ANALYZING INSTRUCTTION BASED CACHE REPLACEMENT POLICIES

by

PING XIANG
B.E. Huazhong University of Science and Technology, 2008

A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science
in the School of Electrical Engineering and Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Summer Term
2010

Major Professor. Huiyang Zhou

ABSTRACT

The increasing speed gap between microprocessors and off-chip DRAM makes last-level caches (LLCs) a critical component for computer performance. Multi core processors aggravate the problem since multiple processor cores compete for the LLC. As a result, LLCs typically consume a significant amount of the die area and effective utilization of LLCs is mandatory for both performance and power efficiency.

We present a novel replacement policy for last-level caches (LLCs). The fundamental observation is to view LLCs as a shared resource among multiple address streams with each stream being generated by a static memory access instruction. The management of LLCs in both single-core and multi-core processors can then be modeled as a competition among multiple instructions. In our proposed scheme, we prioritize those instructions based on the number of LLC accesses and reuses and only allow cache lines having high instruction priorities to replace those of low priorities. The hardware support for our proposed replacement policy is light-weighted. Our experimental results based on a set of SPEC 2006 benchmarks show that it achieves significant performance improvement upon the least-recently used (LRU) replacement policy for benchmarks with high numbers of LLC misses. To handle LRU-friendly workloads, the set sampling technique is adopted to retain the benefits from the LRU replacement policy.

ACKNOWLEDGMENTS

This thesis would not have been possible without the help and support of a number of people. First and foremost, I would like to express my sincerest gratitude to my advisor, Dr. Huiyang Zhou, for the tremendous time, energy and wisdom he invested in my graduate education. His inspiring and constructive supervision has always been a constant source of encouragement for my study. I also want to thank my other thesis committee members, Dr. Mark Heinrich, and Dr. Liqiang Ni, for spending their time to review the manuscript and providing valuable comments.

I would like to thank my group members: Jingfei Kong, Martin Dimitrov, Yi Yang, Zhaoshi Zheng and Jacob Staples. I want especially to thank Yi, who is my best friend and lab mate, for the inspiring discussions and sharing each step of graduate study with me. I would also like to give a special thanks to Jingfei, for his tremendous help during my graduate study. My enormous debt of gratitude goes to Jacob Staples, who proof-read my dissertation and provided many suggestions to help me improve it.

I dedicate this thesis to my family: my parents Xiaoxin Xiang and Yingxiang Ming, for all their love and encouragement through my life.

Last but not least, I would also like to extend my thanks to my friends who have cared and helped, in one way or another. My graduate studies would not have been the same without them.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1. INTRODUCTION

1.1. Motivation

Microprocessor-based systems have inspired and revolutionized the scientific and engineering communities for over 40 years. The four decades of the history of microprocessors also tell a remarkable story of dramatic technological advances driven by Moore's Law, which advocates that integrated circuits effectively double in performance every 18 months. While Moore's Law is still expected to hold during the next decade, we believe that the rate of improvement in microprocessor speed will continue.

However, this phenomenal increase of microprocessor speed places significant demands on the memory system in both memory latency and capacity aspects. Unfortunately, due to semiconductor manufacturing technology and cost issues, the rate of improvement in microprocessor speed has largely exceeded the rate of improvement in DRAM memory latency over the past few years. The tradeoff between memory capacity and performance makes this processor-memory gap even worse, as shown in Figure1[1].

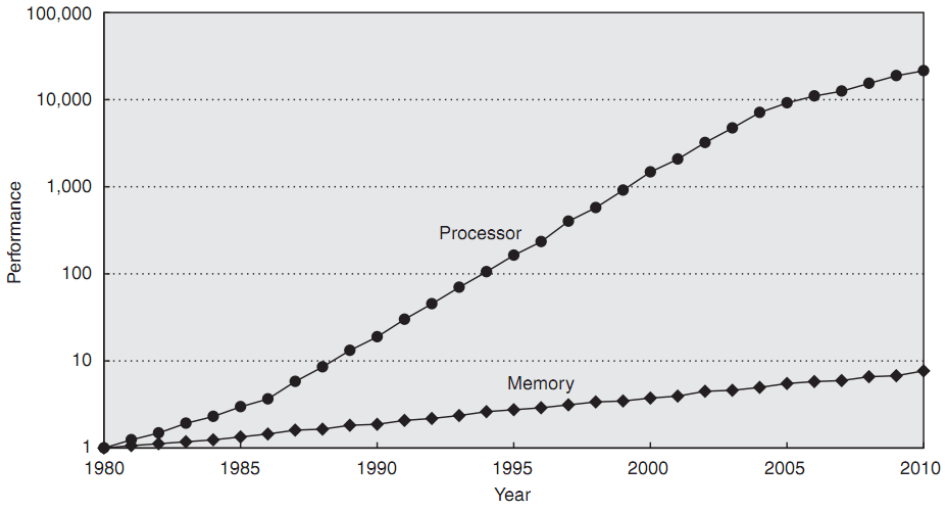

Figure 1.The processor-memory gap, taking performance in 1980 as a baseline

Thus, the increasing speed gap between microprocessors and off-chip DRAM makes a cache-based memory hierarchy a critical component for computer performance. Multi-core processors aggravate the problem since multiple processor cores compete for the last-level cache (LLC). As a result, LLCs typically consume a significant amount of the die area and effective utilization of LLCs is mandatory for both performance and power efficiency.

Given its significant impact on cache hit rates, the replacement policy of a cache plays an important role in filling the speed gap and sustaining the performance growth of microprocessors. Many approaches have been proposed to exploit the principle of locality, like probabilistic analysis [2], data reuse frequency [3], Re-Reference Interval Prediction [8] and etc., yet most of them are based on the analysis of address streams to determine reuse patterns and then design replacement policies accordingly.

1.2. Contribution

This thesis tackles the problem from a new perspective. Instead of inspecting address streams, we examine the memory access instructions that generate the address streams and view cache sharing as a competition of multiple (static) instructions. With this perspective, a unified LLC replacement policy is proposed for both single-core and multi-core processors.

Our experiments show that the majority of memory accesses are usually issued by a small number of access instructions, which are critical for the overall performance. And the memory accesses issued by these access instructions may have different reuse-patterns as well as hit/miss rates.

Our goal, therefore, is to analyze the different hit/miss rates and reuse-patterns of memory accesses issued by these instructions, and prioritize these memory accesses issued by different access instructions accordingly.

In our approach, we first prioritize memory access instructions using the following criteria: the instructions generating a large number of LLC accesses and high hit rates have high priorities while instructions with small numbers of accesses and low hit rates are assigned with low priorities. The replacement policy is such that a cache line is only allowed to be replaced by those having a higher priority. Since most LLC accesses are generated by only a few static memory access instructions, maintaining such instruction-based priorities incurs very limited hardware overhead. The instruction-based information is also used to detect streaming data so that it can bypass the LLC. Furthermore, replacement using instruction-based priority can be integrated with data prefeching techniques [4] [5] and criticality analysis [6]. If an instruction is

on the critical path, e.g. leading to a branch misprediction, it can be prioritized to occupy more cache space.

1.3. Organization of the Thesis

Chapter 2 presents a background on the study of cache replacement policy, including commonly used LRU (least recent used), Random replacement policies. We also discuss some multi-core focused cache replacement policies that have been proposed recently. Chapter 3 talks about our instruction-based cache replacement policy in detail, including the architectural design and storage concerns. Experimental results and analysis are shown in chapter 4. Finally, conclusions are drawn in chapter 5.

CHAPTER 2. BACKGROUND

Given its impact on cache performance, the cache replacement policy has been studied extensively and many replacement policies have been proposed. In this chapter, we briefly talk about selected work relevant to cache replacement policies.

2.1. LRU

According to the LRU (least recent used) algorithm, the least recently used cache line will always be selected as the victim to be replaced when a conflict occurs. The newly arrived cache line will be assigned as most recently used, so this policy effectively keeps track of the access order of cache lines in the same set. This requires an LRU bit for each cache line and needs to update all the cache lines in the set once one of the cache lines is used.

Based on LRU, a number of caching algorithms are proposed to reduce implementation cost or better exploit cache locality, such as the MRU, Pseudo-LRU [7] and adaptive insertion [8] policies.

Although LRU is very simple, it exploits the principle of locality well for many applications and thus is largely used in modern cache structures.

2.2. Utility-based Cache Partition

The utility-based cache partition (UCP) [9] was proposed to manage the shared cache among multi core processors. It tries to partition the cache resource on a utility basis, unlike LRU

which explicitly partitions the cache on a demand basis (the core that issues more memory accesses will be able to occupy a larger portion of the cache). Each core is assigned a utility monitoring circuit (UMON) which tracks the utility information of the application executing on the core as described in Figure 2. The UMON is expected to sample the shared cache by storing the cache lines issued by its owned core within that set range. A hit-counter is maintained to store the hits among different ways.

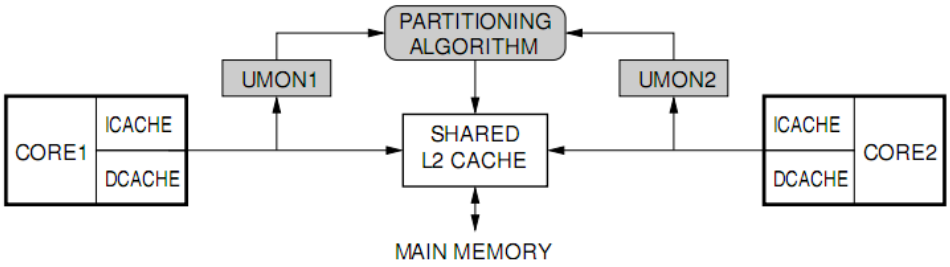

Figure 2.Architecture of Utility-Based Cache Partitioning

After collecting the reuse information from different UMONs, the partitioning algorithm will try to partition the way of the shared cache in a manner that minimizes the total misses and maximizes the total hits incurred by all applications.

By doing this, the UCP attempts to assign more cache ways to the cores that are more likely to benefit from larger cache sizes. At the same time, those cores running with a saturating utility application (an application having a small working set that fits in a small size cache) can be assigned a relatively small number of ways, so that the total miss rate can be reduced.

However, this scheme is more effective when the applications running on a multi core machine have different cache utilization features.

2.3. Promotion/Insertion Pseudo-Partitioning (PIPP)

Another algorithm called Promotion/Insertion Pseudo-Partition (PIPP) [10] was proposed to partition the cache implicitly instead of through explicit way-partitioning.

The PIPP algorithm decomposes the cache management policies into 3 parts: victim selection, insertion policy and promotion policy. It also maintains an access order for each cache set; when a miss occurs, the least recently accessed cache line will be selected as the victim as in any LRU-based replacement policy. It also has monitoring circuits similar to UCP which are used to monitor the utility information of each core and calculate the best way partitioning configuration. However, instead of explicit partitioning the whole shared cache, the configuration information is used to decide where to insert incoming cache lines. For example, for a 4-core system with a 16-way set associative last-level shared cache, core 1 is assigned 6 of the total 16 ways. PIPP thus will insert the memory data from that core into the 6th place of the total 16 LRU list. When there is a hit in the cache, instead of promoting that set into the Most Recently Used (MRU) place, it either promotes the cache line one step ahead or does not promote the cache line at all.

Y. Xie, et. al, demonstrate that PIPP is more effective at controlling the occupancy of shared caches than way partitioning algorithms. They also show that PIPP balances the per-core way configurations in a fashion that comes reasonably close to the ideal cache way allocation for each core while reducing the amount of time that dead-on-arrival cache lines reside in the shared cache.

CHAPTER 3. INSTRUCTION-BASED PRIORITIZATION

As mentioned in the previous chapter, most existing cache replacement algorithms are based on the analysis of address streams, determining the reuse patterns in those streams and then designing replacement policies accordingly. We, however, model cache sharing (for both single and multi-core machines) as a competition among multiple static memory access instructions and our proposed replacement policy is based on the prioritization of those instructions.

3.1. Overall design structure

As we can see from the architecture of our design in Figure 3, there are three hardware modifications to the cache: the PC-index field appending to the tag of each cache line, the instruction priority table and the instruction filter.

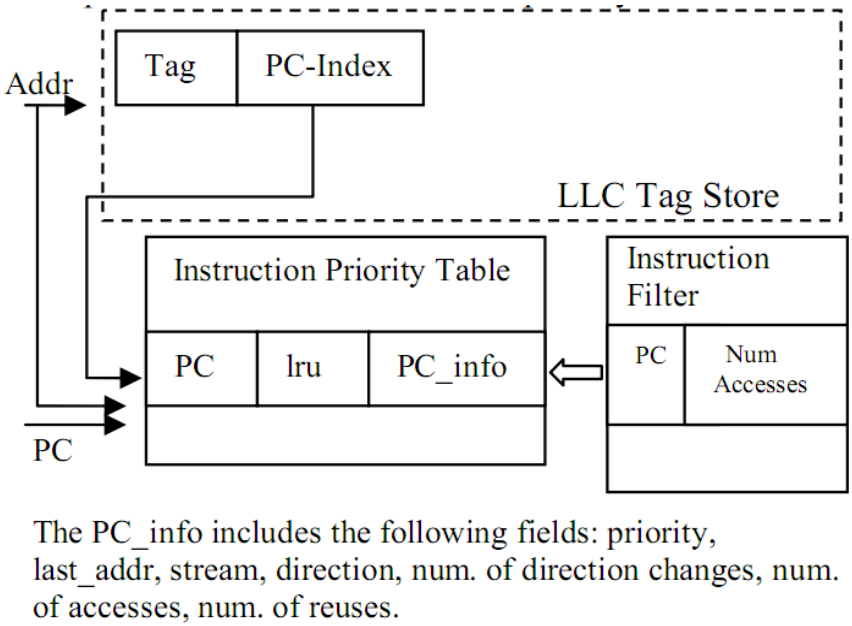

Figure 3.Architecture of Instruction-based Prioritization

The instruction priority table is a fully associative structure. It maintains the information of memory access instructions (e.g. PC address of that instruction, total hits/misses due to that instruction. etc) that frequently access the current level cache. Access order of these instructions is also kept so that this table can be maintained in an LRU manner.

The instruction filter is also a fully associative structure. It is used to filter out rarely accessed memory load/store instructions. It is organized as a circular buffer. The filter keeps track of the number of LLC accesses of each instruction. If the number of accesses of an instruction reaches a threshold within a timer interval, the instruction is then promoted into the instruction priority table. By doing this, we maintain a relatively smaller instruction table and reduce the storage cost.

3.2. Principle of Instruction-Based Prioritization

The LLC with the added structures operates as follows. For an access to the LLC at data address X generated from the instruction p, the LLC control logic decodes the address X and checks whether it is a hit or miss. Then, both the instruction priority table and the instruction filter are updated and a cache line replacement may be performed in the case of a miss.

Both the data address, X, and the program counter (PC), p, are used to update the instruction priority table as shown in Figure 4. If p hits in the instruction priority table, the associated pc_info field is updated, including the number of accesses and the number of reuses/hits to that instruction. The states related to streaming data detection are also updated. The 1-bit direction flag shows whether the instruction accesses data with an increasing or decreasing address offset. If 'direction' disagrees with the sign of (current address X – last address), the number of direction changes is incremented. The following condition is used to detect whether an instruction is accessing streaming data: (num. of accesses > threshold1) && (num. of reuses < threshold2) && (num. direction changes < threshold3). In other words, if the instruction generates many LLC accesses with few reuses and its data addresses keep increasing or decreasing, we decide that this instruction is accessing streaming data. In this case, we set the 'stream' flag so that data accessed by this instruction can be bypassed from LLC.

After streaming data detection, the last address and the direction flag are updated. If p misses in the instruction priority table, the instruction filter is searched. If p hits in the filter, the corresponding number of accesses is incremented. If the access number is large enough, the instruction is promoted to the instruction priority table. If p misses in the instruction filter, a new

entry is allocated in the circular buffer to start tracking how often this instruction accesses the LLC.

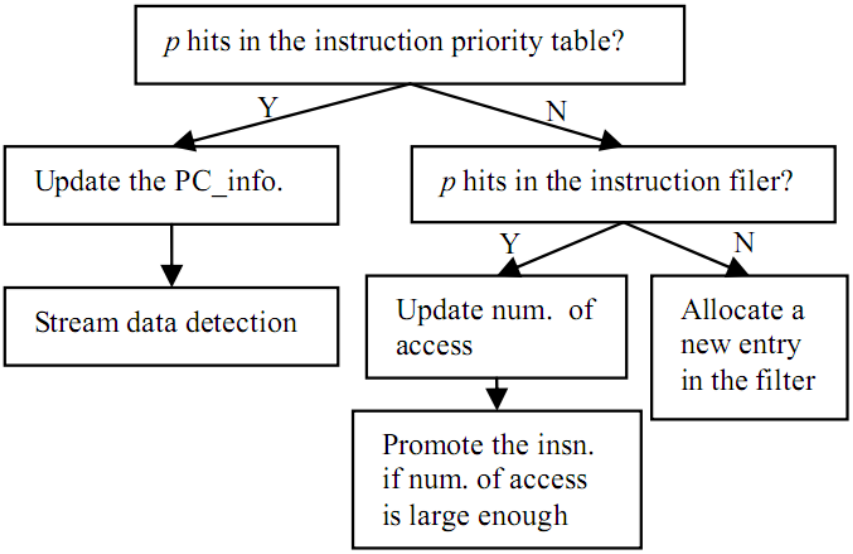

Figure 4.The logic for updating the instruction priority table and the instruction filter. p is the program counter (PC) of the instruction generating the data access

After we use the instruction priority table to monitor those instructions that frequently access the LLC for a certain interval, we set the priority among the instructions based on their number of hits/reuses. If there is a tie, we use the number of accesses to break the tie. When the data access at address X results in a cache miss and the instruction p is not accessing streaming data, a victim may be selected and replaced with the incoming cache block. The replacement policy is described with the pseudocode shown in Figure 5.

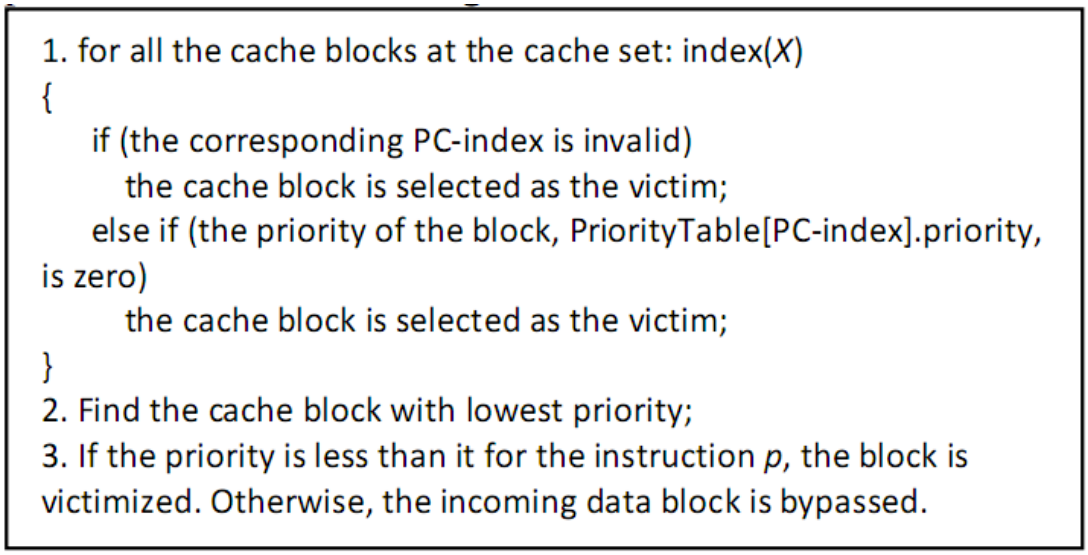

Figure 5.The replacement policy using instruction-based priority

From Figure 5, we can see that the replacement policy first chooses to victimize the blocks that have either an invalid PC index (meaning that the instruction accessing the block is not present in the priority table) or zero priority. If there is no such block, it searches for the block with the lowest priority (using Priority Table [PC- index].priority) and compares the priority against the incoming data block. If the incoming data block has a higher priority, the replacement is performed. Otherwise, the incoming block is bypassed from the LLC.

A subtle question related to the replacement policy in Figure 3 is that before the priorities are set up, what replacement policy is used? The answer is that since all cache blocks have zero priority initially and we follow the same search order through the blocks, we always replace the most-recently inserted (MRI) data blocks before the priorities are set up. The goal is to allow data to stay in the LLC long enough to account for long reuse distances.

Since our replacement policy is designed to capture long reuse distances, it does not fit well with applications featuring short reuse distances for which the least recently used (LRU) policy works well. To combine the benefits of both policies, the set sampling technique proposed

in [13] is adopted such that certain cache sets follow the LRU policy while the rest use instruction-based prioritization for replacement. The hit rates for both policies are tracked to determine the winner. Since, for any cache set, either the LRU or instruction priority replacement is used, we can reuse the PC-index field as the LRU bits. In other words, for the cache sets using the LRU policy, we do not keep track of their instruction information. For the cache sets using the instruction priority replacement scheme, the LRU information is lost. When the replacement policy changes from instruction priority to LRU, the LRU bits are simply re-initialized.

Besides the set-dueling technique, we also designed another version of our PC-prioritization policy: LRU-based PC prioritization. While the hit rates of the LRU set samples are still collected and compared to our PC prioritization policy, we also collect the hit number of the PCs in these LRU sets. When PC prioritization fails to outperform the LRU policy, instead of letting the whole cache switch to LRU we set the priority of each PC according to the corresponding hit number in the 32 LRU sets.

By using the LRU-based PC prioritization scheme, we can utilize the LRU policy to quickly pick up PCs with high reuse rates and replace non-active PCs that previously showed high reuse features.

Our proposed replacement policy is essentially the same for single-core and multi-core processors. Two differences are first that the PC-index field includes the thread ID information and second that the instruction priority table and instruction buffer can be either private or shared among the different cores.

3.3. Storage Cost

The storage cost of our proposed replacement policy mainly comes from the PC-index field, the instruction priority table and the instruction filter. Next, we itemize their storage requirement based on our current design implementation.

3.3.1 PC-index field

We use a 31-entry instruction priority table. Therefore, each PC-index field takes 5 bits. The value 0b11111 is reserved as the invalid PC index. For the 1MB cache with the block size of 64 bytes, the overall PC-index field takes 1k cache sets x16 blocks per set x 5 bits per block = 80k bits. For the 4MB cache with a block size of 64 bytes, the overall PC-index fields take 4k cache sets x 16 blocks per set x 5 bits per block = 320k bits. If we include the 2-bit thread id information in the PC-index, the storage becomes 4k x 16 x (5+2) = 448k bits. As discussed in chapter 3, the PC-index field bits are also reused as LRU bits (4 bits per block) when the sets are not selected to use the instruction-based priority replacement policy. Therefore, we do not need to allocate separate storage for LRU bits.

3.3.2 The instruction priority table

In each entry of the table, there are the following fields: PC (32bits), LRU (5 bits), priority (6 bits), last address (26 bits), stream flag (1 bit), direction flag (1 bit), number of direction changes (3 bits), number of accesses (16 bits), and number of reuses (16 bits). As there are 31 entries in the table, it costs 31x (32+5+6+26+1+1+3+16+16) = 3286 bits. For the multi-core configuration there are at most four priority tables, one for each core. Therefore, the cost is 3286x4 = 13144 bits.

3.3.3 The Instruction filter

Each entry in the filter has two fields: PC and access number. Since we choose to promote an instruction into the priority table once the access number reaches 256, we need an 8-bit counter for this field. We also choose to have 256 entries in the filter. Thus, the overall storage cost is 256 x (32+8) + 8 (the tail pointer used for circular buffer) = 10248 bits. For multi-core configurations we use at most four filters, one per core. The cost is 4 x 10248 = 40992 bits.

Additionally, 32-bit counters for overall hits/reuses, overall accesses, hits in sets using the LRU policy, and accesses to sets using the LRU policy are required. Another 4-bit counter for selecting replacement policy from either LRU or instruction-based prioritization is also required for a total of 4 x 32 + 4 = 132 counter bits.

In summary, for single-core configurations the storage cost of our design is 80k (PC-index) + 3286 (instruction priority table) + 10248 (instruction filter) + 132 (counters) = 95586 bits or 93.3kbits. For 4-core configurations the cost is 448k (PC-index) + 13144 (four instruction priority tables) + 40992 (four instruction filters) + 132 (counters) = 513020 bits or 501k bits.

CHAPTER 4. EXPERIMENTAL RESULTS

4.1. Simulation Methodology

Table 1 shows the baseline configuration used in our experiments. We model the proposed replacement policy and the associated hardware structures using a modified version of CMP$im [14] [17], a Pin-based [15] trace-driven x86 simulator, for our performance studies.

Table 1.Baseline configuration

Processor core	an 8-stage, 4-wide pipeline, perfect branch prediction, 128-entry instruction window with no scheduling restrictions
	The L1 I-cache is 32KB 4-way set associative cache with LRU and 64B line size. The L1D-cache is 32KB 8-way set-associative with LRU and 64B line size. The L2 D-cache is 256KB, 8-way set-associative with LRU and 64B line size
LLC	16-way set-associative cache with 64B line size, 1MB for single core and 4MB for four-core.
Memory	200-cycle latency.

We used gcc 4.1.2 on a 32-bit x86machine to compile a set of memory intensive SPEC 2006 benchmarks. For each benchmark a trace was generated by skipping the first 40 billion instructions and recording the next 100 million instructions.

4.2. Single-core Results

Although our policy was designed to support multi-core applications, its single-core application performance is still useful as a comparison among different cache replacement policies.

Figure 6 shows the performance improvement measured in instructions per cycle (IPC) of our proposed PC prioritization policy. The IPC is normalized to the baseline LRU policy performance.

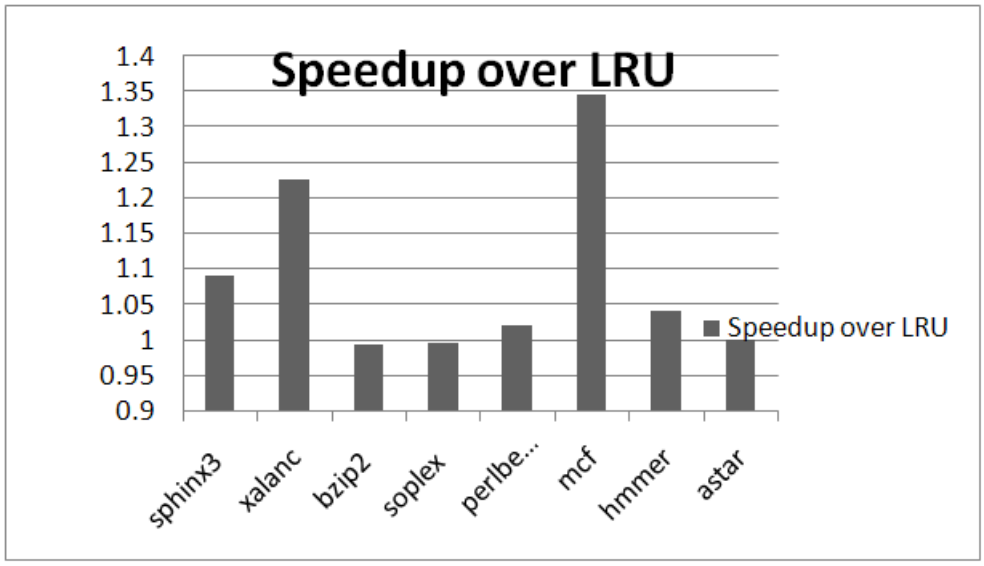

Figure 6.Single-Core results for our proposed PC-prioritization, normalized to LRU

From these results it can be seen that our policy works well with memory-intensive benchmarks like mcf (35%), xalanc (23%) and sphinx3 (9%). For benchmarks with smaller data sets, our policy can still achieve minor speedups as seen in hmmer (4%), perlbench (2%). Although our policy does not show much speedup on astar over LRU, it indeed reduces the number of cache misses by up to 24% for this benchmark. On the other hand, by using the set-sampling/dueling approach mentioned previously, the benefits of the LRU policy are mostly retained for LRU-friendly benchmarks like bzip2 and soplex.

Table 2.Total lookups in the LLC

Name of the benchmark	Misses in the LLC
482.Sphinx3	1601079
483.Xalancbmk	1149515
401.Bzip2	856023
450.Soplex	1912105
400.Perlbench	209229
429.Mcf	5102112
456.Hmmer	241126
473.Astar	375597

To further understand the characteristics of our policy, we present the total cache misses in the LLC in Table 2. It is obvious that our policy is more effective for benchmarks with larger data sets. The reason is that, for larger data sets, the reuse distance is usually longer compared with benchmarks having smaller data sets, meaning LRU cannot keep useful data long enough for it to be used. Although there are exceptions such as bzip2, the total number of misses in bzip2 is relatively large compared to hmmer and astar. Bzip2 also contains many memory accesses with short reuse distances, meaning LRU's performance is acceptable with a nearly 60% hit rate. Thus, it would be more accurate to say our policy favors applications with long reuse distances than favoring larger data sets alone.

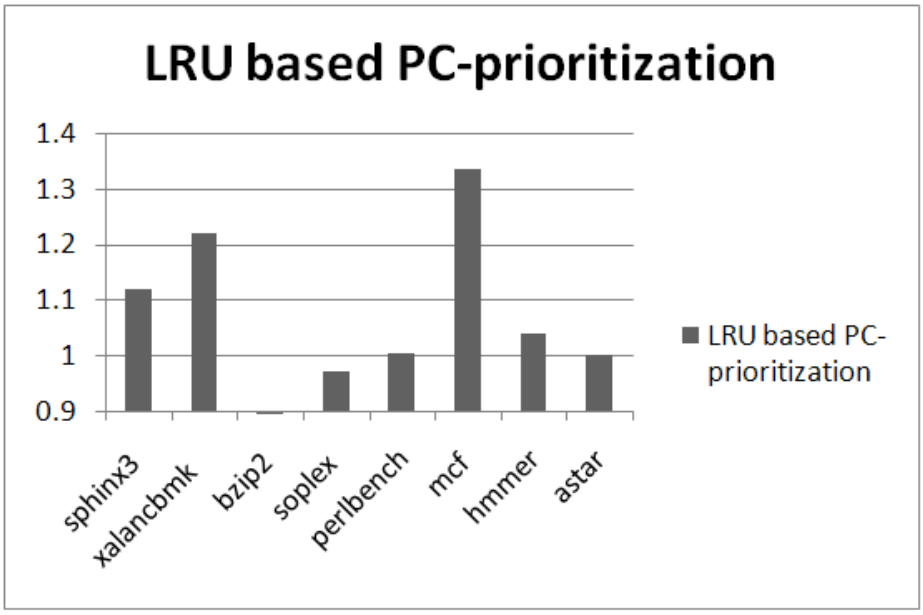

Figure 7.Single-Core results for our LRU-based PC-prioritization, normalized to LRU

Figure 7 shows our LRU-based PC prioritization results. Again, the results are normalized to the baseline LRU performance. This time, however, we do not switch the entire cache set to LRU when our policy fails to outperform the LRU. Instead, when our PC-prioritization policy fails to find the right PCs to prioritize, we count the number of hits in the LRU sets for each PC and use that information for prioritization. If our policy works better than LRU, we still take into account these LRU hits by adding the LRU hits and the hits gained by our policy together. In this fashion, those PCs corresponding to short reuse distance memory access patterns as well as those with long reuse distances will still be quickly found even if they were not given high priority in the beginning (which leads to low cache hit number) and would be overlooked during our first skim. However, the bzip2 benchmark also experiences a 10% slowdown because we do not revert to LRU when LRU outperforms our proposed policy.

The differences between these two policies can be better illustrated in terms of cache miss reduction as shown in Figure 8. For benchmarks with long reuse distances, these two

policies work almost equally well. Since we have shown LRU is not suitable for these benchmarks, using the LRU hit information when prioritizing PCs introduces some noise into our policy. This is why we observe a slight slowdown for these benchmarks. For soplex, sphinx3 and astar, we can see that considering the LRU hits for these benchmarks helps to quickly and more accurately identify useful PCs and reduce the total cache misses.

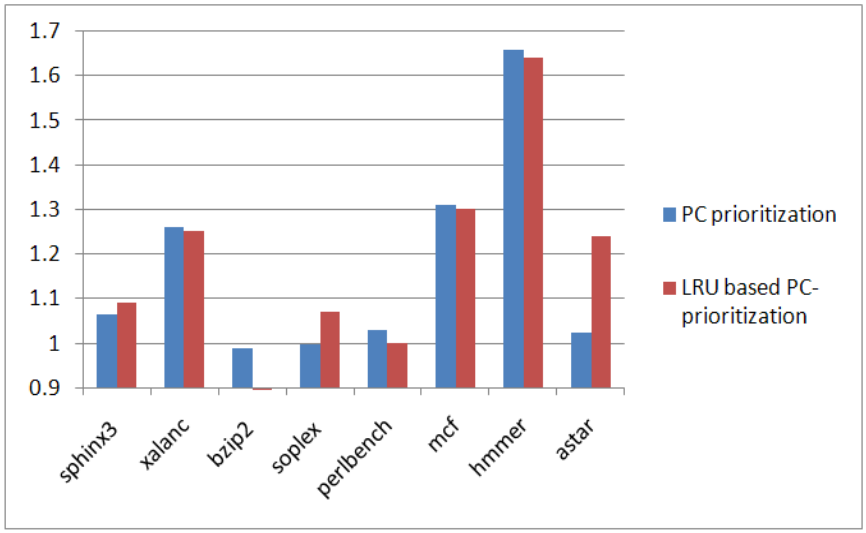

Figure 8.Total Cache Misses for PC-prioritization and LRU-based PC-prioritization, normalized to LRU

4.3. PC-based prioritization sensitivity to cache configuration

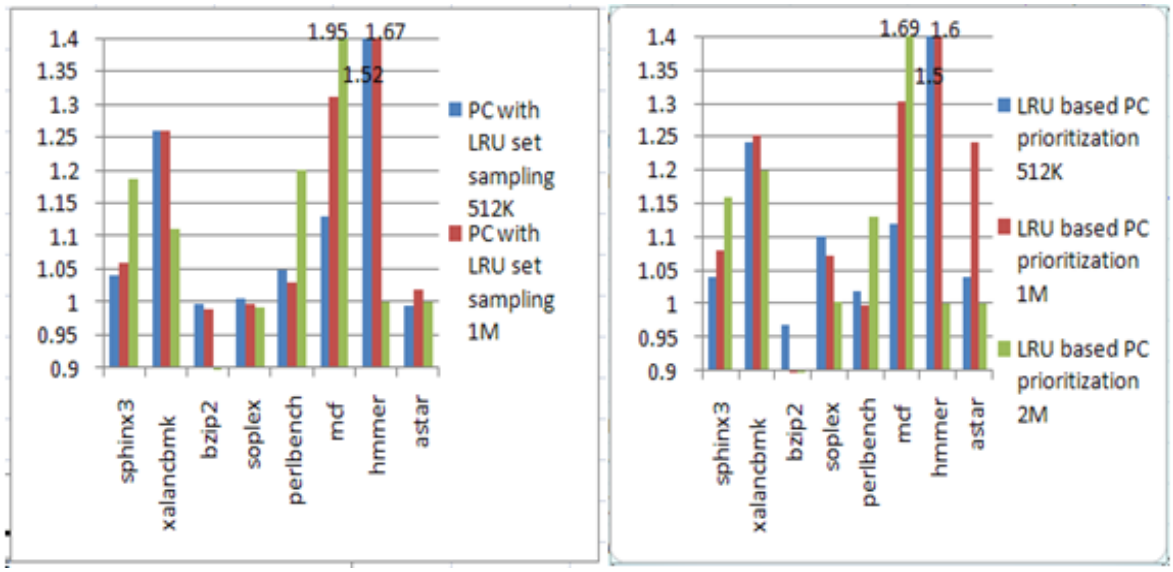

Figure 9.Number of cache misses for PC-prioritization and LRU based PC-prioritization, normalized to LRU

Figure 9 presents the performance of our proposed two policies on different LLC cache sizes: 512KB, 1MB and 2MB. The results show that both policies achieve speedups over LRU with larger cache sets. The reason the speedups are smaller for xalanc and hmmer with 2MB LLC is that 2MB is large enough for LRU to cache the entire data sets, thus it achieves similar performance (CPI, cache hits.ect) to PC prioritization.

4.4. Multi-core results

Given the large number of possible workload mixes, we choose to study the representative ones shown in Table 3.

Table 3.Multi-core Workload

MIX1: 429, 456, 403, 473	MIX2: 482, 456, 401, 473
MIX3: 450, 456, 403, 473	MIX4: 483, 456, 401, 473
MIX5: 400, 456, 403, 473	MIX6: 429, 482, 456, 403
MIX7: 450,483,401,473	MIX8: 429, 400, 456, 403
MIX9: 483, 400, 401, 473	MIX10: 429, 482, 450,456
MIX11: 429,483, 400, 403	MIX12: 429, 482, 483, 400

We choose 4 subsets of the 9 single-core benchmarks and merge them into 12, 4-way multiprogrammed workloads with the following criteria: MIX1 to MIX5 are composed from 1 memory intensive workload and 3 non-intensive ones, MIX6 to MIX9 are combined from 2 memory intensive workloads with 2 non-intensive ones, the rest of the workloads are primarily selected from memory intensive benchmarks.

All of workload mixes are simulated until each core has finished its own 100 million instructions. The faster workloads will be auto-rewound so that they can still be actively creating cache contention in the LLC. The final results will only take into account the first one million instructions of each core.

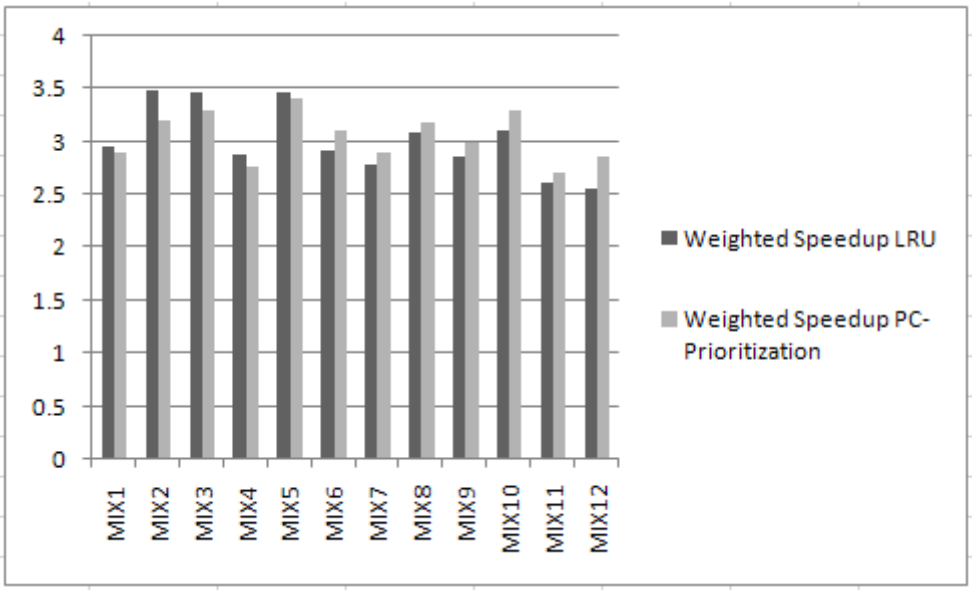

Figure 10.Weighted speedups of IPC for LRU and PC-prioritization with 4M LLC

Figure 10 shows the weighted IPC speedup for the LRU policy and our proposed PC prioritization policy. It can be seen that from MIX1 to MIX5, the LRU has a higher weighted IPC speedup. One reason is that, because MIX1 to MIX5 are composed of a single memory-intensive workload and 3 non memory-intensives ones, the IPCs of the 3 non memory-intensive workloads do not change much whether they share the LLC with other cores or occupy the LLC exclusively. Another reason is that our policy is good at long reuse distance workloads which are usually memory-intensive. Thus, we observe that with 2 or more memory-intensive workloads in the mix our policy outperforms LRU.

Figure 11 shows the normalized average CPI, which is computed using $(\frac{1}{4}\sum_i CPI_i^{NEW})/(\frac{1}{4}\sum_i CPI_i^{LRU})$ [11]. Our proposed PC-prioritization achieves much better performance when there are more memory intensive applications in the multi-core workload mix. Even among the first 5 mixes which contain only a single memory-intensive benchmark, our policy achieves similar or better performance than LRU.

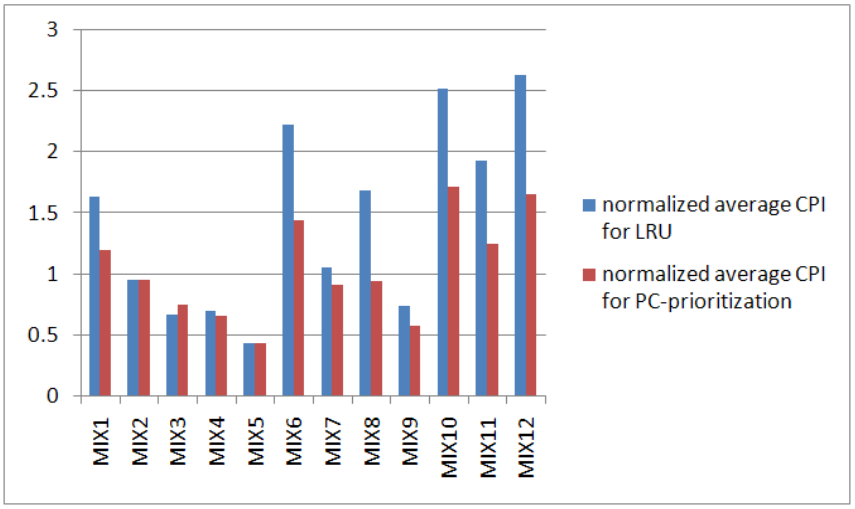

Figure 11.Normalized average CPI for LRU and PC-prioritization

Next we show the harmonic mean of weighted speedup $\frac{c}{\sum_{i=1}^{c}(IPC_{sa}[i]/IPC[i])}$ which accounts for both fairness and performance [16]. It can be seen from Figure 12 that the trend is very similar to weighted speed up.

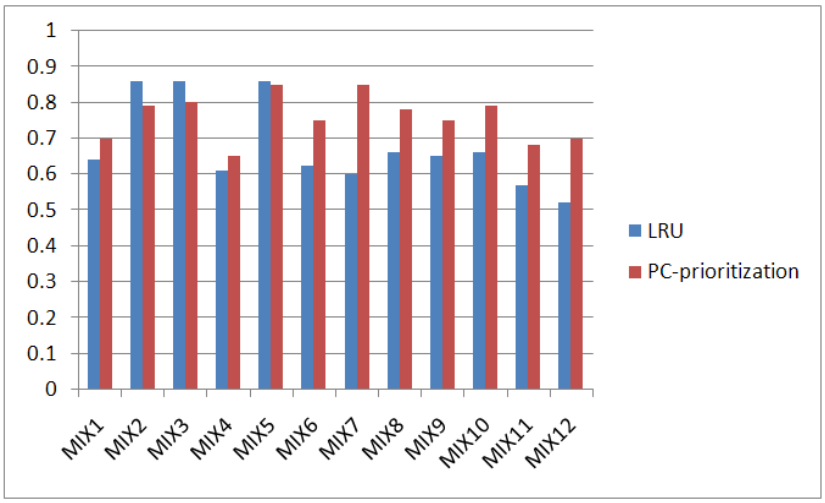

Figure 12.Harmonic weighted speedup for LRU and PC-prioritization

CHAPTER 5. CONCLUSIONS

In this chapter we review the contributions of this thesis and discuss future work related to the policy proposed in this thesis.

We have presented a unified cache replacement policy for both single-core and multi-core processors. The essence of our approach is that we view the cache as being shared by multiple static memory access instructions. From this perspective we can assign priorities to those instructions based on their potential data reuses and only allow cache blocks to be replaced by those having a higher instruction-based priority. By keeping data which has a higher likelihood of reuse in the cache, the overall performance of the cache can be improved. As last-level caches typically feature data blocks with long reuse distances, we use the most-recently-inserted replacement policy to estimate the data reuses of each instruction in order to set its priority. For workloads with short reuse distances, we adopt a set sampling/dueling technique to combine the benefits from the least-recently used (LRU) replacement policy. The performance results based on a set of SPEC 2006 benchmarks show that the proposed replacement policy achieves significant speedups over the LRU policy for both single-core and multi-core processors.

Although our proposed policy works well with workloads whose PCs have long reuse distances, it fails to outperform LRU for workloads whose memory accesses exhibit short reuse distances. Even for PCs with long reuse distances between memory accesses, it would be beneficial to differentiate between long and short reuse distance accesses.

Since our policy can extract those PCs that issue the majority of memory accesses in the workload, it can be combined with data prefetching to prefetch useful data blocks. Moreover, we can use data prefetching structures to identify cache misses that could be potentially be a cache hit with proper replacement strategy.

Finally, our policy updates the priority table of the PCs based on an arbitrarily chosen time interval. It would be beneficial if we could automatically detect changes in program behavior at run time and update the priority table on demand when the PC is inactive or less active rather than simply waiting some arbitrarily chosen amount of time before reprioritizing.

LIST OF REFERENCES

[1] John L. Hennessy, David A. Patterson, Computer Architecture A Quantitative Approach, Elsevier, Inc, 2007.

[2] L. Belady, A study of replacement algorithms for virtual-storage computer, IBM Systems Journal 1966.

[3] John T. Robinson, Murthy V. Devarakonda, Data cache management using frequency-based replacement. SIGMETRICS 1990

[4] K. Nesbit, A. Dhodapkar, and J. E. Smith, AC/DC: An adaptive data cache prefetcher, PACT 2004

[5] M. Dimitrov, H. Zhou Combining Local and Global History for High Performance Data Prefetching, JILP 2009

[6] Brian Fields, Shai Rubin, Rastislav Bodık. Focusing processor policies via critical-path prediction, ISCA 2001.

[7] Hassan Ghasemzadeh, Sepideh S. Mazrouee, Mohammad R. Kakoee, Modified Pseudo LRU Replacement Algorithm. ECBS 2006

[8] M. Qureshi, et. al., Adaptive insertion policies for high performance caching, ISCA 2007.

[9] Moinuddin K. Qureshi, Yale N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches, MICRO 2006

[10] Yuejian Xie, Gabriel H. Loh. PIPP: Promotion/Insertion Pseudo-Partitioning of Multi-Core Shared Caches, ISCA 2009

[11] M. Chaudhuri. Pseudo-LIFO: The Foundation of a New Family of Replacement Policies for Last-level Caches. Micro 2009.

[12] Aamer Jaleel, Kevin Theobald, Simon C. Steely Jr, and Joel Emer. High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP). ISCA 2010

[13] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon C. Steely Jr, and Joel Emer. Adaptive Insertion Policies for Managing Shared Caches on CMPs. PACT 2008

[14] A. Jaleel, R. Cohn, C. K. Luk, B. Jacob. CMP$im: A Pin-Based On- The-Fly Multi-Core Cache Simulator. MoBS 2008.

[15] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, K. Hazelwood. "Pin: building customized program analysis tools with dynamic instrumentation. PLDI 2005.

[16] J. Chang, G. Sohi. Cooperative Cache Partitioning for Chip Multiprocessors. ISC 2007.

[17] The official CRC website: http://www.jilp.org/jwac-1/