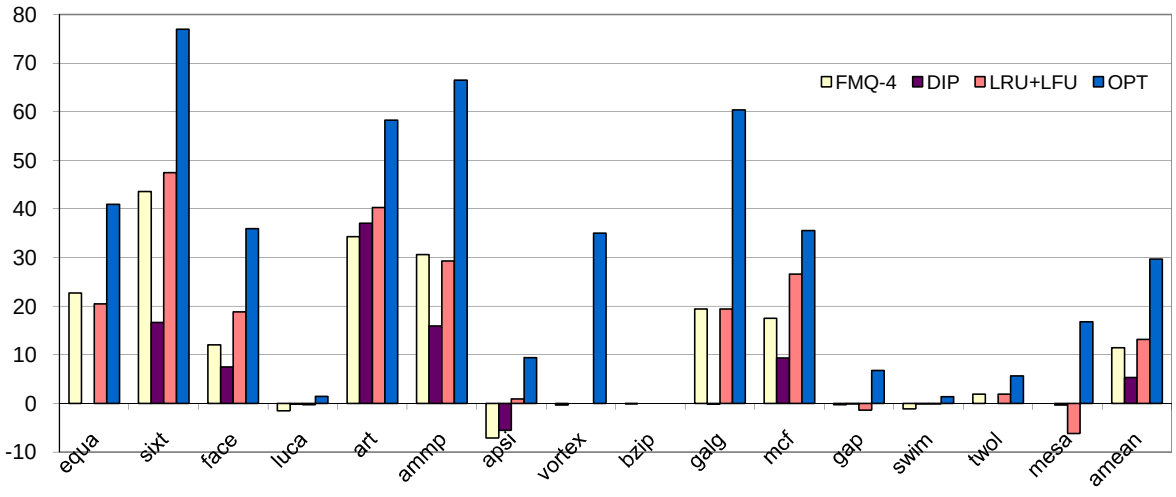


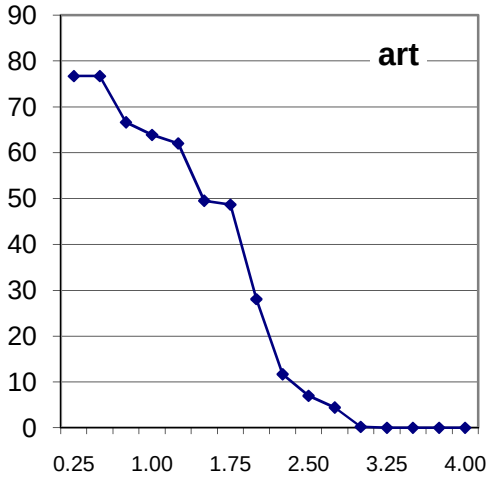
Figure 83: The percentage MPKI reduction of FMQ-4, DIP, SBAR (LRU+LFU), and OPT over the LRU baseline.

Table 20: Comparison with Related Work

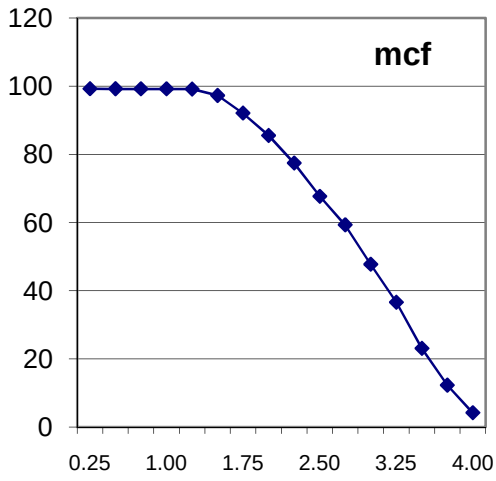
Replacement Policy	Insertion	% MPKI Reduction over LRU
LRU	Insert to MRU	baseline
SBAR(LRU+LFU)	Dynamic selection	13%
DIP	Partially Random Insertion	6%
FMQ	Insert to Bottom	12%
OPT		30%

We fed the address traces collected in our study to the code downloaded from [76][77][78] (provided by the author of the DIP and SBAR) and show the results in Figure 83. The SBAR (LRU+LFU) reduces the MPKI with an average of 13%, which is similar to the results (14.7%) reported in [78]. The MPKI reduction of DIP, however, is on average 6% in our study, which is lower than the results reported in [78]. This may be due to the following two reasons. First, we used precompiled benchmarks binaries, while the DIP uses their own compiled binaries. Second, we used the program phases suggested by the SimPoint, while the DIP selects the program phases using different criteria.

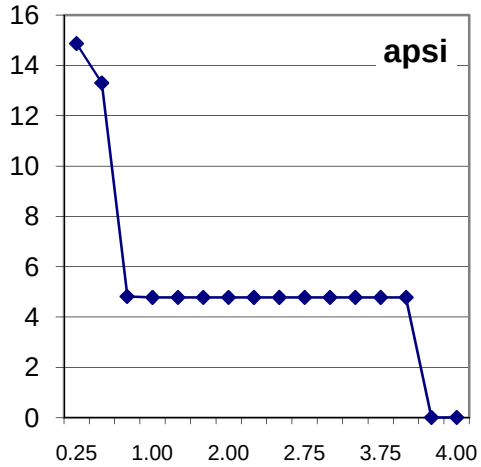
The FMQ outperforms the DIP by 6% on average. We show the MPKI of the benchmarks in different cache size in Figure 84. From Figure 84, we can see that the working set of benchmarks *art* and *mcf* are larger than 1MB. These results are similar to the results reported from the DIP. The DIP improves the MPKI by storing part of



Cache size in MB
(a) art



Cache size in MB
(b) mcf



Cache size in MB
(c) apsi

Figure 84: MKPI vs. Cache Size for the Benchmarks

the cache lines into the cache to avoid the cache thrashing. For benchmark *galgel*, as shown in Figure 37, Figure 38, and Figure 39, because there are no cyclic access

patterns, DIP chooses the LRU instead of the BIP. For benchmark *ammp*, FMQ outperforms DIP since FMQ takes advantage of the frequency information of the *ammp*. The SBAR (LRU+LFP) achieves similar MPKI reductions compared to the FMQ because the SBAR can dynamically choose the better policy between the LRU and LFU. However, SBAR requires extra structures and counters for the LFU and the dynamic selection. In addition, the SBAR needs 16-way traditional LRU, which requires more design effort on circuits than FMQ because FMQ's circuit complexity is equivalent to a 4-way LRU.

Johnson presented Two Queue (2Q) algorithm [47], a low overhead high performance buffer management replacement for database systems. Zhou et al. proposed the Multi-Queue (MQ) [114] replacement policy for buffer caches. Multiple LRU queues store the pages that have been seen. Two queues and a history buffer are required in the MQ. On a page hit, the page frequency is incremented and the page is moved to the MRU position, which is different from our techniques.

CHAPTER 8

SUMMARY AND FUTURE WORK

8.1 Summary

We discuss the widely used cache replacement including LRU, LFU, and their derivations. It is well known that LRU suffers several problems: (1) LRU does not exploit access frequency information of cache line; (2) LRU performs poorly when a program exhibits cyclic access patterns and the working set is greater than the available cache size; (3) LRU is also expensive in hardware implementation for high associative cache.

This dissertation presents two novel replacements – FSR and FMR. The FMQ (queue-number >1) is proposed based on the FSR (queue-number =1). The FMQ combines the specialty of the FSR and the MQ replacement to improve the cache performance. The FMQ replacement exploits both the recency and frequency information for LLC with low additional hardware cost and minimum design changes. We use the locations of cache lines in the queue as a relative access frequency indicator of cache lines. There is a difference from using counters to record the access frequency of cache lines in previous proposals. Frequently accessed blocks are stored in the top of the queue through the promotion policy that moves the hit cache line upward along the queue. Recency information is also considered through exchanging the position of a hit cache line with the cache line above in the queue, even if the

access number of the hit cache line may be lower than the cache line above in the queue. On a cache miss, the incoming cache line randomly selects one subqueue and enters this subqueue from the bottom where the less frequently accessed cache line is stored. A victim cache line can be selected from either the bottom of the subqueue or the top line of the subqueue, which is controlled only by a single bit set-hit flag per set. We reduce the length of queue by dividing the queue into multiple subqueues so that the whole cache capacity can be fully exploited.

The performance for low level cache in the FMQ depends on these factors: cache capacity, line size, set associativity, the number of queues, the working set size and cache blocks access behavior for workload. We select 1MB 16-way cache with a line size of 128B and 4 sub-queues as the baseline of the FMQ. Case studies of the SPEC2000 benchmarks are described in chapter 5. The impact of the access behaviors of the SPEK2K benchmarks is analyzed in 4 groups. Cyclic access patterns and the access frequency information in the benchmarks affect the performance of the FMQ.

The experimental results show that the proposed replacement policy can significantly reduce cache misses for memory-intensive workloads with reduced hardware design and circuit complexities. System performance and energy consumption of the FMR are analyzed in chapter 6. On average, the proposed FMQ achieves a 12% MPKI reduction over the baseline 1MB 16-way unified L2 cache. The average IPC improvement can be 7% over the baseline cache. Some benchmarks can

have up to 33% of improvements. The storage requirement and the updates of the status bits are reduced to 48% and 92%, respectively.

8.2 Future Work

There are several possible extensions of our work. Our goal is “do more with less cost”. Given the performance potential of the FMQ, future work on the replacement continues along the following directions.

8.2.1 Dynamic Frequency based Multiple Queue

In our future work we plan to combine the FMQ with other cache optimizing techniques to reduce cache misses and effective miss penalties. One method is Dynamic Frequency based Multiple Queue (DFMQ) replacement.

The DFMQ replacement dynamically selects a better replacement between a recency-based replacement policy (e.g. LRU) and the FMQ with low hardware complexity as shown in Figure 85. Programs exhibit various behaviors, including memory access patterns and data structure layout among others. DFMQ detect intelligently the access patterns and switch into suitable replacement policy. For example, for LRU friendly application, LRU will be used to employ in L2 cache. A suitable cache replacement that can be tuned for a particular program has the potential to approach the performance of the OPT. DFMQ determines the appropriate cache replacement that improves a program’s spatial or temporal locality, and thus overall performance.

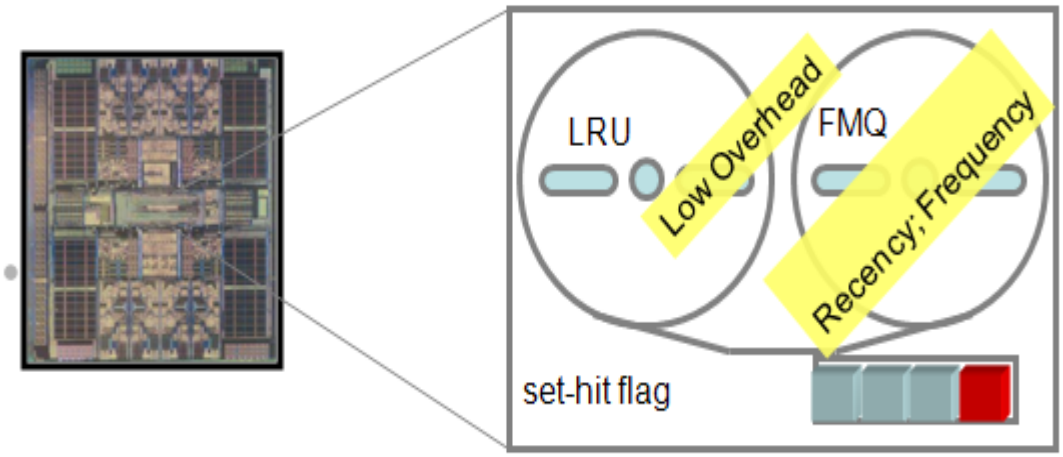


Figure 85: DFMQ Replacement

8.2.2 Cooperation FMQ Replacement and Prefetching

We believe more potential opportunities exist for the cooperation of FMQ replacement and data prefetching techniques for low level caches in future processor design. We will investigate the cooperation of the FMQ and the OBL. Removing L2 cache misses is critical to resolve the gap between the processor and memory performance. Even the OPT replacement still exhibits significant amount of caches misses. We will investigate prefetching combined with FMQ replacement to implement effective and efficient cache management. The frequency of the rollover operation in a limited period will increase or decrease as data patterns prefetched change.

8.2.3 Multi-rule Based Replacement

Based on the discussion in chapter 6, programs exhibit varied behaviors. This motivates us to propose multiple rules based replacement to locate the OPT victim. Each rule is used to divide the block into two parts, with one part containing the OPT victim with high probability. The common set of these rules would increase the accuracy of locating the optimal victim. We will investigate how many rules should be used and the interactions among these rules.

8.2.4 Leading Sets Based Replacement

Memory access exhibits temporal and spatial localities. Previous proposals on temporal stream and spatial stream based memory-streaming show that memory accesses can be fetched in streams instead of individual blocks. These streams will be stored in many sets in traditional caches. Stream based fetching motivates the stream-based replacement. We propose to classify the sets into groups. If we can determine the optimal replacement for one set, then we can exploit the streaming access behavior to predict the victim blocks stored in other sets of the same group.

8.2.5 Cooperation Multi-rule Based Replacement and Prefetching

We will investigate Multi-rule Based replacement combined with prefetching. Prefetching is hindered by the fact that accessed address patterns may not be easy to predict. We propose to classify the addresses into two groups. The “cold” group is not easy to predict while the “hot” group is easy to predict. We will design a replacement that tries to keep those non-easy to predict addresses into cache, while leaving those

easy to predict addresses to prefetching. We will investigate mechanisms on how to represent addresses into two groups.

REFERENCE LIST

[1] Alameldeen, A. R. and Wood, D. A. Interactions Between Compression and Prefetching in Chip Multiprocessors, In Proceedings of the 13th Int'l Symposium on High Performance Computer Architecture (2007), pp. 228-239.

[2] Alghazo, J., Akaaboune, A., and Botros, N., Sf-lru cache replacement algorithm. In Proceedings of the 2004 International Workshop on Memory Technology, Design and Testing (2004), pp. 19-24.

[3] Al-Zoubi, H., Milenkovic, A., and Milenkovic, M. Performance evaluation of cache replacement policies for the SPEC CPU2000, In Proceedings of the 42nd annual Southeast regional conference (2004), pp. 267-272.

[4] Baer, J.-L., and Chen, T.-F. Dynamic improvements of locality in virtual memory systems. IEEE Transactions on Software Engineering (March 1976), pp. 54-62.

[5] Baer, J.-L., and Chen, T.-F. An effective on-chip preloading scheme to reduce data access penalty. In Proceedings of Supercomputing 1991 (November 1991), pp. 176-186.

[6] Bansal, S. AND Modha, D. S. CAR: Clock with Adaptive Replacement, In Proceedings of the 2004 USENIX Conference on File and Storage Technologies (2004), pp. 187-200.

[7] Basu, A., et al. Scavenger: A new last level cache architecture with global block priority. In Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (2007), pp. 421-432.

[8] Belady, L. A. A study of replacement algorithms for virtual storage computers. IBM Systems Journal (1966), 78-101.

[9] Beyls, K. and Hollander, E. D. Platform-independent cache optimization by pinpointing low-locality reuse. In Proceedings of the International Conference on Computational Science, volume 3038, (2004), pp. 448-455.

[10] Beyls, K. and D'Hollander, E. Reuse distance as a metric for cache behavior. In Proceedings of the IASTED Conference (2001), pp. 617-662.

[11] Burger, D. AND Austin, T.M. The SimpleScalar Tool Set, Version 2.0. Univ. of Wisconsin-Madison Computer Sciences Dept. Technical Report #1342, June 1997.

[12] Calder, B. 2010. Available from http://www.cse.ucsd.edu/~calder/simpoint/points/standard/spec2000-multiple-std-100M.html; Accessed in 2010.

[13] Calder, B., Grunwald, D., and Emer, J. Predictive sequential associative cache. In Proceedings of 2nd International Symposium on High Performance Computer Architecture (1996), pp. 244-253.

[14] Chang, J. and Sohi, G.S., Cooperative caching for chip multiprocessors. In Proceedings of the 33rd International Symposium on Computer Architecture (2006), pp. 264-276.

[15] Charney, Mark J. and Reeves, Anthony P. Generalized correlation-based hardware prefetching. Technical Report EE-CEG-95-1, School of Electrical Engineering, Cornell University, February 1995.

[16] Chaudhuri, M. Pseudo-LIFO: the foundation of a new family of replacement policies for last-level caches. In Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (2009), pp. 401 - 412.

[17] Chen, T.-F., and Baer, J.-L. Reducing memory latency via non-blocking and prefetching caches. In Proceedings of the Fifth ASPLOS V (October 1992), pp. 51- 61.

[18] Chilimbi, T. M. and Hirzel, M. Dynamic hot data stream prefetching for generalpurpose programs. In Proceedings of the SIGPLAN '02 Conference on Programming Language Design and Implementation (June 2002), pp. 199-209.

[19] Chilimbi, T. M., Davidson, B., and Larus, J. R. Cache-concious structure definition. In Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation (May 1999), pp. 13 - 24.

[20] Chilimbi, T.M., Hill, M.D., and Larus, J.R., Cache-conscious structure layout. In Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation (May 1999), pp. 1-12.

[21] Chilimbi, T.M., Hill, M.D., and Larus, J.R., Making pointer-based data structures cache conscious. Computer. 33, 12 (2000), 67-74.

[22] Chilimbi, T.M. and Larus, J.R. Using generational garbage collection to implement cache-conscious data placement. In Proceedings of the 1st international symposium on Memory management (1998), pp. 36-48.

[23] Chishti, Z., Powell, M.D., and Vijaykumar, T.N. Distance associativity for high-performance energy-efficient non-uniform cache architectures. In Proceedings of the 36th Annual International Symposium on Microarchitecture (2003), pp. 55 - 66.

[24] Chou, Y., Fahs, B. and Abraham, S. Microarchitecture optimizations for exploiting memory-level parallelism. In Proceedings of the 31st Annual International Symposium on Computer Architecture (June 2004), pp. 76-89.

[25] Collins, J. D. et al. Speculative precomputation: Long-range prefetching of delinquent loads. In Proceedings of the 28th Annual International Symposium on Computer Architecture (July 2001), pp. 14-25.

[26] Dropsho, S., Buyuktosunoglu, A., Balasubramonian, R., Albonesi, D.H., Dwarkadas, S., Semeraro, G., Magklis, G., and Scott, M.L. Integrating Adaptive On-Chip Storage Structures for Reduced Dynamic Power. In Proceedings of 11th Int'l Conf. Parallel Architectures and Compilation Techniques (Sept. 2002), pp. 141-154.

[27] Durham, C. M. AND Hanley, B. P. Method for implementing a pseudo least recent used (LRU) mechanism in a four-way cache memory within a data processing system. US Patent 6240489; Available from http://www.patentstorm.us/patents/6240489/description.html; accessed in 2010.

[28] Dybdahl, H. An LRU-based replacement algorithm augmented with frequency of access in shared chip-multiprocessor caches. ACM SIGARCH Computer Architecture News, 35, 4, (Sept. 2007), 45-52.

[29] Fang, C., Carr,S. , Önder ,S., Wang, Z., Reuse-distance-based miss-rate prediction on a per instruction basis. In Proceedings of the 2004 workshop on Memory system performance (2004), pp. 60-68.

[30] Frigo, M., Leiserson, C. E., and Prokop, H. Cache-oblivious algorithms. In Proceedings of the 40th Annual Symposium on Foundations of Computer Science (Oct. 1999), pp. 285-298.

[31] Ghasemzadeh, H., Mazrouee, S., Moghaddam, H. G., Shojaei, H., AND Kakoee, M. R. Hardware Implementation of Stack-Based Replacement Algorithms. In Proceedings of world academy of science, engineering and technology (2006), pp. 135-139.

[32] Ghasemzadeh, H., Mazrouee, S.S., and Kakoee, M.R., Modified pseudo LRU replacement algorithm. In Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems (2006), pp.368-376.

[33] Gill, B.S. and Modha, D.S. SARC: Sequential prefetching in adaptive replacement cache. In Proceedings of the 2005 conference on USENIX Annual Technical Conference (2005), pp. 293-308.

[34] Gniady, C., Butt, A.R., and Hu, Y.C. Program-counter-based pattern classification in buffer caching. In Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6 (2004), pp. 395-408.

[35] Gonzalez, A. et al. A data cache with multiple caching strategies tuned to different types of locality, in Proceedings of the 9th international conference on Supercomputing (1995), pp. 338-347.

[36] Glass, G. and Cao, P. Adaptive page replacement based on memory reference behavior. In Proceedings of the 1997 ACM SIGMETRICS (1997), pp. 115-126.

[37] Hallnor, E. G. AND Reinhardt, S. K. A fully-associative software-managed cache design. In Proceedings of the 27th Annual International Symposium on Computer Architecture (2000), pp. 107-116.

[38] Hallnor, E.G. and Reinhardt, S.K. A compressed memory hierarchy using an indirect index cache. In Proceedings of the 3rd workshop on Memory performance issues: in conjunction with the 31st international symposium on computer architecture (2004), pp. 9-15.

[39] Hopkins, M.E. and Nair, R. Exploiting instruction level parallelism in processors by caching scheduled groups. In Proceedings of the 24th Annual International Symposium on Computer Architecture (1997), pp. 13-25.

[40] Hu, Z., Kaxiras, S., and Martonosi, M. Timekeeping in the memory system: predicting and optimizing memory behavior. In Proceedings of the 29th International Symposium on Computer Architecture (May 2002), pp. 209–220.

[41] Inoue, K., Ishihara, T., and Murakami, K. Way-predicting set-associative cache for high performance and low energy consumption. In Proceedings of the 1999 international symposium on Low power electronics and design (1999), pp. 273-275.

[42] Jaleel, A. and Theobald, K. SCS, J, and Emer, J. High performance cache replacement using re-reference interval prediction (RRIP). In Proceedings of the 37th annual international symposium on Computer architecture (2010), pp.60-71.

[43] Jeong, J. and Dubois, M. Cost-sensitive cache replacement algorithms. In Proceedings of 9th Int'l Symposium on High Performance Computer Architecture (2003).

[44] Jiang, S. and Zhang, X. LIRS: An Efficient Low Inter-Reference Recency Set Replacement Policy to Improve Buffer Cache Performance. In Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (2002), pp. 31-42.

[45] Jiang, S., Chen, F., and Zhang, X. CLOCK-Pro: an effective improvement of the CLOCK replacement. In Proceedings of the 2005 USENIX Technical Conference (2005), pp. 323-336.

[46] John, L.K., More on finding a single number to indicate overall performance of a benchmark suite. ACM SIGARCH Computer Architecture News. 32, 1 (2004), 3-8.

[47] Johnson, T. and Shasha, D. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In VLDB-94 (1994), pp. 439-450.

[48] Johnson, T. et al. Run-time Cache Bypassing, IEEE Transactions on Computers, 48, 12 (Dec. 1999), 1338-1354.

[49] Joseph, D. and Grunwald, D. Prefetching using Markov Predictors. IEEE Transactions on Computers, 48, 2 (Feb. 1999), 121–133.

[50] Jouppi, N. "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," in Proceedings of the 17th International Symposium on Computer Architecture (1990), pp. 364-373.

[51] Kadayif, I. et al. Studying interactions between prefetching and cache line turnoff. In Proceedings of the 2005 Asia and South Pacific Design Automation Conference (2005), pp.545-548.

[52] Rajan, K. and Ramaswamy, G., Emulating Optimal Replacement with a Shepherd Cache, In Proceedings of IEEE/ACM International Symposium on Microarchitecture (2007), pp. 445-454.

[53] Keramidas, G., Petoumenos, P., and Kaxiras, S. Cache replacement based on reuse-distance prediction. In Proceedings of the 25th International Conference on Computer Design (2007), pp. 245-250.

[54] Kharbutli, M., AND Solihin, Y. Counter-Based Cache Replacement and Bypassing Algorithms. IEEE Transactions on Computers. 57, (April 2008), pp.433-447.

[55] Kharbutli, M., Irwin, K., Solihin, Y., and Lee, J. Using prime numbers for cache indexing to eliminate conflict misses. In Proceedings of the 31th International Symposium on Computer Architecture (2004), pp. 288-299.

[56] Kim, C., Burger, D., and Keckler, S.W. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (2002), pp. 211-222.

[57] Kim, H. and Ahn, S. BPLRU: a buffer management scheme for improving random writes in flash storage. In Proceedings of the 6th USENIX Conference on File and Storage (2008), pp. 239-252.

[58] Koufaty, D. A., Chen, X., Poulsen, D. K. and Torrellas, J. Data forwarding in scalable shared-memory multiprocessors. In IEEE Transactions on Parallel and Distributed Systems. 7, 12 (1996), 1250-1264.

[59] Lai, A.-C., Fide, C., and Falsafi, B. Dead-Block Prediction and Dead-Block Correlating Prefetchers, In Proceedings of the 28th International Symposium on Computer Architecture (2001), pp. 144-154.

[60] Lee, D. et al. 1999. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies, In Proceedings of the 1999 ACM international conference on Measurement and modeling of computer systems (May, 1999), pp. 134-143.

[61] Lin, W.-F., and Reinhardt, S. "Predicting Last-Touch References under Optimal Replacement," Technical Report CSE-TR-447-02, Univ. of Michigan, 2002.

[62] Liptay, J. S. Structural Aspects of the System/360 Model 85, Part II: The cache. IBM Journal of Research and Development. 7, 1, 1968, 15-21.

[63] Lira, J., Molina, C., and González, A. LRU-PEA: a smart replacement policy for non-uniform cache architectures on chip multiprocessors. In Proceedings of the 27th International Conference on Computer Design (2009).

[64] Luk, C.-K. and Mowry, T. C. Automatic compiler-inserted prefetching for pointer based applications. IEEE Transactions on Computers. 48, 2, (Feb. 1999), pp. 134-141.

[65] Megiddo, N. AND Modha, D. S. ARC: A self-tuning, low overhead replacement cache. In Proceeding of the 2nd USENIX Conference on File and Storage Technologies (2003).

[66] Megiddo, N. and Modha, D.S., Outperforming LRU with an adaptive replacement cache algorithm. Computer. 37, 4 (2004), 58-65.

[67] Mehrotra, S. and Harrison, L. Examination of a memory access classification scheme for pointer-intensive and numeric programs. In Proceedings of the 1996 International Conference on Supercomputing (May 1996), pp. 133–139.

[68] Mutlu, O., Stark, J., Wilkerson, C., and Patt, Y. N. Runahead execution: an effective alternative to large instruction windows. IEEE Micro. 23, 6 (November/ December 2003), 20–25.

[69] Milutinovic, V. et al. A new cache architecture concept: the split temporal/spatial cache, In Proceedings of 8th Mediterranean Electro technical Conference (1996).

[70] Nesbit, K. J. and Smith, J. E. Data cache prefetching using a global history buffer. In Proceedings of the Tenth IEEE Symposium on High-Performance Computer Architecture (February 2004), pp. 96-105.

[71] O'Neil, Elizabeth J. et al. The LRU-K page replacement algorithm for database disk buffering, In Proceedings of ACM SIGMOD Conference (1993), pp. 297–306.

[72] Palacharla, S and Kessler, R. E. Evaluating stream buffers as a secondary cache placement. In Proceedings of the 21st Annual International Symposium on Computer Architecture (April 1994), pp. 24–33.

[73] Park, S., et al. CFLRU: a replacement algorithm for flash memory. In Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems (2006), pp. 234-241.

[74] Perelman, E., Hamerly, G., Biesbrouck, M.G., Sherwood, T. AND Calder, B. Using Simpoint for Accurate and Efficient Simulation. SIGMETRICS Perform. Eval. Rev. 31, 1 (2003), 318–319.

[75] Petoumenos, P., Keramidas, G., and Kaxiras, S. Instruction-based reuse-distance prediction for effective cache management. In Proceedings of the International Symposium on Systems, Architectures, Modeling, and Simulation (2009), pp. 49-58.

[76] Qureshi, M.K., Thompson, D., and Patt, Y.N. The v-way cache: Demand based associativity via global replacement. In Proceedings of the 32nd International Symposium on Computer Architecture (2005), pp. 544-555.

[77] Qureshi, M. K. et al. A case for MLP-aware cache replacement. In Proceedings of the 33rd International Symposium on Computer Architecture (2006), pp. 167-178.

[78] Qureshi, M. K. et al. Adaptive Insertion Policies for High-Performance Caching. In Proceedings of the 34th International Symposium on Computer Architecture (2007), pp. 381-391.

[79] Rivers, J.A., et al. 1998. Utilizing reuse information in data cache management, in Proceedings of the 12th international conference on Supercomputing (1998), pp.449-456.

[80] Robinson, J. AND Devarakonda, M. Data Cache Management Using Frequency-Based Replacement. In Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems (1990), pp.134-142.

[81] Roth, A., Moshovos, A. and Sohi, G. S. Dependence based prefetching for linked data structures. In Proceedings of the Eighth ASPLOS (October 1998), pp.115-126.

[82] Seznec, A. A case for two-way skewed-associative caches. In Proceedings of the 20th International Symposium on Computer Architecture (1993), pp.169-178.

[83] Smaragdakis, Y., Kaplan, S., and Wilson, P. The EELRU adaptive replacement algorithm. Performance Evaluation. 53, 2 (2003), 93–123.

[84] Smith, A. J. Cache Memories. ACM Computing Surveys. 14, 3 (September 1982), 473-530

[85] Smith, A. J. Disk cache-miss ratio analysis and design considerations. ACM Transactions on Computer Systems. 3, 3 (1985), pp. 161-203.

[86] Smith, J. E. Decoupled access/execute computer architectures. In Proceedings of the 9th annual symposium on Computer Architecture (June 1982), pp. 112-119.

[87] Smith, J.E., Characterizing computer performance with a single number. Communications of the ACM. 31, 10 (1988), pp. 1202-1206.

[88] So, K. AND RECHTSCHAFFEN, R. N. Cache Operations by MRU Change, IEEE Transactions on Computers. 37, 6 (1988), pp. 700-709.

[89] Stolte, C., Bosch, R., Hanrahan, P., Rosenblum, M. Visualizing application behavior on superscalar processors, In Proceedings of IEEE Symposium on Information Visualization (1999), pp. 10-17.

[90] Srikanth, T. et al. Continual flow pipelines. In Proceedings of the ASPLOS XI (October 2004), pp. 107-119.

[91] Srinath, S. et al. Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers. In Proceedings of the 13th Int'l Symposium on High Performance Computer Architecture (2007), pp.63-74.

[92] Subramanian, R., Smaragdakis, Y., AND Loh, G. H. Adaptive caches: Effective shaping of cache behavior to workloads. In Proceedings of the International Symposium on Microarchitecture (2006), pp.385-396.

[93] Suh, G.E., Rudolph, L., and Devadas, S., Dynamic partitioning of shared cache memory. The Journal of Supercomputing. 28, 1 (2004), 7-26.

[94] Sundaramoorthy, K., Purser, Z., and Rotenberg, E. Slipstream processors: improving both performance and fault tolerance. In Proceedings of the ASPLOS IX (November 2000), pp. 257-268.

[95] Standard Performance Evaluation Corporation. Available from http://www.specbench.org/osg/cpu2000/; accessed in 2010.

[96] Tyson, G., Farrens, M., Matthews, J., Pleszkun, A. R. A modified approach to data cache management. In Proceedings of the 28th annual international symposium on Microarchitecture (1995), pp. 93-103.

[97] Vandierendonck, H. et al. XOR-based hash functions. IEEE Transactions on Computers. 54, 7 (2005), 800-812.

[98] Wang, Z. et al. Guided region prefetching: A cooperative hardware/software approach. In Proceedings of the 30th Annual International Symposium on Computer Architecture (June 2003), pp. 388-398.

[99] Weaver, C. T. Pre-compiled SPEC2000 Alpha Binaries. Available from http://www.simplescalar.org; accessed in 2000.

[100] Wilkes, M. V. The memory wall and the CMOS end-point. ACM Computer Architecture News. 23, 4 (September 1995), pp. 4-6.

[101] Wong, W., AND Baer, J.-L. Modified LRU Policies for Improving Second-Level Cache Behavior, In Proceedings of the 6th Int'l Symposium on High Performance Computer Architecture (2000), pp. 49-60.

[102] Wood, D., Hill, M., AND Kessler, R. A Model for Estimating Trace-Sample Miss Ratios, In Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (1991), pp.79-89.

[103] Wulf, W. A et al. Hitting the memory wall, implications of the obvious. ACM Computer Architecture News. 23, 1 (1995), 20-24.

[104] Yoon, J., Min, S.L., and Cho, Y. Buffer cache management: predicting the future from the past. In Proceedings of International Symposium on Parallel Architectures, Algorithms and Networks (2002), pp.105-110.

[105] Zhang, C. Balanced instruction cache: reducing conflict misses of direct mapped caches through balanced subarray accesses, IEEE Computer Architecture Letter.5, 1 (2006), 2-5.

[106] Zhang, C., Vahid F., and Najjar, W. A highly configurable cache architecture for embedded systems, in Proceedings of the 30th International Symposium on Computer Architecture (June 2003), pp.136-146.

[107] Zhang, C., Vahid F., and Najjar, W. Energy benefits of a configurable line size cache for embedded systems, in Proceedings of IEEE International Symposium on VLSI Design (2003), pp. 87-91.

[108] Zhang, C., Vahid F., and Lysecky, R. A self-tuning cache architecture for embedded systems, in the Special Issue on Dynamically Adaptable Embedded System, ACM Transactions on Embedded Computing Systems. 3, 2 (May 2004), 1-19.

[109] Zhang, C. and Vahid, F. Using a victim buffer in an application-specific memory hierarchy. In Proceedings of the conference on Design, automation and test in Europe - Volume 1 (2004), pp. 220-227.

[110] Zhang, C., and Xue, B. Two dimensional highly associative level-two cache design. In Proceedings of the 26th International Conference on Computer Design (2008), pp.679-684.

[111] Zhang, C., and Xue, B. Divide-and-conquer: a bubble replacement for low level caches. In Proceedings of the 23rd International Conference on Supercomputing (2009), pp.80-89.

[112] Zhang, C., and Xue, B. A Tag-Based Cache Replacement, In Proceedings of the 28th International Conference on Computer Design (Oct. 2010).

[113] Zhou, P. et al. Dynamic tracking of page miss ratio curve for memory management. In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (2004), pp.177-188.

[114] Zhou, Y., Chen, P. M., and Li, K. The multi-queue replacement algorithm for second level buffer caches. In Proceedings of the 2001 USENIX Technical Conference (June 2001), pp. 91-104.

[115] Zhu, Q., Shankar, A., and Zhou, Y. PB-LRU: a self-tuning power aware storage cache replacement algorithm for conserving disk energy. In Proceedings of the 18th annual international conference on Supercomputing (2004), pp.79-88.

VITA

Bing Xue is a Ph.D. candidate in Department of Computer Science and Electrical Engineering at University of Missouri–Kansas City (UMKC). His research interests include high-performance computing and low energy embedded system design with Dr. Chuanjun Zhang. Prior to attending UMKC, he received BS and MS degrees from Beijing Jiaotong University and Florida International University, in 1998 and 2005, respectively.

In August 2006, he joined the Interdisciplinary Ph.D. program at UMKC with Electrical Engineering and Computer Science as his Coordinating discipline and Co-discipline, respectively. During his Ph.D. study, he published three conference papers in IEEE 26th ICCD, IEEE 28th ICCD, and ACM 23rd ICS. Additionally, he submitted one journal manuscript to ACM TACO.

After completion of his degree requirements, he will pursue opportunities in either industry or academia, where he can apply and continue to do research in related fields.