ABSTRACT

Title of dissertation:	Using Locality and Interleaving Information to Improve Shared Cache Performance
	Wanli Liu, Doctor of Philosophy, 2009
Dissertation directed by:	Professor Donald Yeung Department of Electrical and Computer Engineering

The cache interference is found to play a critical role in optimizing cache allocation among concurrent threads for shared cache. Conventional LRU policy usually works well for low interference workloads, while high cache interference among threads demands explicit allocation regulation, such as cache partitioning. Cache interference is shown to be tied to inter-thread memory reference interleaving granularity: high interference is caused by fine-grain interleaving while low interference is caused coarse-grain interleaving. Profiling of real multi-program workloads shows that cache set mapping and temporal phase result in the variation of interleaving granularity. When memory references from different threads map to disjoint cache sets, or they occur in distinct time windows, they tend to cause little interference due to coarse-grain interleaving. The interleaving granularity measured by run-length in workloads is found to correlate with the preference of cache management policy: fine-grain interleaving workloads perform better with cache partitioning, and coarse-grain interleaving workloads perform better with LRU.

Most existing shared cache management techniques are based on working set locality analysis. This dissertation studies the shared cache performance by taking both locality and interleaving information into consideration. Oracle algorithm which provides theoretical best performance is investigated to provide insight into how to design a better practical policy. Profiling and analysis of Oracle algorithm lead to the proposal of probabilistic replacement (PR), a novel cache allocation policy. With aggressor threads information learned on-line, PR evicts the bad locality blocks of aggressor threads probabilistically while preserving good locality blocks of non-aggressor threads. PR is shown to be able to adapt to the different interleaving granularities in different sets over time. Its flexibility in tuning eviction probability also improves fairness among thread performance. Evaluation indicates that PR outperforms LRU, UCP, and ideal cache partitioning at moderate hardware cost.

For single program cache management, this dissertation also proposes a novel technique: reuse distance last touch predictor (RD-LTP). RD-LTP is able to capture reuse distance information, which represents the intrinsic memory reference pattern. Based on this improved LT predictor, an MRU LT eviction policy is developed to select the right victim at the presence of incorrect LT prediction. In addition to LT predictor, another predictor: reuse distance predictors (RDPs) is proposed, which is able to predict actual reuse distance values. Compared to various existing cache management techniques, these two novel predictors deliver higher cache performance with higher prediction coverage and accuracy at moderate hardware cost.

Using Locality and Interleaving Information to Improve Shared Cache Performance

by

Wanli Liu

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2009

Acknowledgments

I owe my gratitude to all the people who make this thesis possible. I couldn't imagine how I would go through my graduate years without you. First and foremost I'd like to thank my advisor, Professor Donald Yeung for giving me invaluable encouragement and strong support over the past years. He has always made himself available for help and inspiration. It has been a pleasure to work with and learn from such an extraordinary individual.

I would also like to thank Professor Bruce Jacob, Professor Rajeev Barua, Professor Manoj Franklin, Professor Steven Tretter, and Professor Alan Sussman for agreeing to serve on my thesis committee and for sparing their invaluable time reviewing the manuscript.

My colleagues at the System Computer Architecture laboratory have enriched my graduate life in many ways and deserve a special mention. Dongkeun, Seungryul, Gautham, Deepak, Sumit and Hameed helped me jump start my research in the group. Xuanhua, Priyanka, Meng-Ju, Inseok, and Xu gave me wonderful help and discussion. Collaboration and communication with Aamer, Sumesh, Brinda, Zahran, Kursad, David, Sada, Ohm, and other friends are highly appreciated.

My experiences at MIT CSAIL were unforgettable with Michael, David, Ian, Saman, and Anant. Stephen, Janice, Dong-In, Jinwoo, and Karan from ISI east also gave me exceptional help.

Finally, I would like to thank my family. My dear wife Zhongqin has been unconditionally supporting me through my graduate years. I owe my deep thanks to our parents, whose help is just indispensable. Eric makes my life complete and my study happy and meaningful.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

1.1.1 Chip Multi-Processor

While the industry continues to improve the performance of both processor and memory systems, the *processor-memory gap*, a disparity between processor and memory speed, keeps increasing, and is now one of the main obstacles to high performance. Hennessy and Patterson [1] predict the gap will grow wider in the future, which is also shown by the state-of-the-art industry products.

Recent commercial architectures rely on a memory hierarchy to alleviate the memory bottleneck. A modern memory system consists of register files, multiple levels of cache, main memory and disk. The first level *L1* cache is usually built on chip, in the closest location to the processor pipeline among all levels of cache. Since the access time for L1 cache is only a couple of CPU cycles, this latency can be effectively hidden through out-of-order execution and lock-up free caches. The second level *L2* cache has larger capacity but longer latency than L1 cache. More recent processors choose to build the L2 cache on chip to minimize access time, and have an off-chip *L3* cache to buffer data between the on-chip processor and main memory. Since the behavior of the memory hierarchy below L1 cache is less affected by pipeline design, the DRAM latency in the range of hundreds of cycles cannot be totally tolerated. Previous research shows that a significant amount of CPU stall time is spent on L2 cache misses [2].

On the other hand, Moore's law remains valid as industry projects to integrate 10Billion transistors by 2010[1]. Both industry (DEC Piranha, Sun Niagara, IBM Cell, Intel 80-core processor) and academia (MIT Raw, Univ. of Texas Trips, Univ. of Washington WaveScalar) are actively looking for solutions for translating higher integration density to higher microprocessor performance. In addition to traditional *superscalar* architecture, two alternatives have been proposed: *simultaneous multithreading* (SMT) [3] and *chip multiprocessor* (CMP) [4]. Single thread in superscalar architecture can only offer diminishing returns due to limited parallelism which can only be extracted with prohibitive complexity and power cost. Unlike the superscalar architecture which only exploits *instruction-level parallelism*, SMT and CMP can also take advantage of *thread-level parallelism* and *process-level parallelism*, which are exposed by compiler and OS scheduler.

1.1.2 CMP Resource Sharing

Multithreaded architectures, such SMT and CMP, permit sharing of hardware resources, including the memory hierarchy. In particular, how threads share the memory system affects the processor-memory gap. The SMT architecture permits the greatest degree of sharing by allowing all threads to have access to the processor pipeline, and memory hierarchy. This design usually results in more efficient utiliza-

[1]http://www.intel.com/technology /mooreslaw

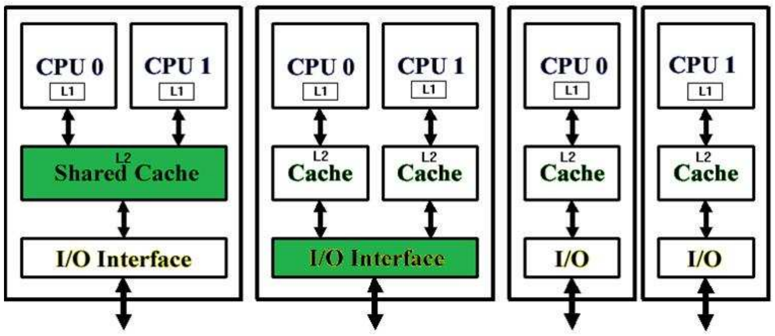


Figure 1.1: Resource sharing in multi-threaded architectures.

tion of processor resources when single thread can be allocated most of the shared resources while the remaining concurrent threads demand little shared resources [5]. However, due to layout and implementation complexity suffered by the wide-issue SMT processor, CMPs are expected to deliver higher throughput with better scalability [6]. Therefore, this dissertation will focus on CMP architectures although some of the techniques presented may also be applicable for SMT architectures.

In CMPs, individual cores usually share some portion of the memory system which are often lower levels of the memory hierarchy (there exists some exceptional design, such as the MIT Raw architecture [7] which provides register-level communication channels between cores). Figure 1.1 shows three sharing strategies for CMP architectures.

The first sharing architecture allows individual cores to have their own private L1 caches which are located close to the cores and provide fast access. These private L1 caches are connected to a globally shared L2 cache (or L3 cache which is the last level of on-chip cache) through a fast on-chip interconnect. The private L1 caches also share the underlying I/O interface. Many commercially available products, for example the Intel Duo Core, are built around such a design.

In the second sharing architecture, individual cores have both private L1 and private L2 caches. Such processors as the AMD Athlon Dual-Core takes advantage of fast on-chip inter-core communication and relies on shared high-speed I/O bandwidth to provide memory access to the threads. Compared to the previous shared cache architecture, the private L1 and L2 cache architecture provides less flexibility in sharing the important on-chip cache capacity among cores.

The third sharing architecture is adopted in traditional SMP's (symmetric multiprocessing) which rely on off-chip network and memory to connect individual single-cores with private L1 and L2 caches. The communication in such processors occurs completely off-chip and therefore suffers from high communication latency. This architecture also suffers from the zero-cache-sharing problem since the L1 and L2 caches are private.

Among the three types of sharing architectures, the first and second designs are more popular since they provide faster on-chip inter-core communication which is the key to better performance for multi-threaded applications. The shared L2 cache design is more appealing than the private L2 cache design due to more flexibility in sharing the highly critical L2 cache capacity. This dissertation will study the shared on-chip L2 cache for CMP architectures.

1.1.3 CMP Shared Resource Management

Although sharing cache capacity among cores enables higher utilization of resources, multiple threads sharing the memory system may sometimes have negative consequences for the following reasons. First, parallel architectures implementing high performance through multiple levels of parallelism also put more pressure on memory latency and capacity. The total working set of multiple threads scales up with the number of threads. This requires larger caches, wider buses and higher bandwidth of both on-chip and off-chip communication, and results in high miss rate, long average access time, high hardware complexity and power consumption. Second, stretched memory systems limit the ability of multithreaded architectures to maximize the total overall IPC because more cycles will be wasted waiting for cache misses. Third, multiple concurrent threads compete for the limited shared cache capacity, and when the total available resources cannot satisfy all the competing threads, there will be winners and losers, which suffer differently from unfair allocation. Therefore, in multithreaded execution, more parallelism is exposed but may not be exploited effectively due to the memory system, and sensible management schemes are necessary to improve both utilization and fairness of cache sharing.

Several policies to manage the shared cache among threads have been studied. The simplest policy is LRU sharing, allowing all threads to compete for shared cache freely. Many policies directly manage the number of cache blocks to be allocated to individual threads by partitioning the whole cache along different dimensions [8].

Most of these techniques try to allocate cache capacity to individual threads based on their working set sizes by assigning ways or sets to a particular thread. While these techniques have had success in managing shared cache, one common problem is that the allocation decision is mostly based on the measurement of individual thread's locality without taking into consideration the fine grain interaction among references from all threads. As shown by the analysis of interaction among threads, various interleaving granularities across sets often lead to non-uniform optimal capacity allocation in different sets. Therefore locality information of individual threads alone results in conservative cache management policies because the interleaving information is not considered in the cache management.

1.2 Improving Shared Cache Policy

1.2.1 Optimal Shared Cache Management

Researchers have been successful in analyzing reference trace to study cache management policies for one thread. A particularly insightful exercise is to study optimally managing cache resources with oracle information [9]. Other realistic online policies can also be evaluated using traces since the intra-thread reference sequence is maintained under various policies. However, it is more complicated to study cache shared by multiple threads than single thread cache, because the inter-thread order of references is subject to the cache management policy applied. Different policies have varied impacts on single thread performance, hence the relative progress variation of individual threads will change the hashing of references from those threads.

Nevertheless, a trace analysis with oracle information is still desired to provide insight into how a better online management policy should behave and how much improvement is possible to achieve over existing policies. To enable such analysis, an important assumption is proposed in this dissertation that the inter-thread reference order is maintained under any management policy in question. With this assumption, various shared cache management ideas can be tested and compared with one another based on common traces.

The first policy studied with this assumption is called Optimal Management Policy for Multi-threading (OPTm). Since the goal of this research is to propose a realistic policy, the OPTm algorithm tries to manage the inter-thread conflict with future information while maintaining realistic policy (*basic policy*) for managing intra-thread references. The basic policy is LRU in this research, since it is a popular policy selection in both academic and industry communities. OPTm policy looks at all the victim candidates of the lists of cache blocks from all the threads in the cache, and uses future information to select the victim candidate with the furthest reuse for replacement. The critical part of OPTm algorithm is to compare the future reuse distances of victim candidates from different threads. The future reuse distance of a reference in a multi-threaded trace is determined by two factors: its intra-thread reuse distance, and interleaved references from other threads. The evaluation of OPTm algorithm shows that there is still room for improvement, which provides motivation for this research.

The success of OPTm algorithm results from two major reasons. One apparent reason is the knowledge of future information. The second one is the consideration of both locality and interleaving. This reason is rather important because it sheds light on the key to further improvement. Existing cache allocation policies base their management decisions on locality information for individual threads. Interleaving information is ignored in such policies. Although some policies [10, 8] try to provide more flexibility into rigid cache capacity allocation, the full potential for improvement cannot be fulfilled without explicitly taking thread interleaving into account. In this dissertation, a new technique called Probability Replacement (PR) is proposed to take advantage of some key observations made in analyzing the behavior of OPTm algorithm,

1.2.2 Probability Replacement Policy

The analysis of OPTm algorithm reveals that it is possible to improve shared cache management by taking thread interleaving into consideration. To take advantage of the potential made possible by considering thread interleaving, some key observations from OPTm algorithm must be understood. The first observation is that in OPTm algorithm, for each set, the boundaries between threads are fluid. In other words, the allocation-based policies enforce unnecessary rigid boundary, which prevents threads from benefiting from transient locality variation. The second observation is that OPTm algorithm looks at victim candidates from all the sharing threads, and selects the one with furthest reuse. Profiling the behavior of the OPTm algorithm indicates that in many cases OPTm has strong bias in selecting optimal candidate based on thread ID of missing thread and the primary candidate (the block at global LRU position). The third observation is that OPTm algorithm often behaves like LRU when the interleaving is coarse-grained.

Based on these observations, a new shared cache management technique, *Probability Replacement*(PR), is proposed in this dissertation. PR is an LRU-style policy, with additional rate control knobs to manage the rate of replacements in shared cache. The rate control knob is a probability by which the policy determines which victim block to evict based on the ID's of missing thread and victim candidate blocks. The values of the knob probability are learned from sampling epochs and applied till different phase is detected. This online learning algorithm enables the hardware to adapt to the phase variation of multi-threaded workload. The evaluation of this technique shows that it is capable of managing shared cache more efficiently than previously proposed techniques.

1.3 Contributions

This dissertation makes the following contributions.

Studying Shared Cache Management Problem as a Locality-Interleaving Problem

The reference trace in multi-threaded workload is the aftermath of interleaved traces from individual threads. Due to the chaotic nature of program behavior, the interleaving is highly irregular and random, hence hard to study. Previous research try to eliminate the effect of interleaving by focusing on locality per-thread. However, this approach cannot achieve high utilization of shared resources. This dissertation suggests that the interleaving factor is equally important and provides opportunities for further improvement.

Development of Trace Analysis Tool to Study both Thread Locality and Thread Interleaving

The high irregularity and randomness of thread interleaving prevents researchers from thoroughly studying the multi-threaded workload. The conventional approach is to separate interleaving from locality information, and use locality per-thread only to manage shared resources. This dissertation proposes to analyze interleaved traces with the awareness that locality interacts with interleaving. Instead of isolating locality from interleaving, new analysis methodology treats interleaved trace in its entirety, which minimizes loss of valuable information as for how multiple threads interact with one another.

Qualitative and Quantitative Analysis of OPTm Algorithm Behavior to Explore Design and Performance Space for Multi-threaded Cache

The assumption that the interleaved trace remains the same under different management policies enables feasible analysis framework to evaluate different policies on the same trace. Although accuracy is sacrificed, the benefit of ease and key observations makes it a desirable approach to study multi-threaded workloads. Experiments based on this assumption shows its effectiveness and validity in revealing insights into how multiple threads interact with one another and directing the design of realistic online management techniques.

Design of Probabilistic Replacement Policy

Based on the knowledge acquired through multi-thread trace, a novel shared cache management policy is proposed. This policy is able to adapt to the dynamic interleaving of threads with different locality. An important portion of the PR policy is the epoch-based online learning of best control ratios, which tracks the phase variation of program behavior and feeds back to control logic. The implementation uses global control ratios to manage threads in all sets, minimizing hardware cost.

Evaluation of Probabilistic Replacement Policy and Comparison with Other Techniques

A faithful evaluation of the performance of probabilistic replacement policy is conducted across a large number of two-program workloads and a number of four-program workloads as well. A group of other allocation based techniques are also evaluated and compared against PR policy qualitatively and quantitatively. The experiment results show that PR policy outperforms LRU and UCP, a recent cache partitioning technique, by about 5% and 3%, respectively.

Design and Evaluation of Reuse Distance based Last Touch Predictor

In addition to the topic on the shared cache for multi-program workloads, the cache management for single-program is also studied. A reuse-distance based last touch predictor is proposed to improve cache performance for single programs. The reuse-distance information is employed to increase both prediction coverage and accuracy, resulting in lower miss rate at cheaper hardware cost, compared to other existing last touch predictors.

1.4 Road Map

The remainder of this proposal is organized as follows. Chapter 2 discusses related work, followed by background and methodology in Chapter 3. Chapter 4 profiles and analyzes the run-time behavior of multi-threaded cache with various management techniques, particularly the Oracle management algorithm. Based on this analysis, Chapter 5 proposes new cache management algorithm: victim eviction.

Probabilistic replacement, a practical online management technique is developed based on victim eviction in Chapter 6, and it is evaluated and compared to other existing techniques. Chapter 7 reports reuse distance information based last touch predictor for single program cache management. Finally, Chapter 8 concludes the dissertation and suggests future directions.

Chapter 2

Related Work

This chapter surveys background material and related work on memory systems. First, it introduces some historical work on the general locality and cache study. It then focuses on some research work on cache management policies and their application in multithreaded execution. Much attention is paid to recent developments in multithreaded cache, such as cache partitioning, locality optimization, capacity allocation, etc.

2.1 Memory Locality Characterization and Application

Characterizing Locality

As the speed gap between the processor and memory system widens, caches became an important component of computing system in mid 60's, 20th century [11]. The cache organization parameters: capacity, associativity and block size, and their impact on cache performance are studied by Smith [12]. Three cache miss categories are defined by Hill [13] as compulsory misses, capacity misses, and conflict misses. Puzak [14] conducted analysis on cache replacement policies to improve cache utilization with shadow tags.

Another aspect of memory study has been driven by workload characterization, such as control flow, instruction level parallelism, memory locality, and basic block behavior. Characterizing the intrinsic quality of application motivates new architectural features and provides insight for explaining performance measurement/simulations. As one important form of program characterization, locality analysis has been the focus of memory research. Early efforts sought to manage page storage and transfer to improve virtual memory system performance [15]. In [9], the theoretical optimum OPT algorithm was proposed to set up the upper bound limit for replacement policy, and concluded LRU (least recently used) replacement is practically a good choice. Denning [16] proposed working set model and defined working set as the set of pages used during a fixed-length sampling window in the past, and suggested working set be measured by monitoring page footprint. Later Denning [17] defined locality using virtual time distance, and suggested three meanings of distance: temporal, spatial and cost.

Some researchers [18] proposed representations of locality in terms of probability density, which can be used to predict and tune cache performance. Instead of using probabilistic model, other researchers [19, 20, 21] used analytical cache modeling to capture locality based on cache models specified with parameters.

Improve Memory Performance with Locality

The principle of locality has wide application in virtual memory, cache, I/O buffer, network interface, video processing, web caching, and search engine. The scope of this literature survey will be focused on cache management. To solve the widening processor-memory gap, many researchers work to improve cache performance through hardware, software, or hybrid solutions.

Hardware solutions have the advantage of exploiting runtime information to improve hit rate and save power consumption.

Column-associative caches [22] improve direct-mapped cache performance by placing conflicting cache blocks through rehashing. Hardware called *locality prediction table* is proposed in [23] to detect different types of locality. The victim cache [24] stores replaced data in a small fully-associative buffer to convert some references with reuse distances slightly larger than what the main cache can allow into additional cache hits. A small fully associative cache is proposed in [25] to bypass references with bad temporal locality and prevent them from polluting cache. Johnson, Merten, and Hwu [26] use Spatial Locality Detection Table to capture the spatial locality variation and dynamically adjust cache block sizes to reduce both capacity and conflict misses. This proposal is particularly indebted to the work of run-time cache bypassing [27] for motivation of using bypassing techniques to optimize locality quality. Line distillation [28] also detects low spatial locality of the less recent part of the LRU stack and retains the used words while evicting the unused words. Hardware history tables for last touch prediction are employed by Lai and Falsafi [29, 30] for self-invalidating dead block in advance for distributed shared memory coherence protocol, as well as for improving cache prefetching. Locality in the form of generational behavior can help to reduce cache leakage power by turning off dead cache lines [31].

Cache replacement policy is a heavily investigated topic. Some researchers try to bring access frequency into replacement policies [32, 33, 34]. Guo and Solihin [35] use an analytical model to explain the performance variation of different replacement policies. In [36], locality is approximated by the combination of access frequency and block age to choose victim to evict. Wong and Baer [37] propose both profile-based scheme and on-line locality table to detect temporal locality and direct replacement policy to retain those cache lines exhibiting strong temporal locality. The v-way cache [38] observes the non-uniform distribution of access frequencies across cache sets and vary associativity on a per-set basis to match the different locality levels in different sets. With replacement policy divided into victim selection and insertion policy, adaptive insertion policies [39] places the incoming line in the LRU position to avoid thrashing in cyclic reference pattern. Cache replacement policies based on last touch prediction [40, 41, 42] provide hardware tables to track reference behavior and predict last touches to enable early eviction replacement.

Since memory locality is an intrinsic property of programs, compiler and operating system can be particularly helpful in detecting locality at source code level and managing cache organization in more flexible way at small or little hardware overhead. Hallnor and Reinhardt [43] use software to manage a fully associative L2 cache, which is called indirect index cache (IIC) with generational replacement policy. This software replacement policy outperforms a set-associative cache using LRU policy by exploiting the ability of fully associative cache to take advantage of temporal locality. Compiler [44] can also play an important role in locality analysis at source code level providing victim selection hints to hardware replacement.

Pretching has been proven to be an effective technique to reduce cache misses. Special hardwares are proposed to facilitate cache prefetching [45, 46, 47, 48, 49]. Compiler can also analyze the program access pattern and predict the block addresses before their real references. As a result, prefetching instructions can be inserted into program by compiler [50, 51]. A similar technique taking advantage of *memory level parallelism* to improve cache efficience is runahead execution [52], which can be supported in both hardware [53] and compiler [54].

More detail about related work on locality optimization can be found in Section 7.2.

2.2 Shared Cache in Multithreaded Workloads

Multithreaded Workloads

When SMT and CMP became the focus of architectural research, people started to study memory behavior in parallel execution by characterizing access pattern [55, 56, 57] and developing analytic model [58]. The benchmarks under study include SPEC2000 suite [59], server workloads [60], bioinformatics benchmarks [61], SPEC OMP suite [62], and SPECjbb [10]. Single-threaded benchmarks are usually grouped together to compose multi-program workload, in which there is zero overlap among the working sets of individual programs. Parallel programs, on the other hand, represent more complicated situations because different thread working sets have both private portion and shared portion. This dissertation is focused on the multi-program workloads.

Evaluation Metric, Performance Target and Management Policy

In [60], shared cache management is described in some important aspects: evaluation metrics, performance targets and management policies. Evaluation matrix are some measurement methods to quantify the performance of multiple threads based on IPC or miss rate. Compared to measuring performance based on miss rate, IPC based evaluation is more frequently adopted in research because it is closely connected with execution time. Three IPC-based metrics are proposed: average IPC, average weighted IPC [63], and harmonic mean of weighted IPC [64]. IPC_i is the IPC of single thread when N threads are executed simultaneously, while $SingleIPC_i$ is the IPC of that thread when it is executed without sharing the cache capacity with other threads.

$$AverageIPC = \frac{\sum_1^N IPC_i}{N} \tag{2.1}$$

$$AverageWeightedIPC = \frac{\sum_1^N \frac{IPC_i}{SingleIPC_i}}{N} \tag{2.2}$$

$$HarmonicMeanWeightedIPC = \frac{N}{\sum_1^N \frac{SingleIPC_i}{IPC_i}} \tag{2.3}$$

Among the three metrics, average weighted IPC and harmonic mean of weighted IPC reveal more about system-level performance by considering relative performance variation due to sharing resources. This relative effect on individual thread performance is quantified as fairness [64, 65], and employed as an important target for optimization of overall throughput. With equal priority set for each individual thread, average weighted IPC has been a popular metric used by many researchers. In this dissertation, the optimization target is to maximize average weighted IPC for multi-program workloads.

The specific technique to reach the optimization target is called the management policy. A simple solution is to inherit cache policies from conventional single-threaded cache, such as LRU. This policy treats all references as if they are from the same program. Unfortunately, this simplest approach does not work well, and researchers found that some regulation is necessary when competition happens and results in unfair allocation among threads. To better manage the competition for shared resources, a number of management techniques are developed.

Static and Dynamic Cache Partitioning Policies

In an early study [66], an optimal static partitioning algorithm of shared memory for multiple programs with miss rate variation outperforms LRU policy. Profiling information determines the static splitting of L2 cache in [62], which outperforms both private L2 and address-interleaved shared L2 caches.

Since programs exhibit phase behavior during their execution, static partitioning cannot adjust to the variation of multiple competing programs. Suh et al. [59, 67] employ on-line miss rate characterization and dynamic variation of partition size to improve overall hit-rate of L2 cache. Chiou et al. [68] use software to dynamically partition cache with the awareness of constructive and destructive interference, which lead to better overall performance. Qureshi et al. [69] define the utility function of program and optimize global miss rate with online utility monitoring technique, and they use this UMON mechanism to implement utility-based cache partitioning (UCP). This technique is selected as a typical practical cache partitioning technique to compare in this dissertation. Improving overall performance alone does not completely solve the problem of fairness, which is addressed by [70] with the notion of fair cache sharing and partitioning. Jaleel et al. [71] change the insertion policy to adaptively control the portions of individual thread working sets to be resident in cache.

Conventional cache partitioning algorithms assign allocations in sets, while Srikantaiah et al. [8] propose partitioning among sets for parallel programs. For parallel programs with inter-thread data reuse, compiler [72] can reorganize the loop iterations of each processor to maximize inter-thread locality. Operating system can also manage shared cache through architectural features [10] to optimize fairness of miss-rate or performance.

Hybrid of Private and Shared Caches

Compared with private cache, shared cache has disadvantage of slow access time and small capacity due to higher complexity. Hybrid of private and shared architectures [73] can provide high global capacity and fast local access through non-uniform accesses and sharing private cache among processors to allow expanding private data. Victim replication [74] replicates local primary cache victims within local L2 cache to minimize local L2 cache hit latency and maximize utilization of effective shared capacity. Cooperative caching [75] uses private caches and aggregate cache to exploit local and global levels of locality. Hardware-managed coherent caches and software-managed streaming memory are compared in [76] to evaluate these two basic models for CMP memory systems. In [77], the separation of L1 and L2 caches are overcome by allowing direct access to remote L1 cache to provide immediate access to data once produced.

Service Quality Management

Shared cache also challenges the Quality of Service (QoS) in CMP systems due to less effective share of resources allocated to a particular application. CQoS [78] is a priority-based QoS scheme for shared cache, in which the priority is enforced by selective cache allocation, set partitioning, and heterogeneous cache regions. In Virtual Private Machines (VPM), Virtual Private Caches (VPCs) [79] provide minimum guarantees of bandwidth and capacity for cache QoS through arbiter and capacity manager.

Chapter 3

Background and Methodology

As discussed in Chapter 1, the memory behavior of shared cache in CMP is an outcome resulting from locality and interleaving. This chapter describes the definitions for locality and interleaving, and how they interact with each other. The experiment setup and methodology are also described in this chapter.

3.1 Locality

To use locality to characterize multithreaded working sets, some quantitative representation methods have been developed. One commonly-used definition of locality of a reference at time T is as follows [80]:

$$Locality = \frac{1}{(T_{next} - T)} \tag{3.1}$$

where T_{next} is the time of the next access to the same address.

Since we assume the data unit for communication in the memory system is a block, temporal locality is a special case of spatial locality. According to this definition, we can say a memory reference has high locality if the time of the next access to the same block is far in the future, and low locality if the next access time is near in the future.

In practice, T is expensive to measure and susceptible to program volatility.

Time Index	0	1	2	3	4	5	6	7
Block Address	X	Y	Z	Y	X	Z	Z	Y
Complete FRD	2	1	2	2	*	0	*	*
Complete FRL	3	1	2	3	*	0	*	*
Complete BRD	*	*	*	1	2	2	0	2
Residual FRD(X)	2	2	1	0	*	0	*	*
Residual FRD(Y)	0	1	0	3	1	1	0	*
Residual FRD(Z)	1	0	2	1	0	0	*	*

Table 3.1: Illustration of the reuse distance concept. (*: the values cannot be computed without the information beyond the trace shown in the table)

The concept of *stack distance* [81] and *reuse distance* [82] are introduced to describe locality using abstract time (the number of accesses) instead of real time. For the convenience of discussion, the most commonly used definition is adopted as follows:

- *reuse distance (stack distance)* $RD(A_m \prec A_n)$ *= the number of unique memory accesses between two consecutive accesses (*A_m *preceding* A_n*) to the same cache block* A.

Some researchers [83] chose to use *reuse distance* to count all intermediate memory accesses. Since the total number of all intermediate memory accesses is also a very important concept in the following discussion, *reuse length(stack length)* is proposed to define it.

Table 3.1 illustrates the concepts defined above, where processor accesses block address X at abstract time 0, then accesses two other cache blocks (Y and Z in the figure) before accessing X again at time 4. Thus, the reuse distance for the first reference to X at time 0 is 2, while its reuse length is 3 because Y is accessed twice between the references to X.

In the following discussion it is sometimes necessary to distinguish between the distances from a cache block reference (Y at time 3) to its immediately previous reference (Y at time 1) and to its immediately next reference (Y at time 7). $RD(Y_3 \prec Y_7)$ is called *forward* reuse distance of Y_3, and $RD(Y_1 \prec Y_3)$ is called *backward* reuse distance of Y_3.

Reuse distance can sometimes be a runtime measurement of how far in the future a block is reused at the reference time of another block. For this purpose, $RD(E_m \prec F_n)$ is called *residual* reuse distance of block F_n at reference time of block E_m, ie, the number of unique memory accesses between reference E_m and reference F_n (including other possible references to block address E between E_m and F_n). On the contrary, $RD(A_m \prec A_n)$ is called *complete* reuse distance of block A.

Before studying multithreaded locality, let us first look at singlethreaded program behavior since it provides a basis for discussing the multithreaded cases. A common way to profile singlethreaded program locality is to look at how miss rate varies over various cache configurations with different levels of capacity, associativity, and replacement policy. Instead of treating every cache miss equally in computing overall miss rate, some studies [84] also try to identify each cache miss as either a conflict miss or a capacity miss. Since LRU is a widely used cache replacement

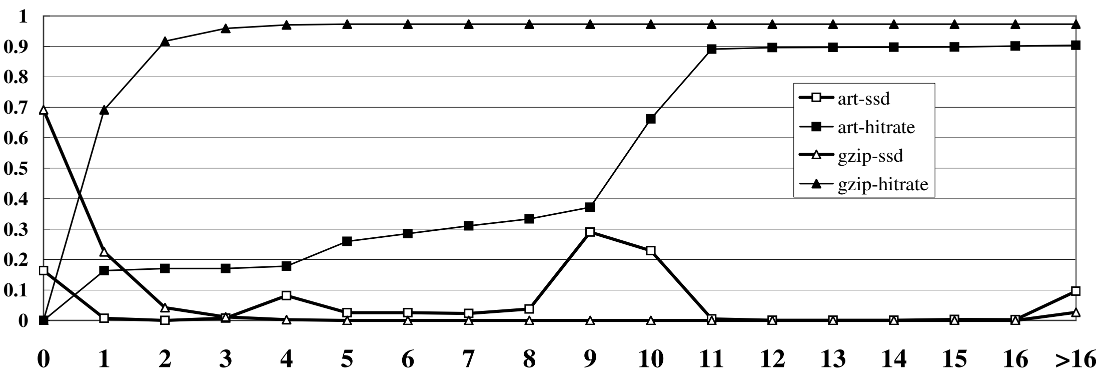


Figure 3.1: Singlethreaded stack distance distribution, and cache hit rate as a function of cache associativity

policy, stack distance (reuse distance) profile [85] has been proposed to reveal more detail in characterizing locality under LRU policy.

Figure 3.1 presents the cache profile of two SPEC2000 benchmarks: *art* and *gzip*, under a set-associative LRU L2 cache. Since the stack distance distribution varies with the number of sets in the cache, the number of sets is set up as 2048 in the experiment. The points labeled "art-ssd" stand for the percentage of references exhibiting a particular stack distance value (X-axis label) among all references in the art benchmark trace. Under LRU replacement policy, all references with future reuse distance smaller than the associativity will cause the next reference to the same block to hit in cache, otherwise a miss will occur. Thus the miss rate for a particular associativity AS is equal to the sum of the percentages of all references with stack distances smaller than AS, ie, Hitrate(AS) $= \sum_{d=0}^{AS-1} ssd(d)$. Assuming 64-byte cache block size, the cache capacity can be computed as $2048 * 64 * AS$ bytes.

From the stack distance distribution in Figure 3.1, it is evident that different benchmarks have various locality characteristics. Benchmark *gzip* has high locality since most of its stack distances are concentrated in the range from 0 to 2, thus high associativity (high capacity) has very limited benefit when associativity increases beyond 2. In contrast, benchmark *art* shows low locality, with its stack distances widely spread from 0 to 16. Hence it takes a cache larger than 1.4MB to reach 90% hit rate.

3.2 Interleaving

When multiple threads are sharing memory system, interference is inevitable due to interleaving. Concurrent threads compete for the limited shared resources. However, depending on the programs, different workloads react differently to interference. Therefore, it is important to study how their references are interleaved in the shared resource, in terms of how frequently each thread is accessing memory and how they are ordered in time.

3.2.1 Interleaving Granularity

Interleaving is interesting yet important to study because it is the cause of memory interference, and as shown in later chapters, it also provides some possibilities to improve memory system performance. One important question is how interleaving happens, and one answer could be *Interleaving Granularity.* Figure 3.2 shows two cases for different interleaving granularities. In fine-grain interleaving, references from different threads are shuffled with each at small distance, while coarse-grain interleaving tends to have references from the same thread cluster to-

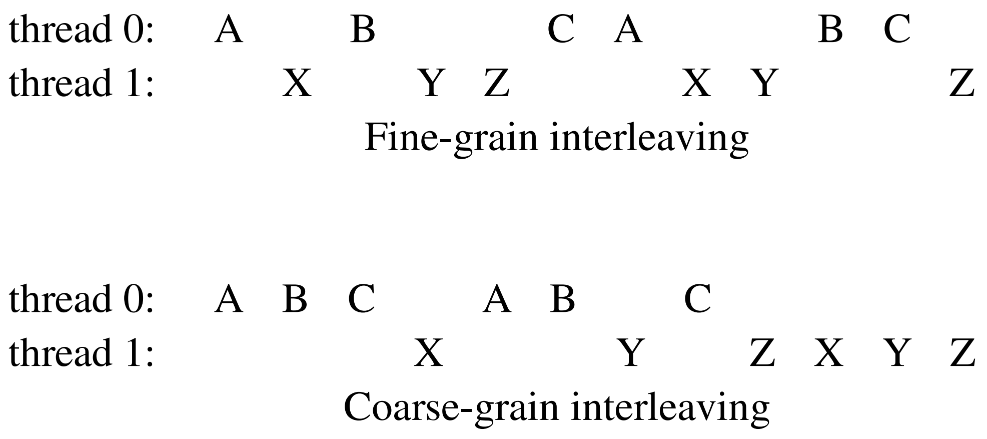


Figure 3.2: Fine-grain and Coarse-grain interleaving

gether. *Run Length* is defined as the number of the references from the same thread without any reference from other thread, and *average runlength* is the average of all the runlength from the same thread in the multi-threaded trace. For example, thread 0 of fine-grain interleaving in Figure 3.2 has different runlengthes, namely 1, 1, 2, and 2. The runlengthes for thread 0 in coarse-grain interleaving are 3, 2 and 1. Thus, thread 0 has different average runlengthes in these two cases, 1.5 for fine-grain interleaving and 2 for coarse-grain interleaving.

The interleaving granularity is workload dependent. Fine-grain interleaving happens to workloads whose concurrent programs access to the same set very close in time, as illustrated by the example workload *swim-twolf* in Figure 3.3. In this Figure, program swim (light dots) is sweeping through all the sets shown, while program twolf (heavy dots) is also accessing almost all the sets in a less regular

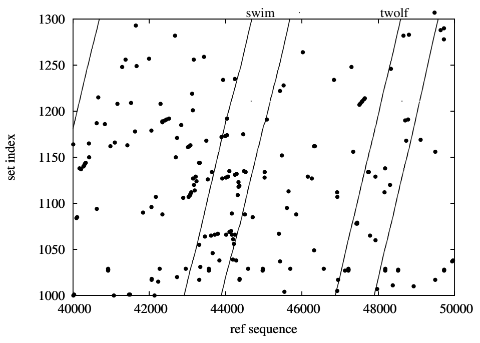


Figure 3.3: Fine-grain interleaving pattern for the swim-twolf workload

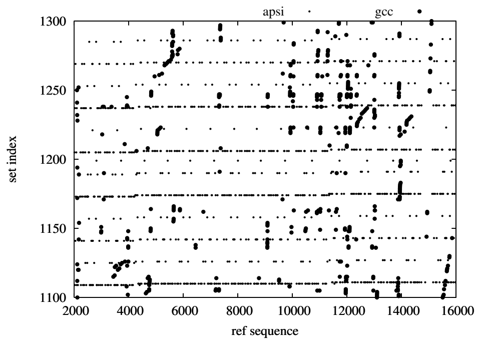


Figure 3.4: Coarse-grain interleaving pattern for the apsi-gcc workload

pattern. Nevertheless, in many sets, there are a few references from twolf between two consecutive references from swim. There is not much clustering occurring for either program. Hence this workload exhibits small runlengthes and small average runlength for both programs. For most workloads to have fine-grain interleaving, concurrent programs usually have a considerable working set spanning most sets and both access cache very frequently.

Figure 3.4 shows a coarse-grain interleaving workload: *apsi-gcc.* Program gcc (heavy dots) still spans most sets, showing the same access pattern of program twolf. Program apsi (light dots) shows a very different access pattern: its references hit a small number of sets with heavy accessing, while avoiding accessing many other sets. In the time window shown in this figure, the access patterns of apsi and gcc show *spatial isolation.* It is possible that in the later profiling windows program apsi will come back to access the sets it does not in the currently shown time window. When that happens, the later references from apsi are said to have *temporal isolation* with

App	applu	mgrid	apsi	sixtrack	bzip2	swim	art	wupwise	perlbmk	crafty	mesa
Skip	1.5B	3.4B	3.3B	8.5B	2.3B	5.7B	18.3B	3.4B	1.7B	5.2B	2.1B

Table 3.2: Different simulation start points for interleaving sensitivity study (B = Billion)

the references from gcc in the time window shown in the figure. It is worthwhile to note that normally in a particular workload, the interleaving granularity varies across sets over time, therefore there is a distribution of different runlengthes, and the shape of the distribution will have important impact on the shared cache performance.

3.2.2 Sensitivity of Interleaving Granularity

In individual programs there are multiple phases with different memory accessing frequencies, working sets, and set mappings. If a program is executed concurrently with different program phases of other program, the interleaving will change because of varied accessing frequency and set mapping. To study the sensitivity of interleaving granularity to program phases, some traces are created by running programs from different starting points. Then the average runlength is measured in these new traces, and compared with the original traces.

Start points for the original traces are shown in Table 3.5. Out of 26 SPEC CPU2000 benchmarks, 11 benchmarks are selected to start from different start points as shown in Table 3.2. 100 new traces are created by simulating workloads which are made of one program with original start point and another program with new start point. The average runlength of the new trace of a particular workload is compared with that of the original trace of the same workload with programs starting from the original start points. Then the percentage of average runlength variation is computed for each workload, and the average variation over the 100 new traces compared to the original traces is within 3%. Although some programs start from very different code regions due to different start points, interleaving granularity remains stable for most workloads. Different execution regions of the same program may have impact on memory access behavior, however, the variation is not significant enough to change the relative interleaving of the memory references from different threads.

3.2.3 Impact of Interleaving on Performance

In order to reveal the impact of interleaving on performance, a large number of two-program workloads (to be introduced in later chapters) are simulated to gain some statistical insights. Shared L2 cache traces are collected from these workloads' dynamic executions and are fed to a set-associative 1MB cache trace simulator with associativity of 8. This simulator is capable of enforcing a number of management policies, including oracle policy, LRU, and some other policies to be mentioned later in this dissertation. Once the simulation completes, the simulator reports miss rate, runlength, locality and other detailed information both per-set and per-cache.

The results show that different workloads react differently to various cache policies because of the interaction of program locality and interleaving. If the interleaving granularity is measured using *average runlength*, the trace miss rate simulations show that certain interleaving correlates with the r elative performance

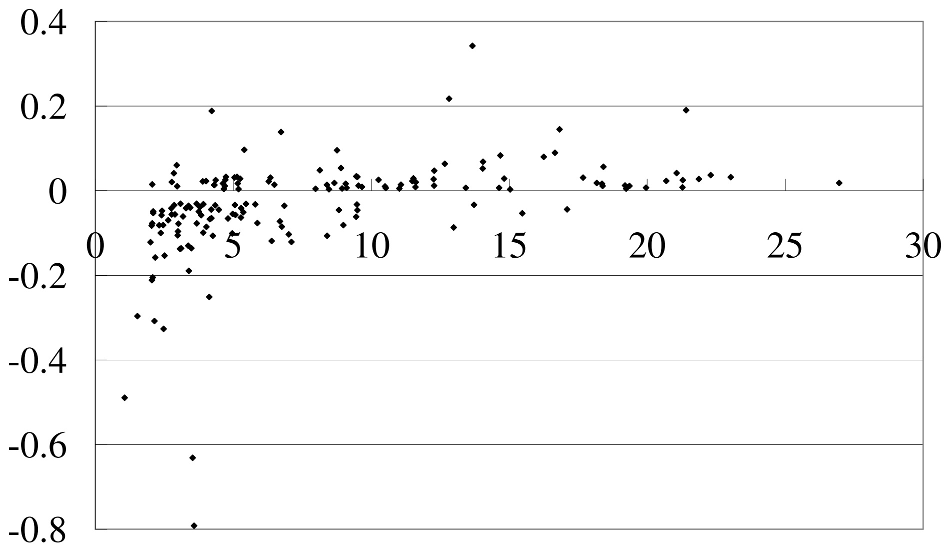


Figure 3.5: Cache performance comparison of LRU and iPART over different interleaving granularities (average runlength)

of cache policies. In Figure 3.5, the cache performance (measured in overall miss rate) is shown across all the 2-program workloads composed in the experiment. The points above 0 are workloads with better performance under LRU policy than under an ideal cache partitioning policy (to be introduced in later chapter), and the points below 0 are workloads with better performance under this ideal cache partitioning policy. This figure shows that cache partitioning usually works better in workloads with finer interleaving, and LRU works better in workloads with coarser interleaving , as well as some workloads with fine interleaving. Another important observation is that there are various levels of granularity across all the workloads, and more workloads with fine interleaving than coarse interleaving. Although cache partitioning seems to work better in the majority of the workloads (67.5%), there are still many workloads preferring the LRU policy.

3.3 Characterizing Multithreaded Workloads

After analyzing the single thread locality of programs and their interleaving, let us look at the interaction of multiple working sets. When multiple threads are running simultaneously, assuming there is no sharing of data among threads (which is true for multi-program workload), the resulting stack distances are incremented due to the interleaving references from other threads. Several models are proposed in [85] to predict inter-thread cache contention, with the conclusion that it is necessary to consider three factors: hit(reuse) frequency, miss frequency, and stack distance distribution.

Based on the definition of stack distance (reuse distance) under singlethreaded working set, multithreaded stack distance (reuse distance) with interleaving can be defined as $SD_m(s) = s + incr(s)$, where $incr(s)$ is the incremental part of $SD_m(s)$ caused by interleaving from other threads. Note that s is the same as stack distance in singlethreaded working set, since interleaving threads has no effect on the internal order of references within a single thread.

Further analysis of $incr(s)$ helps understand what factors contribute to the incremental part of multithreaded stack distance. By definition, $incr(s)$ is the number of unique memory accesses from other threads during the reuse stack of distance s. Thus $incr(s)$ can be represented as $distance(intv(s))$, where $intv(s)$ is the total number of memory accesses during the reuse stack of distance s, and function $distance()$ counts the number of unique memory accesses. There are 2 factors which determine $intv(s)$: the stack length $sl(s)$ corresponding to stack distance s, and the ratio $r(s)$ of the frequencies of all the other threads over local thread. In summary, $incr(s) = distance(sl(s) * r(s))$, and $r(s)$ is a key measurement of the degree of intervention caused by interleaving.

3.3.1 Case Study

Now let us look at the same workload composed of two SPEC2000 benchmarks *art* and *gzip*. Figure 3.6 shows that cache hit rate variation when *art* and *gzip* are simultaneously running with a shared 1MB L2 cache. Compared with Figure 3.1, it is evident that the distributions of stack distances of different benchmarks are expanded towards larger values at different rate. Compared to the singlethreaded case, the stack distance distribution of *art* experiences small variation, with stable high concentrations at stack distance values of 0, 4, 9 and 10 in both figures. At the same time, *gzip* expands its stack distance concentration from the range of $0 \sim 2$ in singlethreaded case to $0 \sim 12$ in multithreaded case, and shows poorer locality than in singlethreaded case (*gzip* hit rate climbs above 90% only at stack distance beyond 14). This observation indicates that *art* loses little locality while *gzip* loses much of its locality due to the competition for limited cache capacity.

Table 3.3 compares the miss rates of *art* and *gzip* under singlethreaded and multithreaded executions. The comparison between singlethreaded 0.5MB and doublethreaded 1MB shows how much benefit a benchmark receives by sharing doubled-capacity cache with another benchmark, while the comparison between singlethreaded 1MB and doublethreaded 1MB shows how much penalty a benchmark suffers by

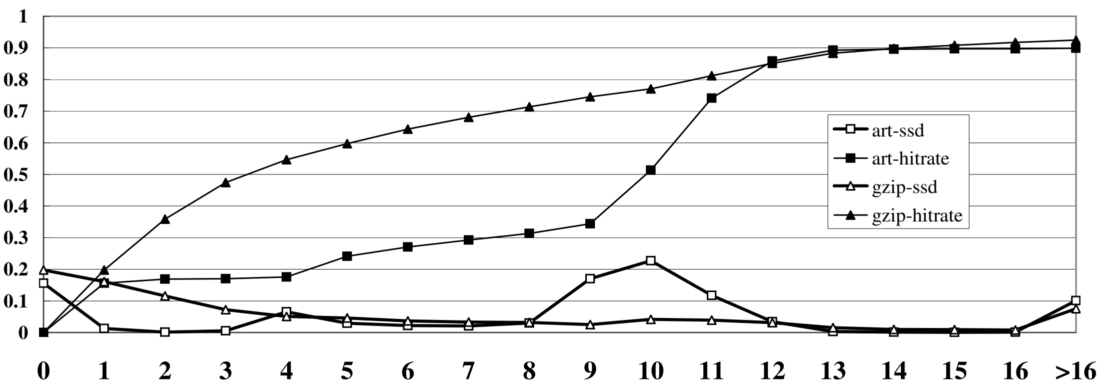


Figure 3.6: Multithreaded stack distance distribution, and cache hit rate as a function of cache associativity

Threads	Single		Double
Cache Size	0.5MB	1MB	1MB
art missrate	0.822	0.666	0.686
gzip missrate	0.029	0.027	0.286

Table 3.3: Comparison of miss rates of singlethreaded and multithreaded cache miss rates of art and gzip, with frequency ratio art:gzip = 0.88:0.12

sharing the same cache with another benchmark. Benchmark *gzip* suffers almost 10X miss rate increase due to contention from *art*. On the other hand, *art* benefits (4.6% miss rate reduction) from sharing 2X capacity with *gzip* while suffering only 18% extra miss rate by sharing 1MB cache. The disparity of miss rate variation of *art* and *gzip* can be explained by their ratio of frequency ($art : gzip = 0.88 : 12$). Such high $r(s)$ $(art_{freq}/gzip_{freq})$ for *gzip* produces high $incr(s)$ and moves the whole stack distribution towards higher locations. For *art*, very small $r(s)$ $(gzip_{freq}/art_{freq})$ makes little difference to its stack distance distribution, hence the locality suffers slightly.

This analysis of multithreaded cache performance indicates some key factors in improving performance, which will be discussed in Chapter 4.

3.4 Methodology

M5 [86], a cycle-accurate event-driven simulator, is employed to quantify the performance of different cache allocation policies. M5 is configured to model both a dual-core and quad-core system. The cores are single-threaded 4-way out-of-order issue processors. Each core also employs its own hybrid gshare/bimodal branch predictor. The on-chip memory hierarchy consists of private L1 caches split between instructions and data, each 32-Kbyte in size and 2-way set associative; the L1 caches are connected to a shared L2 cache that is 1-Mbyte (2-Mbytes) in size and 8-way (16-way) set associative for the dual-core (quad-core) system. The latency to the L1s, L2, and main memory is 2, 10, and 200 cycles, respectively. Table 3.4 lists the detailed simulator parameters.

Cache partitioning techniques under study are applied to the shared L2 cache architectures. To model UCP, M5 is modified to implement the UMON-global profiler for acquiring SDPs, as well as the analysis to determine the partitioning from the SDPs. In each epoch, all possible allocations of the 8 ways in shared L2 cache to different threads are simulated. While 2-program workloads only require trying 7 allocations (each program is allocated at least 1 cache way), simulations become prohibitively time-consuming for larger workloads (there are 455 different allocations per epoch for 4 threads). Due to the combinatorial number of allocations,

Processor Parameters	
Bandwidth	4-Fetch, 4-Issue, 4-Commit
Queue size	32-IFQ, 80-Int IQ, 80-FP IQ, 256-LSQ
Rename reg / ROB	256-Int, 256-FP / 128 entry
Functional unit	6-Int Add, 3-Int Mul/Div, 4-Mem Port, 3-FP Add, 3-FP Mul/Div
Memory Parameters	
IL1	32KB, 64B block, 2 way, 2 cycles
UL2-2core	1MB, 64B block, 8 way, 10 cycles
UL2-4core	2MB, 64B block, 16 way, 10 cycles
Memory	200 cycles (6 cycle bw)
Branch Predictor	
Branch predictor	Hybrid 8192-entry gshare/2048-entry Bimod
Meta table	8192
BTB/RAS	2048 4-way / 64

Table 3.4: Simulator parameters

App	Type	Skip	App	Type	Skip	App	Type	Skip
applu	FP	187.3B	mgrid	FP	135.2B	apsi	FP	279.2B
sixtrack	FP	299.1B	bzip2	Int	67.9B	swim	FP	20.2B
art	FP	14B	twolf	Int	30.8B	equake	FP	26.3B
wupwise	FP	272.1B	facerec	FP	111.8B	vpr	Int	60.0B
fma3d	FP	40.0B	eon	Int	7.8B	galgel	FP	14B
gzip	Int	4.2B	gap	Int	8.3B	perlbmk	Int	35.2B
gcc	Int	2.1B	vortex	Int	2.5B	lucas	FP	2.5B
crafty	Int	177.3B	mcf	Int	14.8B	mesa	FP	49.1B
ammp	FP	4.8B	parser	Int	66.3B			

Table 3.5: SPEC CPU2000 benchmarks used to drive our study (B = Billion)

the ideal cache partitioning is simulated only for 2-program workloads. The performance objective is average weighted IPC (WIPC) [63]. This is the optimization metric when searching for the best partitioning, as well as the metric in other IPC-based performance results. For both UCP and ideal cache partitioning, the epoch size is 1 million cycles, which is comparable to what's used in other dynamic cache partitioning techniques.

To drive simulations, multiprogrammed workloads are created from the complete set of 26 SPEC CPU2000 benchmarks shown in Table 3.5. Many of the experimental results are demonstrated on 2-program workloads: all possible pairs of SPEC benchmarks–in total, 325 workloads, are formed in the experiment. To verify insights on larger systems, 13 4-program workloads are also composed, which are

ammp-applu-gcc-wupwise	equake-galgel-mcf-sixtrack	apsi-gcc-ammp-swim
bzip2-art-lucas-crafty	apsi-bzip2-swim-vpr	perlbmk-twolf-vortex-wupwise
eon-sixtrack-facerec-mgrid	gap-mesa-gzip-lucas	art-vortex-facerec-fma3d
eon-mcf-perlbmk-vpr	applu-fma3d-galgel-equake	gzip-mesa-parser-gap
crafty-parser-mgrid-twolf		

Table 3.6: 4-program workloads used in the evaluation

listed in Table 3.6.[1] The benchmarks were provided as the pre-compiled Alpha binaries with the SimpleScalar tools [87] which have been built with the highest level of compiler optimization.[2] All the benchmarks use the reference input set. Before conducting detailed simulation of workloads, each benchmark is fastforwarded to its representative simulation region; the amount of fast forwarding is reported in the columns labeled "Skip" of Table 3.5. These were determined by SimPoint [88], and are posted on the SimPoint website.[3] After fast forwarding, detailed simulation starts, and the workload is simulated for 500 million cycles (for ideal cache partitioning, this does not include the exhaustive partitioning trials). On average, the 2- and 4-program workloads are simulated for 1 and 2 billion instructions, respectively.

When detailed simulations are executed to measure performance, the cache reference profiler embedded in the M5 simulator also records the memory reference traces at shared L2 level. For some experiments, trace-driven simulation is con-

[1]In the 4-thread workloads, each benchmark appears in 2 workloads.

[2]The binaries are available at http://www.simplescalar.com/benchmarks.html.

[3]Simulation regions we use are published at http://www-cse.ucsd.edu/ calder/simpoint/multiple-standard-simpoints.htm.

ducted to analyze memory reference interleaving and cache performance in terms of miss/hit rate. Because the traces collectively consume significant disk storage, traces are acquired over a smaller simulation window of 100M cycles rather than the 500M cycles used for performance simulations. After acquiring the traces, we replay them on a cache simulator that models Oracle, LRU, and other ideal cache management policies. For ideal cache management policies, the exhaustive search technique looks for the partitioning that minimizes cache misses rather than WIPC.

Chapter 4

Shared Cache Performance Analysis

In this chapter, some typical multithreaded workloads for shared memory in CMP are described in terms of locality and interleaving. An oracle algorithm is proposed to analyze the optimal management policy with future information. The analysis also indicates that better solutions are possible by taking advantage of locality and interleaving characteristics to maximize the utilization of memory system and improve overall performance.

4.1 Oracle Management Policy

As introduced in Chapter 1, the Oracle management policy is based on the conventional policy, LRU, for intra-thread management. The assumption which makes the Oracle algorithm possible is that the interleaved trace remains the same under any applied policy. It is apparent the interleaving will change with the cache management policy. However, for the set-associative cache under study in this dissertation, the variation is small because only the ordering of references in the per-set trace matters and small variation of ordering in per-cache trace will be filtered out.

Considering this inaccuracy, the invariant assumption still enables observing the details about how an oracle algorithm will work. Let us suppose there are N

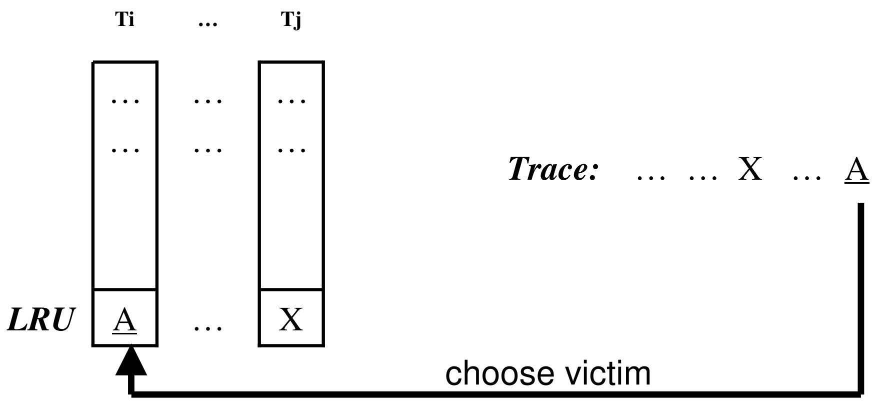


Figure 4.1: Oracle management policy for multiple threads

threads sharing a set (there is only one single set in fully-associative cache) with associativity of A. Each thread has a list of cache blocks in LRU order (assuming LRU policy for intra-thread management). On a cache miss, the cache management hardware will look at N (or fewer if some thread has no block in cache) intra-thread LRU blocks, and pick the one with the furthest reuse (or no reuse in the future). This decision making process is illustrated in Figure 4.1. Each thread has a list of cached blocks, and each list has a block on its local LRU position. Suppose there are two threads T_i and T_j ($0 \leq$ i, j $<$ N) have LRU candidates A and X for victim selection, the Oracle algorithm will compare these two blocks' future reuse in the trace, and choose the one with further reuse (block A in the figure) as the victim for eviction. For more than 2 candidates, the one with the furthest will be chosen. The global LRU block (from thread i) is chosen as victim (thread i) is called *aggressor block*, and usually an aggressor block belongs to a thread that cannot efficiently utilize its cache allocation because of bad locality (furthest reuse). As shown later, in one workload, usually blocks from one or more threads are selected as aggressor

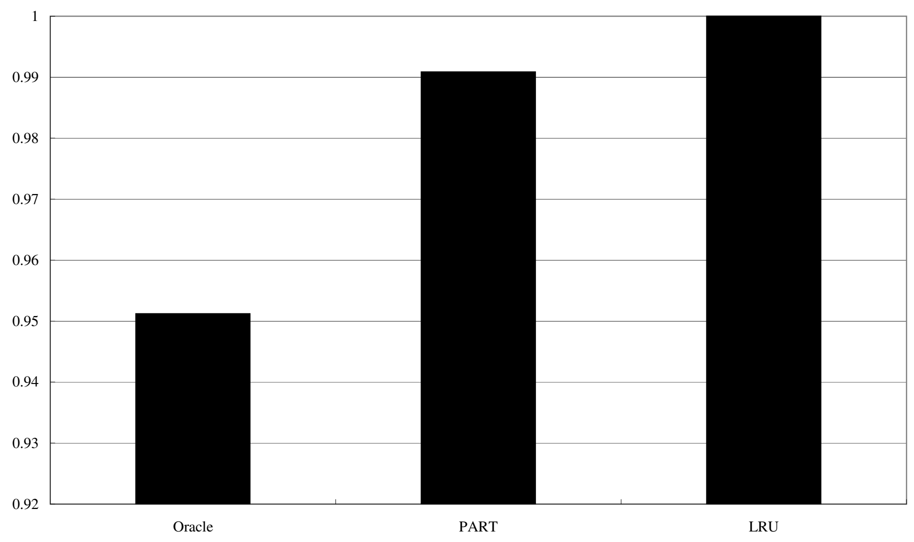


Figure 4.2: Average miss rates achieved by the Oracle management policy, PART, and LRU

block most of time, and such threads are called *aggressor thread*. Next, let us look at the performance and the behavior of the Oracle algorithm.

4.1.1 Oracle Management Performance

Now let us look at the difference of cache performance achieved by some policies in Figure 4.2. This figure uses LRU miss rate averaged across the workload suite as baseline (100% line), and the rest are normalized to it. Another management policy shown in this figure is PART algorithm, which is shown in Figure 4.3. PART is an ideal off-line searching based optimization technique that tries all the possible cache partitionings and omnisciently picks the best partitioning that reaches the best cache performance for each epoch. This algorithm has two versions, and the version shown in Figure 4.2 minimizes the whole cache miss rate as defined as *minMiss* in [69], which is also the optimization target for all the algorithms in trace

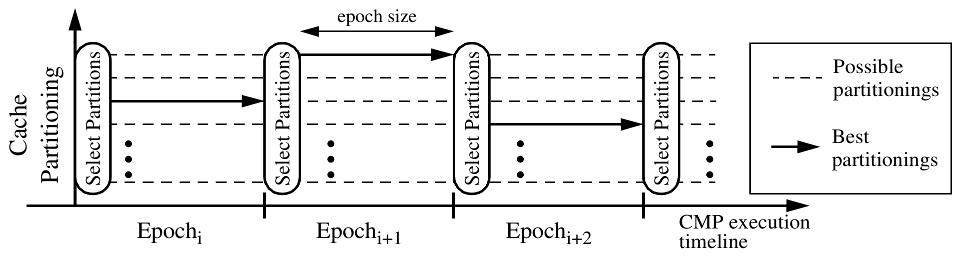


Figure 4.3: Cache partitioning with exhaustive searching

study. Another version of PART algorithm is built to optimize average weighted IPC, which will be presented in later chapter. Compared to LRU policy, the Oracle algorithm on average reaches about 5% reduction in misses, while PART algorithm can only reduce about 1%.

4.1.2 Comparing LRU with the Oracle Algorithm

LRU allows all sharing threads to compete for shared cache freely. Upon a cache miss from a thread, if the block on the global LRU position happens to belong to that thread too, this thread will evict that block, with every thread's allocation remaining unchanged. Otherwise, the thread owner of the global LRU block loses one block, and the cache missing thread gains one more block allocation. The thread with high access frequency or high miss rate (bad locality) end up occupying more cache capacity than it can effectively use, because at any moment, the capacity owned by a thread results from past access frequency and locality, while whether it can effectively utilize the current capacity is dependent on the future reuse of all the cache blocks in cache (including other blocks). Therefore the performance problem with LRU policy is that the cache capacity one thread gains in the contention usually cannot be justified because this contention rule favors high access frequency and bad locality.

In contrast, the Oracle algorithm's eviction decision is totally dependent on future inter-thread reuse. It preserves capacity of a thread with good locality or less intervention from other threads and reduces allocation of a thread with bad locality or more intervention. The advantage of the Oracle algorithm over LRU policy is illustrated in Figure 4.4 and Figure 4.5, in which a typical set behavior is profiled. This typical set with similar per-thread miss rates and overall miss rate as the whole cache performance is chosen to represent the whole cache. Figure 4.4 shows how the whole set (associativity of 8) is allocated to two programs, with the capacity below the division line allocated to ammp and the capacity above the line allocated to swim. Because swim has much higher access frequency than ammp (more than 5X), swim gains more capacity than it can effectively utilize. This can be shown by comparing the number of hits of swim (triangles) in both figures, and it is evident although swim uses more capacity under LRU than under the Oracle algorithm, it does not enjoy many more hits.

Oracle algorithm addresses this problem by evicting more blocks belonging to swim because of its bad locality. As a result, ammp gains more capacity (the area under division line) than under LRU policy and generates more hits than ammp could given the same capacity (more crosses in Figure 4.5 than in Figure 4.4).

In this workload, ammp is more often chosen by the Oracle algorithm as victim, therefore the aggressor in this workload is ammp, and swim is the non-aggressor. Comparing the cache division lines in both figures, the Oracle algorithm prevents ammp from using more capacity than it can utilize effectively. Besides the Oracle

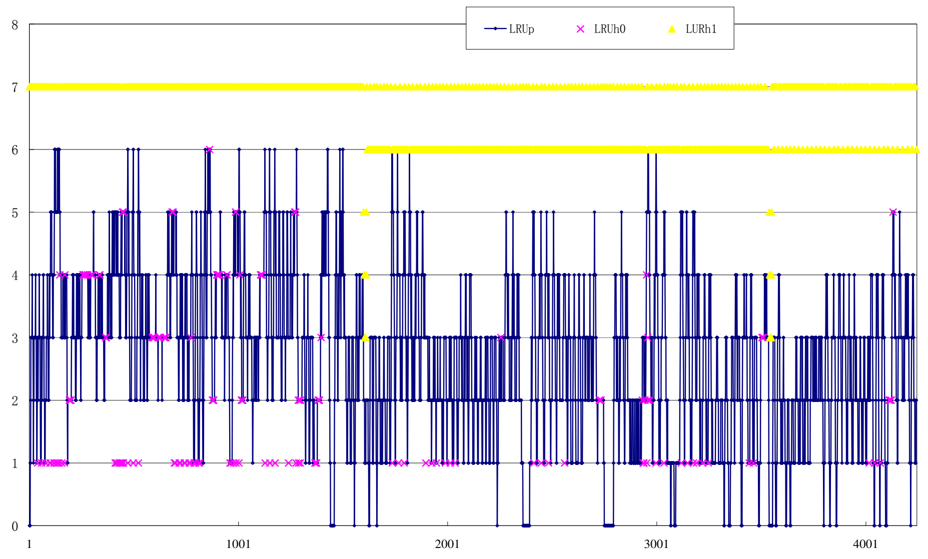


Figure 4.4: LRU policy behavior in ammp-swim workload: line (cache capacity division), crosses (hits of ammp), triangles (hits of swim)

algorithm, the cache partitioning techniques can also reach the same effect by setting up an isolation boundary between two programs to protect each working sets. The simulation results show that a proper cache partitioning can also improve the overall cache performance.

4.1.3 Comparing PART with Oracle Algorithm

It has been shown that both the Oracle algorithm and cache partitioning can help to address the imbalanced allocation problem in LRU policy. Cache partitioning algorithms consider the intra-thread locality information of all the sharing threads, which is not considered in the management of LRU policy. The benefit of knowing the locality of all sharing threads is that the management policy can favor the allocation to a thread which can utilize the given capacity most effectively (the

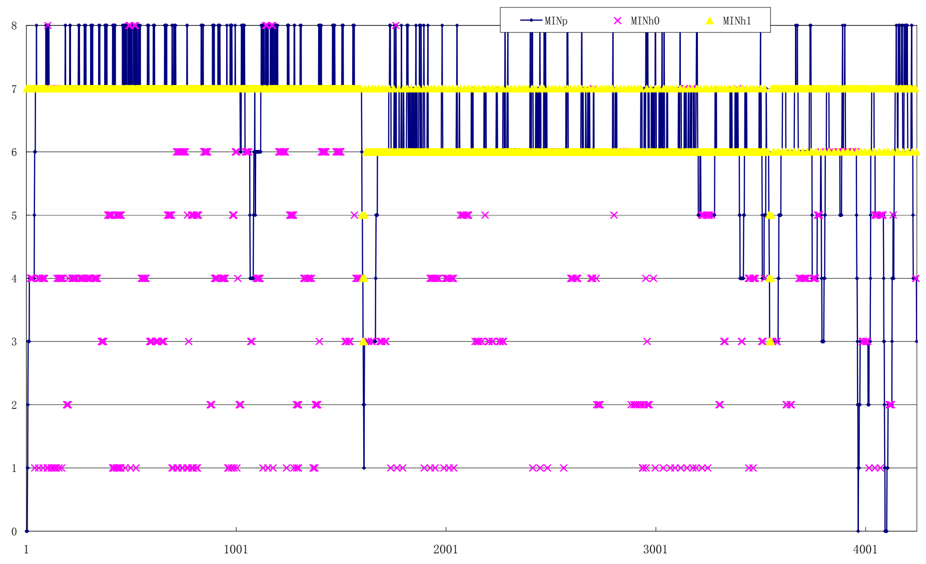


Figure 4.5: The Oracle algorithm behavior in ammp-swim workload: line (cache capacity division), crosses (hits of ammp), triangles (hits of swim)

marginal cache hit gain is defined as *utility* in [69]). As a result, cache partitioning is successful in managing shared cache capacity using intra-thread locality information to maximize global hit rate.

The PART algorithm introduced in 4.1.1 is an ideal off-line algorithm which provides the theoretical upper limit of cache partitioning. This algorithm has two parameters, one is the number of sets that share partitioning control, another one is the epoch (or time interval) granularity. The first one is called *spatial control granularity* and the second is called temporal control granularity. The spatial control granularity determines how independently each set makes their own partitioning decisions, and the temporal control granularity controls how fast the algorithm can react to transient variation in program localities. When these two granularity controls are both extremely fine, the PART algorithm will approach the performance of

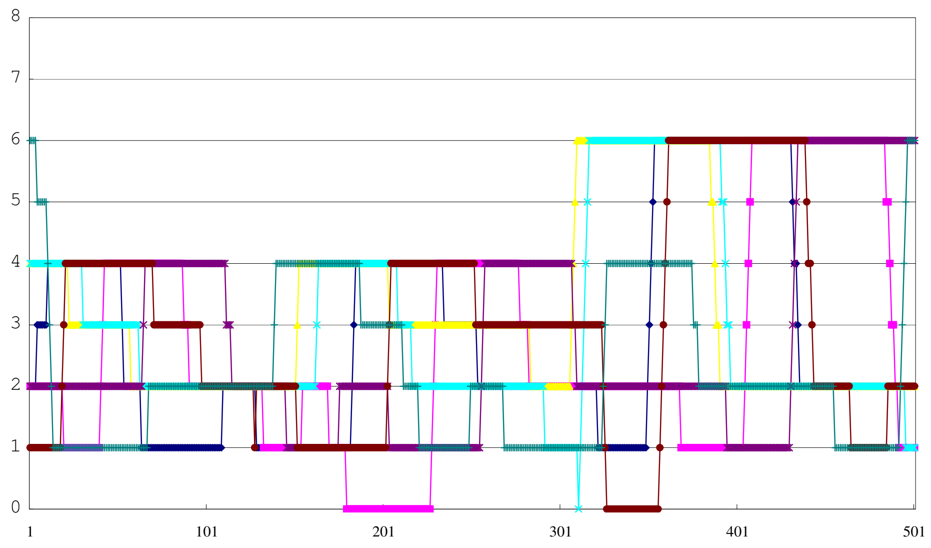


Figure 4.6: Cache division lines for 7 different sets in apsi-gcc workload

the Oracle algorithm because it essentially tests every eviction option and makes the same decision as the Oracle algorithm, and this decision is completely independent of other sets.

However, there are some limitations to the granularity control in realistic online cache partitioning techniques. Because the hardware cost grows linearly with the number of sets to monitor each and every set independently, most proposed cache partitioning techniques either use global counters [70] or dynamic set sampling [69] to reduce hardware overhead. Although Qureshi et. al propose UMON-local [69] to perform partitioning per-set, its expensive overhead makes it unattractive. Thus a trade-off of implementation cost and performance has to be considered in choosing uniform or per-set cache partitioning.

To illustrate this limitation, the cache division per-set is profiled for the PART algorithm introduced in 4.1.1 configured with partitioning per-set. Figure 4.6 shows cache division lines for 7 different cache sets profiled for apsi-gcc workload out of

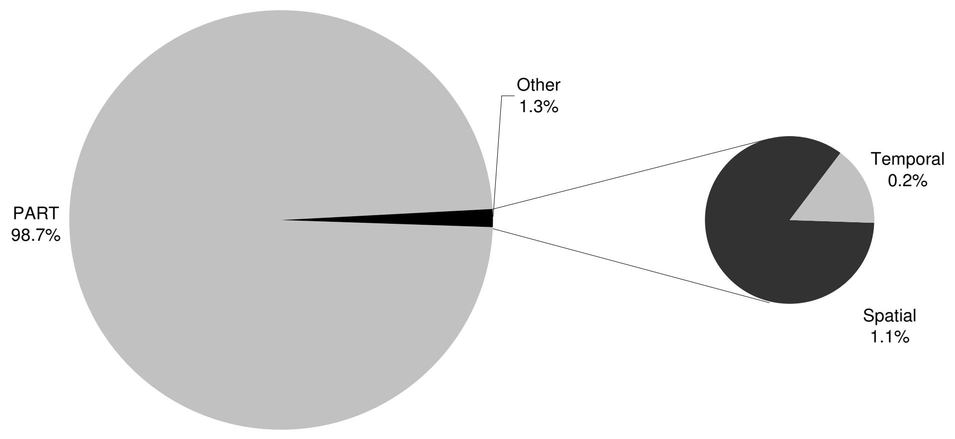


Figure 4.7: Cache performance breakdown between PART and Oracle algorithms

a total of 2048 cache sets. In this figure, different sets make different partitioning decisions independently. Performance simulation results also shows hit rate of partition per-set is 10.4% higher than uniform partition, demonstrating that optimal partition for each set is different and performance will suffer from enforcing a single partitioning cache wide.

To further analyze the impact of spatial granularity and temporal granularity on cache performance, PART algorithm is configured with different spatial granularity levels. One is uniform partitioning, maintaining a per-cache partitioning during the epoch, and another is per-set partitioning, allowing different sets to make individual partitioning decision. This breakdown is shown in Figure 4.7, in which the PART with per-cache uniform partitioning and 5 million cycles epoch size achieves 98.7% of the hits of the Oracle algorithm. Spatial granularity and temporal granularity totally contributes 1.3% degradation in cache performance. PART with per-set partitioning increases another 1.1%, while epoch granularity accounts for the rest 0.2% gap. This shows that on average spatial granularity has more impact on cache performance than temporal granularity.

PART is a highly unrealistic algorithm because it tries all the possible partitionings off line. In reality, since any on-line algorithm does not have the luxury to try every possible partitioning, it is usually *simultaneously executing and profiling.* In any epoch, cache partitioning control applies a specific partitioning based on profiling obtained in the previous epoch, and at the same time, the locality monitoring component is profiling the locality information, which drives the partition decision made at the end of the current epoch, and to be applied in the following epoch. This process requires that the epoch size be chosen appropriately to provide some level of predictability of locality information. If the epoch size is smaller than appropriate length, the profiling is subject to noise such as transient variation of locality, and also loses accuracy because it takes time for the monitoring counters to give meaningful reading. When the epoch size is overly large, the locality profiling tends to average over different phases in the same epoch and also loses accuracy. Researchers often choose the best performing epoch size based on their simulation results, for example 5 million cycles in [69], and 10 thousand L2 cache accesses in [70].

To illustrate the effect of epoch size on a realistic online management, a cache partitioning technique is simulated with UMON-global management algorithm [69], which partitions cache uniformly by ways for all the sets, with epoch of 5 million cycles based on experiments on this workload. Figure 4.8 and Figure 4.9 report some behavior detail of UMON partitioning and the Oracle algorithm in a typical set chosen the same way as in 4.1.2.

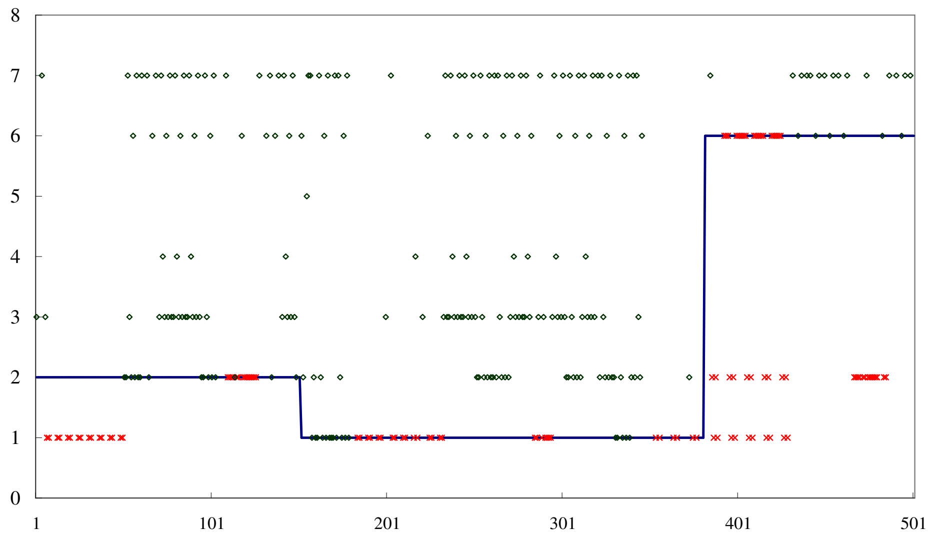


Figure 4.8: UMON partitioning behavior in apsi-gcc workload: line (cache capacity division), crosses (hits of apsi), diamonds (hits of gcc)

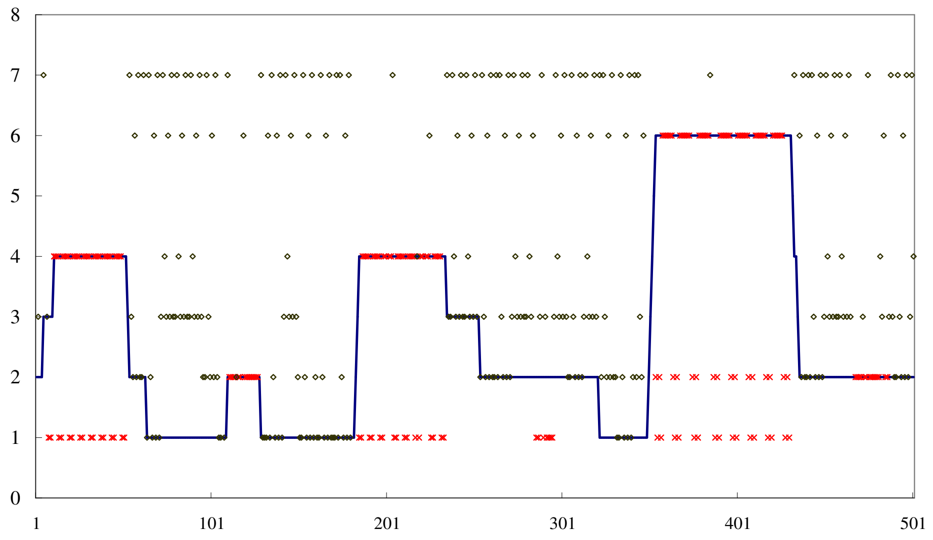


Figure 4.9: The Oracle algorithm behavior in apsi-gcc workload: line (cache capacity division), crosses (hits of apsi), diamonds (hits of gcc)

There are some key observations that can be made from comparing these two figures. First, the allocation division in the Oracle algorithm is varying faster than UMON policy due to a prefixed epoch size. As explained above, although it is possible to capture such high-frequency phase variation with smaller epochs, more noise will also be introduced, and as a matter of fact, smaller epoch size does not improve performance in this workload. Second, the transient phases shown in the Oracle algorithm behavior exhibit various granularity sizes. There is no one single prefixed partitioning epoch size that can fit all transient phases. This is why designers need to test many sizes to find the best performing one, and often this best epoch size is workload dependent. The disadvantages shown here contribute to about 14% more misses in the UMON policy than the Oracle algorithm.

4.2 Characterizing Oracle Algorithm Behavior

4.2.1 Advantages of Oracle Algorithm Behavior

After discussing the behavior of LRU and cache partitioning policies in different types of workloads, let us look at why the Oracle algorithm can perform better. In order to develop a better realistic shared cache management policy, one possible approach is to study the behavior of the Oracle algorithm and improve existing policies by approximating the Oracle behavior.

In short, the Oracle algorithm uses future information to direct current replacement decisions. This future information incorporates both locality and interleaving information, which is the key to the success of the Oracle algorithm.

If we look at the Oracle algorithm from the point of view of cache allocation, it is equivalent to enforcing cache partitioning at both the finest spatial granularity and the finest temporal granularity because it completely decouples all cache sets and makes replacement choices solely based on the effect of that eviction decision. The larger spatial and temporal granularities by realistic cache partitioning policies for implementation reasons therefore contribute to the performance degradation compared to the Oracle algorithm.

These two limitations, however, do not exist in the LRU policy. The LRU policy allocates capacity totally within individual sets, and lets individual references decide the allocation without relating to other previous references. However, the LRU policy is dependent on global LRU stack to give hint on victim selection. The position of an appropriate victim in LRU stack is subject not only to locality, but also to interleaving. And these two factors are interacting in an unmanaged way.

In summary, the secret to the effectiveness of the Oracle algorithm is the elimination of spatial and temporal granularities with the assistance of managing harmful interleaving. It also reveals the possible path to a better realistic shared cache management by adding these working elements into existing policies.

4.2.2 Quantitative Profiling of Oracle Algorithm Behavior

The key step in the Oracle algorithm is to select the intra-thread LRU block with the furthest reuse. According to the inter-thread locality discussion in Chapter 3, the multi-threaded reuse distance of any cache block has two components, namely intra-thread reuse distance, which is determined by its intra-thread locality, and the intervention due to interleaving. When a local LRU block is chosen by the Oracle algorithm as victim, it might be because of its long intra-thread reuse distance, or because of heavy intervention, or both. To understand how the Oracle algorithm behaves in real workloads, a detailed profiling is conducted on all 2-program workload traces in the experiment.

The first insight obtained from the detailed profiling information is how the Oracle algorithm selects victims from multiple threads. The process of replacement in shared cache has two steps. The first step is to choose the right victim among the blocks from the same thread. Since LRU is used as default intra-thread replacement policy, this step is to determine the LRU block within the individual threads. The second step is to choose the final victim from the victim candidates from multiple threads. Oracle algorithm has the benefit of future knowledge to direct victim selection, while any realistic policy does not. However, the analysis of profiling results shows that there is still a hint to direct online algorithm to make a reasonably good choice to improve the cache performance.

To enable the profiler to track the information of victim selection, each cache block in the cache is labeled with PID(thread id). During the process of the Oracle algorithm, upon each cache miss, the profiler records the missing thread ID, the final victim PID, and all the candidate victims' PID. From this information, one can gain insight into the pattern of the Oracle algorithm behavior, specifically, which thread is missing, which thread's block is evicted, and frequency of such events. In addition to the Oracle algorithm profiling, the profiler also keeps information related to LRU

event	missing PID	victim PID
E_{00}: 0	0	0
E_{01}: 1	1	1
E_{10}: 2	0	0
E_{11}: 3	1	1

Table 4.1: The event categories of missing PID and victim PID in the Oracle algorithm for 2-program workloads (thread 0 and 1)

and cache partitioning policies. For LRU information, the profiler maintains inter-thread LRU order of all the blocks in the cache. It enables the comparison of the Oracle algorithm and LRU policy behavior. There is also block-counting mechanism embedded in the profiler to provide cache capacity allocation information. This feature reveals the difference between the Oracle algorithm and cache partitioning algorithm.

This dissertation is particularly interested in showing the difference between the Oracle algorithm and LRU policy, because LRU policy provides a good baseline to develop novel cache policy. Analysis of such profiling information for 2-program workloads reveals that the Oracle algorithm sometimes evicts the global LRU block, but not always. It seems to be a random choice if we only look at the LRU stack position of victims. However, if the PID information of missing thread and the victim block is taken into consideration, there are some interesting regularities in the distribution of all 4 possible events (for thread 0 and 1) as shown in Table 4.1.

In this experiment, when a cache miss occurs, the PID of cache missing thread, the LRU stack position of each candidate victim, and the PID of the selected victim block, are captured by profiler. After a simulation of workload, the count of each category of event is computed and compared with each other. The analysis is focused on the difference of victim selection in the Oracle algorithm and LRU policy. For example, the profiler provides N_{01} as the count of the event E_{01} that thread 0 incurs a cache miss and a block of thread 1 is on the global LRU position. M_{01}, counted by profiler, is the count of the event that when the event E_{01} occurs, the Oracle algorithm chooses the block of thread 1 as victim. In one instance of E_{01}, the Oracle algorithm would choose victim from 2 candidate victims if there are two of them, while LRU would choose the candidate victim on global LRU position. According to this description, M_{01} instances of event E_{01} are the subset of N_{01} instances in which the Oracle algorithm makes the same decision as LRU policy. In other words, the matching ratio M_{01}/N_{01} denotes how well LRU victim selection matches the Oracle algorithm. If the ratio M_{01}/N_{01} is equal to 100%, LRU always makes the same victim decision as the Oracle algorithm in every instance of event E_{01}, while 0% indicates that LRU always makes the wrong choice of victim.

Let us look at the comparison of LRU and Oracle policies presented in Figure 4.10. There are four points for each workload to represent the matching ratio M_i/N_i in all four event categories. Each point is between 0 and 1 (100%), to represent LRU is either close to matching the Oracle behavior (close to 1) or close to the opposite of the Oracle behavior (close to 0). In order to highlight the workloads when LRU and the Oracle do not agree, we do not show the workloads when LRU nearly matches the Oracle (all four points for that workload are all close to 1). One

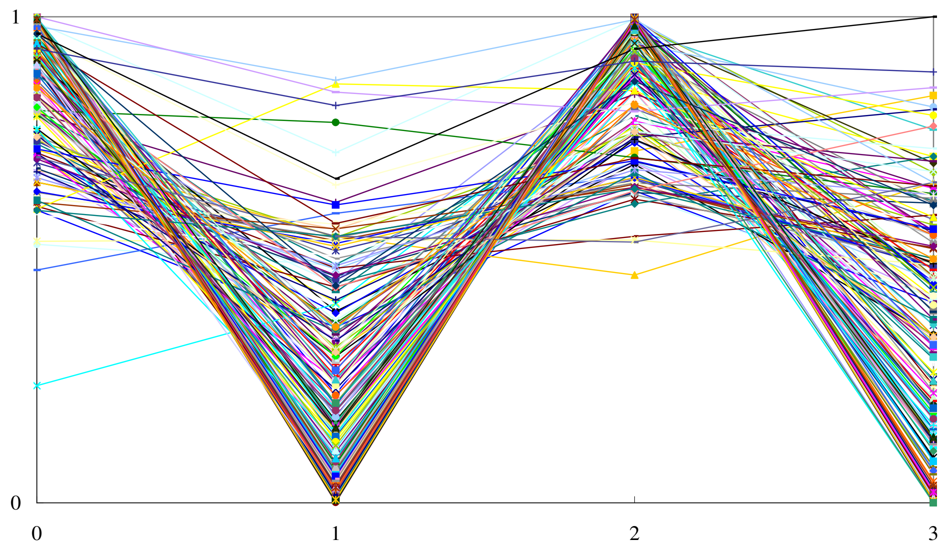


Figure 4.10: The percentage of LRU victim choice matching the Oracle victim choice for 4 event categories (0: E_{00}; 1: E_{01}; 2: E_{10}; 3: E_{11})

interesting observation from this figure is that most unmatching workloads share a similar pattern (the PID's of a workload are swapped to fit the pattern if necessary). This shared pattern is that LRU closely matches the Oracle behavior in most workloads for event categories 0 and 2, while for event categories 1 and 3 LRU and the Oracle strongly disagree.

The study of this pattern shows some insight into how the Oracle adapts to the locality and interleaving situations in two-program workloads. From the definition of the event category, we can see that both event 0 and 2 are evicting the blocks from thread 0 when they are on global LRU position. For event 1 and 3, in most workloads, the Oracle does not always evict the global LRU block which belongs to thread 1 as LRU policy. Instead, the Oracle most likely evicts the victim candidate of thread 0, which is not on global LRU position. In summary, the Oracle in any event shows strong bias in selecting the victim candidate of a particular thread, which is thread 0 in our experiment, no matter the missing thread is 0 or 1. In other words, the Oracle attempts to evict the LRU block among thread 0 for most of the time. This behavior restricts the cache capacity allocated to thread 0 by evicting thread 0's LRU block, and protects the cache capacity allocated to thread 1.

4.2.3 Aggressor Eviction Bias in Oracle Algorithm

Why does the Oracle treat these different threads differently? In Figure 4.10, thread 0 acts as an *aggressor* in the workload, not because thread 0 obtains more cache capacity than thread 1, but because thread 0 cannot utilize some of the obtained cache capacity more efficiently than thread 1. If thread 0 can give up some of its allocation to thread 1, and this allocation transfer will help to improve overall cache performance, the Oracle will select the victim candidate of thread 0 most of time. In this situation, we define thread 0 as aggressor, and thread 1 as non-aggressor. The critical key insight to the success of the Oracle algorithm is that it most likely selects aggressor's LRU block to evict upon a miss.

This pattern makes sense by taking both interleaving and locality into consideration. In Table 4.2, there are blocks with either high or low external interleaving, and either good or bad intrinsic locality. The target of shared cache management is to keep the blocks with good intrinsic locality in the cache, and evict references with bad intrinsic locality. In total four categories of blocks are shown in the table, and top 2 categories are of good locality and should be retained in the cache.

	High external interleaving	Low external interleaving
Good intrinsic locality	close to or on global LRU position	close to MRU position
Bad intrinsic locality	LRU position	close to or on global LRU position

Table 4.2: The block categories of different intrinsic locality and external interleaving, and their likely positions in LRU stack

The blocks with good intrinsic locality and low external interleaving are likely to stay close to the bottom of LRU stack (MRU position) because they are frequently reused and kept from moving up to LRU position with little external interleaving. Since these blocks usually stay low in LRU stack, most of them are not exposed to eviction policy. However, a good locality block suffering from high external interleaving will likely to be pushed up to LRU stack because the references from other threads increase its inter-thread reuse distance. Such blocks should be protected by cache policy, but LRU policy cannot help here because of its lack of ability to prevent such blocks from moving up in LRU stack. Usually the blocks falling into this category belong to non-aggressor, because the aggressor obtains too much allocation and victimizes the good locality blocks of non-aggressors. When the Oracle sees such blocks on global LRU position, it usually chooses to evict aggressor's LRU block instead of global LRU block, as shown by events 1 and 3 in Figure 4.10.

Now let us study how to handle blocks with bad intrinsic locality, which are the bottom 2 categories in Table 4.2. Blocks with bad intrinsic locality should be evicted from cache when threads compete for capacity upon a cache miss. The bad locality with high external interleaving are easy to detect because they often show up at global LRU position. They have high reuse distance and the high external interleaving makes their inter-thread distance even bigger, which increases their chances to move to global LRU position. LRU matches the behavior of the Oracle for these blocks because there is a good chance they are pushed to global position. On the other hand, bad locality blocks with low interleaving from other threads are likely to be moved to global LRU position or close to it. LRU will take care of such blocks when they are global LRU blocks, otherwise only the Oracle can detect such blocks belonging to the aggressor even before they move towards the global LRU position. For events 1 and 3 in Figure 4.10, the Oracle will frequently evict these blocks when they are not on global LRU position because the global LRU blocks usually belong to non-aggressor.

Therefore, the Oracle automatically captures the location of bad locality blocks, and the profiling of the Oracle for 2-program workloads show that such behavior is statistically biased towards the aggressor thread. Later in this dissertation, this pattern is exploited to approximate the behavior of the Oracle algorithm in the multi-program workloads and achieve considerable improvement over the existing management policies.

Chapter 5

Aggressor Eviction Algorithm

This chapter introduces a new cache management algorithm *aggressor-eviction*, which takes advantage of biasing pattern shown by the Oracle algorithm to approximate the Oracle behavior and achieves much of the potential of the Oracle to improve cache performance.

5.1 Using Aggressor Information in Victim Selection

In Section 4.2.3, the aggressor thread is identified by checking the difference between LRU and the Oracle behavior through the trace analysis. Whether a thread is aggressor or non-aggressor is determined by both interleaving and locality information from all the concurrent threads. Therefore, in the trace analysis the identification of aggressor can only be accomplished by comparing the relative performance of the Oracle policies against other policies.

Let us assume the knowledge of aggressor thread is available when we apply cache management policy on shared cache (the problem of identifying aggressor thread dynamically will be solved in Chapter 6). With this information, it is possible to improve performance by approximating the Oracle behavior. The strong bias shown in Figure 4.10 gives hints on how to take advantage of aggressor information. In that figure, thread 0 is the aggressor, while thread 1 is the non-aggressor. For events 0 and 2, LRU matches most of the Oracle decisions in many cases, therefore, when these events occur, global LRU blocks belonging to aggressor thread should be evicted. For events 1 and 3, when the global LRU block belongs to a non-aggressor thread, the victim candidate from aggressor thread should instead be selected for eviction.

The Oracle behavior shows a strong bias in Figure 4.10. In many cases (though not always), the Oracle algorithm selects the aggressor LRU block for eviction. It is especially the case for some workloads in events 1 and 3, where the matching percentage varies in a wide range. This is due to the disparity of aggressor thread across different sets and temporal epochs. Interaction among two programs are very complicated in two dimensions. First dimension of disparity is across all the cache sets. Since the behavior of each set is isolated from one another, there could be different aggressors in different sets simultaneously. Due to temporal phase behavior of programs, there also could be different aggressors in different time windows. With the presence of such inconsistency of aggressor thread, the Oracle still applies strong biasing in evicting aggressor's blocks in the workloads under study in our experiment. This pattern is exploited in a new shared cache management algorithm, which is called *Aggressor-Eviction*, as described in Table 5.1.

Upon every cache miss, the aggressor-eviction algorithm looks at the PID of the block at global LRU position. If that block belongs to the aggressor, no matter which thread is causing the cache miss, the global LRU block will be selected as victim and evicted from the set. Otherwise, a block belonging to non-aggressor is at the global LRU position when cache miss happens. Aggressor-eviction algorithm

missing thread	global LRU block	eviction victim
aggressor	aggressor	global LRU block
aggressor	non-aggressor	LRU aggressor block with probability = 0.99 global LRU block with probability = 0.01
non-aggressor	aggressor	global LRU block
non-aggressor	non-aggressor	LRU aggressor block with probability = 0.99 global LRU block with probability = 0.01

Table 5.1: Aggressor-Eviction Algorithm (assuming there are both aggressor blocks and non-aggressor blocks in the cache set)

will first check if there is any aggressor block in the cache set. If there is none, normal LRU policy will be activated; otherwise, the LRU aggressor block will be picked up as victim with some probability. The fraction value 0.99 is adapted in this algorithm to apply a strong bias towards aggressor eviction. The reason for using 0.99 is that it is necessary to allow aggressor to gain some capacity even if it is a small chance; otherwise, the non-aggressor block will never leave the cache unless evicted by the non-aggressor itself. The experiment results show that 0.99 works well for the cases that a strong bias is needed. Thus, with probability of 0.01, global LRU block belonging to non-aggressor is evicted. A similar thing happens to the case when a non-aggressor cache misses and the global LRU block belongs to a non-aggressor.

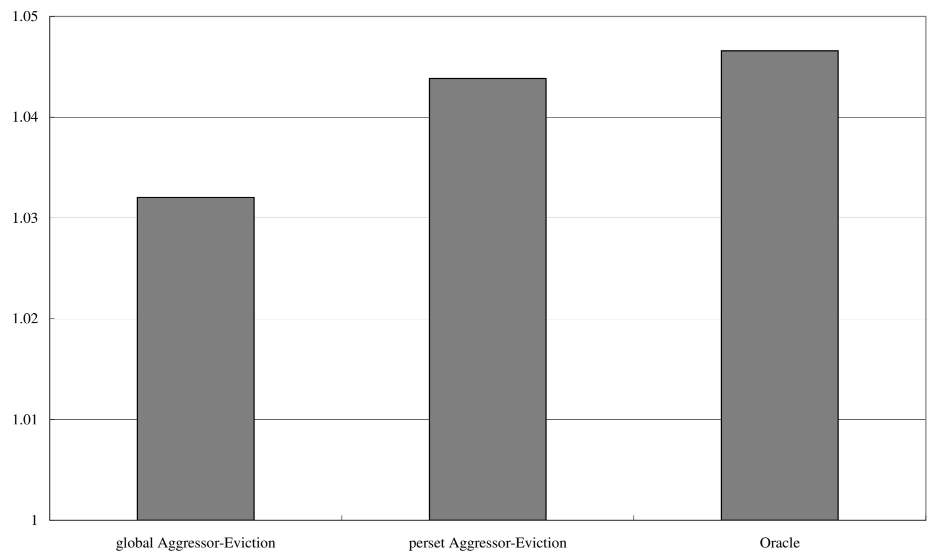


Figure 5.1: Cache performance (hit rate) comparison of global and perset aggressor-eviction with the Oracle algorithm (normalized to LRU hit rate).

5.2 General Aggressor-Eviction Algorithm

Just as cache partitioning can be applied using different spatial and temporal granularities, so can the aggressor-eviction algorithm. In order to compare aggressor-eviction algorithm with the Oracle algorithm, we study a more general form of aggressor-eviction algorithm, which we call *per-set aggressor-eviction*. The aggressor-eviction introduced in Section 5.1 is called *global aggressor-eviction* because only one aggressor is assumed for the whole cache. But in the per-set aggressor-eviction algorithm, each set can determine its own aggressor thread for the same epoch size as the global algorithm, which is small enough to detect temporal variation. We still keep the probability at 0.99 for eviction bias towards aggressor thread blocks.

Figure 5.1 shows the cache performance (hit rate) of both global (per cache) and perset aggressor-eviction algorithms and compares them to the Oracle algorithm. The Oracle algorithm can increase hit rate more than both aggressor-eviction algorithms, while there is only about 0.2% difference between the Oracle and global aggressor-eviction algorithm. The gap between global and perset aggressor-eviction algorithms is caused by the assumption that there exists a single aggressor thread across the entire cache, while the gap between the Oracle and perset aggressor-eviction algorithm is caused by the temporal granularity.

To reveal more detail, some profiling has been conducted to capture the PID of aggressor thread in each set, and less than 5% of all sets on average have a different aggressor thread from the global aggressor thread. Although there is about a 1% gap between global and perset aggressor-eviction algorithms, global aggressor-eviction algorithm is able to close the gap between LRU and the Oracle by about 70%. One advantage of global aggressor-eviction algorithm is that it only uses one global aggressor for the whole cache to adapt to most sets' individual behavior, thus avoiding the expensive hardware cost to monitor every set.

5.3 Comparing Aggressor-Eviction Algorithm to Cache Partitioning

Since the aggressor-eviction algorithm is motivated by the Oracle algorithm, it is expected to outperform existing shared cache policies. We compare global aggressor-eviction algorithm to iPART introduced in Chapter 3. In addition to the 2-program workloads, we also simulate the traces of 13 4-program workloads to check

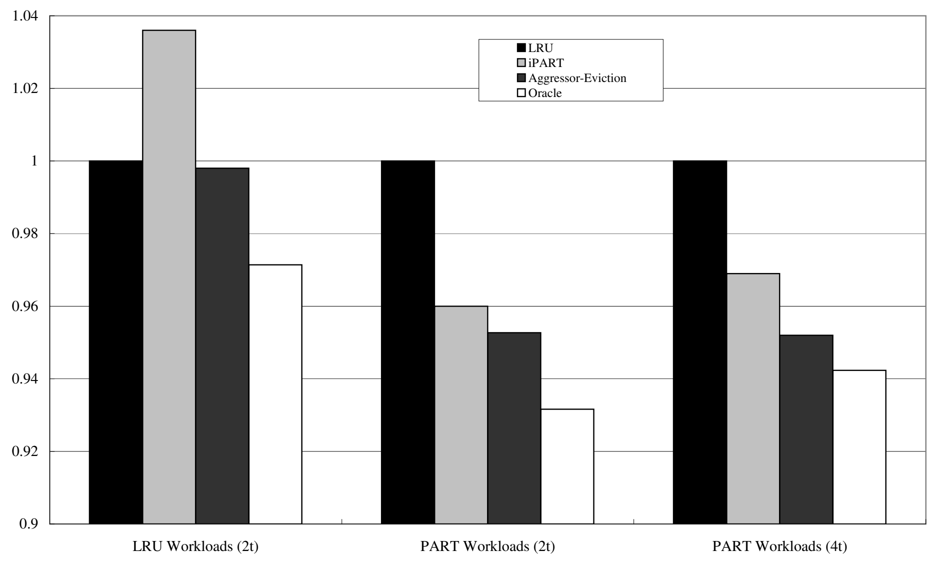


Figure 5.2: Cache performance (miss rate) comparison for LRU, iPART and the Oracle in workloads preferring LRU or partitioning (normalized to LRU miss rate)

the cache performance for workloads with larger thread counts. In this experiment, we divide the 325 2-program workloads into two groups based on their reaction to different policies. Each workload trace is simulated with LRU and iPART policies and cache performance is measured in miss/hit rate. Those workloads experiencing considerable miss rate difference ($\geq 1\%$) are grouped as LRU and partitioning workloads. LRU policy outperforms iPART in LRU workloads, while iPART is a higher performer in partitioning workloads. For the 4-program workloads, most are partitioning workloads except for *apsi-gcc-ammp-swim*. Figure 5.2 reports the cache performance measured in the *policy-sensitive* workloads for both 2-program and 4-program traces.

First, iPART reduces miss rate in partitioning workloads of 2 programs and 4 programs by 4.0% and 3.1%, respectively. However iPART increases miss rate in LRU workloads by 3.6% on average. On the other hand, the Oracle algorithm reaches the best performance in all the groups. The Oracle is better than the best performer, either LRU or iPART, by 2.8% and 2.7%, respectively. In the partitioning workloads of both 2 programs and 4 programs, the Oracle algorithm is able to utilize shared cache capacity more efficiently by applying adaptive allocation which fits each set. For LRU workloads, the Oracle algorithm also performs better than other algorithms (2.9% better than LRU).

It is interesting to see that aggressor-eviction, although it only approximates the Oracle algorithm, also outperforms both LRU and iPART in all groups. The aggressor-eviction algorithm is slightly better than LRU in LRU workloads, and almost 1% better than iPART in 2-program partitioning workloads. Aggressor-eviction outperforms iPART and LRU even more in the 4-program workloads because there are more conflicting working sets and more complicated interleaving with more competing threads. Although a large number of threads make it more difficult to apply uniform partitioning or no allocation regulation, experiments reveal that the aggressor identity is quite stable per workload. With correct aggressor information, aggressor-eviction is able to adapt to the variation across the cache sets, thus reaching higher utilization.

5.4 Aggressor-Eviction Algorithm Analysis with Interleaving

As discussed in earlier chapters, cache performance under a certain management policy is closely related to the interleaving of the references from concurrent

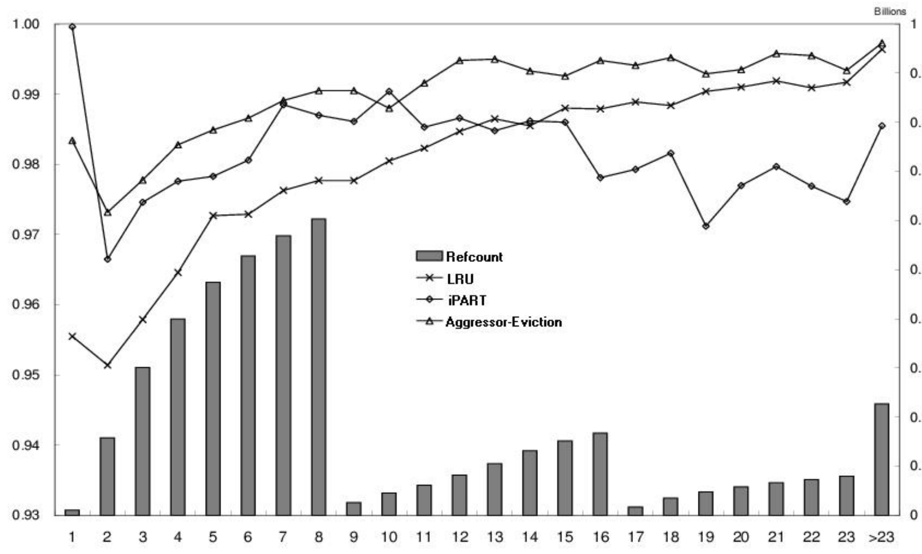


Figure 5.3: Cache performance (hit rate) comparison for LRU, iPART and the Aggressor-Eviction in workloads preferring LRU or partitioning (normalized to Oracle hit rate)

threads. To reveal the relation between interleaving granularity and cache performance under the aggressor-eviction algorithm, the trace simulator also measures the average runlength of each set in each epoch. Then we correlate cache performance under various policies in a certain set in one epoch with the average runlength measurement for the same set in the same epoch.

Figure 5.3 shows the performance comparison for LRU, iPART and the Aggressor-Eviction algorithm, and the distribution of different interleaving granularities. The performance points are measured in the sets with different average runlength. The counts of references in the sets with different average runlength are also shown in three regions: 1-8, 9-16, and beyond 17. The counts are shown in accumulative fashion, so the bar heights at 8, 16, and >23 on X-axis are the total reference counts for

	Coarse-Grain	Fine-Grain
LRU (2-program)	78.4%	21.6%
Partitioning (2-program)	28.0%	72.0%
Partitioning (4-program)	9.5%	90.5%

Table 5.2: Percent memory references performed in sets with coarse-grain interleaving and fine-grain interleaving for 2 and 4 program LRU/partitioning workloads.

the three regions. This figure shows that the aggressor-eviction algorithm performs better than both iPART and LRU except in a few cases. Cache partitioning usually outperforms LRU in the fine-grain interleaving region (average runlength smaller than 9), while LRU performs better in the coarse-grain region (average runlength larger than 16). In the medium grain interleaving region (average runlength from 9 up to 16), partitioning and LRU exhibit about the same performance. This is because the sets exhibiting medium runlength usually behave as a mixture of fine grain interleaving and coarse grain interleaving. A single policy, either partitioning or LRU, is not able to adapt to the mixture of interleaving granularities. Aggressor-eviction, however, can adapt to different interleaving granularities. As shown in the very fine grain interleaving region, aggressor-eviction outperforms partitioning while in very coarse grain interleaving region aggressor-eviction performs better than LRU.

This profiling matches the cache performance reported in Figure 5.1 because there are more coarse grain interleaving sets in LRU workloads, while fine grain interleaving sets dominate PART workloads. The distribution of memory references interleaved at different granularities is presented in Table 5.2. For 2-program workloads, LRU workloads and partitioning workloads have the opposite distribution of fine-grain and coarse-grain interleaving references. Fine-grain interleaving references dominate in 4-program partitioning workloads because with more threads accessing the cache, the average runlength remains small. After discussing the interleaving granularities at reference and set level, let us look back at Figure 3.5, which shows the performance impact at the per-cache level. The correlation between the per-cache comparison of LRU and partitioning and runlength is supported by the profiling at set and reference level.

In summary, the aggressor-eviction algorithm is able to adapt to different interleaving granularities, and achieves most of the cache performance demonstrated by the Oracle algorithm. However, to be practical, the aggressor-eviction algorithm requires identifying aggressor threads in runtime. Next chapter will propose an on-line mechanism to address this issue.

Chapter 6

Probabilistic Replacement Policy

Chapter 5 introduces the aggressor-eviction algorithm which has the potential to outperform existing shared cache management policies. However, the aggressor-eviction algorithm has some issues to solve before it can be employed in real systems. First, the key information to its effectiveness is the aggressor identity information. Since the aggressor is neither defined uniquely by locality information nor by interleaving information, it requires both information to accurately determine the identity of aggressor, which can incur a large hardware cost. Second, aggressor-eviction algorithm is focused on maximizing overall cache performance (minimizing total miss rate). However, as pointed out by other researchers [60, 70], to optimize overall performance in terms of IPC, cache management policy also has to consider fairness.

This chapter presents a practical online shared cache management policy based on aggressor-eviction algorithm: *Probabilistic Replacement policy (PR)*, which achieves performance gain at moderate hardware cost. Detailed performance evaluation is also reported and compared with other policies.

6.1 Probability Replacement Ratios

In aggressor-eviction, once the identity of aggressor thread is known, a strong bias, i.e., a probability of 0.99, is applied to evict the least recently used aggressor block when the global LRU block belongs to a non-aggressor thread. This mechanism is effective in improving overall cache hit rate; however, actual performance gain is determined by factors other than overall cache hit rate. In order to measure the overall performance by considering every thread, researchers tend to use weighted IPC to consider not only absolute IPC values but also IPC variation due to shared resource competition. If the optimization target is to minimize overall miss rate, the management policy tends to favor the thread which can reduce misses most given extra capacity. But this strategy can hurt weighted IPC if the favored thread eliminates a large number of its cache misses without improving its IPC proportionally, especially if other threads suffer significant IPC degradation as a consequence of.

Researchers have shown [70, 69] that IPC improvements are often not linear with cache performance improvements. For example, in a 2-program workloads (A, B), it is often the case that if program A's performance is more sensitive to miss rate than program B, then it makes sense to favor program A by allocating more cache capacity to it and reducing its miss rate. This is true even if doing so hurts the overall miss rate by increasing the number of cache misses incurred by B. To improve overall cache miss rate, we would allocate more cache capacity to program B in this example. Then the allocation decisions are in conflict due to different

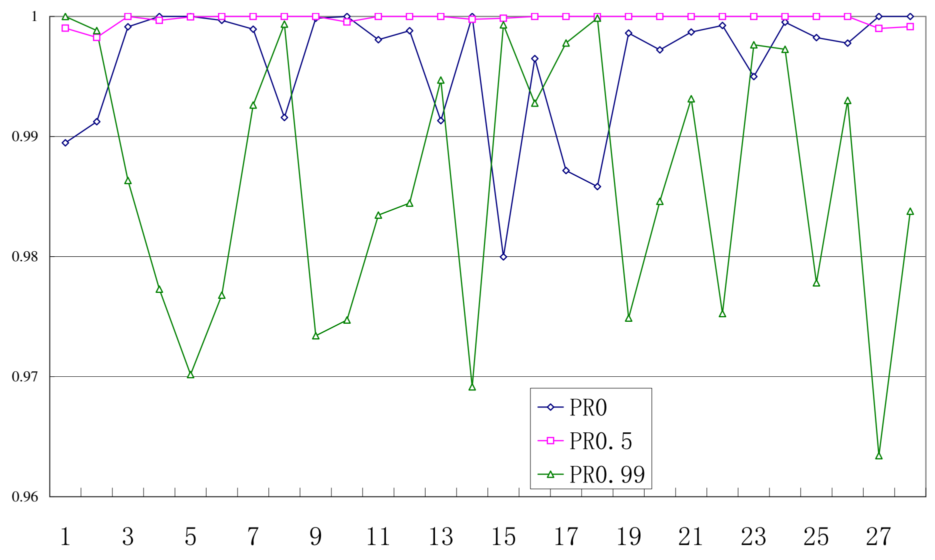


Figure 6.1: Performance (weighted IPC) per epoch comparison of different Probability Ratios in ammp-twolf workload (normalized to the highest performance for each epoch)

goals. In other words, the optimal cache allocations are different given different goals (WIPC vs miss rate). There are also numerous cases in which these two goals are in line with each other, where the aggressor-eviction algorithm will deliver the optimal performance in terms of both overall miss rate and weighted IPC, but a large number of cases also exist in which the two goals are in opposition.

In order to handle these cases, the aggressor-eviction algorithm is extended with more options than the single 0.99 probability. Instead of relying on the single probability of 0.99, a new policy is developed that provides more flexibility in biasing towards aggressor thread at different levels. This variable ratio is called the Probability Ratio, *pr*. A large value of *pr* at 0.99 will victimize aggressor threads almost all the time, while protecting non-aggressor threads fully. If instead we choose

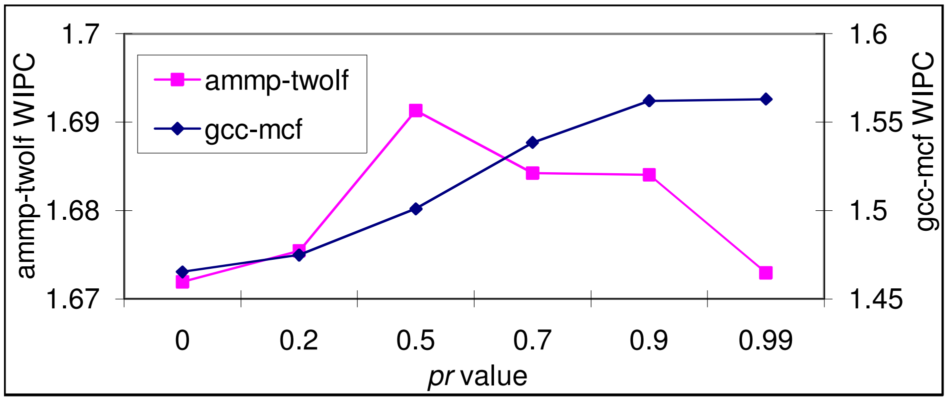


Figure 6.2: Performance (WIPC) as function of different *pr* values for two typical workloads

a smaller *pr* value, aggressor threads will be allowed to keep more cache allocation because aggressor threads' blocks are less likely to be selected as victims. If a *pr* value is set to be 0.0, the cache will follow the LRU decision.

The benefit of probability flexibility is realized in the detailed performance experiments, as shown in Figure 6.1. Various *pr* ratios from 0.0 up to 0.99 are applied in every epoch for the ammp-twolf workload. The weighted IPC results for each epoch are registered for three *pr* ratios: 0.0, 0.5, and 0.99, normalized to the highest WIPC among all the ratios. This workload shows phase behavior across epochs. The performance achieved by using 0.99 *pr* is not always the highest, nor is LRU policy ($pr = 0$). However, a moderate *pr* value 0.5 approaches the highest performance in most epochs. This workload represents a category of workloads that shows concave relation between performance and *pr* values.

In order to reveal what this relation looks like for other *pr* values, the relative performance of all the *pr* values is reported in Figure 6.2. All six *pr* values: 0.0, 0.2, 0.5, 0.7, 0.9, and 0.99, are sampled for two typical workloads: ammp-twolf and gcc-mcf, which represent two different categories of workloads. Workload ammp-twolf represents a category of workloads with concave relation between performance and *pr*, whose performance reaches peak at a moderate *pr* value, 0.5. Such workloads usually have a program whose cache miss rate will decrease with larger *pr* value, but its IPC performance will saturate as miss rate drops below a certain level. With a program that exhibits such a performance characteristic, it makes sense to bias towards this program to a certain point but not beyond that point. Statistics show that a *pr* value of 0.5 works well with most workloads in this category. Although for some workloads in this category, an optimal *pr* value may not occur at exactly 0.5, for the sake of reducing runtime overhead (discussed later in this chapter), we choose 0.5 as an extra *pr* level rather than allowing every *pr* value in Figure 6.2.

Workload gcc-mcf represents another category of workloads in which performance increases monotonically as a function of *pr* with peak performance at the *pr* value 0.99. In these workloads whose performance bottleneck is often off-chip bandwidth, minimizing overall miss rate usually also maximizes overall performance. For such workloads, evicting aggressor thread blocks can achieve performance gain by increasing *pr* values, as shown by the gcc-mcf example. These two main categories dominate the workloads in our experiments. This observation can help to reduce the complexity of designing an online algorithm.

6.2 Online Sampling and Learning of Aggressor Information

The aggressor-eviction algorithm introduced in Chapter 5 is an idealized policy because it assumes perfect information about the aggressors' identity. However, as discussed in previous chapters, both interleaving and locality information is required to determine aggressor threads, which can incur a high implementation cost to acquire. To avoid prohibitive cost, an online sampling technique is proposed to dynamically obtain aggressor information. Another factor to consider in designing a real policy to optimize IPC performance is the potentially non-deterministic relationship between overall miss rate and weighted IPC, as shown in the last section. Thus, the online policy is targeted at optimizing IPC performance instead of overall miss rate. The observation from last section that three most likely optimal *pr* ratios (0.0, 0.5, and 0.99) can help to avoid testing numerous possible ratios. The aggressor thread has a positive *pr* ratio, either 0.5 or 0.99, while a non-aggressor thread uses -0.5 or -0.99 as its *pr*. A *pr* of 0.0 will be applied on a thread which is not sensitive to the policy, and follow LRU behavior.

There are a number of runtime performance gauging proposals [60] which involves the measurement of single program performance under both single-threaded execution and multi-threaded execution. Each sampling technique, including our technique, has its own overhead in terms of hardware and sampling delay. The hardware cost of our online technique will be covered later in this chapter, while this section focuses on performance.

The first step in our technique is to gauge single-threaded performance. Each

```
/* T = number of threads */
/* $wipc_i$ = multithread_$IPC_i$ / singlethread_$IPC_i$ */
/* WIPC = $\Sigma$ $wipc_i$ */
/* do_epoch($pr_1$, ..., $pr_T$): execute 1 epoch using $pr_i$'s */
/* NewPhase(): true if $wipc_i$ order changes, else false */
main() {
    while (1) {
        Sample();
        do {
            do_epoch($pr_1$, $pr_2$, ..., $pr_T$);
        } while (!NewPhase());
    }
}
```

```
Sample() {
    $WIPC_{LRU}$ = do_epoch(0, ..., 0);
    for (i = 0; i < T; i ++) {
        $WIPC_{i+}$ = do_epoch(0, ... +0.5 ..., 0);
        if ($WIPC_{i+}$ > $WIPC_{LRU}$) {
            $pr_i$ = +0.5;
            $WIPC_{i++}$ = do_epoch(0, ... +0.99 ..., 0);
            if ($WIPC_{i++}$ > $WIPC_{i+}$) $pr_i$ = +0.99;
            continue;}
        $WIPC_{i-}$ = do_epoch(0, ... -0.5 ..., 0);
        if ($WIPC_{i-}$ > $WIPC_{LRU}$) {
            $pr_i$ = -0.5;
            $WIPC_{i--}$ = do_epoch(0, ... -0.99 ..., 0);
            if ($WIPC_{i--}$ > $WIPC_{i-}$) $pr_i$ = -0.99;}
    }
}
```

Figure 6.3: Online sampling algorithm for detecting aggressor threads and testing *pr* value for each thread

program of the multi-program workload will be executed for an epoch without executing any other thread. This overhead is linear with the total number of threads in the workload. We assume that single program IPC is stable through the entire simulation window, and the performance loss due to this sampling is not significant. We do not include this sampling overhead in the performance evaluation.

The second step is to identify aggressor threads, which is conducted more often than single program IPC measurement due to the frequent variation of interaction among threads. This step is illustrated by the *Sample*() function in Figure 6.3. Before testing each thread, one epoch is simulated with LRU policy, which provides a baseline performance with each thread at 0.0 *pr*. For every thread in the workload, we run one epoch in which a positive *pr* ratio of 0.5 is applied on that thread while all the other threads are using 0.0 ratio. If the resulting WIPC is better than the performance of the baseline LRU WIPC, we decide that this thread is an aggressor, and another epoch is run with 0.99 applied on this thread while other threads remain with the 0.0 ratio. This second epoch for this thread is to fine tune performance, and if 0.99 pr ratio gives even higher WIPC, we will use 0.99 for this thread until the next resampling of this thread and move on to test the next thread without testing negative pr ratios. If 0.5 is shown to deliver the highest performance (higher than both LRU and 0.99), this ratio will be applied on this thread until the next resampling. However, if positive 0.5 turns out to be underperforming the LRU performance, negative 0.5 will be used to test this thread for an epoch without further testing positive 0.99. If negative 0.5 outperforms LRU performance, a further test will be conducted with negative 0.99, otherwise we conclude that this thread reacts best with the LRU policy and will use 0.0 until the next resampling. This process will be repeated on each thread, so the total cost is linear with the total number of threads. For each thread, the number of sampling epochs is 2 or 3. There is little parallelism lost in sampling epochs, and resampling ($Sample()$) is not necessary until a new phase is detected ($NewPhase()$). We monitor the order of individual weighted IPC $wipc_i$ during the simulation, and any variation of their relative order hints at a new phase, and resampling is conducted to retest each thread for their appropriate pr values.

After this step, each thread is assigned with a pr ratio. An aggressor thread has a positive ratio, while a non-aggressor thread has a negative ratio. The behavior upon cache miss is controlled by pr ratio which is illustrated in Table 6.1. This new policy is based on aggressor-eviction algorithm, but extended with support for more than 2 threads. When a cache miss occurs, the pr value of the cache missing thread is checked. If the cache missing thread has pr of 0.0, this cache miss is handled by the LRU policy, and the global LRU block will be victimized, no matter its

missing thread pr	global LRU block pid	eviction victim
positive (r)	non-aggressor	probability r: the LRU aggressor block probability $1 - r$: global LRU block
negative ($-r$)	non-aggressor	probability r: the LRU aggressor block probability $1 - r$: global LRU block
0.0	any	global LRU block

Table 6.1: Victim selection for thread with different pr values

PID. The change from LRU behavior occurs for aggressor or non-aggressor cache missing threads when the global LRU block belongs to a non-aggressor. Under such a condition, the conventional LRU policy would choose the non-aggressor global LRU block as victim, whereas our policy will attempt to preserve non-aggressor's allocation by instead evicting the least recently used block among all aggressors' blocks with probability pr.

6.3 Hardware Cost

Our probabilistic replacement policy is based on the LRU policy, which is a popular implementation choice. On top of the hardware required by basic LRU policy, each block is labeled with its PID, which is also demanded by other shared cache management techniques such as cache partitioning. For a CMP with N cores, it takes $log_2 N$ bits to encode a PID from 0 to N. For the baseline 1M byte 8-way set-associative shared L2 cache in our study, 16k bits are added to support 2 threads,

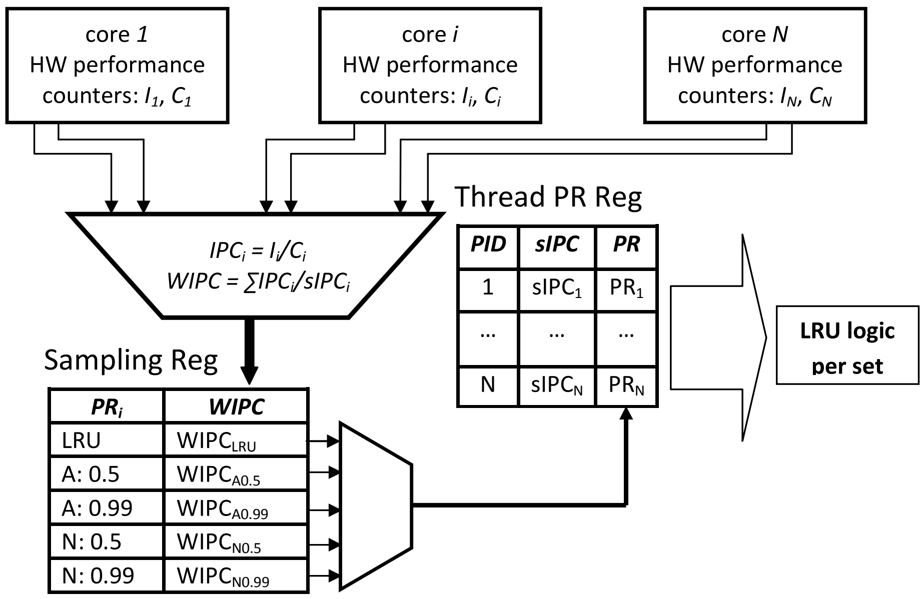


Figure 6.4: PR central control hardware

and 32k bits for 4 threads (both within 1% overhead).

The conventional LRU policy has a per-set LRU stack to keep track of each block's last use. To distinguish aggressor threads from non-aggressors, there is an extra bit to label aggressor or non-aggressor. The thread with 0.0 *pr* is labeled with non-aggressor, and can be identified with this label and *pr* ratio information. In addition to the bits and logic to support per-set LRU order, we also add bits and logic to maintain another LRU stack for aggressor threads because the algorithm described in Table 6.1 demands the LRU block among all aggressor blocks. We allow aggressor threads to share the same LRU stack while cache partitioning techniques demand individual LRU lists per thread for every concurrent thread.

Having described the hardware overhead for blocks and sets, let us examine the hardware for the centralized control logic, as shown in Figure 6.4. There is a global "Thread PR Registers" for all the threads, at most 3 bits per thread is needed to encode all 5 possible *pr* ratios. This central control logic uses a random number generator to drive the probabilistic decision. There are also “Sampling Registers” in the central control for the resampling process to store the temporary IPC measurement and compute WIPC in evaluating appropriate *pr* values. The hardware complexity of the control logic does not grow with cache size, but scales up linearly with the total number of threads. The probabilistic replacement policy is activated upon a cache miss, and the victim selection process can completely overlap with missing block fetch latency, so the hit and miss latencies are not compromised. To compute performance, the control logic obtains IPC readings from hardware performance monitoring infrastructure such as Intel performance counters (I: instruction count, C: cycle count), which is widely available in modern processors.

Further hardware cost reduction is possible with software assistance. Most of the centralized control logic can also be implemented in OS handler to execute resampling process at the beginning of each epoch. Compared with the normal epoch size (at least 1M cycles), at most hundreds of cycles spent on executing $Sample()$ will not incur much performance penalty but scale well with larger number of threads.

6.4 Performance Evaluation

On-line probabilistic replacement policy is implemented in M5 simulator to support multiple program workload simulation. The simulation tools are also able to simulate other shared cache management policies: LRU, UCP, iPART, and iPR (per-set aggressor-eviction introduced in Chapter 5). Like other studies, LRU policy

	ammp	applu	apsi	art	bzip2	crafty	eon	equake	facerec	fma3d	galgel	gap	gcc	gzip	lucas	mcf	mesa	mgrid	parser	perlbmk	sixtrack	swim	twolf	vortex	vpr
applu	12.6%																								
	13.2%																								
apsi	-5.8%	1.1%																							
	0.0%	0.9%																							
art	22.3%	2.1%	-8.7%																						
	18.7%	5.6%	0.0%																						
bzip2	-3.2%	8.2%	-4.6%	10.7%																					
	0.0%	9.7%	0.0%	0.0%																					
crafty		7.5%	-4.9%	9.3%																					
		9.3%	0.0%	5.4%																					
eon		3.5%		5.3%																					
		3.6%		5.2%																					
equake	2.2%			0.5%	4.0%	2.2%																			
	0.0%			0.0%	0.0%	0.4%																			
facerec				-0.7%	1.4%			1.5%																	
				1.0%	0.0%			0.0%																	
fma3d	1.1%	1.3%		-3.5%	1.2%	2.1%																			
	1.6%	0.0%		0.0%	2.2%	2.6%																			
galgel	1.4%	5.3%	-2.7%	-2.0%	4.4%	1.1%		2.3%		-0.6%															
	0.0%	5.4%	0.0%	0.0%	1.9%	0.0%		0.0%		1.6%															
gap	0.8%		-1.5%	-2.4%	2.8%	1.8%			1.6%	0.4%	1.6%														
	1.1%		0.0%	1.3%	3.6%	2.4%			0.1%	0.0%	4.4%														
gcc	-1.7%	19.1%	-14.1%	9.9%	-4.8%	-9.7%	-5.9%	9.5%	-4.4%	-1.8%	18.8%	-1.4%													
	4.4%	29.8%	0.0%	5.9%	0.0%	0.0%	0.0%	14.0%	3.1%	7.3%	28.0%	7.9%													
gzip	1.0%	15.0%	-3.1%	17.9%				4.2%	1.5%	3.6%	4.9%	2.3%	-3.0%												
	0.0%	14.0%	0.0%	13.7%				0.0%	0.0%	3.1%	1.0%	0.0%	0.0%												
lucas	3.3%	1.0%	1.2%	0.0%	5.6%	5.0%	1.1%				4.2%		9.1%	7.4%											
	3.6%	0.0%	1.8%	3.5%	5.9%	5.7%	1.3%				7.3%		21.5%	7.5%											
mcf		0.3%		-1.0%	1.8%	-0.1%	-5.4%		0.3%		1.3%	1.2%	-2.8%	2.5%											
		4.4%		1.6%	2.1%	1.0%	0.0%		2.7%		6.4%	3.8%	9.4%	0.0%											
mesa		8.6%	-2.0%	5.3%				1.0%	-1.0%	1.1%	1.8%		-5.2%		2.2%	-2.5%									
		7.3%	0.0%	0.0%				0.0%	0.0%	0.3%	1.0%		2.4%		2.2%	0.0%									
mgrid	2.3%	2.2%		0.9%	5.5%	3.1%	1.5%				3.5%		23.7%	6.2%		-6.1%	2.9%								
	2.5%	2.9%		5.0%	5.6%	4.3%	1.6%				5.6%		35.7%	5.7%		0.0%	1.8%								
parser	1.2%	10.0%	-6.0%	5.6%	3.4%	-1.3%		8.5%	4.8%	5.5%	7.7%	6.3%	-2.4%		11.1%	6.7%		9.5%							
	0.0%	11.2%	0.0%	0.2%	0.0%	0.0%		2.6%	2.8%	6.4%	5.9%	6.0%	0.0%		11.5%	7.0%		10.6%							
perlbmk		10.9%	-2.9%	27.5%						1.5%			4.2%		3.9%	-2.8%		1.4%							
		10.9%	0.0%	27.2%						1.5%			4.2%		3.9%	0.0%		1.4%							
sixtrack				-2.3%	0.1%	0.9%					-1.6%		-7.1%			-1.3%									
				0.0%	0.6%	1.0%					0.0%		0.0%			8.1%									
swim	32.9%	0.2%	1.6%	8.6%	24.3%	9.5%	3.9%		4.0%	3.0%	8.2%	1.6%	34.8%	21.7%		5.3%	9.1%	7.3%	15.3%	17.6%					
	32.8%	-1.1%	1.8%	7.9%	23.5%	12.7%	3.6%		1.0%	0.0%	6.4%	0.5%	36.6%	22.5%		0.5%	8.5%	0.0%	15.2%	17.4%					
twolf	-3.7%	25.1%	-5.7%	6.1%	-1.2%	-4.9%	-3.7%	19.1%	8.3%	12.0%	19.0%	11.5%	0.0%		21.4%	14.8%	-1.8%	22.6%		-7.4%	0.3%	33.9%			
	0.0%	25.5%	0.0%	6.8%	1.0%	0.0%	0.0%	14.1%	9.6%	14.5%	1.3%	13.1%	7.6%		23.2%	17.4%	0.0%	23.5%		0.0%	4.4%	31.0%			
vortex		9.8%	-6.6%	18.6%	-2.2%			2.2%		1.8%	3.4%	1.1%	-8.8%		6.0%			4.2%	-2.4%			14.5%	-6.9%		
		9.6%	0.0%	15.6%	0.0%			0.5%		1.8%	0.0%	0.0%	0.2%		6.3%			4.3%	0.0%			14.7%	0.0%		
vpr		4.2%	-6.1%	1.7%	-0.3%	-1.2%	-1.1%	4.5%	1.9%	1.9%	3.2%	3.0%	-2.8%	1.0%	5.1%	3.6%		4.3%		-3.1%		9.4%	6.3%	-1.7%	
		4.3%	0.0%	0.2%	0.0%	0.0%	0.2%	4.4%	1.2%	2.6%	0.6%	3.7%	0.0%	0.2%	5.8%	4.8%		4.6%		0.0%		6.6%	0.0%	0.0%	
wupwise		1.4%			1.8%	1.1%					1.4%		-0.8%	2.1%					5.5%			2.4%	11.4%		2.1%
		0.0%			1.6%	0.9%					3.2%		7.6%	0.7%					5.0%			0.0%	13.4%		2.8%

Figure 6.5: Comparison of PR and iPART on 2-program workloads

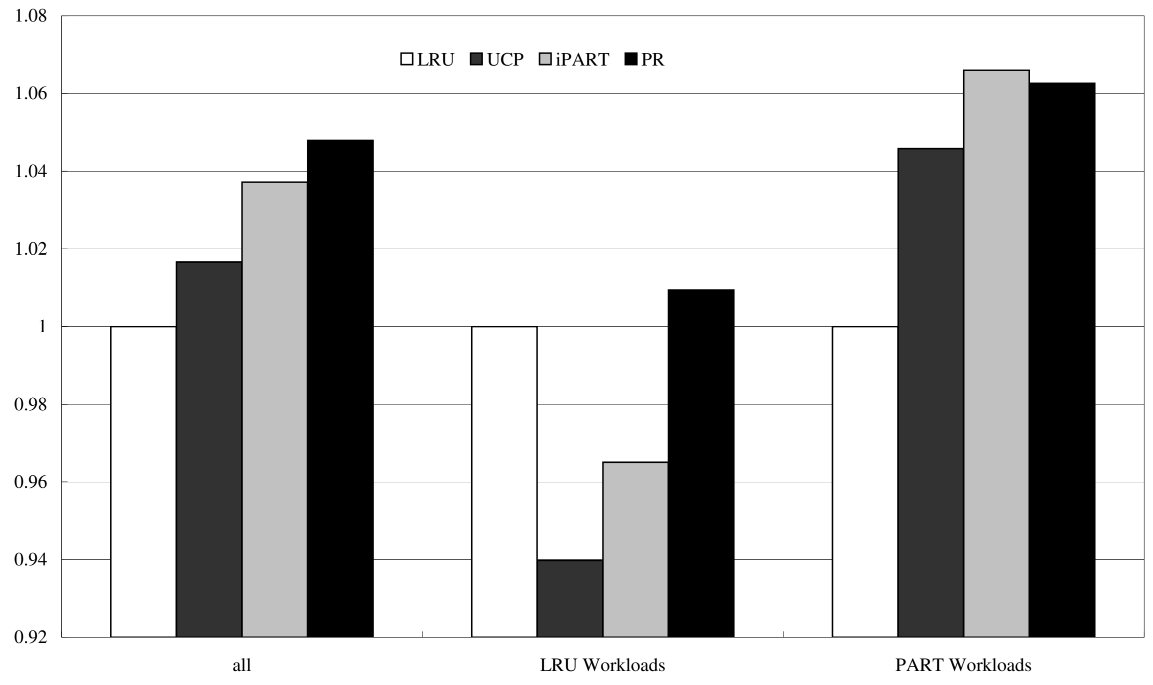


Figure 6.6: Comparison of PR, UCP and iPART on our 2-program workloads (normalized to LRU performance)

is employed as the performance baseline. UCP is a recent shared cache policy based on program locality information. Single program reuse distance profiling is captured by hardware utility monitors to drive optimization logic which chooses a uniform partitioning for each set to minimize overall cache miss rate. We implement UCP with utility monitor for all the sets to minimize the performance penalty of incomplete profiling, and consider the performance of UCP as a reasonable representation of conventional practical cache partitioning techniques. These three practical policies: LRU, UCP, and PR, are simulated in the same 500 million cycle window, with epoch size of 1M cycles.

As introduced in Chapter 4, iPART is an ideal cache partitioning algorithm which picks the best performance from all the possible uniform cache partitionings.

We design iPR (ideal PR) using the same exhaustive searching technique to select the best performance from all possible *pr* combinations of all threads. These two ideal techniques stand for the best possible performance of cache partitioning and PR with zero online prediction penalty. Due to high simulation cost for exhaustive searching, these two ideal algorithms are simulated in the same 100 million window (starting from the same point as the 500 million window for practical policies).

First, let us compare PR and iPART. Each two-program workload has a pair of results in Figure 6.5. Each workload is composed by two programs on the same row and the same column. The first value is the performance difference between PR and LRU, and the second value is the difference between iPART and LRU, both normalized by LRU WIPC. Out of all 325 workloads, 109 workloads do not have values because neither difference for these workloads is more than 1%, and we regard them as insensitive to any policy. When the working sets of two programs can fit in the cache together, they are compatible under most policies. There are 154 workloads that perform better under cache partitioning as compared to LRU, while 62 workloads perform better under LRU. We call these partitioning workloads and LRU workloads, respectively.

For LRU workloads, UCP and iPART underperform the LRU policy by 6% and 3.5%, while online PR outperforms LRU by 0.9%. This performance difference matches the trace analysis that most such workloads have coarse grain interleaving, and the interaction of locality and interleaving demands different cache allocation in different sets but uniform partitioning can not satisfy this requirement. PR does not only perform better than cache partitioning, but also has slightly higher performance

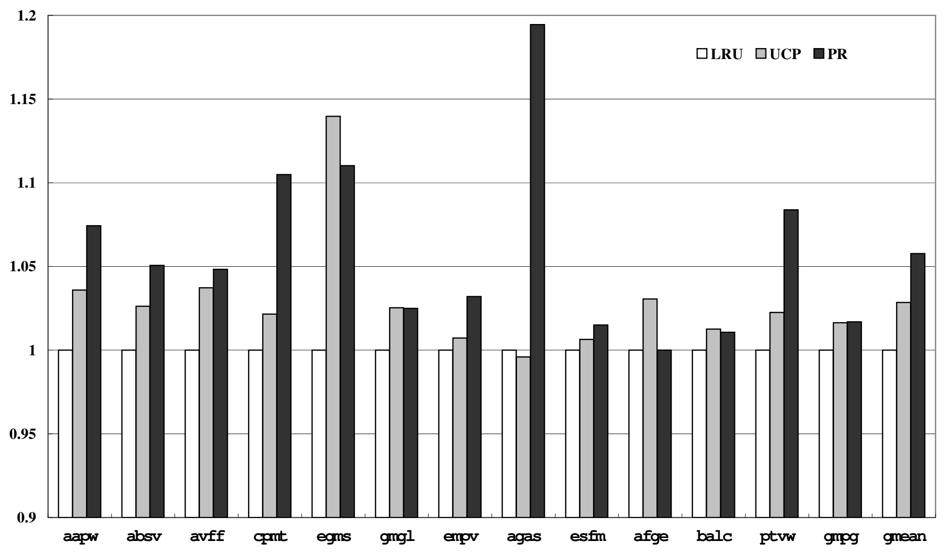


Figure 6.7: Comparison of PR, UCP and iPART on 4-program workloads (normalized to LRU performance)

than LRU. This is because the interleaving granularity is highly uneven across the sets. Even when the majority of the sets prefer LRU, there are still some sets preferring allocation regulation (cache partitioning), as shown in Table 5.2. This slight performance edge of PR over LRU shows the versatility in handling sets with different interleaving granularities. This advantage of PR is also shown by the trace analysis in Figure 5.2.

Cache partitioning policies do much better in partitioning workloads, achieving more than 4% gain over LRU. As discussed in previous chapters, partitioning workloads usually require explicit allocation regulation which is provided by cache partitioning techniques. Although PR outperforms UCP by almost 3%, it is still slightly outperformed by iPART. Note that iPART is an ideal algorithm which suffers zero searching penalty, while PR spends a number of epochs in resampling when a new phase arrives. Another price PR has to pay is the limited option in *pr* ratios. It is shown to be near-optimal, but in some cases the true optimal *pr* falls in between the available *pr* values in our design. UCP also suffers from searching inaccuracy, and this penalty plus the mismatch between optimal miss rate and optimal weighted IPC accounts for over 2% difference between UCP and iPART.

If we look at all the workloads, PR is the best performer, 1% above iPART, more than 3% above UCP and almost 5% above LRU. We also find a similar relative performance for 4-program workloads, as shown in Figure 6.7. In the simulation of 4-program workloads, the cache capacity and associativity are scaled up to 2MB and 16. Because of the high cost of exhaustively simulating all possible cache partitionings, we study LRU, UCP and PR for 4-program workloads excluding ideal policies. In all 13 4-program workloads in our experiment, there is only one LRU workload apsi-gcc-ammp-swim. All the remaining 12 workloads have fine grain interleaving. This result matches the trace interleaving analysis in Table 5.2, which shows more than 90% of references are interleaved at fine granularity. Although most references in most 4-program workloads are overwhelmingly interleaved at fine granularity, the presence of small number of coarse grain interleaved references still makes it hard for cache partitioning to reach high performance. PR again shows its advantage in adapting to different interleaving granularities by outperforming UCP in 8 workloads, matching it in 3 workloads, and noticeably underperforming in only 2 workloads. On average, with 5.9% performance improvement, PR almost doubles the performance gain reached by UCP over LRU.

Although PR is better than UCP by only about 3% on average, PR outper-

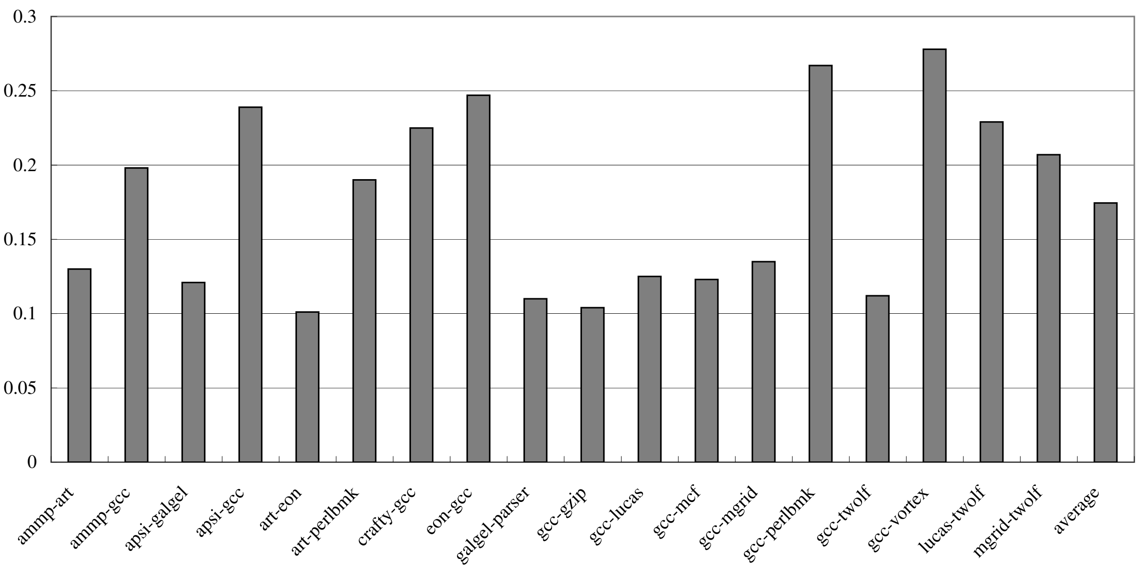


Figure 6.8: PR's performance advantage over UCP on 2-program workloads (with larger than 10% performance advantage)

forms UCP significantly in a number of benchmarks. Figure 6.8 shows all the 2-program workloads with PR outperforming UCP by at least 10%. The average performance edge of PR over UCP is 17.5% in the 18 workloads shown in Figure 6.8, with performance gain up to 28%. For these workloads, cache partitioning techniques such as UCP cannot adapt to all the cache sets because there are various interleaving granularities across most sets. PR can adapt to different interleaving granularities in these workloads which have stable aggressor thread identity across various sets and epochs. In contrast, there are only two workloads with UCP outperforming PR by at least 10%: mgrid-swim (10%), and bzip2-gzip (11.7%). In such workloads, PR suffers from online algorithm latency, or incorrect aggressor threads identification. If a workload experiences frequent new phases, PR has to activate resampling very often. Thus it results in slow reaction to phase changes.

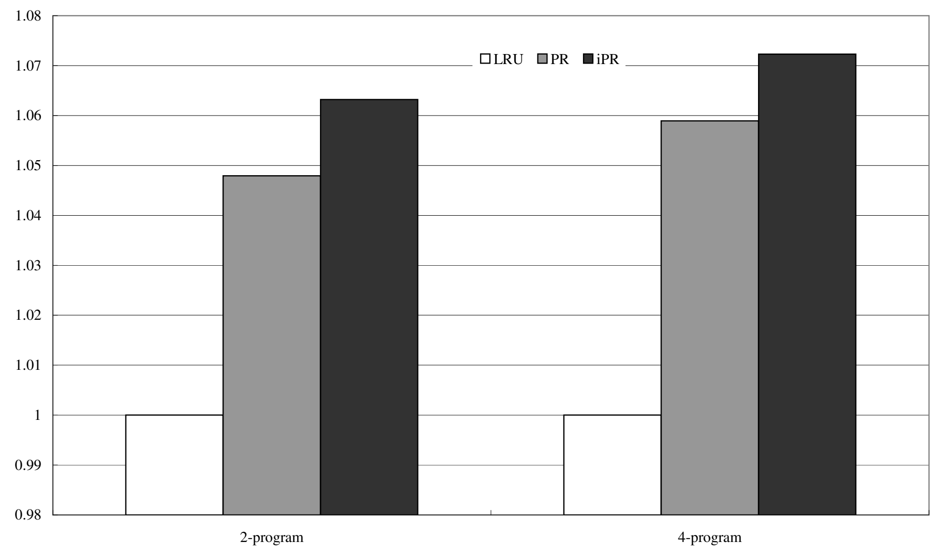


Figure 6.9: Comparison of PR and iPR on 2-program and 4-program workloads (normalized to LRU performance)

Finally, we present the performance of PR and ideal PR in both 2-program and 4-program workloads in Figure 6.9. In both cases, there is a 1.5% performance gap between online PR and ideal PR, because of searching penalty and inaccurate aggressor identification in some workloads. Nevertheless, our online PR effectively achieves over 70% of the performance gain potential of ideal PR.

In summary, the simulation of both 2-program and 4-program workloads indicate that PR is an effective cache management policy which can flexibly adapt to the different interleaving and locality across the different sets in shared caches. It can improve both efficiency and performance of multi-program workloads on CMPs.

Chapter 7

Reuse Distance Based LTP-Driven Cache Management

7.1 Introduction of Last Touch Prediction

In addition to multi-threaded shared cache research, single program cache management is also one of the research topics in this dissertation. As in shared caches, for most processors, uniprocessor caches employ the LRU policy to drive replacement decisions on cache misses. Belady's study on the MIN algorithm [9] sets up the theoretical upper limit for single thread cache performance. Recent works show that there is still a wide gap between practical replacement policies and the MIN algorithm for modern set-associative caches [40, 37].

Working set studies reveal that the key to the success of the MIN algorithm is its perfect future knowledge, which is employed to direct its eviction decisions. The oracle information employed by the MIN algorithm is *reuse distance* of memory references. It is impossible for a practical policy to gain perfect oracle information about the future, but it has been proven that prediction of reference locality is possible. Researchers proposed numerous techniques to provide such information, and one of these is *last touch prediction* [41, 29, 30, 40]. The predicted last touch (LT) blocks are labeled so that they can be evicted earlier than by LRU eviction, and other LT blocks can be retained in the cache longer so that they can cause more cache hits because they are expected to have nearer future reuse than those evicted

LT blocks.

This dissertation proposes an LT predictor called the *reuse distance last touch predictor* (RD-LTP). RD-LTP is able to capture reuse distance information which represents the intrinsic memory reference pattern. Based on this improved LT predictor, an MRU LT eviction policy selects the right victim in the presence of incorrect LT predictions. In addition to the LT predictor, another predictor, called the *reuse distance predictor* (RDP) is able developed to predict actual reuse distance values. Compared to various existing cache management techniques, these two novel predictors deliver higher cache performance with higher prediction coverage and accuracy at moderate hardware cost.

7.2 Related Work

There are two typical last touch predictors (LTPs): execution signature history based predictor [29, 30, 40], and block life time history based predictor [41, 31]. The execution signature based LT predictors extract information from the execution context such as instruction trace, while block life time based LT predictors look at the cache block life time history. Both types of LT predictors combine the collected information and block address to build an index which accesses the prediction table to predict and update the block status.

To illustrate how these LTPs work, Figure 7.1 shows a sample memory reference trace with execution context and memory information: reference block address, PC, last touch status, and reuse distance. Most LTPs are based on block address, as shown in the "Address" row in Figure 7.1. The difference among LTPs is what other information associated with block address are exploited. For example, execution signature based LTP examines the PC values of instructions accessing a particular block address. Preceding PCs accessing block address A: PC_1, PC_3 and PC_1 are concatenated and then truncated appropriately to form a signature for the fourth instance of reference to block address A. This technique is shown to be more effective for L1 cache than for L2 cache with large capacity and high associativity. A better association with last touch prediction is proposed based on the last touch history of a particular block which can be encoded as "1" for a previous last touch and "0" for a previous reference which is not last touch, as shown in the "Last Touch" row in Figure 7.1. The prediction signature for block A is then constructed by concatenating these history bits of A. A deep last touch history is often required to provide good prediction accuracy [40].

Life time based LTPs look at different history information associated with a particular block. The cache life time is quantified in different ways: memory reference count is used in [41], and cycle count is used in [31]. The block life time quantification is used differently in these techniques. In [41], the number of references to the target block is defined as *life time*, while the number of any references between two neighboring references to the target block is defined as *access time*. Expiration of life or access time results in prediction of last touch for the target block. In Cache Decay [31] and Adaptive Mode Control [89], the expiration of life times indicates a dead block which can help to reduce power in L2 caches. Cache block liveness can also be detected in runtime system execution [90] to save power

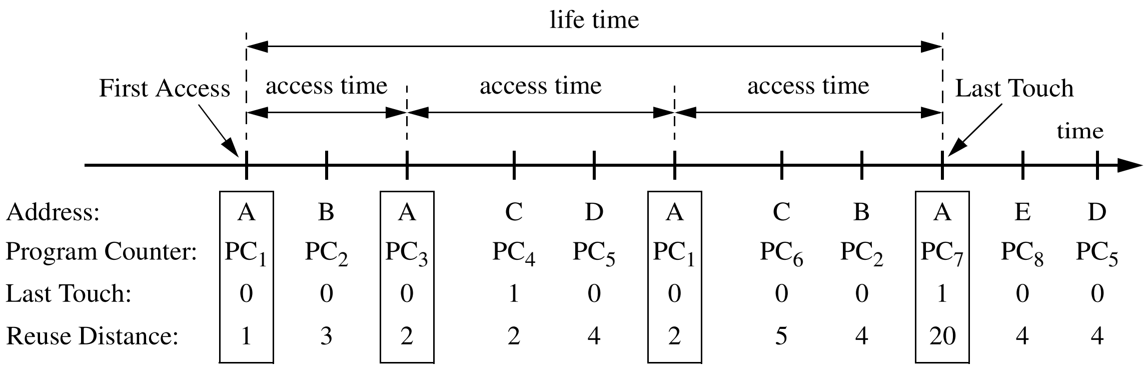


Figure 7.1: A memory address trace, and various information associated with the trace used to predict last touch and/or reuse distance. Information includes memory addresses, memory reference program counters, last touch history, and reuse distance. Access and live times are indicated for a sequence of references to the memory address A.

by turning off the blocks predicted to be dead. In addition to hardware solutions, compiler techniques [44] are proposed to identify possible last touch references by examining program memory behavior and providing hints to cache control logic.

The signature in Inter-Reference Gap [91] is based on the reuse history of the target block. The reuse information is quantified by counting the number of references between two neighboring references to the target block which is *reuse length.* This information is similar to the reuse information exploited by the technique proposed in this dissertation. In this dissertation, the reuse information is measured as *reuse distance.*

Generally speaking, reuse distance based predictors are also signature based. However, they exhibit some important differences compared to existing techniques. The first difference is that the novel LTP proposed in this dissertation constructs a new kind of signature using reuse distances instead of instruction traces, last touch history, or reuse lengths. The reuse distance will be shown to be a more effective way of capturing a program's memory access patterns. The second difference is the way the new LTP treats signature components. The reuse distance information from preceding memory references accessing different memory locations is collected from the same set, whereas other signatures only contain information from the same memory location regardless of its memory reference context. The third difference has to do with additional information provided by the new prediction mechanism. In addition to a binary last touch outcome, a similar predictor can provide a numeric prediction of the block's future reuse distance which is shown to be able to help differentiate OPT last touches and LNO [40] (i.e., the evictions that are last touches under LRU but not under OPT). Finally, similar to some existing techniques, the new LTP uses shadow tags to save history information beyond the cache associativity. However, shadow tags are used to keep accurate reuse information to form high quality signatures, while Puzak [14] uses shadow tags to save blocks for potential near reuse. Therefore, the shadow tags in this technique keep block tag information only rather than the data blocks themselves.

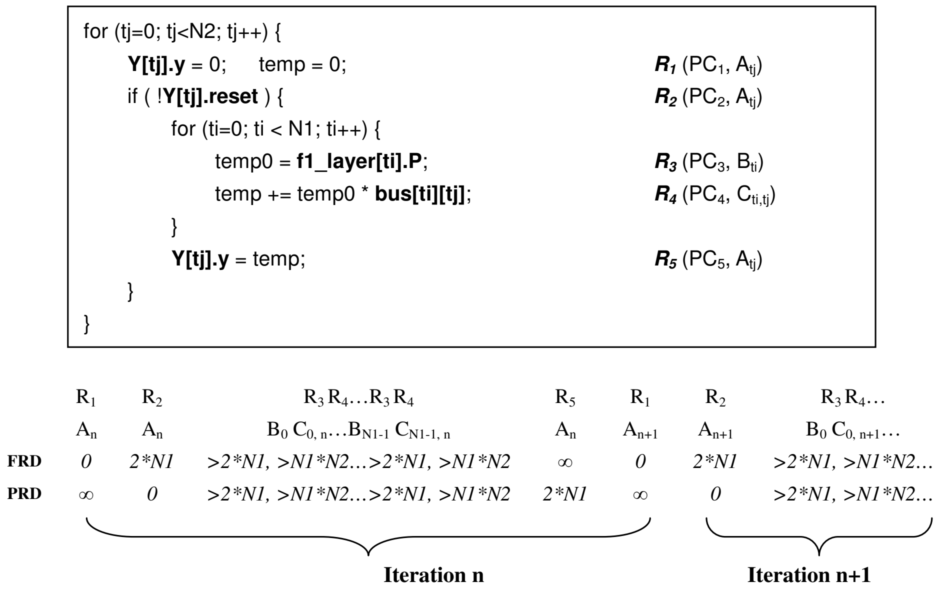
7.3 Predicting Last Touch References

This section presents new predictors. The reuse distance history is introduced first, followed by a motivating example to show its effectiveness. Then, this section describes how RD-LTP constructs signatures and makes predictions. LNO last touch is also discussed followed by an introduction to RDPs.

7.3.1 Global Reuse Distance History

Reuse distance based predictors predict reference locality with *reuse distance history*, which is a sequence of reuse distances of consecutive memory references per set. An example of reuse distance history is shown in "Reuse Distance" row of Figure 7.1. Each reference is labeled with its reuse distance which is defined as the number of distinct memory blocks referenced from the last reference to this block and the current reference. Note that this defines *previous reuse distance* (PRD) of a memory reference while the predictor tries to predict its *future reuse distance* (FRD) which is defined as the number of distinct memory blocks referenced from the current reference and the next reference to the same block. For example, the reuse distance history for the last reference to A is composed of the PRDs of A's preceding memory references to C and B. These references' PRD are 2 and 3 respectively, therefore, A's reuse distance history is {2, 3} if history of depth 2 is considered.

With the definition of reuse distance history, it is easy to compute previous reuse distance information in hardware. In a cache implementation supporting an LRU stack, the previous reuse distance is the location of a block in the stack on a cache hit. For a cache miss, the exact reuse distance is unknown because the block is evicted from the cache in the past; however, its previous reuse distance is guaranteed to be at least equal to the associativity. For the purpose of encoding PRD, all references with reuse distance equal to or larger than the cache associativity



```
for (tj=0; tj<N2; tj++) {
    Y[tj].y = 0;    temp = 0;                    R1 (PC1, Atj)
    if ( !Y[tj].reset ) {                        R2 (PC2, Atj)
        for (ti=0; ti < N1; ti++) {
            temp0 = f1_layer[ti].P;              R3 (PC3, Bti)
            temp += temp0 * bus[ti][tj];         R4 (PC4, Cti,tj)
        }
        Y[tj].y = temp;                          R5 (PC5, Atj)
    }
}
```

	R_1	R_2	$R_3 R_4 \ldots R_3 R_4$	R_5	R_1	R_2	$R_3 R_4 \ldots$
	A_n	A_n	$B_0\ C_{0,n} \ldots B_{N1-1}\ C_{N1-1,n}$	A_n	A_{n+1}	A_{n+1}	$B_0\ C_{0,n+1} \ldots$
FRD	*0*	*2*N1*	*>2*N1, >N1*N2...>2*N1, >N1*N2*	∞	*0*	*2*N1*	*>2*N1, >N1*N2...*
PRD	∞	*0*	*>2*N1, >N1*N2...>2*N1, >N1*N2*	*2*N1*	∞	*0*	*>2*N1, >N1*N2...*

Figure 7.2: Memory reference pattern from the ART benchmark.

is encoded as the cache associativity.

7.3.2 Motivating Example

Having defined global reuse distance history, let us study why it can be effective for reuse distance prediction. A motivating example, a frequently executed loop nest, is taken from *art*, a memory-intensive SPEC2000 benchmark. Figure 7.2 shows the loop nest code. Within the outer loop of the code, there are five memory references, R_1 to R_5, performed by five different instructions, PC_1 to PC_5. The memory reference trace for two consecutive iterations of the outer loop is shown under the code, along with each reference's *future* reuse distance (FRD) and PRD. The FRD and PRD values are computed assuming each element of the Y array is smaller than a cache block so that R_1 can access the same cache block as R_2 in the same iteration of the outer loop, as well as R_5 if $Y[tj].reset$ is false.

From the sequence of PRD values in Figure 7.2, a few reuse distance history patterns can be identified. In particular, assuming a history length of 3, possible patterns include $[\infty, 0, > 2*N1]$ for R_4, $[0, > 2*N1, > N1*N2]$ for R_3, $[> 2*N1, > N1*N2, > 2*N1]$ for R_4, and $[> N1*N2, 2*N1, \infty]$ for R_2. As described in Section 7.3.1, these pattern histories–along with the associated referencing PCs–can be used to predict memory references' FRD values. Take R_2 for example. The FRD of the first R_2 instance is 2*N1 because the if condition is met for iteration n. This information can be captured by a predictor and used to predict the following iteration, where the reuse distance history is the same for the second R_2 instance. Notice, however, reuse distance history alone is not enough to distinguish certain cases. For example, the reuse distance history for the last instance of R_3 is identical to the reuse distance history for R_5. In such cases, we must augment the ambiguous reuse distance histories with memory references' PCs to distinguish between them. At the same time, PC alone (without reuse distance history) clearly cannot provide good predictability either. For example, the FRD of R_2 depends on the outcome of the if statement. Predicting solely on the PC value, PC_2, cannot disambiguate between the two possible if statement outcomes, whereas the reuse distance history can provide the context for performing such disambiguation. This is why we combine reuse distance history with instruction PCs.

Not only does Figure 7.2 show how an RD-based predictor works, it also illustrates why it can be effective. Despite executing a large number of iterations (N1 is typically very big), there are a relatively small number of reuse distance history patterns that arise in the `art` code. This is because the code contains only a few memory references, and exhibits fairly simple control flow. Hence, a small predictor table can capture all of the important patterns. In contrast, signatures that track individual cache blocks, like those used in LvP and AIP, can generate significantly more patterns due to the enormous number of cache blocks referenced by the code. For per-block signatures, much larger predictor tables are required to store the prediction state. The compact prediction state associated with RD-based predictors also facilitate very fast training. For example, after only a single outer-loop iteration, the reuse distance history for R_2 can be captured and used to make predictions for the second instance of R_2 in the following iteration. However, for per-block predictors, enough accesses to each cache block must occur to generate sufficient history to train a predictor. For example, in the *art* code, it is difficult to predict for cache block A_n since there are only 3 references (at most) to the block each outer-loop iteration.

Although our analysis of reuse distance prediction does not take cache organization into consideration (essentially, we assume a fully associative cache), we find the same behavior for reuse distance history patterns occurs in individual sets of set-associative caches as well. Hence, our motivating example also illustrates why global reuse distance history can be effective for reuse distance prediction in a set-associative cache.

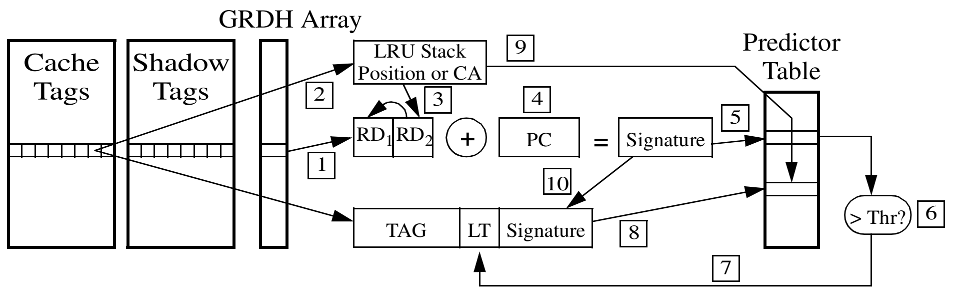


Figure 7.3: Reuse distance last touch predictor organization and actions. Steps 1–7 perform a prediction. Steps 8–10 update the predictor.

7.3.3 Predictor Hardware

The reuse distance information used by reuse distance based predictors is captured by the hardware shown in Figure 7.3. Compared to a conventional cache, the additional hardware structures are as follows: a *global reuse distance history array* (GRDH array) to save global reuse history per set, a *last touch (LT) bit* per block to label last touch predictions, a *signature field* per block to allow the signature to be computed, a *shadow tag array* [14] per set to save tag and signature information for evicted blocks, and finally, a central *predictor table* to save prediction for each signature. In particular, combined with the shadow tag array, the normal cache tags form a unified LRU stack with a depth larger than the cache associativity.

This hardware organization performs two steps: predictor table based prediction and table updating. The prediction process starts from step labeled "1" in Figure 7.3. Upon a cache access, contents of the GRDH array for that set are read (RD_1 and RD_2). GRDH always keeps track of the last two preceding cache accesses' PRD information (label "2" and label "3"). The reading from GRDH is XORed with the PC of the reference instruction incurring this cache access to construct a signature (label "4") for the cache block hit by this access (either in tags of normal tags or the shadow tags). If there is no hit in the cache (either in shadow tags or normal tags), this signature is discarded. This signature is used as an index to access the predictor table (label "5"). The corresponding entry in the predictor table is a saturating counter which gives a prediction based on a threshold (label "6"). The prediction result is used to label that accessed block's LT bit (label "7").

The updating of the predictor table happens upon each cache access. If the reference hits in the top half of the unified LRU stack, this indicates a reuse distance smaller than the cache associativity. Otherwise the reuse distance is equal to or larger than the cache associativity. If the reference hits in either the normal tags or the shadow tags, this outcome is directed to access the predictor table with the signature saved in the signature field (label "8"), and to update the predictor entry counter (label "9"). The saturating counter is incremented if the reuse distance is equal to or larger than associativity, and decremented otherwise. The blocks evicted from either normal tags or shadow tags have to update the predictor table before the eviction finishes because the signature information would be lost after eviction. Finally, once a new signature is created, it has to be saved in the signature field for the current referenced block (label "10").

From the description of how the hardware works, it is critical to keep track of block location in the unified LRU stack for the sake of true reuse distance. The combination of normal tags and shadow tags provides a complete LRU ordering for blocks resident in cache and blocks evicted in near past. For those newly evicted blocks, only tag and signature information are valuable in generating signature and update predictor table. If the set associativity is CA, it requires $CA - 1$ shadow tags to maintain the true reuse distance of newly evicted blocks. This also allows the hardware to identify the mistakenly evicted blocks due to incorrect predictions and avoid global reuse history corruption.

7.3.4 LNO vs OPT Last Touches

The OPT algorithm sets up a theoretical upper limit for cache performance which achieves more cache hits than any other policy. The difference between LRU and OPT is studied in [40] which observes that some LRU last touches are not last touches under OPT. These LRU last touches are defined as LRU non-OPT (LNO). LNO last touches usually are the references with reuse distance slightly larger than the cache associativity. Thus they can be converted into cache hits by OPT because OPT keeps them in the cache longer than LRU by evicting other LRU last touches which are less likely to convert to cache hits. These LRU last touches evicted by OPT are called OPT last touches, and they usually have longer reuse distance than the LNO last touches when they are present in the same set simultaneously.

Compared to other LTP techniques, this dissertation proposes two different methods to exploit the distinction between LNO and OPT last touches. In RD-LTP, a simple rule is employed to decide which last touch block should be evicted early when there are multiple blocks predicted to be last touches: the most-recently-used predicted last touch block should be evicted. The performance achieved by this rule will be presented later in this chapter. Since OPT last touches usually have larger reuse distance than LNO last touches, a predictor with reuse distance prediction can more accurately distinguish between LNO and OPT last touches. Since both OPT and LNO last touches have reuse distances larger than the cache associativity, it requires a deeper stack than set associativity to keep track of their reuse distance information. For this purpose, the shadow tags used in RD-LTPs can also be employed to track reuse distance beyond associativity but below $CA + SA$.

This dissertation proposes reuse distance predictors (RDPs) to predict exact reuse distance values for last touch blocks. The prediction mechanism is similar to RD-LTPs with a minor difference that the predictor table saves the reuse distance value information instead of saturating counters for binary output. Since the unified stack depth is $CA + SA$, this hardware can only track reuse distance from 0 up to $CA + SA$, while larger reuse distance than $CA + SA$ is encoded as $CA + SA$. Therefore, this RDP cannot predict reuse distances beyond $CA + SA$. However, as shown in the later sections, this limited ability of reuse distance still can provide further cache performance gain.

7.4 Experimental Methodology

The evaluation in this dissertation considers the 24 SPEC CPU2000 benchmarks shown in Table 7.1. These benchmarks are simulated with the reference input set by using their pre-compiled Alpha binaries provided with the SimpleScalar tools [87] which have been built using the highest level of compiler optimization.[1] Each benchmark is first fast-forwarded to its representative region (the columns labeled "Skip Ins" in Table 7.1 report the number of fast forwarded instructions) according to SimPoint [88] by consulting the SimPoint website.[2]. Then, their traces of L2 references with reference address and accessing instruction PC information are collected by simulating 2 billion dynamic instructions on the M5 simulator [86]. The architectural parameters in M5 simulator is reported in Table 3.4.

There are two groups of benchmarks in the experiment: "High Potential" benchmarks and "Low Potential" benchmarks. The simulation on a 1MB L2 cache shows that high potential benchmarks exhibit more than 10% difference in the miss rates of LRU and OPT, whereas low potential benchmarks see less than 10% difference. Since the verification shows that most techniques show little performance improvement for low potential benchmarks, the rest of this chapter is focused on high potential benchmarks.

To evaluate the techniques proposed against other techniques, some recent representative techniques are also simulated, such as the AIP [41], LvP [41], and DIP [39] techniques described in Section 7.2. AIP and LvP represent the state-of-the-art for LTP-driven cache management, shown to outperform some earlier LTPs [41]. DIP is a more recent cache insertion policy which also shows performance improvement for single program L2 cache. DIP is evaluated with the simulator

[1]The binaries we used are available at http://www.simplescalar.com/benchmarks.html.

[2]Simulation regions for the Alpha binaries we use are published at http://www-cse.ucsd.edu/~calder/simpoint/multiple-standard-simpoints.htm.

High Potential				Low Potential			
App	Skip Ins	MPKI	Type	App	Skip Ins	MPKI	Type
ammp	4.75B	3.27	FP	perlbmk	1.7B	0.01	Int
art	2.0B	100.70	FP	eon	7.8B	0.00	Int
bzip2	1.8B	1.07	Int	gzip	4.2B	0.15	Int
facerec	69.3B	3.00	FP	gap	8.3B	0.98	Int
galgel	14B	1.41	FP	apsi	2.3B	2.15	FP
gcc	2.1B	3.73	Int	fma3d	2.6B	0.00	FP
mcf	14.75B	70.04	Int	equake	4.8B	13.58	FP
mesa	2.1B	0.08	FP	lucas	1.5B	9.84	FP
parser	13.1B	1.26	Int	swim	5.7B	17.56	FP
sixtrack	3.8B	0.12	FP	applu	1.5B	14.30	FP
twolf	2.0B	3.01	Int				
vortex	2.5B	0.49	Int				
vpr	7.6B	4.77	Int				
wupwise	3.4B	2.05	FP				
AVG		13.93		AVG		5.33	

Table 7.1: SPEC CPU2000 benchmarks used to drive our cache simulations (B = Billion).

Cache Parameters			
L1 I-cache		16 Kbyte, 2-way set associative, 64 byte blocks	
L1 D-cache		16 Kbyte, 2-way set associative, 64 byte blocks	
L2 U-cache		1 Mbyte, 8-way set associative, 64 byte blocks	
Predictor Parameters			
History Length	2	RD-LTP Shadow Tags	7 per cache set, 7 bits each
Reuse Distance Values	3 bits	RDP Shadow Tags	8 per cache set, 7 bits each
Predictor Table	1024 entries	RD-LTP Table Entries	2 bits
Signature Size	10 bits	RDP Table Entries	4 bits

Table 7.2: Cache and predictor parameter settings.

provided on the authors' web site.[3]

The cost of techniques proposed in this dissertation is analyzed in terms of area, power, and cycle time. Based on the cache and predictor configuration parameters in Table 7.2, RD-LTP and RDP incur 53.5 and 64 Kbyte of additional storage, respectively. Table 7.3 shows the cost for each component of the additional storage. Given the 1 Mbyte L2 cache, the hardware estimation adds at most a 6% area overhead for both predictors to the conventional cache implementation. As for power consumption, the additional storage is expected to increase the L2 power in proportion to its area overhead. Since the policy operation is only active upon each L2 cache access, and L2 cache access frequency is normally low, the overall CPU power impact should be below 6%. RD-LTP and RDP have little impact on

[3]The DIP simulator is available at http://users.ece.utexas.edu/~qk/dip/.

GRDH array (2 * 3-bit/set * 2048 sets)	1.5 kB
data blocks signature (10-bit/block * 8-block/set * 2048 sets)	20 kB
RD-LTP LT prediction bit (1-bit/block * 8-block/set * 2048 sets)	2 kB
RD-LTP predictor table (1024 * 2-bit)	0.25 kB
RD-LTP shadow tags ((10 + 7) bit * 7 entry/set * 2048 sets)	29.75 kB
RDP RD prediction (4-bit/block * 8-block/set * 2048 sets)	8 kB
RDP predictor table (1024 * 4-bit)	0.5 kB
RDP shadow tags ((10 + 7) bit * 8 entry/set * 2048 sets)	34 kB
RD-LTP total cost	53.5 kB
RDP total cost	64 kB

Table 7.3: Storage Cost of RD-LTP and RDP for 1MB cache.

L2 cache access time because the latency (which is estimated to be at most 3 or 4 cycles per table access) is off the CPU's critical path as the predictor operations can be performed *after* the cache data is returned to the CPU. There should be ample time to completely hide the predictor's latency before the next L2 reference, which is normally more than 10 cycles away.

In comparison, Kharbutli and Solihin report 61 and 57 Kbyte of additional storage for AIP and LvP, respectively, assuming a 512 Kbyte L2 cache [41]. For a 1 Mbyte L2 cache, this overhead increases to 82 and 73 Kbytes. Their hardware overhead is very similar to RD-LTP and RDP. However, the predictor table evaluated in Kharbutli and Solihin's previous study achieved poor performance for several of our benchmarks. Instead, infinite predictor tables are simulated for LvP and AIP

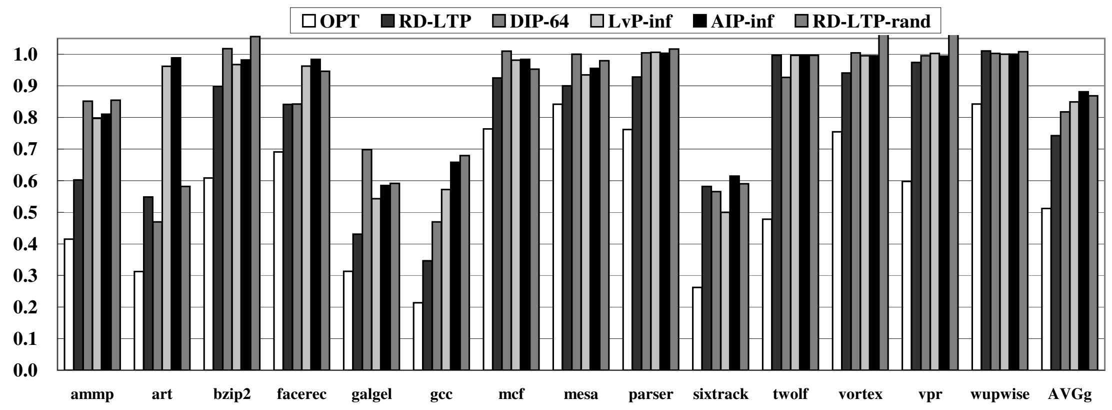


Figure 7.4: Cache miss rates for the high potential benchmarks achieved by OPT, RD-LTP, DIP, LvP, AIP, and RD-LTP-Rand. All of the miss rates are normalized to the LRU miss rate.

(With infinite tables, simulation performance of LvP and AIP is similar to what is reported in [41]). DIP [39] incurs negligible hardware overhead compared to last touch predictors.

7.5 Last Touch Prediction Results

RD-LTP is evaluated in this section and compared with other similar techniques: DIP, LvP, AIP, and victim cache. High potential benchmarks are evaluated for these techniques.

7.5.1 LTP and Insertion Policy Evaluation

Figure 7.4 reports the performance of various LTPs and compares them to OPT and the DIP insertion policy. The figure reports miss rates, normalized to LRU policy, and "AVGg" reports the geometric mean average.

A significant gap of about 50% between OPT and LRU reveals opportunities for potential improvement over LRU. This gap is closed by RD-LTP by 25.8% on average, more than other realistic policies. In particular, RD-LTP reaches miss rate reduction of 12.6% and 15.8% compared to LvP and AIP, respectively. Out of the 14 benchmarks, RD-LTP outperforms both LvP and AIP in 11 benchmarks, outperforms AIP alone in 1 benchmark, and matches the performance of LvP and AIP in 1 benchmark. The miss rate gap between LvP/AIP and OPT is reduced by 30% on average. In addition to performance gains over other LTPs, RD-LTP also brings miss rate 9.3% below DIP on average, and outperforms DIP in 9 out of 14 benchmarks, while matching DIP in 1 benchmark.

Compared to LvP and AIP, RD-LTP reaches higher prediction coverage, which makes the MRU last touch eviction more effective in selecting the right victim. The prediction results will be discussed in greater detail later in this section. Compared to DIP, RD-LTP can address a wide range of memory use patterns while DIP is effective in handling only certain access pattern such as circular access patter. Insertion policies work well for working sets slightly larger than the cache capacity by avoid thrashing. By locating incoming cache blocks at LRU position rather than the MRU position, DIP retains a portion of the working set and increases reuse on that portion. However, LRU insertion adapts poorly for other memory use patterns; hence, DIP reverts back to MRU insertion. In contrast, RD-LTP can outperform DIP by identifying dead blocks and performing early eviction. The little hardware cost required by DIP makes it quite a competitive solution. Nonetheless, RD-LTP still provides higher performance.

7.5.2 Cache Organization Evaluation

As discussed previously, RD-LTP and RDP demand hardware cost at about 60 Kbyte storage. This hardware cost can also be invested in building cache with larger capacity, so it takes a comparison between RD-LTP and other cache larger organization to justify the investment on RD-LTP. In this evaluation, RD-LTP is compared with a normal cache with higher associativity and victim caches as following: a 9-way set associative LRU cache with 128 Kbyte added to the baseline 1 MB cache; a 1 Mbyte LRU cache with a 64 Kbyte victim cache [92](1024 cache blocks, random replacement policy). Figure 7.5 compares the performance of these techniques against RD-LTP and OPT assuming the baseline 1 MB cache. Out of the 14 high potential benchmarks, RD-LTP outperforms the 9-way cache in 8 benchmarks, and outperforms the victim cache in 7 benchmarks. For the remaining benchmarks, RD-LTP does not perform as well as the 9-way cache or the victim cache because the majority of the benchmarks' working sets can fit in the increased cache capacity. In particular, the 9-way cache reduces miss rates in *galgel* and *sixtrack* even more than OPT. And for *gcc* the victim cache reduces miss rate by 97%, while OPT can only reach an 88% reduction. The victim cache also performs best in *ammp* and *sixtrack*, which helps the victim cache to reach similar cache performance as OPT on average.

Figure 7.5 shows that the extra cache capacity can give the 9-way cache and victim cache a considerable performance edge for a few benchmarks whose working sets are just slightly larger than 1 MB. Without these special cases are eliminated,

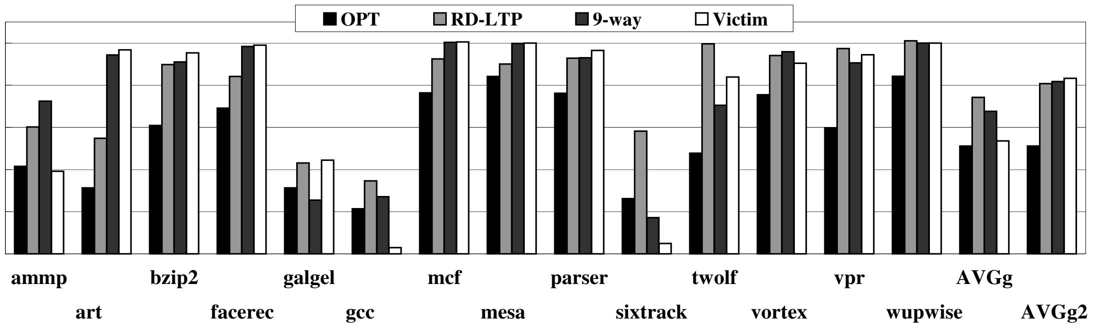


Figure 7.5: Cache miss rates for the high potential benchmarks achieved by OPT, RD-LTP, 9-way cache, and victim cache. All of the miss rates are normalized to the LRU miss rate.

RD-LTP becomes a more favorable technique, as shown by another average miss rate, labeled AVGg2 in Figure 7.5 (excluding *gcc* and *sixtrack*). For the remaining 12 benchmarks, RD-LTP outperforms both 9-way and victim cache on average. Based on these results, we conclude that when benchmarks' working sets are very close to the cache size, using the extra hardware to increase cache capacity performs better than implementing an RD-LTP. However, for the more general case that benchmarks' working sets are noticeably larger than the baseline cache capacity, the investment of additional hardware in an RD-LTP reaches better cache performance.

By increasing baseline cache capacities from 2 MB to 8 MB while keeping the associativity at 8-way(except for the 9-way cache), Figure 7.6 shows the performance of LRU, 9-way, victim cache, LvP, AIP, DIP and RD-LTP. RD-LTP outperforms or matches other techniques up to 8MB as more working sets fit into the larger caches.

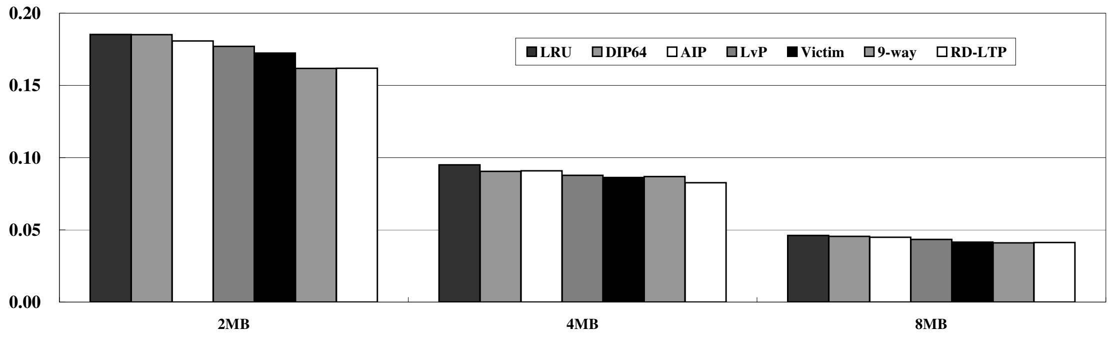


Figure 7.6: Average cache miss rates for different cache sizes achieved by LRU, 9-way, victim cache, LvP, AIP, DIP, and RD-LTP. All of the miss rates are normalized to the LRU miss rate for 1 MB cache.

7.5.3 Prediction Rate

As mentioned early in this section, RD-LTP achieves its performance gains based on better prediction. Figure 7.7 summarizes the prediction quality of RD-LTP, LvP and AIP. The last touch prediction outcomes in Figure 7.7 has 3 parts: correct last touch prediction (labeled "Correct Prediction"); last touches not predicted (labeled "Not Predicted"); and non last touches incorrectly predicted to be last touch (labeled "Wrong Prediction"). All bars are normalized to the total number of true LRU last touch references in each benchmark, with the last group of bars reporting the average across the 14 benchmarks.

As Figure 7.7 shows, RD-LTP is able to correctly predict more last touches than either LvP or AIP. On average, RD-LTP correctly identifies 71.2% of the LRU last touches compared to only 19.2% and 15.6% for LvP and AIP, respectively. The high correct prediction coverage of RD-LTP provides the *potential* to perform a larger number of beneficial early evictions. (How RD-LTP can capitalize on this po-

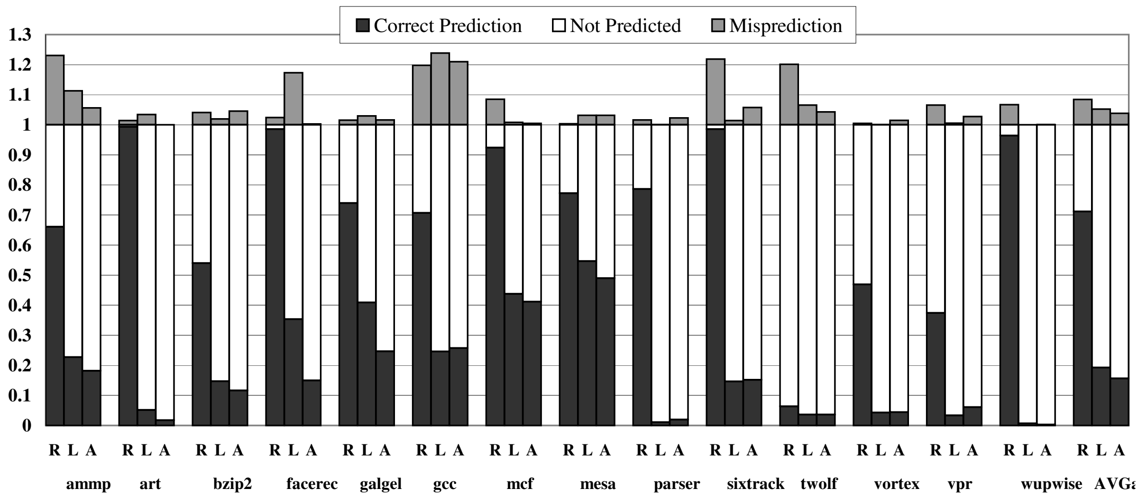


Figure 7.7: Prediction accuracy of RD-LTP, LvP, and AIP.

tential will be discussed later). Unfortunately, the higher prediction rate is achieved at the cost of higher percentage of mispredictions, as shown by the "Wrong Prediction" components in Figure 7.7. RD-LTP incurs 8.5% mispredictions whereas LvP and AIP incur only 5.2% and 3.8%, respectively. Such mispredictions result in premature evictions, converting some LRU cache hits into cache misses. However, the benefit of RD-LTP's higher prediction rate far outweighs the negative impact of its mispredictions.

With high prediction coverage and accuracy, RD-LTP is a more effective last touch predictor than LvP and AIP. There are three factors contributing to RD-LTP. First, last touch events are more predictable when associated with global reuse distance history. Our reuse distance based signatures help to identify more last touches. Second, with the assistance of shadow tags, RD-LTP improves predictor training. As discussed in the last section, without shadow tags, once the cache management hardware begins acting on predictions and performing early evictions, the LRU last touch outcomes of blocks that leave the cache early cannot be captured. Shadow tags allow us to continue tracking recently evicted blocks, thus permitting continued observation of LRU last touches even when replacement deviates from the normal LRU order. Finally, RD-LTP's higher accuracy makes it possible to apply more aggressive eviction. RD-LTP provides last touch prediction for *every* memory reference, while LvP and AIP avoid predicting those memory references with low accuracy (both predictors employ confidence mechanisms). The smaller coverage of predicted memory references in LvP and AIP further weakens their ability to make correct predictions.

7.5.4 Victim Selection

Next, let us look at how the victim selection among multiple predicted last touches impacts cache performance. RD-LTP is shown to be able to predict *multiple* cache blocks as last touches. Table 7.4 reports how often this multiple prediction happens. The column labeled "RDMrk" reports the average number of marked blocks available for an eviction is 4.7, while the column labeled "RD$\geq$2" reports for 77.9% evictions, multiple blocks are predicted as last touches. So, most of the time, RD-LTP has to choose between multiple blocks predicted as last touch for eviction. This *victim selection problem* becomes a design issue only because RD-LTP predicts a large number of LRU last touches, as discussed in Section 7.5.3. In [41], Victim selection is not an issue for LvP/AIP. The same statistics for the LvP technique (the results for AIP are similar) is shown in the columns labeled "LPMrk" and "LP$\geq$2".

	RDMrk	RD≥2	LPMrk	LP≥2	CRD
ammp	4.3	89.8	0.6	12.1	12
art	7.6	100.0	0.07	1.6	12
bzip2	3.3	82.0	0.3	3.8	14
facerec	7.4	99.8	1.5	31.8	20
galgel	4.3	83.4	1.6	45.7	9
gcc	5.0	92.1	0.9	1.6	9
mcf	7.3	100.0	0.5	4.4	30
mesa	6.2	86.5	1.1	28.1	44
parser	5.2	91.5	0.07	0.9	14
sixtrack	0.9	21.8	1.1	19.2	9
twolf	0.3	3.7	0.1	0.7	10
vortex	3.4	72.1	0.2	2.4	20
vpr	2.4	68.5	0.09	0.4	16
wupwise	7.6	100.0	0.03	0.6	44
AVG	4.7	77.9	0.6	11.0	18.8

Table 7.4: Number of marked blocks and percentage of evictions with at least 2 marked blocks for RD-LTP and LvP. The last column reports CRD for each benchmark.

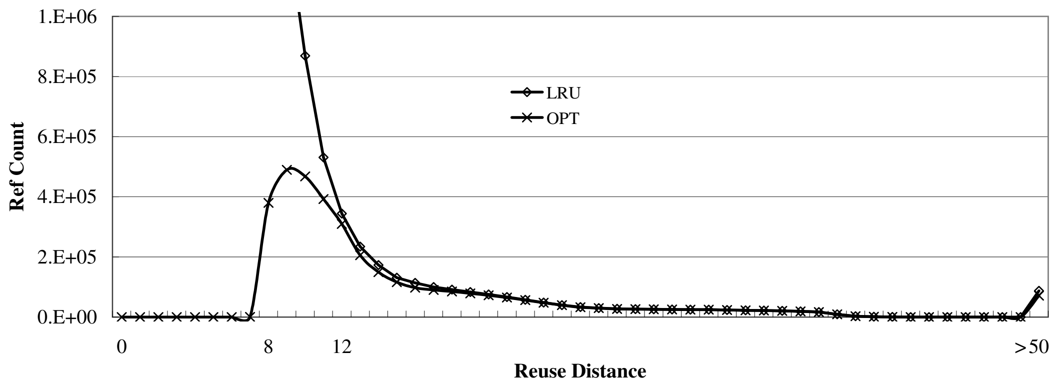


Figure 7.8: Last touch reference histograms under LRU and OPT cache management for the AMMP benchmark. CRD = 12.

LvP marks only 0.6 blocks on average during each eviction, and encounters 2 or more marked blocks in only 11% of the evictions. Most of the time, there are either none or one marked block because of LvP's low prediction rate.

RD-LTP's identification of LRU last touches can not be very beneficial without an intelligent algorithm to select the appropriate victim, as discussed in Section 7.3.4. The reason is that LNO last touches can be converted into cache hits by retaining them in cache a bit longer. Since OPT evictions are a subset of LRU last touches, blindly evicting last touch predictions does not guarantee high performance.

To provide insight into which LRU last touches are the best victims, it is necessary to examine the evictions made by LRU and OPT. Figure 7.8 shows a histogram of last touch references under the LRU and OPT for the AMMP benchmark. The X-axis plots reuse distance from 0 up to beyond 50. For different reuse distances, the histogram plots counting of last touch references(Y-axis) exhibiting that reuse distance. LRU has more last touches than OPT in Figure 7.8, and the area below LRU and above OPT constitutes the LNO last touches. Most importantly, there

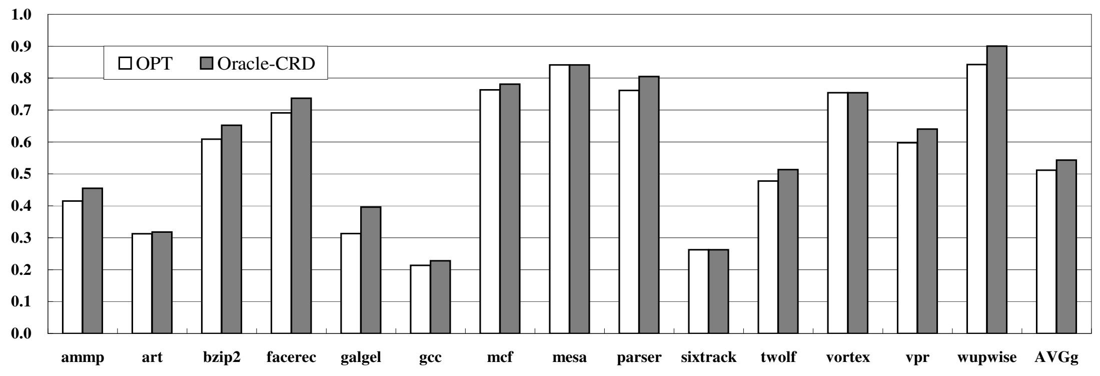


Figure 7.9: Cache misses under the OPT and Oracle-CRD policies.

are very few LNO last touches beyond some reuse distance, where most OPT last touches overlap with LRU last touches. For example, in AMMP, beyond a *critical reuse distance* (CRD) of 12, more than 90% of LRU last touches are also OPT last touches. This makes sense: LRU last touches with larger reuse distances are less likely to become cache hits, and they are also likely to be evicted by OPT. This implies that beyond CRD, *every LRU last touch should be selected as victim as long as it is predicted.* The portion below CRD needs further information to make a reasonable distinction between OPT and LNO last touches. This distribution is similar in all benchmarks with different CRD values, as reported in Table 7.4 as the column labeled "CRD". The average CRD is 18.8.

The benefit of evicting LRU last touches beyond CRD is shown in Figure 7.9. Each benchmark is simulated with OPT and an ideal eviction policy, called Oracle-CRD, which employs a perfect last touch predictor that is able to mark all the LRU last touches correctly with no wrong predictions–i.e., it has 100% coverage and 0% wrong predictions. In addition, Oracle-CRD also has the knowledge of each last touch's reuse distance and whether it is beyond CRD (i.e., the long-reuse). Based on oracle knowledge, Oracle-CRD always evicts the long-reuse blocks upon a miss. If there are only short-reuse last touch blocks available for eviction, it selects one of the short-reuse last touch blocks randomly. As Figure 7.9 shows, Oracle-CRD is worse than OPT by only 3.1%, which demonstrates that the identification of the long-reuse last touch blocks can lead to the optimal performance.

The CRD information is hard to obtain online, but the likely location of long-reuse last touch blocks is found to be close to the MRU marked block. Figure 7.10 profiles the evictions performed by RD-LTP. The components labeled “Long” and “Short” quantify the fraction of evictions involving a long-reuse last touch and short-reuse last touch block, respectively. And the components labeled “Misprediction” indicate the fraction of evictions involving an incorrectly predicted block. RD-LTP always evicts the MRU marked block, while an alternate version, called RD-LTP-Rand, evicts a randomly selected marked block. On average, the MRU policy identifies a long-reuse last touch block 52% of the time, while a random policy identifies a long-reuse block only 48% of the time.

In addition to improved identification of long-reuse blocks, MRU selection can also avoid evicting mispredicted blocks more effectively. Figure 7.10 shows that the random policy evicts about twice as many mispredicted LRU last touches compared to the MRU policy (12.8% versus 6%). With low misprediction rate (see Figure 7.7), the average interval between two mispredictions is fairly large. As a result, when a cache miss occurs, it is rare for a mispredicted block to be the MRU marked block after several other (correctly) predicted LRU last touches have likely occurred since the last misprediction. Therefore, selecting the MRU block helps to reduce

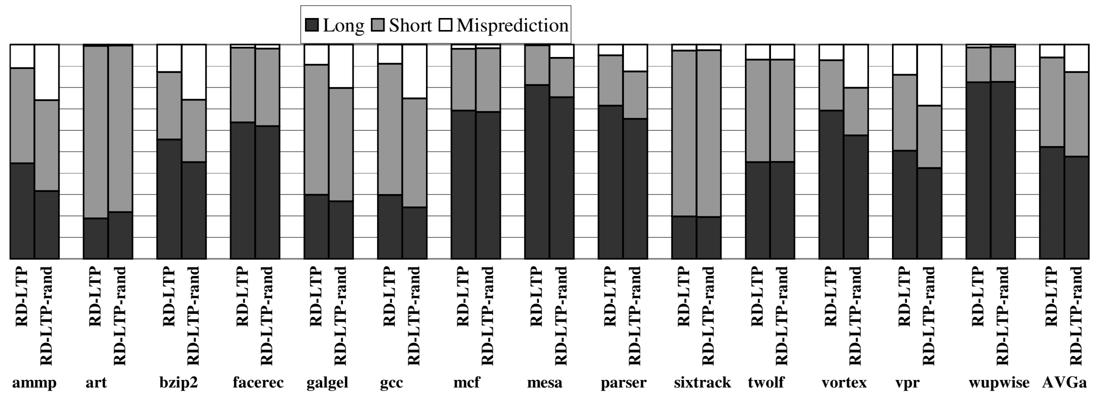


Figure 7.10: Breakdown of the evictions performed by the RD-LTP and RD-LTP-Rand techniques.

the negative consequences of mispredictions.

The advantages of the MRU policy result in performance gains shown in Figure 7.4. The miss rate achieved by RD-LTP-Rand is shown in bars labeled "RD-LTP-Rand", which is outperformed by the MRU policy by 14.5%. These results validate MRU victim selection as a good policy for our RD-LTP technique.

7.6 Reuse Distance Prediction Results

This section presents the performance evaluation of RDP, as shown in Figure 7.11. The bars labeled "RDP" report the miss rates achieved when driving cache management decisions using an RDP across high potential benchmarks. For comparison, the miss rates achieved by RD-LTP and OPT from Figure 7.4 are also shown in Figure 7.11. All bars are normalized to the LRU miss rate for each benchmark, and the group of bars labeled "AVGg" report the geometric mean across all the benchmarks. RDP, compared with RD-LTP, provides an additional 2.7% miss

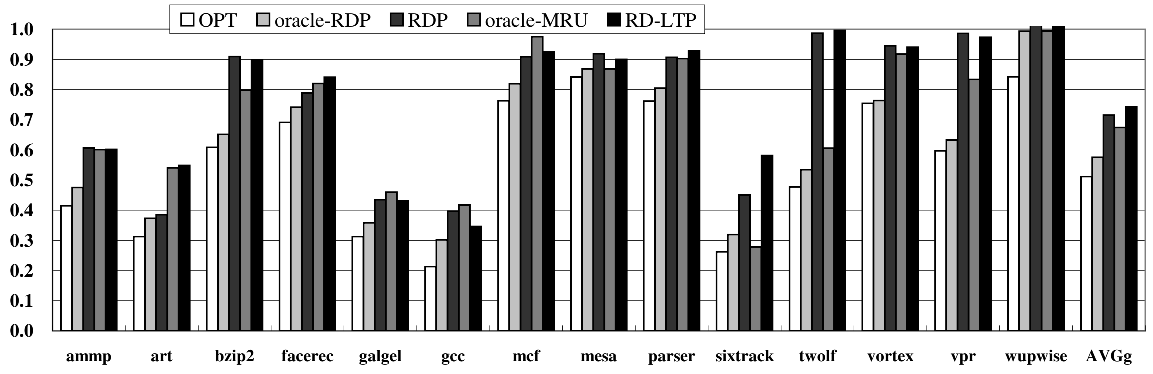


Figure 7.11: Cache miss rates achieved by OPT, Oracle-RDP, RDP, Oracle-MRU, and RD-LTP for the high potential benchmarks.

rate reduction on average, and improves over LvP and AIP by 14.9% and 17.8%, respectively.

The additional benefit achieved by RDPs is due to more information about reuse distance provided by the predictor than the binary last touch prediction from RD-LTP. As discussed in Section 7.5.4, while some predictions identify LRU last touches beyond CRD which are likely to be OPT last touches, many predictions identify LRU last touches below CRD that may possibly be LNO last touches. In the latter case, the exact reuse distance prediction from RDP can better identify good victim block. Hence, when multiple blocks are predicted, the block referenced farthest in the future can be better identified using reuse distance prediction than selecting the MRU marked block.

Another two ideal cache management algorithms, Oracle-MRU and Oracle-RDP are simulated to provide more insight into why reuse distance prediction works better. Oracle-MRU is MRU selection with perfect last touch block identification. [4]

[4]Although Oracle-MRU has perfect last touch information, the MRU policy may still mistakenly

Oracle-RDP is longest reuse distance block selection with perfect reuse distance information. Figure 7.11 shows that Oracle-MRU improves upon RD-LTP by 12.5%. This represents the performance lost by RD-LTP due to predictor inaccuracy (i.e., the "Not Predicted" and "Wrong Prediction" components in Figure 7.7). In addition, Figure 7.11 also shows Oracle-RDP improves upon Oracle-MRU by 13.8%. This performance difference reveals actual potential benefit of exact reuse distance information under perfect condition. Unfortunately, RDP does not fully achieve this potential, as demonstrated by only 2.7% performance gain over RD-LTP. RDP's suffers from the wrong victim selection due to its inaccuracies in predicting the exact reuse distance.

evict LNO last touches over OPT last touches. In fact, Oracle-MRU may suffer if the additional correct last touch predictions expose more LNO last touches for eviction.

Chapter 8

Conclusion

8.1 Summary

This dissertation studies cache management techniques for multi-threaded and single-threaded workloads. For shared cache with multi-threaded workloads, we study how the individual working sets interact with each other in terms of locality and interleaving. The interleaving and its impact on locality are simulated and profiled to provide insight. A number of ideal cache management algorithms are also studied to provide hints on how to improve existing techniques. We propose techniques which approximate the behavior of ideal algorithms while improving cache performance, overall performance, and thread fairness. For a cache with single-threaded programs, the reuse distance pattern is studied to provide a better last-touch prediction solution. Our reuse distance based last-touch predictor improves both prediction coverage and accuracy and achieves better cache performance compared to existing techniques.

This research help us to draw the following conclusions. First, we found that the memory behavior in multi-threaded workloads has two important factors: locality and interleaving. While locality has been studied extensively in the past, interleaving has not. This dissertation shows that interleaving factor provides further opportunity to improve cache performance over existing locality-based techniques. Second, we study the ideal shared cache management algorithm with oracle information, and this ideal management algorithm provides an upper limit for cache performance. The profiling of the Oracle algorithm behavior reveals that it is critical to identify the aggressor thread from the conflicting threads. A practical eviction algorithm can approach the performance of the ideal algorithm by approximating its eviction bias. Third, to optimize overall system performance (weighted IPC), we need to also improve thread fairness, which can be optimized by fine tuning the degree of bias (*pr* ratio) against aggressor threads. Our results show PR policy outperforms LRU, UCP, and ideal cache partitioning by 4.86%, 3.15%, and 1.09%, respectively. Fourth, for the last-touch prediction for single program, prediction coverage and accuracy can be significantly improved using reuse distance information. As a result, the cache performance for single program can be improved at lower hardware cost. Our results show that for an 8-way 1MB L2 cache, a 54KB RD-LTP reduces the cache miss rate by 12.6% and 15.8% compared to LvP and AIP, and by 9.3% compared to DIP. An RDP, which can predict exact reuse distances, improves the miss rate compared to an RD-LTP by an additional 2.7%.

8.2 Future Directions

In this dissertation, we show that probabilistic replacement policy can effectively manage the conflicting working sets and achieve higher utilization of shared cache compared to other practical techniques. We also show the case for optimizing working set by evicting the blocks with bad locality. The principles of interleavingaware locality management and working set optimization provide opportunities to design more efficient shared resource management techniques, and can be extended to other resources shared by conflicting threads.

For example, the PR policy can be extended to optimize the working sets on chip. The PR policy described in this dissertation manages multiple working sets without any working set reduction. As our LTP study shows, the original working set size can be reduced without suffering more cache misses. For a shared cache with limited capacity, it is easier to fit multiple reduced working sets into cache. The reuse distance information can be useful in determining the targets for working set reduction because it provides information about the distinction between good locality and bad locality blocks. However, it is not enough to know the static information of cache blocks. A good management policy also has to know the interleaving of each reference with other references close in time and their locality information to make reasonable decisions as for which bad locality block belonging to which thread is to be evicted, and which good locality block belonging to which thread is to be kept in cache. There are also fairness issues involved in eviction decision making for the purpose of overall system throughput.

Bibliography

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann, May 2002.

[2] W. Lin, "Reducing dram latencies with an integrated memory hierarchy design," in *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, (Washington, DC, USA), p. 301, IEEE Computer Society, 2001.

[3] D. M. Tullsen, S. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proceedings of the 22th Annual International Symposium on Computer Architecture*, pp. 392–403, 1995.

[4] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang, "The case for a single-chip multiprocessor," in *ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, (New York, NY, USA), pp. 2–11, ACM Press, 1996.

[5] S. Choi, "Hill-Climbing SMT Processor Resource Distribution." April 2006.

[6] J. Burns and J.-L. Gaudiot, "Area and system clock effects on smt/cmp processors," in *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, (Washington, DC, USA), p. 211, IEEE Computer Society, 2001.

[7] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams," in *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, (Washington, DC, USA), p. 2, IEEE Computer Society, 2004.

[8] S. Srikantaiah, M. Kandemir, and M. J. Irwin, "Adaptive Set Pinning: Managing Shared Caches in Chip Multiprocessors," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, (Seattle, WA), pp. 135–144, March 2008.

[9] L. A. Belady, "A Study of Replacement Algorithms for a Virtual-Storage Computer," *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, 1966.

[10] N. Rafique, W.-T. Lim, and M. Thottethodi, "Architectural support for operating system-driven cmp cache management," in *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, (New York, NY, USA), pp. 2–12, ACM Press, 2006.

[11] M. V. Wilkes, "Slave memories and dynamic storage allocation," pp. 371–372, 2000.

[12] A. J. Smith, "Cache memories," *ACM Comput. Surv.*, vol. 14, no. 3, pp. 473–530, 1982.

[13] M. D. Hill, *Aspects of cache memory and instruction buffer performance.* PhD thesis, 1987. Chairman-Smith,, Alan Jay.

[14] T. R. Puzak, "Analysis of Cache Replacement Algorithms." February 1985.

[15] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner, "One-level storage system," *IRE Transactions EC-11*, pp. 223–235, Apr. 1962.

[16] P. J. Denning, "The working set model for program behavior," *Commun. ACM*, vol. 11, no. 5, pp. 323–333, 1968.

[17] P. J. Denning, "Working sets past and present," *IEEE Transactions on Software Engineering*, vol. SE-6, pp. 64–84, Jan. 1980.

[18] T. Conte and W. m W. Hwu, "Benchmark characterization for experimental system evaluation," in *Hawaii International Conference on System Sciences (HICSS)*, vol. I, pp. 6–18, 1990.

[19] A. Agarwal, J. Hennessy, and M. Horowitz, "An analytical cache model," *ACM Trans. Comput. Syst.*, vol. 7, no. 2, pp. 184–215, 1989.

[20] A. Mendelson, D. Thiebaut, and D. K. Pradhan, "Modeling live and dead lines in cache memory systems," *IEEE Trans. Comput.*, vol. 42, no. 1, pp. 1–14, 1993.

[21] T. C. Mowry and C.-K. Luk, "Predicting data cache misses in non-numeric applications through correlation profiling," in *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, (Washington, DC, USA), pp. 314–320, IEEE Computer Society, 1997.

[22] A. Agarwal and S. D. Pudar, "Column-associative caches: a technique for reducing the miss rate of direct-mapped caches," in *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, (New York, NY, USA), pp. 179–190, ACM Press, 1993.

[23] A. González, C. Aliagas, and M. Valero, "A data cache with multiple caching strategies tuned to different types of locality," in *ICS '95: Proceedings of the 9th international conference on Supercomputing*, (New York, NY, USA), pp. 338–347, ACM, 1995.

[24] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache prefetch buffers," in *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, (New York, NY, USA), pp. 388–397, ACM Press, 1998.

[25] J. Rivers and E. S. Davidson, "Reducing conflicts in direct-mapped caches with a temporality-based design," in *Proceedings of the 1996 ICPP*, pp. 154–163, 1996.

[26] T. L. Johnson, M. C. Merten, and W.-M. W. Hwu, "Run-time spatial locality detection and optimization," in *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, (Washington, DC, USA), pp. 57–64, IEEE Computer Society, 1997.

[27] T. L. Johnson, D. A. Connors, M. C. Merten, and W. mei W. Hwu, "Run-time cache bypassing," *IEEE Trans. Comput.*, vol. 48, no. 12, pp. 1338–1354, 1999.

[28] M. Qureshi, M. Suleman, and Y. Patt, "Line distillation: Increasing cache capacity by filtering unused words in cache lines," in *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, Feb. 2007.

[29] A.-C. Lai and B. Falsafi, "Selective, Accurate, and Timely Self-Invalidation Using Last-Touch Prediction," in *Proceedings of the 27th International Symposium on Computer Architecture*, 2000.

[30] A.-C. Lai, C. Fide, and B. Falsafi, "Dead-Block Prediction and Dead-Block Correlating Prefetchers," in *Proceedings of the 28th International Symposium on Computer Architecture*, 2001.

[31] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power," in *Proceedings of the 28th International Symposium on Computer Architecture*, pp. 240–251, 2001.

[32] N. Megiddo and D. S. Modha, "Arc: A self-tuning, low overhead replacement cache," in *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, (Berkeley, CA, USA), pp. 115–130, USENIX Association, 2003.

[33] S. Bansal and D. S. Modha, "Car: Clock with adaptive replacement," in *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, (Berkeley, CA, USA), pp. 187–200, USENIX Association, 2004.

[34] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "On the existence of a spectrum of policies that subsumes the least recently used (lru) and least frequently used (lfu) policies," in *SIGMETRICS '99: Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, (New York, NY, USA), pp. 134–143, ACM, 1999.

[35] F. Guo and Y. Solihin, "An analytical model for cache replacement policy performance," in *SIGMETRICS '06/Performance '06: Proceedings of the joint international conference on Measurement and modeling of computer systems*, (New York, NY, USA), pp. 228–239, ACM Press, 2006.

[36] J. T. Robinson and M. V. Devarakonda, "Data cache management using frequency-based replacement," in *SIGMETRICS '90: Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, (New York, NY, USA), pp. 134–142, ACM Press, 1990.

[37] W. A. Wong and J.-L. Baer, "Modified LRU Policies for Improving Second-Level Cache Behavior," in *Proceedings of the 6th International Symposium on High-Performance Computer Architecture*, pp. 49–60, 2000.

[38] M. K. Qureshi, D. Thompson, and Y. N. Patt, "The v-way cache: Demand based associativity via global replacement," in *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, (Washington, DC, USA), pp. 544–555, IEEE Computer Society, 2005.

[39] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, (New York, NY, USA), pp. 381–391, ACM Press, 2007.

[40] W.-F. Lin and S. K. Reinhardt, "Predicting Last-Touch References under Optimal Replacement," CSE-TR 447-02, University of Michigan, 2002.

[41] M. Kharbutli and Y. Solihin, "Counter-Based Cache Replacement Algorithms," in *Proceedings of the International Conference on Computer Design*, (San Jose, CA), October 2005.

[42] W. Liu and D. Yeung, "Enhancing LTP-Driven Cache Management Using Reuse Distance Information," UMIACS-TR 2007-33, University of Maryland, 2007.

[43] E. G. Hallnor and S. K. Reinhardt, "A fully associative software-managed cache design," in *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, (New York, NY, USA), pp. 107–116, ACM Press, 2000.

[44] Z. Wang, K. S. McKinley, A. L. Rosenberg, and C. les C. Weems, "Using the compiler to improve cache replacement decisions," in *PACT '02*, p. 199, 2002.

[45] W. Y. Chen, R. A. Bringmann, S. A. Mahlke, R. E. Hank, and J. E. Sicolo, "An efficient architecture for loop based data preloading," in *Proc. IEEE International Symposium on Microarchitecture*, pp. 92–100, 1992.

[46] J. W. C. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," in *MICRO 25: Proceedings of the 25th annual international symposium on Microarchitecture*, (Los Alamitos, CA, USA), pp. 102–110, IEEE Computer Society Press, 1992.

[47] D. Joseph and D. Grunwald, "Prefetching using markov predictors," in *ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture*, (New York, NY, USA), pp. 252–263, ACM, 1997.

[48] J.-L. Baer and T.-F. Chen, "Effective hardware-based data prefetching for high-performance processors," *IEEE Trans. Comput.*, vol. 44, no. 5, pp. 609–623, 1995.

[49] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," in *HPCA '04: Proceedings of the 10th International Symposium on High Performance Computer Architecture*, (Washington, DC, USA), p. 96, IEEE Computer Society, 2004.

[50] A. K. Porterfield, *Software methods for improvement of cache performance on supercomputer applications.* PhD thesis, Houston, TX, USA, 1989. Chairman-Kennedy,, K. W.

[51] D. Kim and D. Yeung, "Design and evaluation of compiler algorithms for pre-execution," in *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, (New York, NY, USA), pp. 159–170, ACM, 2002.

[52] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *HPCA '03: Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, (Washington, DC, USA), p. 129, IEEE Computer Society, 2003.

[53] Y. Chou, B. Fahs, and S. Abraham, "Microarchitecture optimizations for exploiting memory-level parallelism," in *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, (Washington, DC, USA), p. 76, IEEE Computer Society, 2004.

[54] V. S. Pai and S. Adve, "Code transformations to improve memory parallelism," in *MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, (Washington, DC, USA), pp. 147–155, IEEE Computer Society, 1999.

[55] F. Darema-Rogers, G. F. Pfister, and K. So, "Memory access patterns of parallel scientific programs," in *SIGMETRICS '87: Proceedings of the 1987 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, (New York, NY, USA), pp. 46–58, ACM Press, 1987.

[56] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: characterization and methodological considerations," in *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, (New York, NY, USA), pp. 24–36, ACM Press, 1995.

[57] J.-S. Kim, X. Qin, and Y. Hsu, "Memory characterization of a parallel data mining workload," in *WWC '98: Proceedings of the Workload Characterization: Methodology and Case Studies*, (Washington, DC, USA), p. 60, IEEE Computer Society, 1998.

[58] D. J. Sorin, V. S. Pai, S. V. Adve, M. K. Vernon, and D. A. Wood, "Analytic evaluation of shared-memory systems with ilp processors," in *ISCA '98: Proceedings of the 25th annual international symposium on Computer architecture*, (Washington, DC, USA), pp. 380–391, IEEE Computer Society, 1998.

[59] G. E. Suh, S. Devadas, and L. Rudolph, "A new memory monitoring scheme for memory-aware scheduling and partitioning," in *HPCA '02: Proceedings of the 8th International Symposium on High Performance Computer Architecture*, pp. 117–, 2002.

[60] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni, "Communist, Utilitarian, and Capitalist Cache Policies on CMPs: Caches as a Shared Resource," in *Proceedings of the International Symposium on Parallel Architectures and Compilation Techniques*, (Seattle, WA), September 2006.

[61] A. Jaleel, M. Mattina, and B. Jacob, "Last level cache (llc) performance of data mining workloads on a cmp - a case study of parallel bioinformatics workloads," in *HPCA '06: Proceedings of the 12th International Symposium on High Performance Computer Architecture*, (Washington, DC, USA), pp. 88–98, IEEE Computer Society, 2006.

[62] C. Liu, A. Sivasubramaniam, and M. Kandemir, "Organizing the last line of defense before hitting the memory wall for cmps," in *HPCA '04: Proceedings of the 10th International Symposium on High Performance Computer Architecture*, (Washington, DC, USA), p. 176, IEEE Computer Society, 2004.

[63] A. Snavely, D. M. Tullsen, and G. Voelker, "Symbiotic Jobscheduling with Priorities for a Simultaneous Multithreading Processor," in *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, June 2002.

[64] K. Luo, J. Gummaraju, and M. Franklin, "Balancing throughput and fairness in smt processors," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, pp. 164–171, 2001.

[65] S. Kim, D. Chandra, and Y. Solihin, "Fair cache sharing and partitioning in a chip multiprocessor architecture," in *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, (Washington, DC, USA), pp. 111–122, IEEE Computer Society, 2004.

[66] H. S. Stone, J. Turek, and J. L. Wolf, "Optimal partitioning of cache memory," *IEEE Trans. Comput.*, vol. 41, no. 9, pp. 1054–1068, 1992.

[67] G. E. Suh, L. Rudolph, and S. Devadas, "Dynamic partitioning of shared cache memory," *J. Supercomput.*, vol. 28, no. 1, pp. 7–26, 2004.

[68] D. Chiou, P. Jain, S. Devadas, and L. Rudolph, "Dynamic cache partitioning via columnization," in *Proceedings of Design Automation Conference, Los Angeles*, June 2000.

[69] M. K. Qureshi and Y. N. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," in *Proceedings of the International Symposium on Microarchitecture*, 2006.

[70] S. Kim, D. Chandra, and Y. Solihin, "Fair cache sharing and partitioning in a chip multiprocessor architecture," in *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, (Washington, DC, USA), pp. 111–122, IEEE Computer Society, 2004.

[71] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer, "Adaptive insertion policies for managing shared caches," in *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, (New York, NY, USA), pp. 208–219, ACM, 2008.

[72] G. Chen and M. Kandemir, "Optimizing inter-processor data locality on embedded chip multiprocessors," in *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, (New York, NY, USA), pp. 227–236, ACM Press, 2005.

[73] Z. Chishti, M. D. Powell, and T. N. Vijaykumar, "Optimizing replication, communication, and capacity allocation in cmps," in *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, (Washington, DC, USA), pp. 357–368, IEEE Computer Society, 2005.

[74] M. Zhang and K. Asanovic, "Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors," in *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, (Washington, DC, USA), pp. 336–345, IEEE Computer Society, 2005.

[75] J. Chang and G. S. Sohi, "Cooperative caching for chip multiprocessors," in *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, (Washington, DC, USA), pp. 264–276, IEEE Computer Society, 2006.

[76] J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, M. Horowitz, and C. Kozyrakis, "Comparing memory systems for chip multiprocessors," in *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, (New York, NY, USA), pp. 358–368, ACM Press, 2007.

[77] H. Hossain, S. Dwarkadas, and M. C. Huang, "Improving support for locality and fine-grain sharing in chip multiprocessors," in *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, (New York, NY, USA), pp. 155–165, ACM, 2008.

[78] R. Iyer, "Cqos: a framework for enabling qos in shared caches of cmp platforms," in *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, (New York, NY, USA), pp. 257–266, ACM Press, 2004.

[79] K. J. Nesbit, J. Laudon, and J. E. Smith, "Virtual private caches," in *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, (New York, NY, USA), pp. 57–68, ACM Press, 2007.

[80] M. Prvulovic, D. Marinov, Z. Dimitrijevic, and V. Milutinovic, "Split temporal/spatial cache: A survey and reevaluation of performance," *IEEE TCCA Newsletter*, pp. 1–10, 1999.

[81] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Systems Journal*, vol. 9, no. 2, p. 78, 1970.

[82] C. Ding and Y. Zhong, "Predicting whole-program locality through reuse distance analysis," in *Conference on Programming Languages Design and Implementation'03. ACM, 2003. 3.4, 3.4, 4.7*, 2003.

[83] E. Berg and E. Hagersten, "Fast data-locality profiling of native execution," in *SIGMETRICS '05: Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, (New York, NY, USA), pp. 169–180, ACM Press, 2005.

[84] J. D. Collins and D. M. Tullsen, "Hardware identification of cache conflict misses," in *MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, (Washington, DC, USA), pp. 126–135, IEEE Computer Society, 1999.

[85] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting inter-thread cache contention on a chip multi-processor architecture," in *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, (Washington, DC, USA), pp. 340–351, IEEE Computer Society, 2005.

[86] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, “The M5 Simulator: Modeling Networked Systems,” *IEEE Micro*, vol. 26, pp. 52–60, July/August 2006.

[87] D. Burger and T. M. Austin, “The SimpleScalar Tool Set, Version 2.0,” CS TR 1342, University of Wisconsin-Madison, June 1997.

[88] T. Sherwood, E. Perelman, and B. Calder, “Basic block distribution analysis to find periodic behavior and simulation points in applications,” in *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[89] H. Zhou, M. C. Toburen, E. Rotenberg, and T. M. Conte, “Adaptive Mode Control: A Static-Power-Efficient Cache Design,” *ACM Transactions on Embedded Computing Systems*, vol. 2, no. 3, 2002.

[90] G. Chen, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, and M. Wolczko, “Tracking Object Life Cycle for Leakage Energy Optimization,” in *Proceedings of the ISSS/CODES joint conference*, (Newport Beach, CA), October 2003.

[91] V. Phalke and B. Gopinath, “An inter-reference gap model for temporal locality in program behavior,” in *SIGMETRICS ’95/PERFORMANCE ’95: Proceedings of the 1995 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, (New York, NY, USA), pp. 291–300, ACM Press, 1995.

[92] N. P. Jouppi, “Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers,” *SIGARCH Comput. Archit. News*, vol. 18, no. 3a, pp. 364–373, 1990.