Software-assisted Cache Mechanisms for Embedded Systems

by

Prabhat Jain

Bachelor of Engineering in Computer Engineering
Devi Ahilya University, 1986

Master of Technology in Computer and Information Technology
Indian Institute of Technology, 1988

Master of Science in Computer Science
University of Utah, 1992

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2008

Author .....
Department of Electrical Engineering and Computer Science
September 17, 2007

Certified by .....
Srinivas Devadas
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Terry P. Orlando
Chairman, Department Committee on Graduate Students

Software-assisted Cache Mechanisms for Embedded Systems

by

Prabhat Jain

Submitted to the Department of Electrical Engineering and Computer Science
on September 17, 2007, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Embedded systems are increasingly using on-chip caches as part of their on-chip memory system. This thesis presents cache mechanisms to improve cache performance and provide opportunities to improve data availability that can lead to more predictable cache performance.

The first cache mechanism presented is an intelligent cache replacement policy that utilizes information about dead data and data that is very frequently used. This mechanism is analyzed theoretically to show that the number of misses using intelligent cache replacement is guaranteed to be no more than the number of misses using traditional LRU replacement. Hardware and software-assisted mechanisms to implement intelligent cache replacement are presented and evaluated.

The second cache mechanism presented is that of cache partitioning which exploits disjoint access sequences that do not overlap in the memory space. A theoretical result is proven that shows that modifying an access sequence into a concatenation of disjoint access sequences is guaranteed to improve the cache hit rate. Partitioning mechanisms inspired by the concept of disjoint sequences are designed and evaluated.

A profile-based analysis, annotation, and simulation framework has been implemented to evaluate the cache mechanisms. This framework takes a compiled benchmark program and a set of program inputs and evaluates various cache mechanisms to provide a range of possible performance improvement scenarios. The proposed cache mechanisms have been evaluated using this framework by measuring cache miss rates and Instructions Per Clock (IPC) information. The results show that the proposed cache mechanisms show promise in improving cache performance and predictability with a modest increase in silicon area.

Thesis Supervisor: Srinivas Devadas
Title: Professor of Electrical Engineering and Computer Science

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

There has been tremendous growth in the electronics and the computing industry in recent years fueled by the advances in semiconductor technology. Electronic computing systems are used in diverse areas and support a tremendous range of functionality. These systems can be categorized into general-purpose systems or application-specific systems. The application-specific systems are also called *embedded systems*. General-purpose systems (e.g., desktop computers) are designed to support many different application *domains*. On the other hand, most embedded systems are typically designed for a single application domain. Embedded systems are used in consumer electronics products such as cellular phones, personal digital assistants, and multimedia systems. In addition, embedded systems are used in many other fields such as automotive industry, medical sensors and equipment, and business equipment. Embedded systems are customized to an application domain to meet performance, power, and cost requirements. This customization feature makes the embedded systems popular as evidenced by their use in a wide range of fields and applications.

Current submicron semiconductor technology allows for the integration of millions of gates on a single chip. As a result, different components of a general-purpose or an embedded system can be integrated onto a single chip. The integration of components on a single chip offers the potential for improved performance, lower power, and reduced cost. However, the ability to integrate more functionality onto a chip results in increased design complexity. Most embedded systems use both programmable components, Field Programmable Logic, and application-specific integrated circuit (ASIC) logic. The programmable compo-

nents are called *embedded processors*. These embedded processors can be general-purpose microprocessors, off-the-shelf digital signal processors (DSPs), in-house application specific instruction-set processors (ASIPs), or micro-controllers. Typically, the time-critical functionality is implemented as an ASIC and the remaining functionality is implemented in software which runs on the embedded processor. Figure 1-1 illustrates a typical embedded system, consisting of a processor, DSP, or an ASIP, a program ROM or instruction cache, data cache, SRAM, application-specific circuitry (ASIC), peripheral circuitry, and off-chip DRAM memory.

The ASIC logic typically provides (predictably) high performance for the time-critical functionality in an embedded system. But, design cycles for ASIC logic can be long and design errors may require a new chip to correct the problem. On the other hand, software implementations can accommodate late changes in the requirements or design, thus reducing the length of the design cycle. Software programmability can also help in the evolution of a product – simply changing the software may be enough to augment the functionality for the next generation of a product. Due to the need for short design cycles and the flexibility offered by software, it is desirable that an increasing amount of an embedded system's functionality be implemented in software relative to hardware, provided the software can meet the desired level of performance and predictability goals.

The software that runs on the embedded processor is supported by on-chip data memory which may be used as a cache and/or scratch-pad SRAM. The goal of this thesis is to bring the performance and predictability of software closer to that offered by an ASIC in an embedded system. This is achieved by focusing on the on-chip memory component of the embedded system. I will describe various problems with embedded memory systems in the following sections. In this thesis, I target performance and predictability improvement of on-chip cache memory through the development of a range of cache mechanisms. I developed these mechanisms based on theoretical results so that these mechanisms can offer some performance guarantees or have some properties relevant to performance and predictability. Some of these mechanisms make use of application-specific information obtained from application traces and incorporate that information in the form of additional instructions or hints in the executable. The mechanisms are explored in the context of embedded systems but are also applicable to general-purpose systems.

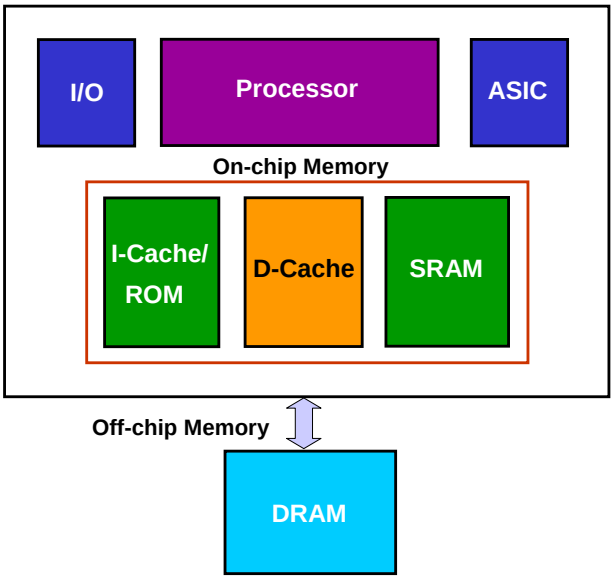


Figure 1-1: Embedded System and On-chip Memory

1.2 Embedded Memory Systems

In many embedded systems, simple pipelined processors (e.g., ARM9TDMI [1]) are used along with on-chip memory. The on-chip static memory, in the form of a cache or an SRAM (scratch-pad memory) or some combination of the two, is used to provide an interface between hardware and software and to improve embedded system performance. Most systems have both on-chip cache and SRAM since each addresses a different need. The components of an on-chip memory in an embedded system are shown in Figure 1-1. The effective utilization of the caches and on-chip SRAMs can lead to significant performance improvement for embedded systems.

Typically, the processor and on-chip memory are used as decoupled components. The only communication between the processor and the on-chip memory is through load/store operations. The on-chip memory serves as data storage for the processor and data is read from or written to an address provided by the processor. On-chip memory when used as a cache serves as transparent storage between the processor and the off-chip memory since it is accessed through the same address space as the off-chip memory. The processor read/write requests are served from the on-chip cache provided the data is in the cache, otherwise data is brought into the cache from the off-chip memory. The hardware management logic handles

data movement and some other functions to maintain transparency. On-chip memory when used as an SRAM does not have any hardware management logic and the data movement is controlled through software.

Typically, on-chip memory simply responds to the access requests by the processor. The processor does not communicate any application-specific information to the on-chip memory. There is little or no software control or assistance to the hardware logic to manage the cache resources effectively.

I consider the on-chip memory to be *intelligent*, if there is some hardware logic associated with the on-chip memory either to manage the data storage or to perform some computation. So, an on-chip associative cache has some intelligence, for example, to make replacement decisions, and the on-chip SRAM is not intelligent. For embedded systems, the capability (intelligence) of the on-chip caches can be enhanced by cache mechanisms that use some application-specific information. There are several problems associated with the use of caches and SRAM when they are used as on-chip memory for an embedded system. My primary focus in this thesis is on the development of a range of cache mechanisms for on-chip cache memory that address these problems. The specific problems addressed in this thesis are described in the next section.

1.3 Thesis Research Problems

In this thesis I focus on the following aspects of the on-chip caches.

1.3.1 Cache Performance

Caches are used to improve the average performance of application-specific and general-purpose processors. Caches vary in their organization, size and architecture. Depending on the cache characteristics, application performance may vary dramatically. Caches are valuable resources that have to be managed properly in order to ensure the best possible program performance. For example, cache line replacement decisions are made by the hardware replacement logic using a cache replacement strategy. In associative caches, commonly used replacement strategies are the Least Recently Used (LRU) replacement strategy and its cheaper approximations, where the cache line that is judged least recently used is evicted. It is known, however, that LRU does not perform well in many situations, such as streaming

applications and timeshared systems where multiple processes use the same cache.

The cache performance directly affects the processor performance even in superscalar processors where the latency of a cache miss can be hidden by scheduling other ready-to-execute processor instructions. Current caches in embedded systems and most general-purpose processors are largely passive and reactive in terms of their response to the access requests. Moreover, current caches do not use application-specific information in managing their resources. and there is relatively little software control or assistance with hardware support to manage the cache resources effectively. As a result the caches may not perform well for some applications.

The performance of a cache can be measured in terms of its hit or miss rate. A miss in a cache can be either a cold miss, a conflict miss, or a capacity miss. Conflict and capacity misses can sometimes be avoided using different data mapping techniques or increasing the associativity of the cache. Cold misses can be avoided using prefetching, i.e., predicting that the processor will make a request for a data block, and bringing the data in before it is accessed by the processor. There have been numerous approaches proposed in the past decades to improve cache performance. Though my approach in this thesis draws on previous work, I have attempted to develop mechanisms with some theoretical basis to provide performance guarantees.

1.3.2 Cache Predictability

Caches are transparent to software since they are accessed through the same address space as the larger backing store. They often improve overall software performance but can be unpredictable. Although the cache replacement hardware is known, predicting its performance depends on accurately predicting past and future reference patterns.

One important aspect to cache design is the choice of associativity and the replacement strategy, which controls which cache line to evict from the cache when a new line is brought in. LRU does not perform well in many situations, including timeshared systems where multiple processes use the same cache and when there is streaming data in applications. Additionally, the LRU policy often performs poorly for applications in which the cache memory requirements and memory access patterns change during execution. Most cache replacement policies, including LRU, do not provide mechanisms to increase predictability (worst-case performance), making them unsuited for many real-time embedded

system applications. The predictability of cache performance can be enhanced by allowing for more control over replacement decisions or by separating data accesses with different characteristics in the cache as described later in the thesis.

1.3.3 Cache Pollution with Prefetching

One performance improvement technique for caches is data prefetching where data likely to be used in the near future is brought into the cache before its use. So, when the prefetched data is accessed, it results in a hit in the cache. Prefetching methods come in many different flavors, and to be effective they must be implemented in such a way that they are timely, useful, and introduce little overhead [123]. But, prefetching can lead to *cache pollution* because a prematurely prefetched block can displace data in the cache that is in use or will be used in the near future by the processor [15]. Cache pollution can become significant, and cause severe performance degradation when the prefetching method used is too aggressive, i.e., too much data is brought into the cache, or too little of the data brought into the cache is useful. It has been noted that hardware-based prefetching often generates more unnecessary prefetches than software prefetching [123]. Many different hardware-based prefetch methods have been proposed; the simple schemes usually generate a large number of prefetches, and the more complex schemes usually require hardware tables of large sizes. The cache pollution caused by a prefetching method can be mitigated by combining the prefetch method with a replacement policy. Such a combination can reduce cache pollution by effectively controlling the prefetch method and the treatment of the prefetched data in the cache.

1.4 Relationship to Previous Research on Caches

Caches have been a focus of research for a long time. For reference, Belady's optimal replacement algorithm was presented in 1966 [5] and since then there has been a lot of research effort focussed on improving cache performance using a variety of techniques. As a result, there is a great deal of work on various aspects of caches and there is a huge number of results in the cache research literature. Sometimes, it is hard to compare the results either due to a very specific nature of the work or the difficulty to reproduce the results in one framework for comparison. Now I briefly describe how the thesis work relates to prior

work in caches. A detailed description of related work is given in Chapter 7.

Cache performance problems have been addressed from different perspectives (e.g., cache architectures [55, 118, 129, 134], locality optimization [30, 48, 76, 78], cache management [46, 53, 86, 112, 116], replacement policies [7, 44, 45, 50, 107, 128, 132]). The intelligent replacement mechanism I developed, as described in Chapter 3, can be viewed as touching the above mentioned perspectives of prior cache research work. Unlike most previous approaches, it offers a lower bound on performance. There are many prefetching methods proposed in the cache literature (e.g., hardware prefetching [18, 91, 108], software prefetching [68, 69, 79, 82, 102], prefetching with other mechanisms [13, 39, 64, 85], and other prefetching approaches [3, 52, 70, 126]). Since prefetching is largely based on prediction and timing, a prefetching method may work for one application and not work for another application. My work has not resulted in new prefetch methods – I have instead focussed on mitigating cache pollution effects caused by aggressive prefetch methods [43], while offering theoretical guarantees. The cache partitioning approaches proposed in the past focus on different aspects of the caches (e.g., performance [23, 80, 87, 94, 98, 100], low power or energy [56, 58, 81, 136], predictability [59, 104]). The cache partitioning mechanisms I developed, as described in Chapter 5, are based on some theoretical results that can provide guarantees of improved behavior.

There have been many mechanisms proposed with no theoretical basis or performance guarantees. We think that theoretical results are important because a theoretical basis for mechanisms can provide a framework to reason about the properties of different mechanisms, separate from benchmark applications. Empirical evaluation of caches based on benchmarks suffers from the problem that results can rarely be generalized to different cache sizes or organizations. So, the focus of this thesis is on a search for theoretical results that can guide the development of suitable mechanisms to solve the on-chip cache problems. The contributions of the thesis in this regard are presented below.

1.5 Thesis Contributions

The main contributions of the thesis are the theoretical results for the proposed mechanisms. A range of cache mechanisms are possible based on these theoretical results. A subset of the possible cache mechanisms is designed, implemented in the Simplescalar [11] simulator,

and evaluated to validate the theoretical results. The experiments also show the efficacy of the proposed cache mechanisms to address the cache performance, predictability, and pollution problems outlined in the previous section.

I considered two categories of cache mechanisms to address the problems outlined in the previous section. These mechanisms use a small amount of additional hardware support logic with software assistance in the form of hints or control instructions. The combination of software/hardware support leads to better data management capability of the on-chip data cache. The specific contributions with regard to these cache mechanisms are described below.

1.5.1 Intelligent Cache Replacement

I introduced the notion of a *kill*-bit and *keep*-bit which are stored as part of the additional cache state. In short, the kill bit is used to mark data that is no longer beneficial to keep in the cache, whereas the keep bit is used to mark data that is still useful and desired to be kept in the cache. I used the kill-bit state information along with the LRU information to derive a replacement policy called *Kill+LRU* with several possible variations. In addition, I used the kill-bit state and the keep-bit state information along with the LRU information to derive a replacement policy called *Kill+Keep+LRU* with several possible variations. In order to address the cache pollution caused by an aggressive hardware prefetch scheme, I used the Kill+LRU replacement policy to derive *Kill+Prefetch+LRU* replacement policy. The LRU cache replacement policy is augmented with the additional cache state bits to make replacement decisions. We refer to the the replacement policies Kill+LRU, Kill+Keep+LRU, Kill+Prefetch+LRU, and their variations as intelligent cache replacement policies. There are several ways to provide hardware/software support to set the additional state information for the intelligent cache replacement policies described above. I consider some of the hardware and software-assisted mechanisms. In this context, the additional contributions are:

- **Theoretical Results:** I derived conditions and proved theorems based on these conditions which would guarantee that the number of misses in the Kill+LRU cache replacement policy is no more than the LRU replacement policy for the fully-associative and set-associative caches. I proved similar theorems for the case where an additional

keep state information is combined with the Kill+LRU replacement policy, i.e., for the Kill+Keep+LRU replacement policy. Some additional theoretical results are derived when prefetching is combined with the Kill+LRU replacement policy, i.e., for the Kill+Prefetch+LRU replacement policy, which show that the integration of Kill+LRU replacement policy and prefetching results in more predictable cache performance by controlling cache pollution.

- **Hardware Mechanisms:** I have designed some hardware mechanisms to approximate the conditions to set the kill-bit and other state information to provide support for the intelligent replacement policies described above. In particular, I designed the hardware mechanisms based on history signature functions. I also designed a hardware mechanism to measure reuse distance which can be used to further enhance the Kill+LRU replacement policy. The hardware mechanisms were implemented in the Simplescalar [11] simulator.

- **Software-assisted Mechanisms:** I also designed some software-assisted mechanisms where an offline analysis is done to determine the conditions appropriate for the chosen intelligent replacement policy. This application-specific information is incorporated into the application code in the form of hints and control instructions to support the intelligent replacement policy. A small hardware support logic module uses the hints and control instructions to handle the additional cache state updates. A trace-based or profile-based approach is used to derive and incorporate the necessary information into the application code. These mechanisms are also implemented in the Simplescalar simulator and evaluated in a stand-alone fashion and with prefetching on the Spec2000 benchmarks.

1.5.2 Disjoint Sequences and Cache Partitioning

I propose the concept of *disjoint sequences*, where two address sequences are disjoint if their memory addresses are disjoint, i.e., there is no common address between the two address sequences. The concept of disjoint sequences facilitates the analysis and transformation of address sequences. Since any address sequence can be considered as a merged address sequence of one or more disjoint address sequences, an address sequence can be transformed into a concatenation of disjoint address sequences. This temporal separation of disjoint

sequences of an address sequence is used to derive conditions for physical separation of disjoint sequences in the cache. A range of cache partitioning mechanisms and a static cache partitioning algorithm are developed based on the disjoint sequences concept to address the problems of cache performance and predictability. The specific contributions are:

- **Theoretical Results:** I prove theorems which show that changing an access sequence into a concatenation of disjoint access sequences and applying the concatenated access sequence to the cache is guaranteed to result in improved hit rate for the cache. I use this result to derive partitioning conditions which guarantee that cache partitioning leads to the same or less number of misses than the unpartitioned cache of the same size.

- **Partitioning Mechanisms:** Several cache partitioning mechanisms are introduced which support disjoint sequences-based partitioning. In particular, partitioning mechanisms based on a modified LRU replacement, cache tiles, and multi-tag sharing approach are proposed in this thesis. In order to use a the cache partitioning mechanism, a partitioning algorithm is required. I have focussed on a static partitioning algorithm which is inspired by the disjoint sequences-based theorems. This algorithm determines disjoint sequences for an application, groups the disjoint sequences, and assigns the cache partition sizes to the groups. The cache partitioning mechanism based on modified LRU replacement policy was implemented in the Simplescalar simulator. A two-sequence static partitioning algorithm with the modified LRU replacement policy-based partitioning mechanism was evaluated on the Spec2000 benchmarks.

1.6 Thesis Organization

In this chapter, I have outlined the performance bottlenecks of current processors, identified research problems associated with data caches, put my thesis work in the context of previous research work on caches, and summarized the contributions of this thesis.

In Chapter 2, a brief overview of the new mechanisms is given along with the overall approach to implement and evaluate the mechanisms. The simulation framework and the benchmarks used for evaluation are also described in this chapter.

In Chapter 3 and 4, the details of the intelligent cache replacement mechanism are described. We give many variants in Chapter 3, including integrating the replacement

schemes with prefetching. Theorems are proven that give performance guarantees in terms of miss rates for the intelligent cache replacement methods. Implementation details of the intelligent cache replacement mechanism are presented in Chapter 4.

In Chapter 5, the concept of disjoint sequences, a theoretical result regarding the hit rates achieved by disjoint sequences, a relationship between disjoint sequences and cache partitioning, and details of the hardware and software aspects of partitioning mechanisms are presented.

In Chapter 6, a trace-based evaluation of the intelligent cache replacement and cache pollution control with intelligent replacement on a subset of Spec95 and/or Spec2000 benchmarks is presented. Also, the evaluation of cache partitioning based on disjoint sequences is presented on a subset of Spec2000 benchmarks.

In Chapter 7, related work on intelligent cache replacement mechanisms and cache partitioning schemes is summarized.

In Chapter 8 a summary of the thesis is followed by some possible extensions to the work presented in this thesis.

Chapter 2

Overview of Approach

2.1 Architecture Mechanisms

I propose approaches to make on-chip memory *intelligent* by adding additional hardware logic and state to/near on-chip memory and by increasing the communication between the processor and the on-chip memory. Application-specific information is communicated to the on-chip memory hardware logic in different ways, as I elaborate on below. These enhancements to the capability of the on-chip memory improve the on-chip memory performance, which in turn improves embedded system performance.

The software support for the mechanisms is provided in the form of processor to on-chip Memory communication. Additional information may be supplied to the memory based on the application-specific information gathered during the program analysis. The following methods are used to increase the interaction between the application and the on-chip memory.

- **hints**: these are hints in the application code to communicate some specific information to the on-chip memory. Such information is used by the hardware support logic.
- **control**: these are specific controls that augment the state of some on-chip memory resources.
- **processor information**: this is processor state conveyed to the on-chip memory (e.g., branch history, load/store type).

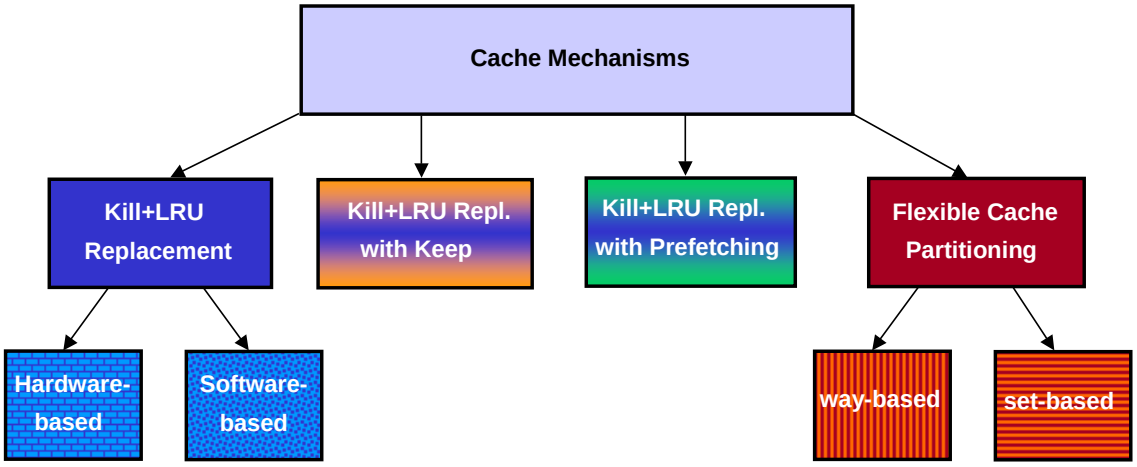


Figure 2-1: Overview of the Mechanisms

Each mechanism I propose targets a particular problem associated with using simple on-chip memory. An intelligent on-chip memory would consist of support for one or more mechanisms. The mechanisms differ in terms of the required hardware support, processor to on-chip memory communication, and additional state information. The mechanisms described in this thesis are targeted for the use of on-chip memory as a cache where the on-chip cache would have more intelligent management of the cache resources. The taxonomy of the proposed mechanisms is shown in Figure 2-1. A brief overview of the mechanisms is given below and the details of the mechanisms are discussed in the following chapters.

An overview of the intelligent replacement mechanism is shown in Figure 2-2. The data cache is augmented to have some additional state information and modified LRU replacement logic. The intelligent cache replacement mechanism can use either the hardware-based or software-assisted augmentation of the LRU replacement strategy.

Cache pollution due to aggressive hardware prefetching is measured for LRU replacement. The proposed cache replacement mechanism is explored in the context of reducing cache pollution when used with sequential hardware prefetching, an aggressive prefetch method.

In order to use the cache space effectively, a cache partitioning approach based on disjoint sequences is explored in this thesis. A high-level overview of the cache partitioning is shown in Figure 2-2. The cache partitioning related information is derived using profile-based analysis or dynamically in hardware. The static cache partitioning information is provided

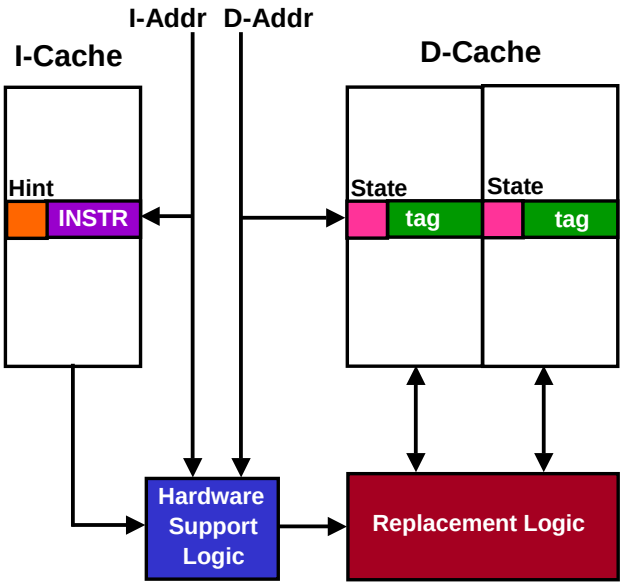


Figure 2-2: Overview of Intelligent Replacement and Cache Partitioning

via some additional instructions at the beginning of the program. This cache partitioning information is maintained in a small hardware partitioning information table. The hardware support for the way-based and set-based partitioning is described in Section 5.5.2 (Modified LRU Replacement) and Section 5.5.2 (Cache Tiles) respectively.

2.2 Evaluation Methodology

2.2.1 Overall Flow

The overall flow of information for intelligent on-chip memory is shown in Figure 2-3. There are two major steps in providing the software support for different mechanisms. First, the program is analyzed to gather information about the application. Second, the information about the application is incorporated into the program generated code. The type of analysis and modification differs for the two on-chip cache mechanisms.

2.2.2 Program Analysis

Two primary ways of program analysis are **static analysis** and **profile analysis**. Static analysis uses the intermediate representation of the source code of the program to analyze the program and provides information about the control and data flow of the program. Static analysis cannot easily determine program behaviors that are dependent on input

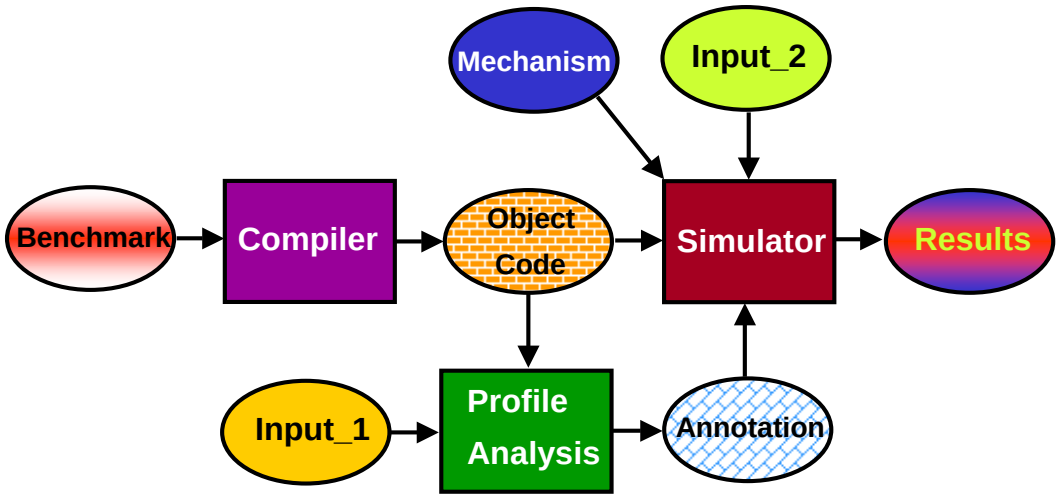


Figure 2-3: Overall Flow

data. Profile analysis of the program uses a training input data set and gets program related information that may not be available through static analysis. The analysis of the program identifies the points in the program where useful information can be communicated to the underlying hardware logic for the chosen mechanism. I focus on profile analysis in this thesis.

2.2.3 Program Code Modification

Program code is annotated with the appropriate instructions to communicate the information gathered from program analysis. The program can be modified in two ways: annotations are introduced during the program code generation, or annotations are introduced into the assembly code after code generation. The code modification approach may depend on the analysis approach used. These modifications to the code use the underlying hardware for the mechanisms at run-time.

2.2.4 Simulation Framework

The cache mechanisms are implemented in the Simplescalar [11] simulator by modifying some existing simulator modules and adding new functionality for each of the new mechanisms. The simulator is also modified to handle the annotations appropriate for each of the software-assisted mechanisms.

2.2.5 Benchmarks

The intelligent cache replacement mechanism is evaluated using a subset of the Spec2000 benchmarks. The Spec2000 benchmarks consist of the Spec2000INT - integer benchmarks and Spec2000FP - floating-point benchmarks.

2.3 Evaluation of the Mechanisms

The cache mechanisms are implemented in the Simplescalar simulator. The information necessary for the hints and control instructions is derived using profile-based analysis. The hints and control instructions are associated with a PC and written to an annotation file. The hints and control instructions are conveyed to the Simplescalar simulator using this annotation file and stored in a PC-based table in the Simplescalar simulator. Some of the processor related information is conveyed to the cache module using the modified cache module interface. A brief description of the implementation and evaluation of the mechanisms is given below.

2.3.1 Intelligent Cache Replacement

The cache module of the Simplescalar was modified with additional state bits and a new replacement policy. The hardware-based intelligent replacement uses some tables in the simulator to maintain and use dynamically generated information. For the software-assisted intelligent replacement, the PC-based table is used in conjunction with the hardware structures. The intelligent cache replacement mechanism is evaluated in stand-alone fashion and with prefetching. The prefetching method is implemented in the Simplescalar simulator and is combined with the replacement mechanism to evaluate the reduction in cache pollution with the intelligent replacement mechanism.

2.3.2 Cache Partitioning

The cache module of the Simplescalar simulator was modified to add partitioning related state bits in the cache structures. Also some hardware structures necessary to maintain partitioning related information were implemented into the Simplescalar. The partitioning mechanism also makes use of the profile-based information using the PC-based table and data range-based tables. The partitioning mechanism was evaluated in stand-alone fashion

for performance and predictability. The way-based partitioning was implemented using a modified LRU replacement policy which takes into account the partitioning information for replacement. In the way-based partitioning evaluation, address segment information of the accesses was used to form two disjoint sequences of accesses. The two disjoint sequences were assigned cache partition sizes using a reuse distance-based partitioning algorithm. The way-based cache partitioning evaluation results for the Spec2000 benchmarks are described in Section 6.3.

Chapter 3

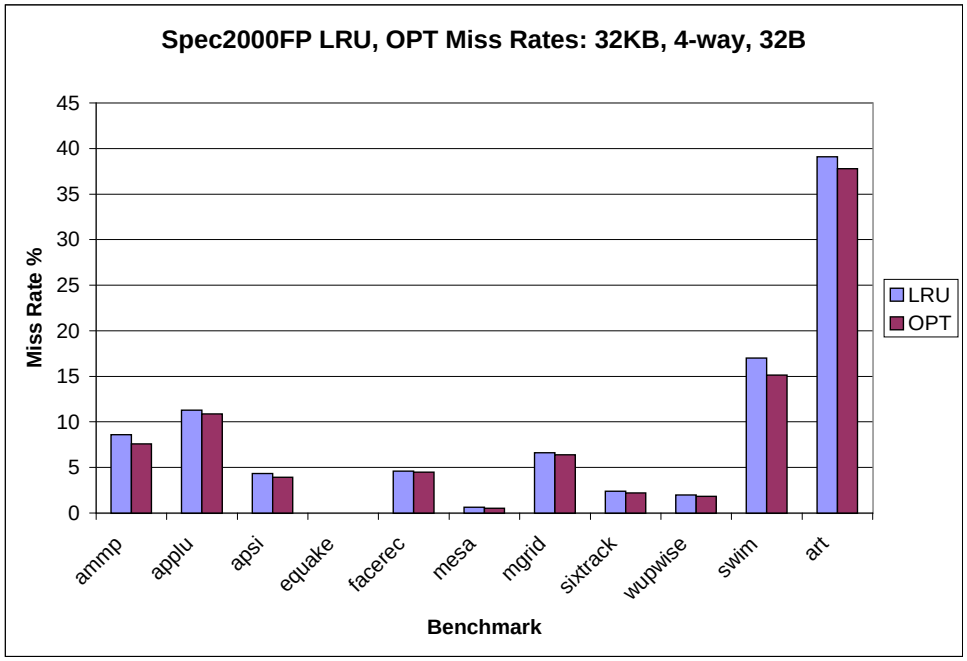
Intelligent Cache Replacement Mechanism

3.1 Introduction

The LRU replacement policy is a common replacement policy and it is the basis of the intelligent cache replacement mechanism which offers miss rate improvement over the LRU policy. The miss rate improvement over the LRU replacement policy offered by the intelligent replacement mechanism is limited by the miss rate of the Optimal replacement policy (OPT) [5] because the OPT replacement policy gives a lower bound on the miss rate for any cache. So, the intelligent replacement mechanism improves cache and overall performance of an embedded system by bringing the miss rate of an application closer to the miss rate using the OPT replacement policy.

Figure 3-1 shows the miss rates for the Spec2000FP and Spec2000INT benchmarks respectively using the LRU and OPT replacement policies. The miss rates were obtained using the Sim-cheetah simulator [11, 110]. One billion instructions were simulated after skipping the first two billion instructions. Similarly, Figure 3-2 shows the miss rates of the LRU and OPT replacements policies for a subset of the Mediabench and Mibench benchmarks. The results show that for some benchmarks there is appreciable potential for improvement in LRU miss rates.

In this chapter the intelligent replacement mechanism based on a modification of the LRU replacement policy is described in detail. Some of the work described herein has been



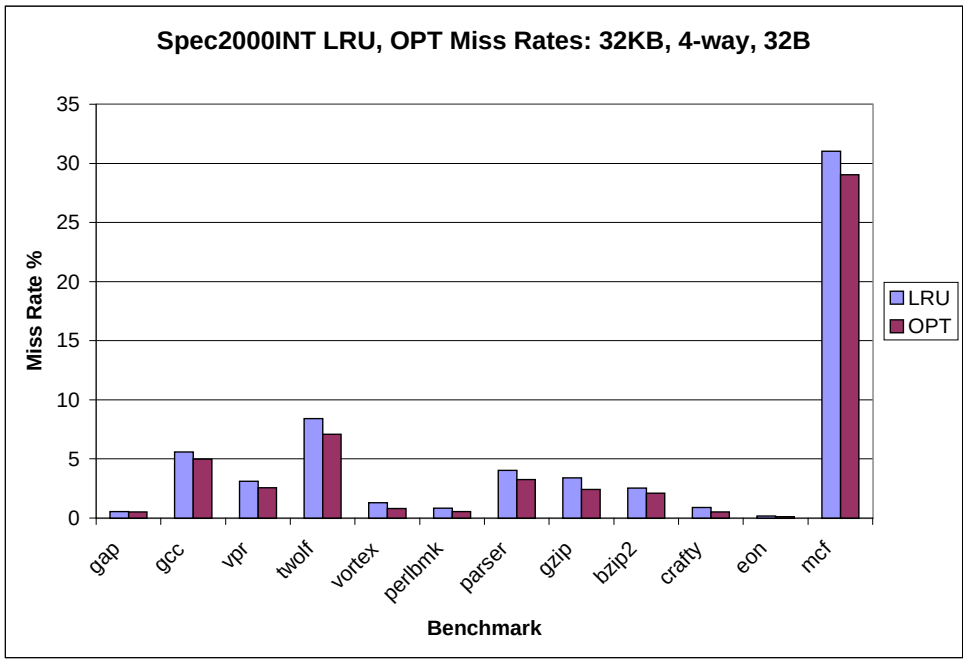


Figure 3-1: LRU and OPT replacement misses for Spec2000FP and Spec2000INT benchmarks using sim-cheetah

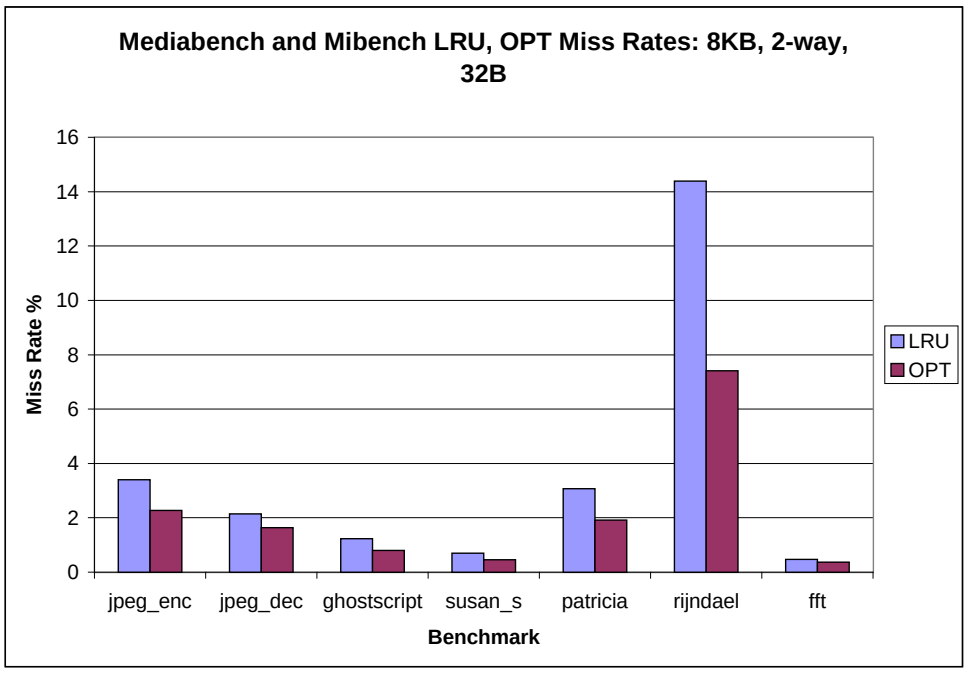


Figure 3-2: LRU and OPT Miss Rates for Mediabench and Mibench

published in [42]. First, the basic concepts of dead block, killing a block, and keeping a block are presented in Section 3.2. Then, the mechanisms based on these concepts are presented in Section 3.3. These include an integration of intelligent replacement with prefetching to control cache pollution. The theoretical results for the proposed mechanisms are presented in Section 3.4. The implementation of the intelligent cache replacement is presented in Chapter 4.

3.2 Basic Concepts

3.2.1 Dead Blocks

If there is a cache miss upon an access to a cache block b, the missed cache block b is brought from the next level of the memory hierarchy. There may be one or more subsequent accesses to the cache block b that result in cache hits. The cache block b is eventually chosen to be evicted for another cache block as a result of the cache miss(es) that follow based on the cache replacement policy. The cache block b is considered *dead* from its last cache hit to its eviction from the cache.

3.2.2 Kill a Block

A cache replacement policy can use the information about dead blocks in its replacement decisions and choose a dead block over other blocks for replacement. The dead block information introduces a priority level in the replacement policy. If the last access to a cache block b can be detected or predicted either statically or dynamically, then the cache block b can be *kill*ed by marking it as a dead block. The condition that determines the last access to a cache block is called a *kill predicate*. So, when the kill predicate for a cache block is satisfied, the cache block can be killed by marking it as a dead block.

3.2.3 Keep a Block

In some cases it may be desirable to assign a cache block b higher priority than other blocks in the cache so that a replacement policy would choose other blocks over the block b for replacement. This can be accomplished by *Keep*ing the block b in the cache by setting some additional state associated with the block b.

3.3 Mechanisms

The LRU replacement policy is combined with the above concepts of dead blocks, killing a block, and keeping a block to form the intelligent replacement mechanisms. In the following description, an *element* refers to a cache block.

3.3.1 Kill with LRU

In this replacement policy, each element in the cache has an additional one-bit state (K_l), called the kill state, associated with it. The K_l bit can be set under software or hardware control. On a hit the elements in the cache are reordered along with their K_l bits the same way as in an LRU policy. On a miss, instead of replacing the LRU element in the cache, an element with its K_l bit set is chosen to be replaced and the new element is placed at the most recently used position and the other elements are reordered as necessary. Two variations of the replacement policy to choose an element with the K_l bit set for replacement are considered: (1) the least recent element that has its K_l bit set is chosen to be replaced (LRK_LRU policy); (2) the most recent element that has its K_l bit set is chosen to be replaced (MRK_LRU policy). The K_l bit is reset when there is a hit on an element with its

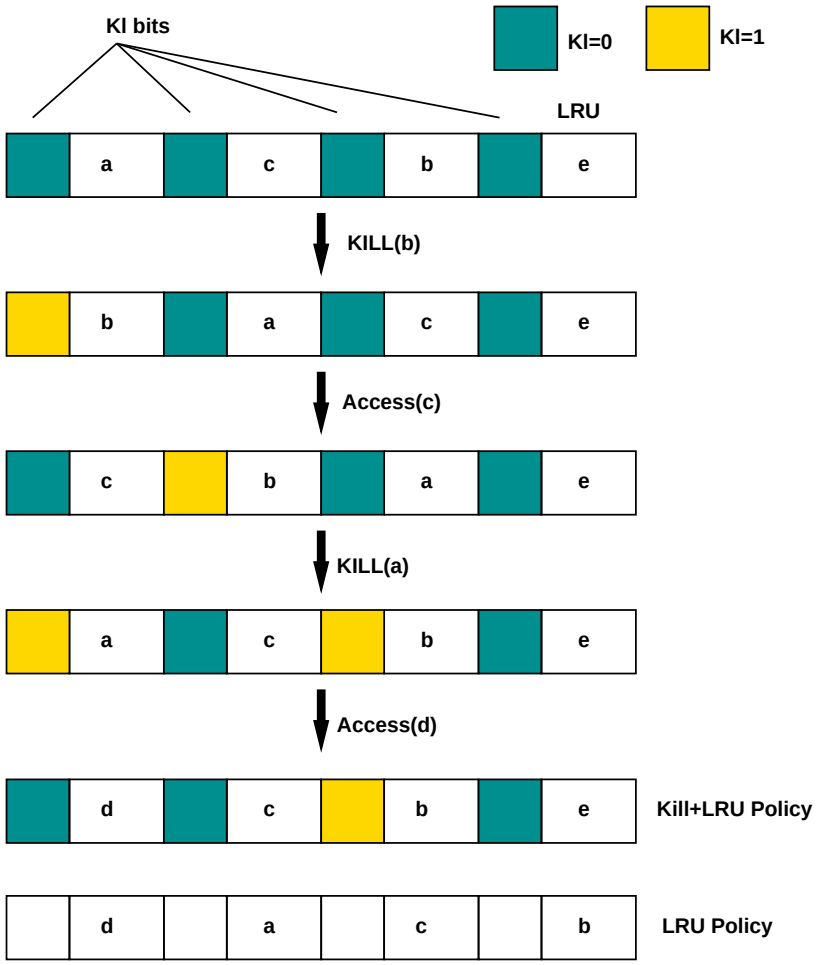


Figure 3-3: Kill+LRU Replacement Policy

K_l bit set unless the current access sets the K_l bit. It is assumed that the K_l bit is changed – set or reset – for an element upon an access to that element. The access to the element has an associated hint that determines the K_l bit after the access and the access does not affect the K_l bit of other elements in the cache.

Figure 3-3 shows how the K_l bit is used with the LRU policy. It shows a fully-associative cache with four elements. Initially, the cache state is $\{a, c, b, e\}$ with all the K_l bits reset. When b is killed (b is accessed and its K_l bit is set) b becomes the MRU element and the new cache state is $\{b, a, c, e\}$. Then, c is accessed and a is killed. After a is killed, the cache state is $\{a, c, b, e\}$ and the K_l bits for a and b are set. When d is accessed, it results in a cache miss and an element needs to be chosen for replacement. In the Kill+LRU replacement policy, using the variation (2), the most recently killed (MRK) element a is chosen for replacement and the new cache state is $\{d, c, b, e\}$. If the LRU replacement policy were used, the LRU

element e would be chosen for replacement and the new cache state would be $\{d, a, c, b\}$.

MRK-LRU based Miss Rate Estimation and OPT Policy

The miss rate for the MRK_LRU (Most Recently Killed) variation (2) of the Kill+LRU replacement policy can be estimated using the reuse distance profile of a program under a uniform reuse distance model. This estimated miss rate gives an idea of the miss rate improvement possible over LRU for the given program. Also, the estimated miss rate improvement with the MRK_LRU policy can be compared to the miss rate improvement under the OPT policy. The miss rate estimation method can be useful in determining how effective the MRK_LRU replacement policy would be for a given program.

Consider a uniform reuse distance model where all the accesses have a distinct block reuse distance d between two accesses to the same block. For a given associativity a and a uniform reuse distance model with d such that $d \geq a$, in steady-state the LRU replacement would miss on every access resulting in miss rate 1.0 and the MRK-LRU would have the miss rate $1.0-(a-1)/d$. So, the miss rate improvement of MRK-LRU over LRU is $(a-1)/d$.

For example, given the access sequence $\{a, b, c, d, e, a, b, c, d, e, a, b, c, d, e, ...\}$ where the uniform reuse distance is 4 and assuming a 4-word fully-associative cache, the access sequence would result in a miss every fourth access in the steady-state (i.e., when the cache reaches the full state from an initially empty state). For this example, the corresponding miss-hit sequence is $\{m, m, m, m, m, h, h, h, m, h, h, h, m, h, h, ...\}$.

The above uniform reuse distance model can be used to estimate the miss rate improvement for a given program if the reuse distance profile of the program is available. Suppose the distribution of the reuse distances for a program is such that W_d is the weight for a reuse distance $d \geq a$, for a given associativity a. The miss rate improvement can be estimated using the following formula based on the uniform reuse distance model:

$$MRImpv = (a-1) \times \sum_{d=a}^{d_{max}} \frac{W_d}{d}$$

The miss rate improvement formula was applied to the reuse distance profile of the Spec2000 benchmarks for a set-associative cache with $a = 4$ and $d_{max} = 256$. The reuse distance profile was collected using the replacement-to-miss distance history with $d_{max} = 256$. The miss rate improvement estimate is compared to the OPT miss rate improvement

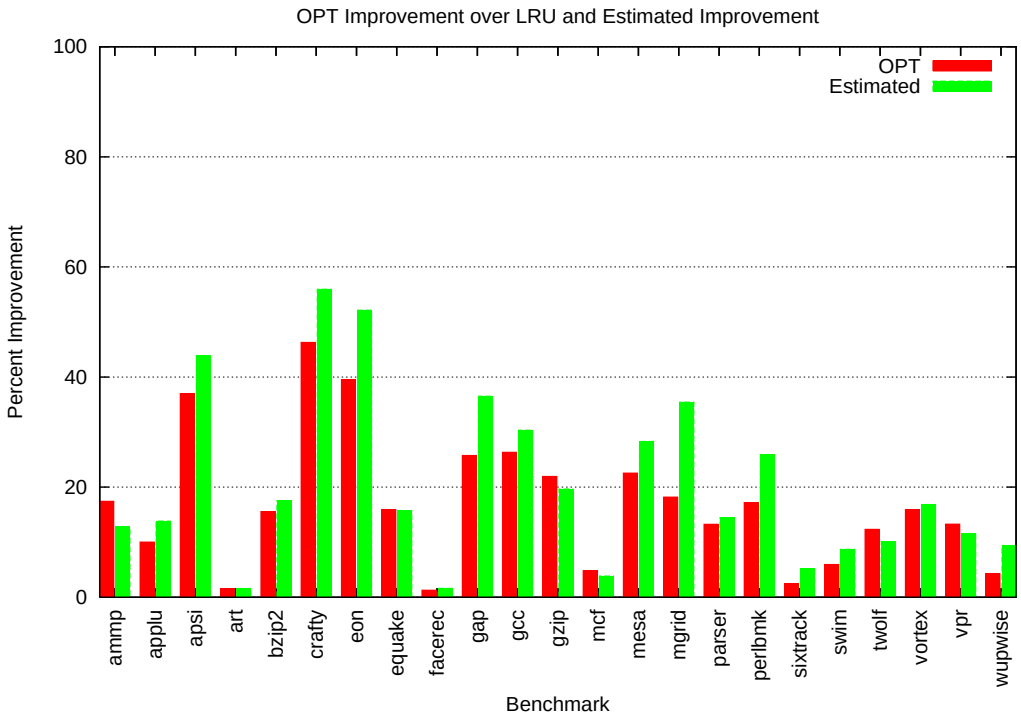


Figure 3-4: OPT Improvement and Estimated Improvement Based on Profile data and Uniform Distance Model

in Figure 3-4.

Since the blocks with a certain reuse distance get replaced by blocks with a different reuse distance, the uniform distance assumption does not hold in all instances for a given a reuse distance profile. As a result, the estimate is sometimes higher or lower than the OPT miss rate improvement. If the estimate is lower than the OPT, then it indicates that OPT was able to find the blocks with higher distance to replace for the ones with lower distance. If the estimate is higher than the OPT, then it indicates that the formula is more optimistic than the actual OPT block replacement choices. As an aside, a more accurate model for miss-rate improvement estimation can be developed using some more parameters derived from the profile-based analysis and some more information about the block replacements.

3.3.2 Kill and Keep with LRU

In this replacement policy, each element in the cache has two additional states associated with it. One is called a kill state represented by a K_l bit and the other is called a keep state represented by a K_p bit. The K_l and K_p bits cannot both be 1 for any element in the cache at any time. The K_l and K_p bits can be set under software or hardware control. On a hit

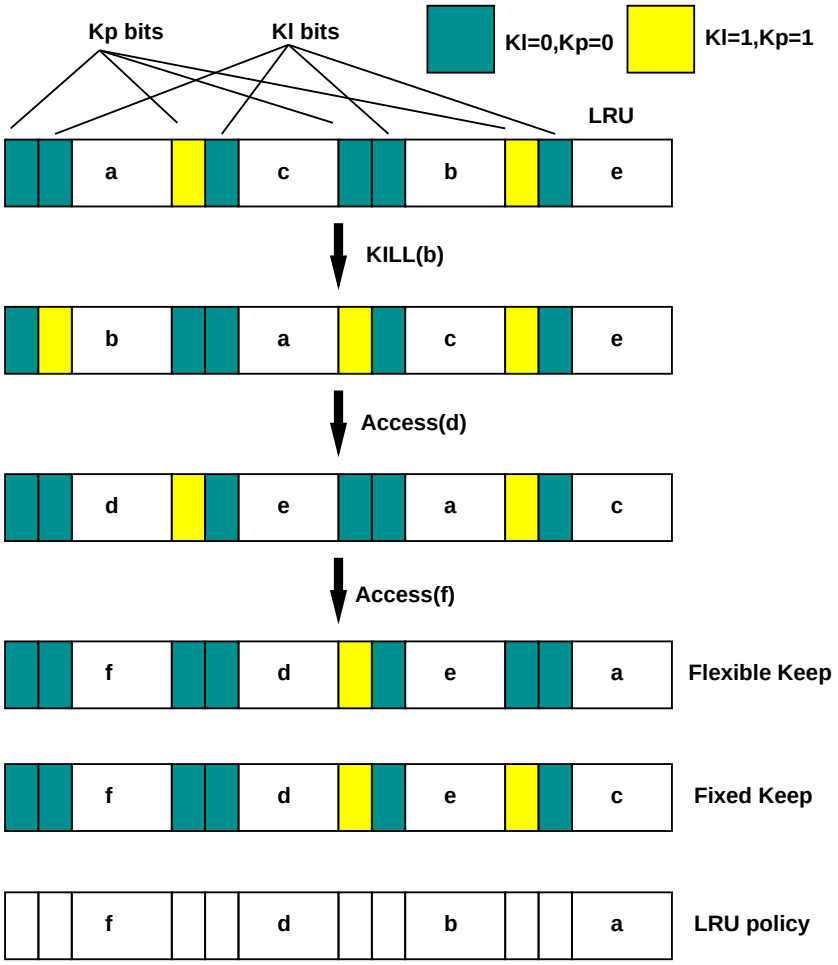


Figure 3-5: Kill+Keep+LRU Replacement Policy

the elements in the cache are reordered along with their K_l and K_p bits the same way as in an LRU policy. On a miss, if there is an element with the K_p bit set at the LRU position, then instead of replacing this LRU element in the cache, the most recent element with the K_l bit set is chosen to be replaced by the element at the LRU position (to give the element with the K_p bit set the most number of accesses before it reaches the LRU position again) and all the elements are moved to bring the new element at the most recently used position. On a miss, if the K_p bit is 0 for the element at the LRU position, then the elements in the cache are reordered along with their K_l and K_p bits in the same way as in an LRU policy. There are two variations of this policy: (a) *Flexible Keep:* On a miss, if there is an element at the LRU position with the K_p bit set and if there is no element with the K_l bit set, then replace the LRU element (b) *Fixed Keep:* On a miss, if there is an element at the LRU position with the K_p bit set and if there is no element with the K_l bit set, then replace the

least recent element with its K_p bit equal to 0.

Figure 3-5 shows how the K_l and K_p bits are used with the LRU policy. It shows a fully-associative cache with four elements. Initially, the cache state is $\{a, c, b, e\}$ with all the K_l bits reset and K_p bits set for c and e. When b is killed (b is accessed and its K_l bit is set) b becomes the MRU element and the new cache state is $\{b, a, c, e\}$. Then, d is accessed and it results in a cache miss and an element needs to be chosen for replacement. In the flexible keep variation of the Keep with Kill+LRU replacement policy, the most recently killed element a is chosen for replacement because e is in the LRU position with its K_p bit set. The new cache state is $\{d, e, a, c\}$. Now, the access to f results in a miss and c is the LRU element chosen for replacement because there is no element with its K_l bit set. The final cache state is $\{f, d, e, a\}$. If the fixed keep variation is used, the cache state before the access to f is the same cache state for the flexible keep variation: $\{d, e, a, c\}$. Now, upon an access to f, there is a cache miss and a is chosen for replacement because c has its K_p bit set and a is the least recently used element with its K_p bit reset. The final cache state is $\{f, d, e, c\}$. If the LRU replacement policy were used, the final cache state would be $\{f, d, b, a\}$.

Flexible Keep and Kill with LRU

The flexible Kill+Keep+LRU allows the blocks marked as Keep blocks to be kept in the cache as long as possible without reducing the hit rate compared to the LRU policy. But, the Flexible Keep can result in less hits than the Kill+LRU replacement if the blocks that are marked to be kept in the cache have the reuse distance longer than the blocks that are not marked as Keep blocks. As a result, some of the blocks not marked as Keep blocks would be replaced and lead to more misses than the hits resulting from the blocks marked to be kept in the cache. Even though the Kill+Keep+LRU results in lower overall hit rate for a set of data marked to be kept in the cache compared to the Kill+LRU, it improves cache predictability of the data in the cache marked to be kept in the cache.

The other approach for keeping important data in on-chip memory is by assigning the data to the SRAM, but this approach requires using different structures for the cache and the SRAM. The data kept in the SRAM is available all the time, but the data in the SRAM needs to be explicitly assigned to the SRAM and it may require some data layout to accommodate different data that needs to be kept in the SRAM at the same time.

3.3.3 Kill and Prefetch with LRU

The above mechanisms considered only the accesses issued by the processor that resulted in a cache miss or a hit. A cache block can be brought into the cache by prefetching it before it is requested by the processor. When prefetching is used in addition to the normal accesses to the cache, there are different ways to handle the prefetched blocks.

Cache pollution is a significant problem with hardware prefetching methods, since hardware methods typically need to be aggressive in order to have simple implementations. Cache pollution can be controlled by using the intelligent cache replacement mechanism, and the resultant prefetch strategies can improve performance in many cases. I will show that a new prefetch scheme where 1 or 2 adjacent blocks are prefetched depending on whether there is dead data in the cache or not works very well, and is significantly more stable than standard sequential hardware prefetching (cf. Section 6).

There are four different scenarios for integrating intelligent cache replacement with prefetching as shown in Figure 3-6.

(a): Figure 3-6(a) illustrates the standard LRU method without prefetching. New data is brought into the cache on a miss, and the LRU replacement policy is used to evict data in the cache. This method is denoted as (LRU, ϕ), where the first item in the 2-tuple indicates that new (normal) data brought in on a cache miss replaces old data based on the LRU policy, and the ϕ indicates that there is no prefetched data.

(b): Figure 3-6(b) illustrates generic prefetching integrated with standard LRU replacement. This method is denoted as (LRU, LRU) – normal data brought in on a cache miss as well as prefetched data replace old data based on the LRU policy.

(c): Figure 3-6(c) integrates Kill + LRU replacement with a generic prefetching method. Normal data brought in on a cache miss as well as prefetched data replaces old data based on the Kill + LRU replacement strategy described in Section 3.3.1. This method is denoted as (Kill + LRU, Kill + LRU).

(d): Figure 3-6(d) integrates Kill + LRU replacement with a generic prefetching method in a different way. Normal data brought in on a cache miss replaces old data based on the Kill + LRU policy, but prefetched data is only brought in if there is enough dead data. That is, prefetched data does not replace data which does not have its

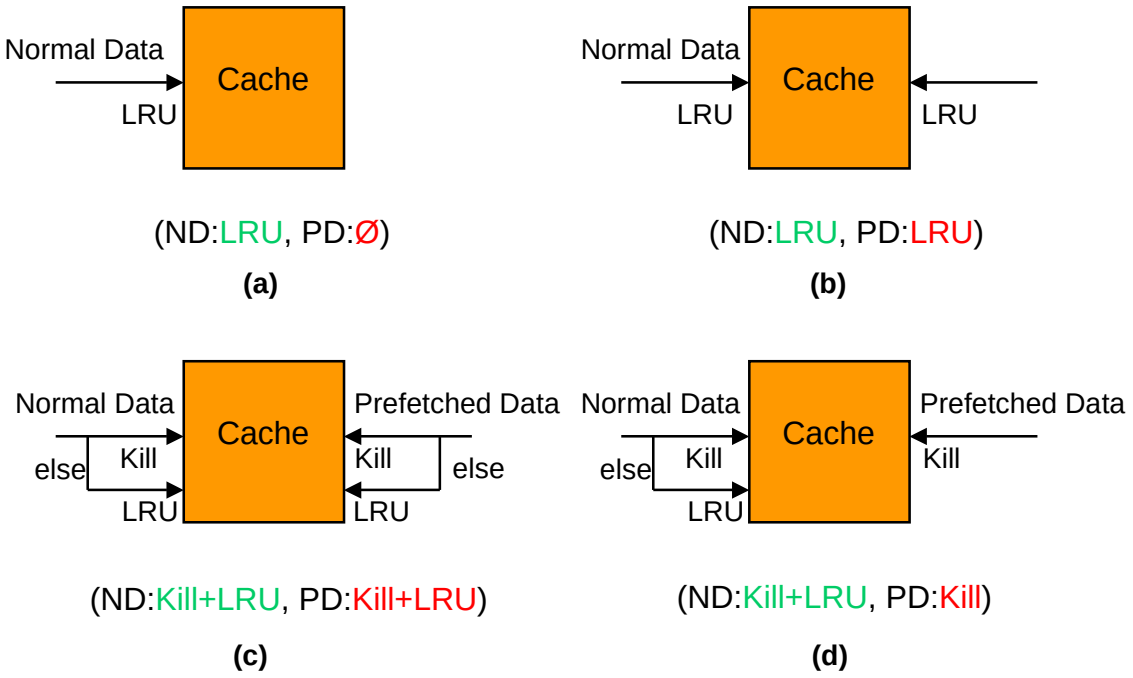


Figure 3-6: Integrating Prefetching with Modified LRU Replacement

kill bit set. Further, when the prefetched data replaces dead data, the LRU order of the prefetched data is *not* changed to MRU, rather it remains the same as that of the dead data. This method is denoted as (Kill + LRU, Kill).

There is one other variant that is possible, the (LRU, Kill + LRU) variant. I do not consider this variant since it does not have any interesting properties.

3.4 Theoretical Results

The theoretical results are presented for the replacement mechanisms that use the additional states to keep or kill the cache lines. The conditions are shown under which the replacement mechanisms with the kill and keep states are guaranteed to perform as good or better than the LRU policy. Also, the theorems for prefetching with LRU and Kill+LRU are presented here.

3.4.1 Kill+LRU Theorem

Definitions: For a fully-associative cache C with associativity m, the cache state is an ordered set of elements. Let the elements in the cache have a position number in the range

$[1, ..., m]$ that indicates the position of the element in the cache. Let $pos(e)$, $1 \leq pos(e) \leq m$ indicate the position of the element e in the ordered set. If $pos(e) = 1$, then e is the most recently used element in the cache. If $pos(e) = m$, then the element e is the least recently used element in the cache. Let $C(LRU, t)$ indicate the cache state C at time t when using the LRU replacement policy. Let $C(KIL, t)$ indicate the cache state C at time t when using the Kill + LRU policy. I assume the Kill + LRU policy variation (1) in the sequel. Let X and Y be sets of elements and let X_0 and Y_0 indicate the subsets of X and Y respectively with K_l bit reset. Let the relation $X \preceq_0 Y$ indicate that the $X_0 \subseteq Y_0$ and the order of common elements $(X_0 \cap Y_0)$ in X_0 and Y_0 is the same. Let X_1 and Y_1 indicate the subsets of X and Y respectively with K_l bit set. Let the relation $X \preceq_1 Y$ indicate that the $X_1 \subseteq Y_1$ and the order of common elements $(X_1 \cap Y_1)$ in X_1 and Y_1 is the same. Let d indicate the number of distinct elements between the access of an element e at time t_1 and the next access of the element e at time t_2.

I first give a simple condition that determines whether an element in the cache can be killed. This condition will serve as the basis for compiler analysis to determine variables that can be killed.

Lemma 1 *If the condition $d \geq m$ is satisfied, then the access of e at t_2 would result in a miss in the LRU policy.*

Proof: On every access to a distinct element, the element e moves by one position towards the LRU position m. So, after $m-1$ distinct element accesses, the element e reaches the LRU position m. At this time, the next distinct element access replaces e. Since $d \geq m$, the element e is replaced before its next access, therefore the access of e at time t_2 would result in a miss. ■

Lemma 2 *The set of elements with K_l bit set in $C(KIL, t) \preceq_1 C(LRU, t)$ at any time t.*

Proof: The proof is based on induction on the cache states $C(KIL, t)$ and $C(LRU, t)$. The intuition is that after every access to the cache (hit or miss), the new cache states $C(KIL, t+1)$ and $C(LRU, t+1)$ maintain the relation $\preceq_1$ for the elements with the K_l bit set.

At $t = 0$, $C(KIL, 0) \preceq_1 C(LRU, 0)$.

Assume that at time t, $C(KIL, t) \preceq_1 C(LRU, t)$.

At time $t+1$, let the element that is accessed be e.

Case H: The element e results in a hit in $C(KIL,t)$. If the K_l bit for e is set, then e is also an element of $C(LRU,t)$ from the assumption at time t. Now the K_l bit of e would be reset unless it is set by this access. Thus, I have $C(KIL,t+1) \preceq_1 C(LRU,t+1)$. If the K_l bit of e is 0, then there is no change in the order of elements with the K_l bit set. So, I have $C(KIL,t+1) \preceq_1 C(LRU,t+1)$.

Case M: The element e results in a miss in $C(KIL,t)$. Let y be the least recent element with the K_l bit set in $C(KIL,t)$. If e results in a miss in $C(LRU,t)$, let $C(KIL,t) = \{M_1, y, M_2\}$ and $C(LRU,t) = \{L, x\}$. M_2 has no element with K_l bit set. If the K_l bit of x is 0, $\{M_1, y\} \preceq_1 \{L\}$ implies $C(KIL,t+1) \preceq_1 C(LRU,t+1)$. If the K_l bit of x is set and $x = y$ then $\{M_1\} \preceq_1 \{L\}$ and that implies $C(KIL,t+1) \preceq_1 C(LRU,t+1)$. If the K_l bit of x is set and $x \neq y$, then $x \notin M_1$ because that violates the assumption at time t. Further, $y \in L$ from the assumption at time t and this implies $C(KIL,t+1) \preceq_1 C(LRU,t+1)$. ■

I show the proof of the theorem below for variation (1); the proof for variation (2) is similar.

Theorem 1 *For a fully associative cache with associativity m if the K_l bit for any element e is set upon an access at time t_1 only if the number of distinct elements d between the access at time t_1 and the next access of the element e at time t_2 is such that $d \geq m$, then the Kill + LRU policy variation (1) is as good as or better than LRU.*

Proof: The proof is based on induction on the cache states $C(LRU,t)$ and $C(KIL,t)$. The intuition is that after every access to the cache, the new cache states $C(LRU,t+1)$ and $C(KIL,t+1)$ maintain the $\preceq_0$ relation for the elements with the K_l bit reset. In addition, the new cache states $C(KIL,t+1)$ and $C(LRU,t+1)$ maintain the relation $\preceq_1$ based on Lemma 2. Therefore, every access hit in $C(LRU,t)$ implies a hit in $C(KIL,t)$ for any time t.

I consider the Kill + LRU policy variation (1) for a fully-associative cache C with associativity m. I show that $C(LRU,t) \preceq_0 C(KIL,t)$ at any time t.

At $t = 0$, $C(LRU,0) \preceq_0 C(KIL,0)$.

Assume that at time t, $C(LRU,t) \preceq_0 C(KIL,t)$.

At time $t+1$, let the element accessed is e.

Case 0: The element e results in a hit in $C(LRU,t)$. From the assumption at time t, e

results in a hit in $C(KIL, t)$ too. Let $C(LRU, t) = \{L_1, e, L_2\}$ and $C(KIL, t) = \{M_1, e, M_2\}$. From the assumption at time t, $L_1 \preceq_0 M_1$ and $L_2 \preceq_0 M_2$. From the definition of LRU and Kill + LRU replacement, $C(LRU, t+1) = \{e, L_1, L_2\}$ and $C(KIL, t+1) = \{e, M_1, M_2\}$. Since $\{L_1, L_2\} \preceq_0 \{M_1, M_2\}$, $C(LRU, t+1) \preceq_0 C(KIL, t+1)$.

Case 1: The element e results in a miss in $C(LRU, t)$, but a hit in $C(KIL, t)$. Let $C(LRU, t) = \{L, x\}$ and $C(KIL, t) = \{M_1, e, M_2\}$. From the assumption at time t, $\{L, x\} \preceq_0 \{M_1, e, M_2\}$ and it implies that $\{L\} \preceq_0 \{M_1, e, M_2\}$. Since $e \notin L$, I have $\{L\} \preceq_0 \{M_1, M_2\}$. From the definition of LRU and Kill + LRU replacement, $C(LRU, t+1) = \{e, L\}$ and $C(KIL, t+1) = \{e, M_1, M_2\}$. Since $L \preceq_0 \{M_1, M_2\}$, $C(LRU, t+1) \preceq_0 C(KIL, t+1)$.

Case 2: The element e results in a miss in $C(LRU, t)$ and a miss in $C(KIL, t)$ and there is no element with K_l bit set in $C(KIL, t)$. Let $C(LRU, t) = \{L, x\}$ and $C(KIL, t) = \{M, y\}$. From the assumption at time t, there are two possibilities: (a) $\{L, x\} \preceq_0 M$, or (b) $L \preceq_0 M$ and $x = y$. From the definition of LRU and Kill + LRU replacement, $C(LRU, t+1) = \{e, L\}$ and $C(KIL, t+1) = \{e, M\}$. Since $L \preceq_0 M$ for both sub-cases (a) and (b), I have $C(LRU, t+1) \preceq_0 C(KIL, t+1)$.

Case 3: The element e results in a miss in $C(LRU, t)$ and a miss in $C(KIL, t)$ and there is at least one element with the K_l bit set in $C(KIL, t)$. There are two sub-cases (a) there is an element with the K_l bit set in the LRU position, (b) there is no element with the K_l bit set in the LRU position. For sub-case (a), the argument is the same as in Case 2. For sub-case (b), let the LRU element with the K_l bit set be in position i, $1 \leq i < m$. Let $C(LRU, t) = \{L, x\}$ and $C(KIL, t) = \{M_1, y, M_2\}$, $M_2 \neq \phi$. From the assumption at time t, $\{L, x\} \preceq_0 \{M_1, y, M_2\}$, which implies $\{L\} \preceq_0 \{M_1, y, M_2\}$. Since y has the K_l bit set, $y \in L$ using Lemma 2. Let $\{L\} = \{L_1, y, L_2\}$. So, $\{L_1\} \preceq_0 \{M_1\}$ and $\{L_2\} \preceq_0 \{M_2\}$. Using Lemma 1, for the LRU policy y would be evicted from the cache before the next access of y. The next access of y would result in a miss using the LRU policy. So, $\{L_1, y, L_2\} \preceq_0 \{M_1, M_2\}$ when considering the elements that do not have have the K_l bit set. From the definition of LRU and Kill + LRU replacement, $C(LRU, t+1) = \{e, L_1, y, L_2\}$ and $C(KIL, t+1) = \{e, M_1, M_2\}$. Using the result at time t, I have $C(LRU, t+1) \preceq_0 C(KIL, t+1)$. ■

3.4.2 Kill+Keep+LRU Theorem

We now prove a theorem regarding the Flexible Keep policy.

Theorem 2 *(a) The* Flexible Keep *variation of the Kill + Keep + LRU policy is as good as or better than the LRU policy. (b) Whenever there is an element at the LRU position with the K_p bit set if there is also a different element with the K_l bit set, then the* Fixed Keep *variation of the Kill + Keep + LRU policy is as good as or better than LRU.*

Proof: I just give a sketch of the proof here, since the cases are similar to the ones in the Kill+LRU Theorem. I assume a fully-associative cache C with associativity m. Let $C(KKL, t)$ indicate the cache state C at time t when using the Kill+Keep+LRU policy. A different case from the Kill+LRU theorem is where the current access of an element e results in a miss in $C(KKL, t)$ and the element x at the LRU position has its K_p bit set.

Consider the *Flexible Keep* variation of the Kill+Keep+LRU policy. If there is no element with the K_l bit set, then the element x is replaced and the case is similar to the Kill+LRU policy. If there is at least one element with the K_l bit set in $C(KKL, t)$, let the most recent element with its K_l bit set is y. Let the cache state $C(KKL, t) = \{L_1, y, L_2, x\}$. The new state is $C(KKL, t+1) = \{e, L_1, x, L_2\}$. There is no change in the order of the elements in L_1 and L_2, so the relationship $C(LRU, t+1) \preceq_0 C(KKL, t+1)$ holds for the induction step. This implies the statement of Theorem 2(a).

Consider the *Fixed Keep* variation of the Kill+Keep+LRU policy. If there is at least one element with the K_l bit set in $C(KKL, t)$, let the most recent element with its K_l bit set be y. Let the cache state $C(KKL, t) = \{L_1, y, L_2, x\}$. The new state is $C(KKL, t+1) = \{e, L_1, x, L_2\}$. There is no change in the order of the elements in L_1 and L_2, so the relationship $C(LRU, t+1) \preceq_0 C(KKL, t+1)$ holds for the induction step. This implies the statement of Theorem 2(b). ■

Intuitively, the Flexible Keep theorem says that keep the data marked as Keep data whenever possible, i.e., if a Keep marked data would be replaced because it is at the LRU position, then a dead block can save the Keep data from being replaced. This is done by moving the Keep data at LRU position in a dead block's place and bringing a new block as in the LRU replacement policy. If there is no dead block available to save the Keep marked data from being replaced, then the Keep marked block is replaced. Both of these cases result in cache states that are supersets of the cache state for the LRU replacement.

In the Fixed Keep variation of the Kill+Keep+LRU the data that is marked as Keep data cannot be replaced by other blocks. If at any one instance in time, there is no dead block available for a Keep data at the LRU position then the least recent non-Keep data is

replaced. This instance can result in more misses in the Fixed Keep variation compared to the LRU policy. That is why the statement of the Fixed Keep Theorem in Section 3.4.2 has an extra condition, which is the weakest condition necessary for the Fixed Keep variation of Kill+Keep+LRU to have as good or better hit rate than the LRU replacement.

Set-Associative Caches

Theorem 1 and Theorem 2 can be generalized to set-associative caches.

Theorem 3 *For a set-associative cache with associativity m if the K_l bit for any element e mapping to a cache-set i is set upon an access at time t_1 only if the number of distinct elements d, mapping to the same cache-set i as e between the access at time t_1 and the next access of the element e at time t_2, is such that $d \geq m$, then the Kill+LRU policy variation (1) is as good as or better than LRU.*

Proof: Let the number of sets in a set-associative cache C be s. Every element maps to a particular cache set. After an access to the element e that maps to cache set i, the cache state for the cache sets 0 to $i-1$ and $i+1$ to $s-1$ remains unchanged. The cache set i is a fully-associative cache with associativity m. So, using Theorem 1, the Kill+LRU policy variation (1) is as good as or better than LRU for the cache set i. This implies the statement of Theorem 3. ■

Theorem 4 *(a) The* Flexible Keep *variation of the Kill + Keep + LRU policy is as good as or better than the LRU policy. (b) Whenever there is an element at the LRU position with the K_p bit set in a cache-set i, if there is a different element with the K_l bit set in the cache-set i, then the* Fixed Keep *variation of the Kill + Keep + LRU policy is as good as or better than LRU.*

Proof: I omit the proof of this theorem because it is similar to Theorem 3. ■

3.4.3 Kill+LRU with Prefetching Theorems

The combination of prefetching with LRU and Kill+LRU replacement as described in Section 3.3.3 have some properties and the theoretical results based on these properties are presented here. One property proven here is that (Kill + LRU, Kill + LRU) will perform as good or better than (LRU, LRU) if any prefetching strategy, which is not predicated on

cache hits or misses, is used, as long as it is the same in both cases. The other property is that (Kill + LRU, Kill) will perform as good or better than (LRU, ϕ) for any prefetching strategy. Note that (Kill + LRU, Kill + LRU) could actually perform worse than (LRU, ϕ) (Of course, if a good prefetch algorithm is used, it will usually perform better). Similarly, (Kill + LRU, Kill + LRU) could perform worse or better than (Kill + LRU, Kill).

Theorem 5 *Given a set-associative cache, Strategy (Kill + LRU, Kill + LRU) where Kill + LRU corresponds to variation (1) is as good as or better than Strategy (LRU, LRU) for any prefetch strategy not predicated on hits or misses in the cache.*

Proof: (Sketch) Consider a processor with a cache C_1 that runs Strategy (Kill + LRU, Kill + LRU), and a processor with a cache C_2 that runs Strategy (LRU, LRU). I have two different access streams, the normal access stream, and the prefetched access stream. Accesses in these two streams are interspersed to form a composite stream. The normal access stream is the same regardless of the replacement policy used. The prefetched access stream is also the same because even though the replacement strategy affects hits and misses in the cache, by the condition in the theorem, the prefetch requests are not affected. Therefore, in the (Kill + LRU, Kill + LRU) or the (LRU, LRU) case, accesses in composite stream encounter (Kill + LRU) replacement or LRU replacement, respectively. Since it does not matter whether the access is a normal access or a prefetch access, I can invoke Theorem 3 directly. ■

I can show that (Kill + LRU, Kill) is always as good or better than (LRU, ϕ).

Theorem 6 *Given a set-associative cache, the prefetch strategy (Kill + LRU, Kill) where the Kill + LRU policy corresponds to variation (1) is as good or better than the prefetch strategy (LRU, ϕ).*

Proof: (Sketch) The prefetch strategy (Kill + LRU, Kill) results in two access streams, the normal accesses and the prefetched blocks. The no-prefetch (LRU, ϕ) strategy has only the normal access stream. Invoking Theorem 3, the number of misses in the normal access stream is no more using Kill + LRU replacement than LRU replacement. The prefetch stream in the (Kill + LRU, Kill) strategy only affects the state of the cache by replacing a dead element with a prefetched element. When this replacement occurs, the LRU ordering of the dead item is not changed. If the element has been correctly killed, then no subsequent

access after the kill will result in a hit on the dead element before the element is evicted from the cache. A dead element may result in a hit using the Kill + LRU replacement due to the presence of multiple dead blocks, but the same block would result in a miss using the LRU replacement. Therefore, I can replace the dead element with a prefetched element without loss of hits/performance. It is possible that prefetched item may get a hit before it leaves the cache, in which case performance improves. ■

Chapter 4

Implementation of Intelligent Cache Replacement

4.1 Overview

In the previous chapter, the condition that determines the last access to a cache block is called a kill predicate and when the kill predicate for a cache block is satisfied, the cache block can be killed by marking it as a dead block. Two approaches to kill cache blocks are considered here: hardware-based and software-assisted. The hardware-based approach performs all the tasks related to building and using the kill predicates in hardware. The software-assisted approach uses compiler analysis or profile-based analysis to introduce additional instructions or annotations into the code that use the associated hardware to build and use the kill predicates.

4.2 Intuition behind Signatures

The hardware-based approach needs to build the kill predicates using some hardware support and use the predicates to kill blocks by marking them as dead blocks. Since the hardware does not have information about the future accesses, it uses some form of history information along with other information to form kill predicates. The history information consists of a sequence of accesses and this information is captured in the form of a signature. The signatures can be formed using load/store instruction PCs and/or additional information as described later. The history information in the form of signatures is used to predict

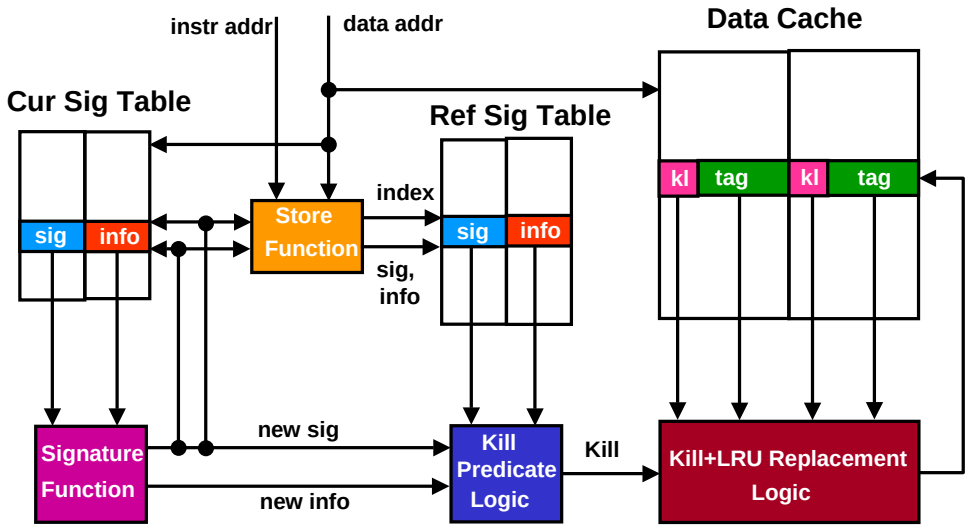


Figure 4-1: Hardware-based Kill+LRU Replacement

the dead blocks, i.e., a signature indicates when a block is dead and can be marked as a dead block. For example, consider a block b and the sequence T_x of PCs $= \{PC_1, PC_2, ..., PC_n\}$ ranging from the miss of block b (PC_1) and the last PC (PC_n) before the block b is replaced from the cache. Suppose T_x is captured in the form of a signature S_x. Now, if the block b is accessed in the future by a sequence T_y, the signature S_y corresponding the next sequence T_y is compared to S_x and if there is a match, then b is marked as a killed block. The success of history-based prediction relies on the repetition of access sequences of blocks that result in dead blocks. There is another way the history information for a block b_1 can be used to predict the dead block for block b_2. This way relies on the same history information based signatures for multiple blocks. Suppose the signature S_1 for the block b_1 indicates that the block b_1 is a dead block. If the signature for some other neighboring blocks $b_2, ..., b_k$ is the same as the signature S_1, then the signature S_1 can be used as the dead block signature for the neighboring blocks $b_2, ..., b_k$. So, even if the access sequence for a block (b_1) does not repeat, the history information from one block (S_1) can be applied to other blocks ($b_2, ..., b_k$).

4.3 Hardware-based Kill+LRU Replacement

In this approach, all the tasks related to computing and using the kill predicates are performed in hardware. The kill predicates are approximated in the form of signatures and

these signatures are matched against the current signatures associated with cache lines to kill the cache lines. The following steps are involved in using the hardware kill predicates based on signatures:

1. Compute Signatures: The signatures can be computed using different signature functions involving different types of information.

2. Store Signatures: The signatures can be stored such that they are associated with load/store instructions, data cache blocks, or data address ranges.

3. Lookup Signatures: The signatures are looked up in the signature table to determine if a cache line block can be killed or not.

The hardware support for hardware kill predicates using signatures is shown in Figure 4-1 and the pseudo-code for the hardware kill predicates algorithm based on signatures is shown in Figure 4-2. The algorithm for hardware kill predicates using signatures works as follows. When a data cache block b is brought into the cache, the current signature s associated with b is initialized. Upon every access to b, the signature s is updated using the chosen signature function. The current signature s is compared to the signatures corresponding to block b in the signature table T. If there is a match, then the block b is marked killed. When the block b is replaced, the signature entry corresponding to b in the signature table T is updated with the signature s, if the value does s not exist in T.

4.3.1 Signature Functions

Signature functions use a variety of information to build the signatures. A kill predicate may consist of other information in addition to a signature. The signature functions are:

- **XOR_LDST**: The PCs of the load/store instructions accessing a data cache block are XOR'ed together.

- **XOR_LDST_ROT**: The PC of a load/store instruction is rotated left by the number of accesses to the accessed cache block and it is XOR'ed with the current signature.

- **BR_HIST**: A path indicated by last n taken/not_taken committed branches is used as a signature. Though a signature based on load/store instructions implicitly includes the path information, a signature based on the branch history incorporates explicit path information.

Algorithm: Hardware Kill Predicates using Signatures

On a Cache Hit:
  Compute the new signature
  Update the signature in the data cache block
  Check the signature in the signature table
  if *found* **then**
    Mark the cache block killed
  end if

On a Cache Miss:
  Check the replaced block signature in the signature table
  if *not_found* **then**
    Replace the last entry for the block in the table
  end if
  Initialize the signature for the new block
  Check the new block's signature entry in the signature table
  if *found* **then**
    Mark the cache line killed
  end if

Figure 4-2: Signature-based Hardware Kill Predicates

- **TRUNC_ADD_LDST**: The PCs of the load/store instructions are added to form a signature. The result of the addition is truncated based on the number of bits chosen for the signature.

- **Signatures based on load/store Classification**: In this approach, different signatures parameters or functions are used for the load/stores based on the load/store classification. The instructions are classified into the different types either statically or dynamically. The classification helps because the signatures reflect the types of accesses associated with the instructions.

A kill predicate may consist of a signature, or some information, or a combination of a signature and some information. Some examples of kill predicates are: Signature, Number of Accesses, Number of Accesses and XOR_LDST, Number of Accesses and XOR_LDST and Cache Set, Number of Accesses and XOR_LDST_ROT, Number of Accesses and XOR_LDST_ROT and Cache Set.

4.3.2 Storing Signatures

There are different ways to store the signatures computed for the data cache blocks in the signature table.

- **Signatures associated with instructions**: In this approach, the signatures are stored in a signature table using partial load/store PCs as store/lookup indices. The partial load/store PC or a pointer to the load/store miss table that keeps the partial load/store PCs is kept as part of the data cache block and initialized when the data cache block is brought into the cache.

  The load/stores that miss in the data cache bring in a new data cache block into the data cache. These load/stores are a small percentage of load/store instructions. This approach works well when the load/store instructions that miss have a small number of distinct signatures for the different data cache blocks they access. So, it requires a small number of signatures for every partial load/store PC. The disadvantage of this approach is that the number of signatures associated with the load/store instructions may vary across the application.

- **Signatures associated with data blocks**: When there are many signatures for the load/store instructions that miss, it is better to store the signatures in the table using data block addresses as indices.

- **Signatures associated with data address ranges**: To take advantage of the similarity of the access patterns for some data address ranges, the signatures can be stored and looked up based on the data address ranges.

- **Signatures stored using Hash Functions**: Some hash functions can be used to store and access the signatures in the signature table. The hash functions distribute the signatures in the signature table to reduce the conflicts.

Using Reuse Distance

The replacement-to-miss distance information can be combined with other information such as a kill bit to make replacement decisions. The replacement-to-miss distance can be measured in hardware. The reuse distance captures some of the difference between the most-recently-killed Kill+LRU policy and the optimal replacement by allowing a choice in

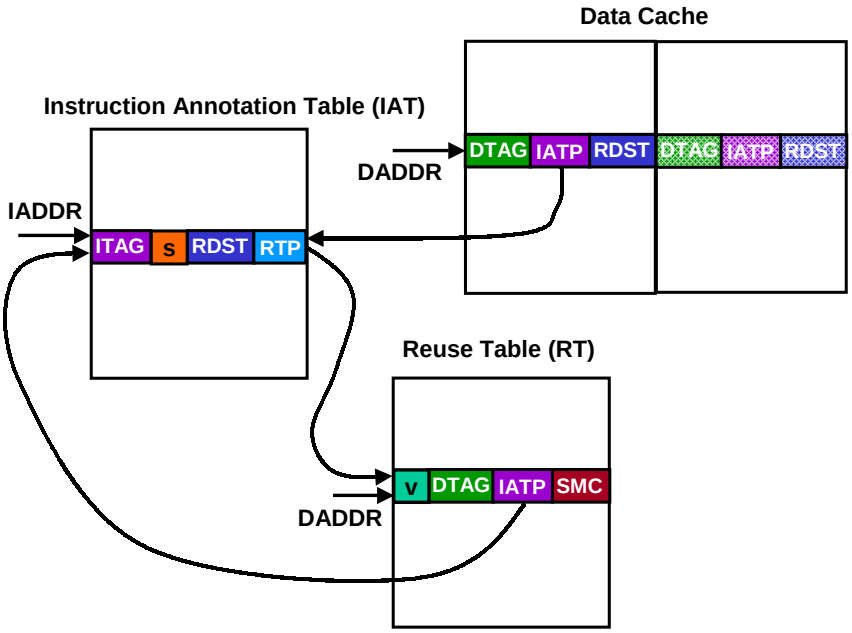


Figure 4-3: Hardware for Reuse Distance Measurement

selecting a killed block for replacement. With the use of reuse distance, the problem of stale killed blocks can be avoided because the killed blocks with the reuse distance greater than the most recently killed block would be selected for replacement before a most recently killed block. Figure 4-3 shows how the reuse distance (replacement-to-miss distance) can be measured in hardware. It uses an Instruction Annotation Table (IAT) and a Reuse Table (RT) to measure the replacement-to-miss distance for the instructions. An instruction annotation table (IAT) entry keeps a pointer to the replacement table (RT). The RT table stores the information about the reuse distance measurements under progress. A status bit `meas_reuse` with values `pending` and `idle` is used to indicate if the reuse distance measurement is in progress. Reuse distance is measured for one set at a time for a given load/store instruction. There is a set miss count (SMC) which keeps the number of misses per set. The number of bits used for the count limit the the maximum reuse distance that can be measured.

4.4 Software-assisted Kill+LRU Replacement

In this approach, a compiler analysis or profile-based analysis is done to determine the kill predicates and these kill predicates are incorporated into the generated code using some

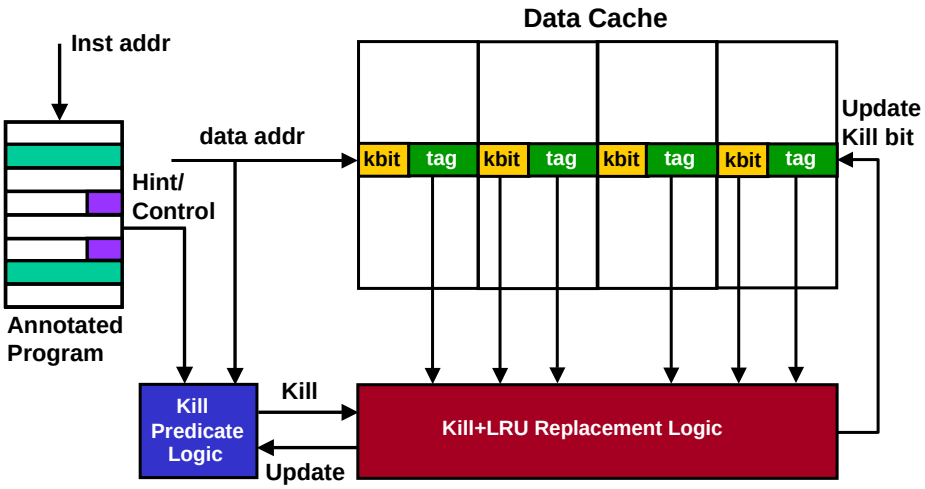


Figure 4-4: Software-assisted Kill+LRU Replacement Hardware

additional instructions. These additional instructions communicate the kill predicates to the associated hardware. The hardware may setup some registers and set the kill bits of the data cache blocks according to the specified kill predicates. The interaction of the hardware and the additional information introduced by the compiler is shown in Figure 4-4.

4.4.1 Kill Predicate Instructions

The kill predicate instructions have different software and hardware costs associated with them. Different kill predicate instructions are suited for different types of kill predicates. The choice of a kill predicate instruction to use at a particular point in the program is up to the compiler algorithm. The compiler algorithm uses the appropriate kill predicate instruction to handle the desired kill predicate.

The kill predicate instructions can be introduced anywhere in the code, but to reduce the overhead of these instructions, it is better to put them either before a loop or after a loop. The kill predicate instructions where the kill predicate is part of the load/store instruction do not introduce any execution time overhead. In order to reduce the execution time overhead, an annotation cache can be used as described later. The following kill predicate instructions are used by the compiler algorithm:

- LDST_KILL: A new load/store instruction that sets the kill bit on every access generated by the instruction. The kill predicate (set kill bit) is part of the instruction encoding.

- LDST_HINT_KPRED: There is a hint or condition attached with the load/store instruction. The hint may require an additional instruction word to specify this instruction. For example, the kill predicate can be count n which indicates that on every n^{th} access generated by the instruction, the kill bit of the accessed cache block should be set.

- LDST_CLS: This instruction conveys the instruction classification information that can help in determining the kill predicates. The class information indicates the type of accesses issued by the load/store instruction (e.g., stream accesses).

- PRGM_LDST_KPRED: Programmable kill predicate load/store instruction specifies the kill predicate and the load/store PC to attach the kill predicate. This instruction can be used before a loop to setup a hardware table that is accessed using the PC. Upon an access, the load/store PC is used to lookup and update the hardware table. This instruction is different from the above instructions because the kill predicate information is maintained in a separate table.

- LDST_AUTO_BLK_KPRED: This instruction sets the auto-kill bit in the cache blocks accessed by the kill predicate instruction. The kill predicate indicates the type of information that would be maintained with each data cache block and when the block is accessed. If the kill predicate is satisfied the block is marked killed.

- STC_DRNG_KPRED: Static kill address range instruction specifies the address range and a kill predicate that is used to set kill bits for the data cache blocks that fall within the range and satisfy the kill predicate. This instruction can be put either before a loop or after a loop.

- DYN_DRNG_KPRED: Dynamic kill address range information where the address range and a kill predicate are used to compare and set the kill bits. The range changes based on the condition specified as part of the instruction. This instruction is used in manner similar to the STC_DRNG_KPRED instruction.

Kill Range Implementation

The kill range hardware implementation is shown in Figure 4-5. A hardware table stores the information about the ranges of addresses and matches those addresses to the addresses

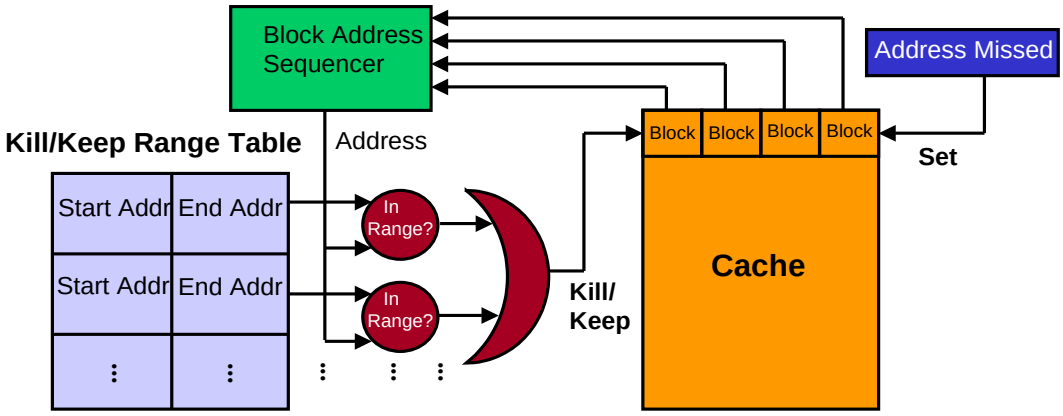


Figure 4-5: Kill Range Implementation Hardware

generated and declares the blocks killed based on the condition in the hardware table. The hardware table consists of the Start_Address and End_Address entries. The entries of the table can be updated in the FIFO manner or using some other policy as desired. When an address is provided for comparison, it is compared with the address ranges in parallel and the associated kill predicates. Upon an access to a set in the cache, if the address results in a miss, then the addresses corresponding to the tags in the set are provided to the hardware range table for sequential comparison and if any address falls within a range and satisfies the kill predicate, then the tag corresponding to that address is marked as the killed tag and this information is used during the replacement.

4.4.2 Profile-based Algorithm

The compiler algorithm looks at the data accesses in terms of the cache blocks. There may be multiple scalar variables assigned to the same block, but this information is generally not available at compile time. But, in some cases, it can be obtained using alignment analysis in the compiler. The compiler algorithm focuses on statically or dynamically allocated array variables and they are called kill candidate variables. Some group of scalar variables that are allocated in contiguous memory locations (e.g., local variables of a procedure) are also considered as kill candidate variables. In order to compute the reuse distance for a cache block, the number of distinct blocks mapping to the same set as the cache block is required. This requires information about the data layout, which the compiler does not typically have. This makes the computation of reuse distances infeasible. An alternative is to compute the total number of distinct blocks (*footprint*) between the two accesses of the

same block and if the footprint is above some threshold, then the first access of the cache block can be assigned a kill predicate instruction/annotation. The footprint computation requires the information about data sizes, values of input parameters that determine the control flow and the number of iterations, etc. The compiler can compute the footprint functions in terms of constants and input parameters and use a kill predicate instruction based on conditional expression at run-time. The steps for a compiler algorithm to identify and assign kill predicates are:

1. Perform the pointer analysis to identify the references that refer to the same variables. This analysis identifies memory references that would refer to the same cache blocks.

2. Perform life-time analysis on variables in the program, and identify the last use of all variables in the program. Associate an appropriate kill predicate instruction (e.g., STC_DRNG_KPRED) for these references.

3. If sufficient information is available to compute a reuse distance d between two temporally adjacent pair of references to the kill candidate variables, then determine the lower bound on the reuse distance d for each such pair.

   For a given pair of references to a kill candidate variable, the number of accesses to distinct blocks mapping to the same set is determined for each control path and a minimum value along these paths is chosen as the value d.

   For kill candidate variables, if any adjacent pair of references has $d \geq m$, where m is the associativity of the cache, associate a kill predicate instruction with the first reference.

4. Otherwise, compute the minimum footprint between each temporally-adjacent pair of references to the kill candidate variables. If the minimum footprint is greater than the data cache size, then the first reference is assigned a kill predicate instruction.

5. If the footprint function cannot be determined fully then a conditional expression is used to conditionally execute the kill predicate instruction at run-time. For example, a footprint function $f(N)$, where N is the number of iterations, and if $f(N) > CacheSize$ for $N \geq 100$, then a kill predicate instruction is executed conditionally based on the value of the conditional expression $N \geq 100$ at run-time.

The above compiler algorithm has not been implemented in a compiler. But a profile-based algorithm for the software-assisted approach uses some input data independent program information that can also be derived using the compiler analysis. The profile-based algorithm is inspired by the above algorithm and can approximate some aspects of the compiler algorithm using some hardware structures.

In the following profile-based algorithm, the static and data-dependent information about the program is derived using a profile run and used to obtain annotations for evaluation using simulation. The kill predicates are determined in hardware but with software assistance in the form of some annotations. The steps for the profile-based algorithm are:

- Run the target program on an input data set and collect input data independent information: type of load/store instruction, loop boundary markers, etc.
- Generate an annotation table for the program information to be incorporated into the simulator. The annotation table is a PC-based annotation table. Additional instructions are incorporated and used in simulation using PC-based entries in the annotation table.
- Simulate the program run on the same input data set or a different data set. Simulation uses the annotations to generate Kill predicates and updates the annotation table dynamically.

The profile-based algorithm uses some aspects of the compiler algorithm. Instead of static analysis, profile-based analysis is performed. Some of the static information is obtained or derived through the application runs on the test, train, and reference inputs. The profile information gathered is data-dependent or data-independent. The data-independent information is static in nature, e.g., load/store type, loop backward branch targets, loop end marker branches. The data-dependent information may vary with inputs, but can provide the information about the relative variations, e.g., reuse distance information for a block above some threshold, condition for marking cache blocks for replacement, last use of variables.

The information obtained using the profile-based analysis is used in deciding whether to use a hint with a load/store instruction or use a range-based instruction at the annotation points (loop boundaries). The decision to assign a hint to a load/store instruction is based

on how predictable the replacement condition is for the dynamic instances of the load/store instruction. If a load/store instruction is to be assigned a hint, then the type of hint is selected for the load/store instruction. The range instruction is chosen for the cases that cannot be captured by load/store hints and where range instruction can specify the condition for a group of load/store instructions. The profile-based annotations can lead to small overhead in terms of the static code size, fetch and issue bandwidth, and the instruction cache.

4.5 Keep with Kill+LRU Replacement

In order to use Keep with Kill+LRU replacement, the information about the Keep data blocks needs to be determined and conveyed to the hardware logic. The Keep information can be determined using compiler analysis or profiled-based analysis. The information can be conveyed to the hardware in the form of hints or additional instructions.

The compiler analysis that determines Keep data blocks can use three types of information: data types, data block usage patterns, and reuse distance information to determine the Keep variables. The data types and data block usage patterns may imply that it is beneficial to keep certain data blocks temporarily in the cache. A reuse distance threshold can also be used to help in determining the Keep variables. The information about the keep variables is conveyed to the hardware either as hints that are part of load/store instructions or Keep range instructions which specify the data address ranges for the Keep variables. The keep range instruction implementation is similar to kill range implementation as described in Section 4.4.1.

The Keep with Kill+LRU replacement has not been implemented, but the algorithm to determine the Keep data blocks can be implemented in a compiler along with the algorithm for the Kill+LRU replacement. The results presented later are based on the Keep variables being determined statically and conveyed to hardware using the annotations for the Keep data blocks.

Keep Range and Cache Line Locking

The use of a keep range instruction is better than cache line locking because in cache line locking the data space is reserved in the cache and less space is available to the other

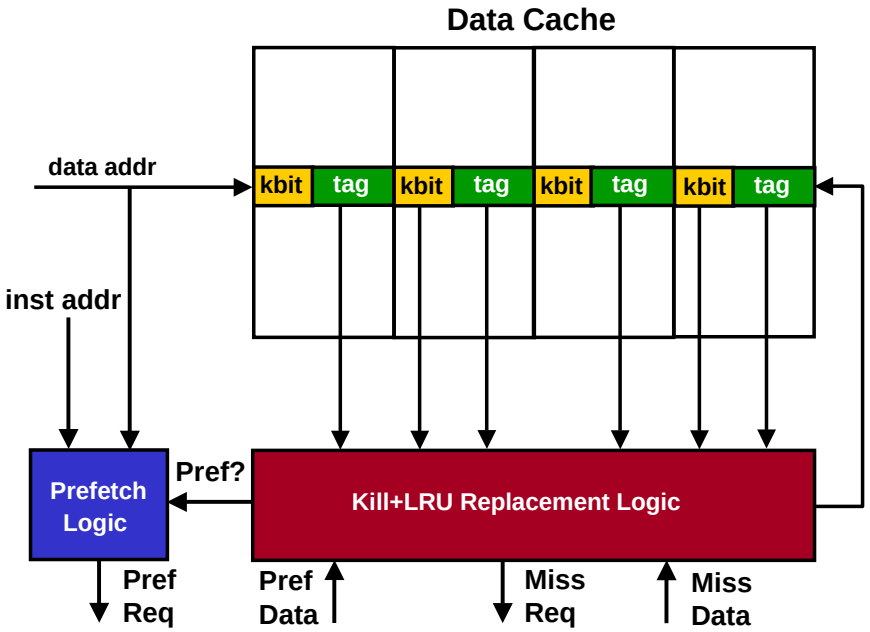


Figure 4-6: Prefetching with Kill+LRU Replacement

data. The cache space need to be released later to be used by other data. The keep range instruction allows for different data ranges to be specified in different phases based on the changing requirements of the different phases of an application.

4.6 Prefetching with Kill+LRU Replacement

Any prefetching scheme can be combined with the Kill+LRU replacement policy. The two ways to combine prefetching with the Kill+LRU replacement are considered here. One approach keeps the prefetching scheme independent of the Kill+LRU replacement by issuing the prefetches independent of the availability of killed blocks in the target cache set. But, the prefetched block replaces a killed block if available, otherwise it replaces the LRU block. The other approach controls prefetching by issuing prefetches only when there is killed data available and replacing only a killed block when the prefetched block is brought into the cache. If there is no killed block available when the prefetched block arrives, it is discarded. There are certain performance guarantees in terms of misses can be made for the two Kill+LRU and prefetching combinations. A hardware view of prefetching with Kill+LRU replacement is shown in Figure 4-6. A hardware prefetching scheme is considered in combination with the Kill+LRU replacement policy as described in the following section.

Hardware Prefetching

A tagged hardware prefetching scheme is used here. In the n-block lookahead tagged hardware prefetching scheme, upon a cache miss to a cache block b, n cache blocks following the block b are prefetched into the cache if they are not already in the cache. When the LRU replacement is used, the prefetched block replace the LRU cache block and the prefetched cache block becomes the MRU cache block. The last prefetched block is tagged, if it is not already in the cache. If there is a hit on a tagged block, the tag is removed from the block and the next block address is prefetched. We now give some details pertaining to the hardware implementation of the prefetching method. Generating the new addresses for the blocks is trivial. Before a prefetch request is made for a block, the cache is interrogated to see if the block is already present in the cache. This involves determining the set indexed by the address of the block to be prefetched. If this set contains the block the prefetch request is not made.

Predictor-based Prefetching

A hardware prefetching scheme that uses stride prediction can be used effectively in combination with the Kill+LRU replacement. The stride prefetching scheme considered here uses a hardware structure called *Reference Prediction Table* (RPT) to predict the strides in the data memory addresses accessed by the load/store instructions. When a load/store misses in the cache, and if the stride prediction is available, in addition to the missed block the next block based on the stride is prefetched into the cache.

The RPT maintains the access stride information about active load/store instructions. The RPT can be a direct-mapped table or a set-associative table indexed by load/store PC address information. Each RPT entry maintains the last address accessed by the PC corresponding to the RPT entry, stable stride flag, and stride value. Upon a data access, RPT is checked and if the load/store PC entry does not exist, then it is created with the last access address. When the same PC initiates another access, the last address is subtracted from the current address to determine the stride and it is stored in the RPT entry for the PC. When the same stride is computed the second time, it is considered stable and can be used in predicting the address for prefetching.

Prefetch Correlation with Kill+LRU Replacement

The use of hardware-based prefetching or predictor-based prefetching with Kill+LRU replacement decouples the prefetching approach from the Kill+LRU replacement. The prefetching of cache blocks can be correlated with the Kill+LRU replacement and can be implemented in the form of a Kill+Prefetch engine. This engine integrates the functions for the Kill+LRU and prefetching by correlating prefetches with the killed blocks or kill predicates. For example, when using the software-assisted approach, the kill predicate instruction PRGM_LDST_KPRED is used in combination with the prediction-based prefetching using the RPT. The RPT determines the prefetch address and the same table structure can be used to predict the dead blocks when the dead block information is associated with the load/store instructions. So, when a dead block condition is satisfied in an RPT entry for a load/store instruction, the same entry contains the information about the next prefetch address.

Chapter 5

Disjoint Sequences and Cache Partitioning

5.1 Introduction

Given two memory address sequences, they are *disjoint* if there is no common address between the two address sequences. Any address sequence can be considered as a merged address sequence of one or more disjoint address sequences. So, when an address sequence is applied to a cache, the whole cache space is shared by the elements of the applied address sequence.

In this chapter, I provide a theoretical basis for improving the cache performance of programs by applying the notion of disjoint sequences to cache partitioning. I prove theorems that show that if a program's memory reference stream can be reordered such that the reordered memory reference stream satisfies a disjointness property, then the reordered memory reference stream is guaranteed to have fewer misses for the cache so long as the cache uses the LRU replacement policy. To elaborate, if a memory reference stream R_{12} can be reordered into a concatenation of two memory reference streams R_1 and R_2 such that R_1 and R_2 are disjoint streams, i.e., no memory address in R_1 is in R_2 and vice versa, then the number of misses produced by R_{12} for *any* cache with LRU replacement is guaranteed to be greater than or equal to the number of misses produced by the stream $R_1@R_2$, where @ denotes concatenation. Thus, reordering to produce disjoint memory reference streams is guaranteed to improve performance, as measured by cache hit rate.

5.2 Theoretical Results

The theoretical results presented here form the basis of the cache partitioning approach explored in this thesis.

5.2.1 Definitions

Let R_1 and R_2 be two disjoint reference sequences merged without re-ordering to form a sequence S. That is, S is formed by interspersing elements of R_2 within R_1 (or vice versa) such that the order of elements of R_1 and R_2 within S is unchanged. For example, consider the sequence $R_1 = a, b, a, c$ and $R_2 = d, d, e, f$. These sequences are disjoint, i.e., they do not share any memory reference. The sequence S can be a, d, d, b, e, a, c, f but not a, d, d, a, b, c, e, f because the latter has reordered the elements of R_1.

Let $R_1^{'}$ be the R_1 sequence padded with the null element ϕ in the position where R_2 elements occur in S such that $\parallel S \parallel = \parallel R_1^{'} \parallel = N$. If $S = a, d, d, b, e, a, c, f$ then $R_1^{'} = a, \phi, \phi, b, \phi, a, c, \phi$. So, based on the above description, if $S(x) \in R_1$ then $S(x) = R_1^{'}(x)$. I define the null element ϕ to always result in a hit in the cache.

Let $C(S, t)$ be the cache state at time t for the sequence S. Let $C(R_1^{'}, t)$ be the cache state at time t for the sequence $R_1^{'}$. There are no duplicate elements in a cache state. Let the relation $X \preceq Y$ indicate that $X \subseteq Y$ and the order of the elements of X in Y is same as the order in X. For example, if $X = \{a, b, e\}$ and $Y = \{f, a, g, b, h, e\}$ then $X \preceq Y$ because $X \subseteq Y$ and the order of a, b, and e in Y is the same as the order in X.

For a direct-mapped cache C is an array; for a fully associative cache with an LRU policy C is an ordered set; and for a set-associative cache with the LRU policy C is an array of ordered sets.

For the following theorems in Section 5.2.2 and Section 5.2.3 it is assumed that the reference sequence element size is 1 word and the cache line size is also 1 word.

5.2.2 Disjoint Sequence Merge Theorem

Theorem 1: Given a cache C with an organization (direct mapped, fully-associative or set-associative) and the LRU replacement policy, and two disjoint reference sequences R_1 and R_2 merged without re-ordering to form a sequence S. The number of misses m_1 resulting from applying the sequence R_1 to $C \leq$ the number of misses M resulting from applying the

sequence S to C.

Sketch of the proof: Let $m_1^{'}$ be the number of misses of R_1 in S. Let $m_2^{'}$ be the number of misses of R_2 in S. So, $M = m_1^{'} + m_2^{'}$ is the total number of misses in S. For a direct-mapped cache I show by induction that for any time t, $0 \leq t \leq N$, $C(S,t) \cap R_1 \subseteq C(R_1^{'}, t)$. For a fully-associative cache I show by induction that for any time t, $0 \leq t \leq N$, $C(S,t) \cap R_1 \preceq C(R_1^{'}, t)$. For a set-associative cache I show by induction that for any time t, $0 \leq t \leq N$, $0 \leq i \leq p-1$, $C(S,t)[i] \cap R_1 \preceq C(R_1^{'}, t)[i]$, where p is the number of ordered sets in the set-associative cache. This implies that if an element of R_1 results in a miss for the sequence $R_1^{'}$ it would result in a miss in the sequence S. So, I have $m_1 \leq m_1^{'}$ and by symmetry $m_2 \leq m_2^{'}$. So, $m_1 + m_2 \leq m_1^{'} + m_2^{'}$ or $m_1 + m_2 \leq M$. Thus I have $m_1 \leq M$ and $m_2 \leq M$.

Direct Mapped Cache

I show that for any time t, $0 \leq t \leq N$, $C(S,t) \cap R_1 \subseteq C(R_1^{'}, t)$. Every element e maps to an index $Ind(e)$ that can be used to lookup the element in the cache state C.

For $t = 0$, $C(S,0) \cap R_1 \subseteq C(R_1^{'}, 0)$.
Assume for time t, $C(S,t) \cap R_1 \subseteq C(R_1^{'}, t)$.
For time t+1, let $f = S(t+1)$ and let $i = Ind(f)$ and let $e = C(S,t)[i]$ and let $x = C(R_1^{'}, t)[i]$.

Case 0 (hit): The element f results in a hit in $C(S,t)$. So, $e \equiv f$ and $C(S, t+1) = C(S,t)$. If $f \in R_1$, $R_1^{'}(t+1) \equiv f$ and from the assumption at time t, $C(R_1^{'}, t)[i] \equiv f$. Since the element f results in a hit in $C(R_1^{'}, t)$, $C(R_1^{'}, t+1) = C(R_1^{'}, t)$. So, $C(S, t+1) \cap R_1 \subseteq C(R_1^{'}, t+1)$. If $f \in R_2$, then $R_1^{'}(t+1) \equiv \phi$ and $C(R_1^{'}, t+1) = C(R_1^{'}, t)$. So, $C(S, t+1) \cap R_1 \subseteq C(R_1^{'}, t+1)$.

Case 1 (miss): The element f results in a miss in $C(S,t)$ and $f \in R_2$ and $e \in R_1$. The new cache state for the sequence S is $C(S, t+1) = C(S,t) - \{e\} \cup \{f\}$. This implies that $C(S, t+1) \cap R_1 \subseteq C(S,t) \cap R_1$. The element $R_1^{'}(t+1) \equiv \phi$ so $C(R_1^{'}, t+1) = C(R_1^{'}, t)$. So, $C(S, t+1) \cap R_1 \subseteq C(R_1^{'}, t+1)$.

Case 2 (miss): The element f results in a miss in $C(S,t)$ and $f \in R_2$ and $e \in R_2$. The new cache state for the sequence S is $C(S, t+1) = C(S,t) - \{e\} \cup \{f\}$. This implies that $C(S, t+1) \cap R_1 \equiv C(S,t) \cap R_1$. The element $R_1^{'}(t+1) \equiv \phi$ so $C(R_1^{'}, t+1) = C(R_1^{'}, t)$. So,

$C(S, t+1) \cap R_1 \subseteq C(R_1^{'}, t+1)$.

Case 3 (miss): The element f results in a miss in $C(S,t)$ and $f \in R_1$ and $e \in R_1$. The new cache state for the sequence S is $C(S, t+1) = C(S,t) - \{e\} \cup \{f\}$. The element $R_1^{'}(t+1) \equiv f$ from the construction of S and $R_1^{'}$. The new cache state for the sequence $R_1^{'}$ is $C(R_1^{'}, t+1) = C(R_1^{'}, t) - \{x\} \cup \{f\}$. From the assumption at time t $x \equiv e$. So, $C(S, t+1) \cap R_1 \subseteq C(R_1^{'}, t+1)$.

Case 4 (miss): The element f results in a miss in $C(S,t)$ and $f \in R_1$ and $e \in R_2$. The new cache state for the sequence S is $C(S, t+1) = C(S,t) - \{e\} \cup \{f\}$. The element $R_1^{'}(t+1) \equiv f$ from the construction of S and $R_1^{'}$. The new cache state for the sequence $R_1^{'}$ is $C(R_1^{'}, t+1) = C(R_1^{'}, t) - \{x\} \cup \{f\}$. Since $e \in R_2$ and the assumption at time t, $(C(S,t) - \{e\}) \cap R_1 \subseteq (C(R_1^{'}, t) - \{x\})$. So, $C(S, t+1) \cap R_1 \subseteq C(R_1^{'}, t+1)$.

Fully-associative Cache

Lemma 1: Let $C(S,t) = \{L, y\}$, where L is an ordered subset of elements and y is the LRU element of $C(S,t)$. Let $C(R_1^{'}, t) = \{M, z\}$, where M is an ordered subset of elements and z is the LRU element of $C(R_1^{'}, t)$. If $y \in R_2$, then $z \notin L$.

Proof: Based on the construction of $R_1^{'}$ and S, if M is not null, the last reference of z in both the sequences should occur at the same time $t_1 < t$ and between t_1 and t the number of distinct elements in $R_1^{'} \leq$ the number of distinct elements in S. If M is null, then there is only one element in C at time t because we have referenced z over and over, so t can be anything but $t_1 = t$ and the number of distinct elements following z in $R_1^{'} = 0$ as in S. Let $\parallel C \parallel = c$. For $C(R_1^{'}, t) = \{M, z\}$, let the number of distinct elements following z be n. Since z is the LRU element in $C(R_1^{'}, t)$, $n = c - 1$. Let us assume that $z \in L$. Let $L = \{L_1, z, L_2\}$ and $\parallel L_1 \parallel = l_1$, $\parallel L_2 \parallel = l_2$, $\parallel L \parallel = c - 1$. For $C(S,t) = \{L, y\} = \{L_1, z, L_2, y\}$, let the number of distinct elements following z be m and $m = l_1$. So, $m < c - 1$. So, $m < n$ and that contradicts the assertion on the number of distinct elements. Therefore, $z \notin L$.

I show that for any time t, $0 \leq t \leq N$, $C(S,t) \cap R_1 \preceq C(R_1^{'}, t)$.

For $t = 0$, $C(S,0) \cap R_1 \preceq C(R_1^{'}, 0)$.
Assume for time t, $C(S,t) \cap R_1 \preceq C(R_1^{'}, t)$.

For time $t+1$, let $x = S(t+1)$.

Case 0 (hit): $x \in R_1$ and x results in a hit in $C(S,t)$. Let $C(S,t) = \{L_1, x, L_2\}$, where L_1 and L_2 are subsets of ordered elements. The new state for the sequence S is $C(S,t+1) = \{x, L_1, L_2\}$. Let $C(R_1^{'},t) = \{M_1, x, M_2\}$, where M_1 and M_2 are subsets of ordered elements. $C(R_1^{'},t+1) = \{x, M_1, M_2\}$. From the assumption at time t, $\{L_1, x, L_2\} \cap R_1 \preceq \{M_1, x, M_2\}$. So, $\{L_1\} \cap R_1 \preceq \{M_1\}$ and $\{L_2\} \cap R_1 \preceq \{M_2\}$. Thus $\{L_1, L_2\} \cap R_1 \preceq \{M_1, M_2\}$. So, $C(S,t+1) \cap R_1 \preceq C(R_1^{'},t+1)$.

Case 1 (hit): $x \in R_2$ and x results in a hit in $C(S,t)$. Let $C(S,t) = \{L_1, x, L_2\}$, where L_1 and L_2 are subsets of ordered elements. The new state for the sequence S is $C(S,t+1) = \{x, L_1, L_2\}$. From the construction of $R_1^{'}$, $R_1^{'}(t+1) \equiv \phi$ and $C(R_1^{'},t+1) = C(R_1^{'},t)$. Since $x \in R_2$, $C(S,t+1) \cap R_1 \equiv C(S,t) \cap R_1$. So, $C(S,t+1) \cap R_1 \preceq C(R_1^{'},t+1)$.

Case 2 (miss): $x \in R_2$ and x results in a miss in $C(S,t)$ and the LRU element $y \in R_2$. Let $C(S,t) = \{L, y\}$, where L is a subset of ordered elements and y is the LRU element of $C(S,t)$. The new state for the sequence S is $C(S,t+1) = \{x, L\}$. Since $x \in R_2$ and $y \in R_2$, $C(S,t+1) \cap R_1 \equiv C(S,t) \cap R_1$. From the construction of $R_1^{'}$, $R_1^{'}(t+1) \equiv \phi$ and $C(R_1^{'},t+1) = C(R_1^{'},t)$. So, $C(S,t+1) \cap R_1 \preceq C(R_1^{'},t+1)$.

Case 3 (miss): $x \in R_2$ and x results in a miss in $C(S,t)$ and the LRU element $y \in R_1$. Let $C(S,t) = \{L, y\}$, where L is a subset of ordered elements and y is the LRU element of $C(S,t)$. The new state for the sequence S is $C(S,t+1) = \{x, L\}$. Since $x \in R_2$ and $y \in R_1$, $C(S,t+1) \cap R_1 \preceq C(S,t) \cap R_1$. From the construction of $R_1^{'}$, $R_1^{'}(t+1) \equiv \phi$ and $C(R_1^{'},t+1) = C(R_1^{'},t)$. So, $C(S,t+1) \cap R_1 \preceq C(R_1^{'},t+1)$.

Case 4 (miss): $x \in R_1$ and x results in a miss in $C(S,t)$ and the LRU element $y \in R_2$. Let $C(S,t) = \{L, y\}$, where L is a subset of ordered elements and y is the LRU element of $C(S,t)$. The new state for the sequence S is $C(S,t+1) = \{x, L\}$. From the construction of $R_1^{'}$, $R_1^{'}(t+1) \equiv x$. Let $C(R_1^{'},t) = \{M, z\}$, where M is a subset of ordered elements. The new state for the sequence $R_1^{'}$ is $C(R_1^{'},t+1) = \{x, M\}$. From Lemma 1, $z \notin L$. So, $\{x, L\} \cap R_1 \preceq \{x, M\}$. So, $C(S,t+1) \cap R_1 \preceq C(R_1^{'},t+1)$.

Case 5 (miss): $x \in R_1$ and x results in a miss in $C(S,t)$ and the LRU element $y \in R_1$. Let $C(S,t) = \{L, y\}$, where L is a subset of ordered elements and y is the LRU element of

$C(S,t)$. The new state for the sequence S is $C(S,t+1) = \{x, L\}$. From the construction of $R_1^{'}$, $R_1^{'}(t+1) \equiv x$. Since $y \in R_1$, the LRU element of $C(R_1^{'},t)$ is also y due to the assumption at time t. Let $C(R_1^{'},t) = \{M, y\}$, where M is a subset of ordered elements. The new state for the sequence $R_1^{'}$ is $C(R_1^{'},t+1) = \{x, M\}$. From the assumption at time t, $\{L\} \cap R_1 \preceq \{M\}$. Thus, $\{x, L\} \cap R_1 \preceq \{x, M\}$. So, $C(S,t+1) \cap R_1 \preceq C(R_1^{'},t+1)$.

Set-associative Cache

Assume that there are p ordered sets in C that can be referred as $C[0], ..., C[p-1]$. Every element e maps to an index $Ind(e)$ such that $0 \leq Ind(e) \leq p-1$ that is used to lookup the element in the ordered set $C[Ind(e)]$. I show for $0 \leq t \leq N$, $0 \leq i \leq p-1$, $C(S,t)[i] \cap R_1 \preceq C(R_1^{'},t)[i]$.

For $t = 0$, $0 \leq i \leq p-1$, $C(S,0)[i] \cap R_1 \preceq C(R_1^{'},0)[i]$.
Assume for time t, $0 \leq i \leq p-1$, $C(S,t)[i] \cap R_1 \preceq C(R_1^{'},t)[i]$.
For time $t+1$, let $x = S(t+1)$, and $j = Ind(x)$.
For $i \neq j$ ($0 \leq i \leq j-1$ and $j+1 \leq i \leq p-1$), $C(S,t+1)[i+1] = C(S,t)[i+1]$ and $C(R_1^{'},t+1)[i+1] = C(R_1^{'},t)[i+1]$ because the element x does not map in the ordered set $C[i]$. So, $C(S,t+1)[i] \cap R_1 \preceq C(R_1^{'},t+1)[i]$.
For $i = j$ using the assumption at time t and the proof for the fully-associative cache I have $C(S,t+1)[i] \cap R_1 \preceq C(R_1^{'},t+1)[i]$.
Therefore, $C(S,t+1)[i] \cap R_1 \preceq C(R_1^{'},t+1)[i]$ for $0 \leq i \leq p-1$.

5.2.3 Concatenation Theorem

Theorem 2: Given two disjoint reference sequences R_1 and R_2 merged in any arbitrary manner without re-ordering the elements of R_1 and R_2 to form a sequence S and given the concatenation of R_1 and R_2 indicated by $R_1@R_2$, the number of misses produced by S for *any* cache with the LRU replacement is greater than or equal to the number of misses produced by $R_1@R_2$.

Proof: Let m_1 indicate the number of misses produced by the sequence R_1 alone and m_2 indicate the number of misses produced by the sequence R_2 alone. The number of misses produced by $R_1@R_2$ is $m_1 + m_2$. Let M be the number of misses produced by S. I show that $m_1 + m_2 \leq M$. Let $m_1^{'}$ indicate the number of misses of R_1 in S and let $m_2^{'}$ indicate

the number of misses of R_2 in S. In Section 5.2.2 (Direct Mapped Cache) I showed for a direct-mapped cache that for any time t, $0 \leq t \leq N$, $C(S,t) \cap R_1 \subseteq C(R_1^{'},t)$ and in Section 5.2.2 (Fully-associative Cache) I showed for a fully-associative cache that for any time t, $0 \leq t \leq N$, $C(S,t) \cap R_1 \preceq C(R_1^{'},t)$. In Section 5.2.2 (Set-associative Cache) I showed for a set-associative cache by induction that for any time t, $0 \leq t \leq N$, $0 \leq i \leq p-1$, $C(S,t)[i] \cap R_1 \preceq C(R_1^{'},t)[i]$, where p is the number of ordered sets in the set-associative cache. This implies that if an element of R_1 results in a miss for the sequence $R_1^{'}$ it would result in a miss in the sequence S. Thus, $m_1 \leq m_1^{'}$. By symmetry, I have $m_2 \leq m_2^{'}$. So, $m_1 + m_2 \leq m_1^{'} + m_2^{'}$ or $m_1 + m_2 \leq M$, where $M = m_1^{'} + m_2^{'}$ is the total number of misses in S. Therefore, the number of misses M produced by S is greater than or equal to the number of misses $m_1 + m_2$ produced by $R_1@R_2$.

5.2.4 Effect of Memory Line Size

For the theorems above in Section 5.2.2 and Section 5.2.3 it was assumed that the reference sequence element size is 1 word and the cache line size is also 1 word. For a given line size, if the sequences are defined such that the elements of the sequences are on cache line boundaries, then the disjoint sequence theorem still holds. So, if the reference sequence element size is equal to the cache line size and the reference elements are aligned to the cache line boundaries, the above theorems still hold. For example, consider a cache line size of 4 words and the element size of 4 words (quad-word). If the elements are aligned on the cache line boundaries, the addresses of the sequence elements would consist of the cache line aligned addresses (e.g., `0xaaaabbb0`, `0xaaaaccc0`). This implies that any reordering of elements is on cache line boundaries and there is one to one correspondence between an element and a cache line. The effect is the same as that for the cache line size of 1 word and the element size of 1 word assumption of the disjoint sequences theorem. So, the disjoint sequences theorem would hold for the above example.

If the elements in sequences are aligned for a cache line size L, then the theorem holds for any line size $\leq L$. For example, if the elements are aligned for 4 words, the theorem would hold for cache line sizes 4, 2, and 1 word.

For an arbitrary cache line size, the disjoint sequence theorem cannot be applied because the sequences that are disjoint for one cache line size l may not be disjoint for a cache line size $l' > l$.

5.3 Partitioning with Disjoint Sequences

The notion of disjoint sequences can be applied to cache partitioning. In this approach, the cache space is divided into cache partitions and the address sequence is divided into disjoint set of groups. Each group consists of a merge sequence of one or more disjoint address sequences. One or more groups are mapped to each cache partition and the addresses belonging to a group can replace only those addresses belonging to the same group. Cache partitioning reduces the effective cache size available to different groups of disjoint sequences. The overall hit rate depends on the potential increase self-interference within the groups due to smaller cache space and decrease in the cross-interference from other groups of addresses. The hit rate guarantees as compared to the LRU replacement policy over the whole cache space hold under certain conditions as described in Section 5.3.2.

5.3.1 Non-Disjointness Example

Assume a two word cache C. Assume that the sequences have read accesses only. Consider a merged sequence S = {1.a 2.b 3.b 4.c 5.c 6.a 7.a 8.b}. Now consider the sequence divided into two sequences one going to cache C1 and the other going to cache C2, where C1 and C2 are same size caches as C. The results for various cases are shown in Table 5.1. If the divisions of the sequence are not disjoint, it results in total of 8 misses. On the other hand, if the sequence divisions S1 and S2 are disjoint and assigned to caches C1 and C2, it results in total of 3 misses. If the sequence S is applied to the cache C, it results in a total of 5 misses. Now, if C is partitioned into two one-word partitions, then it results in a total of 4 misses. So, the partitioning of the cache C based on disjoint sequences S1 and S2 saves one miss compared to the case of the sequence S being mapped to cache C.

5.3.2 Two Sequence Partitioning Example

Given two sequences, s1 and s2, with the number of accesses n1 and n2 let the number of misses of these sequences total m for cache size C. If used individually, the number of misses are m1 and m2. I have m1 + m2 $\leq$ m. Now if f is the fraction of the cache C assigned to S1, then the goal is to find f such that mfs1 + mfs2 $\leq$ m. That is, the sum of the number of misses mfs1 of s1 for the size f*C and number of misses mfs2 of s2 for the size (1-f)*C is $\leq$ m. If footprint(s1)+footprint(s2) $\leq$ C, then m1+m2=m. Otherwise, m1+m2 $\leq$ m and

Non-disjoint Sequences				
C1:	1.a	2.b	4.c	6.a
	miss	miss	miss	miss
C2:	3.b	5.c	7.a	8.b
	miss	miss	miss	miss
Total Misses = 8				

Disjoint Sequences {a} {b, c}					
C1:	1.a	6.a	7.a		
	miss	hit	hit		
C2:	2.b	3.b	4.c	5.c	8.b
	miss	hit	miss	hit	hit
Total Misses = 3					

Only One Cache With Two Words								
C:	1.a	2.b	3.b	4.c	5.c	6.a	7.a	8.b
	miss	miss	hit	miss	hit	miss	hit	miss
Total Misses = 5								

One Word Partition for Each Sequence					
Cp1:	1.a	6.a	7.a		
	miss	hit	hit		
Cp2:	2.b	3.b	4.c	5.c	8.b
	miss	hit	miss	hit	miss
Total Misses = 4					

Table 5.1: Non-disjoint Sequences Example

let m = m1+m2+mi, where mi are the interference misses. For a given partition, if mp1 = m1 + mi1 and mp2 = m2 + mi2, then the condition required is: mi1 + mi2 <= mi given the curves for both the sequences of number of misses vs cache size. For a solution to be feasible, there should be a cache size c1 and c2 such that mc1 < m1 + mi, mc2 < m2 + mi, and c1 + c2 <= C. Or the superimposed curves have a point in the range m1+m2 : m.

5.3.3 Partitioning Conditions

The condition for a partitioning based on disjoint sequences to be as good as or better than LRU is derived here. Assuming a Modified LRU replacement policy as a partitioning mechanism. Given a set of sequences $S_1, S_2, ..., S_n$, and a fully-associative cache C. Assume the cache partitions C_1 and C_2 where S_1 is assigned to C_1 and $S_2, ..., S_n$ are assigned to C_2. There will not be any additional misses in the partitioning based on C_1 and C_2, if the

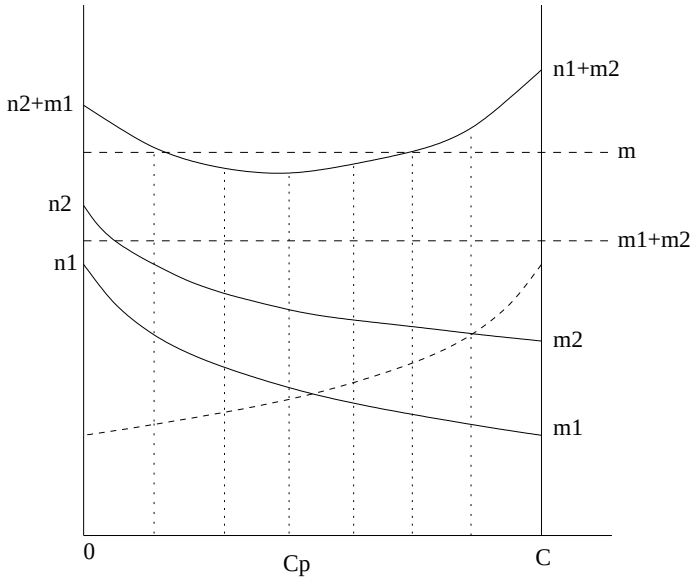


Figure 5-1: Two Sequence Partitioning

following statement holds for both the partitions:

Whenever a K_l bit can be set in C_1 or C_2 based on their sizes, it can also be set in C for S. This means that whenever an element is replaced from a partition, it would also be replaced in the case where the merged sequence was applied to the whole cache.

In terms of the reuse distances, whenever the reuse distance of an element in the sequence S_1 (S_2) is $\geq C_1$ ($\geq C_2$), then the reuse distance for the same element in S is $\geq C$. This means that whenever the reuse distance of an element is greater than the partition size, its reuse distance in the merged sequence is greater than the full cache size.

5.4 Information for Partitioning

There are different types of information that can be gathered and used in cache partitioning based on disjoint sequences. For example, the information about a sequence can be the footprint for the sequence, the cache usage efficiency for the sequence, or the reuse distance information for the sequence.

5.4.1 Cache Usage

The cache usage increases and the conflict misses reduce as the associativity increases for the same number of sets. There are different ways to measure the cache usage information

of the sequences, namely,

- H/(M*b), where H is the number of hits and M is the number of misses and b is the block size, or
- the percentage of words used in a cache line (H*(percentage of b)/(M*b)).

The cache usage information provides a metric to measure the effect of cache partitioning on the sequences' average use of the cache lines on a miss.

5.4.2 Sequence Footprints

The footprint size of a sequence gives an indication of the active cache lines of the sequence in the cache. Some profile information can be used as a footprint indicator to estimate the relative footprint sizes of the sequences.

5.4.3 Sequence Coloring

There are different ways to determine number of sequences for partitioning and the constituents of the sequences. The sequences may consist of different address ranges or load/store instructions. The sequences are combined into groups by sequence coloring. The groups are mapped to a cache partition. The information for sequence coloring consists of:

- usage pattern classes (they can be associated with the access method),
- random accesses with asymptotic footprint of certain size with certain usage efficiency, and
- stride-based accesses with usage efficiency less than some number and reuse distance more than some number.

The load/store instructions can be classified according to their cache line usage and strides of accesses. This load/store classification information gives an indication of the sharing possible. The load/store classification information can be obtained statically or dynamically.

Sequence Coloring Information

The cache lines are divided into groups and each group has some partitioning information associated with it. Some cache lines may not be part of any group. Each group is assigned a code that is used to maintain partitioning information and to make replacement decisions. There are three types of partitioning information considered for each group:

- **exclusive flag:** when this flag is set for a group g, it indicates that only the cache lines belonging to group g can replace a cache line in group g. This partitioning information requires some way of resetting this flag to release the cache lines from this restriction.
- **minimum size:** this value for a group specifies the minimum number of cache lines per set for that group. This requires some way of avoiding locking the whole cache.
- **maximum size:** this value for a group specifies the maximum number of cache lines per set for that group.

5.5 Static Partitioning

The cache partitioning should improve the hit rate compared to the unpartitioned cache using the LRU replacement policy. So, the goal is to reduce the misses for the groups of sequences. If the cost of self-interference is smaller than the cross interference then choose the self-interference that reduces the sharing among other sequences. The groups would get the cache partition size that is not going to increase their miss rate when compared to the case where the group was sharing the whole cache space with the other sequences.

5.5.1 Algorithm

Given a number of sequences, determine the grouping of sequences along with the partition sizes for the groups such that number of misses is minimized and the partition sizes add up to the cache size. The function *estimate_misses* uses the information about the sequences in the group, the interleaving of sequences, and the group partition size to estimate the misses. The algorithm partitions recursively assuming two partitions (2) and the sizes of the partitions are variable in the steps of the granularity of partitioning.

```
max_misses = INF
sol_g1 = {}
sol_g2 = {}

cluster_nodes(G, g1, g2) {
  if (G not empty) {
    nd = select_node(G); /* smallest sum of edge weights */
    g1' = assign_node(nd, g1);
    cluster_nodes(G-{nd}, g1', g2);
    g2' = assign_node(nd, g2);
    cluster_nodes(G-{nd}, g1, g2');
  }
  else {
    m_g1 = estimate_misses(g1);
    m_g2 = estimate_misses(g2);
    if (m_g1+m_g2 < max_misses) {
      sol_g1 = g1;
      sol_g2 = g2;
    }
  }
}
```

Figure 5-2: Clustering Algorithm

1. For each partition size pair, construct a Sequence Interaction Graph and Cluster the sequence nodes into two groups using the Clustering Algorithm shown in Figure 5-2.

2. Choose the partition size pair with the minimum estimated misses.

3. If a partition in the chosen pair contains more than one sequence, recursively partition the partitions in the pair.

Example

Consider six sequences `S1, S2, S3, S4, S5, S6`. Start with two partitions `1/4 C` and `3/4 C`.

```
a. {S1}                  -> P1 = (1/4 C),
   {S2, S3, S4, S5, S6} -> P2 (3/4 C)
b. {S2, S3}              -> P2_1 (1/4 C),
   {S4, S5, S6}          -> P2_2 (1/2 C)
```

Start with `1/2 C` and `1/2 C` and Clustering should lead to:

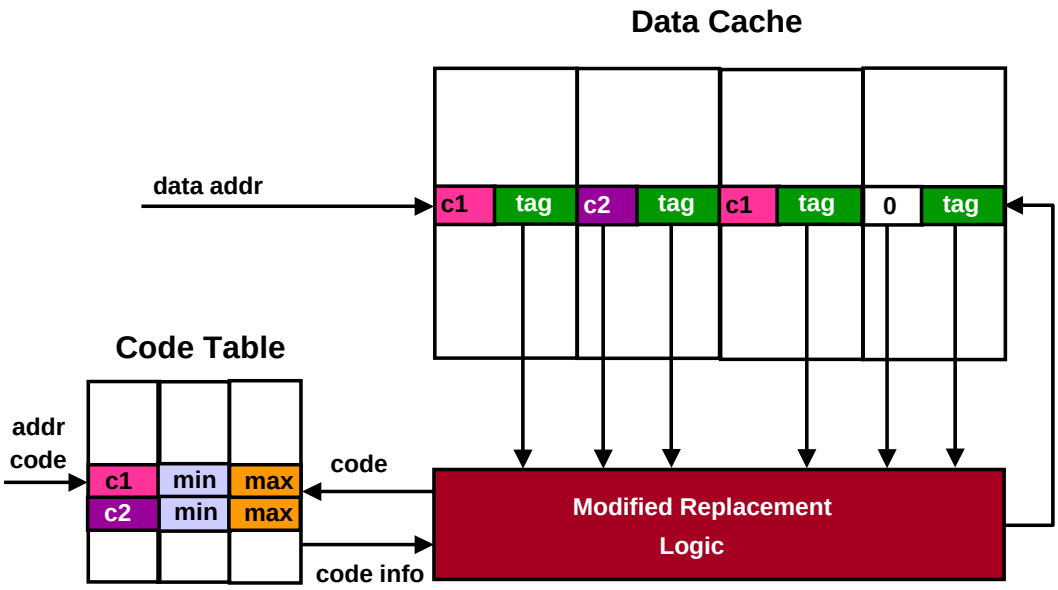


Figure 5-3: Hardware Support for Flexible Partitioning

```
a. {S1, S2, S3} -> P1 (1/2 C),
   {S4, S5, S6} -> P2 (1/2 C) then
b. {S1}         -> P1_1 (1/4 C),
   {S2, S3}     -> P1_2 {1/4 C},
   {S4, S5, S6} -> P2 (1/2 C)
```

5.5.2 Hardware Support

Modified LRU Replacement

The way-based partitioning is implemented using a modified LRU replacement algorithm. The replacement algorithm is a modified LRU replacement algorithm with the cache line group partitioning information. On a hit in a cache set, the cache lines in the set are reordered according to the LRU ordering of the cache lines in the accessed set. On a miss in a cache set, the partitioning information for the missed cache line group is checked from the group partitioning information table. If the minimum size is specified, and the number of cache lines belonging to the missed cache line's group are less than the minimum, then a cache line from other group is replaced. Otherwise, if the maximum size is specified and the current number of cache lines is equal to the maximum size, then the new cache line replaces the LRU cache line in the same group. Otherwise, a cache line from other group

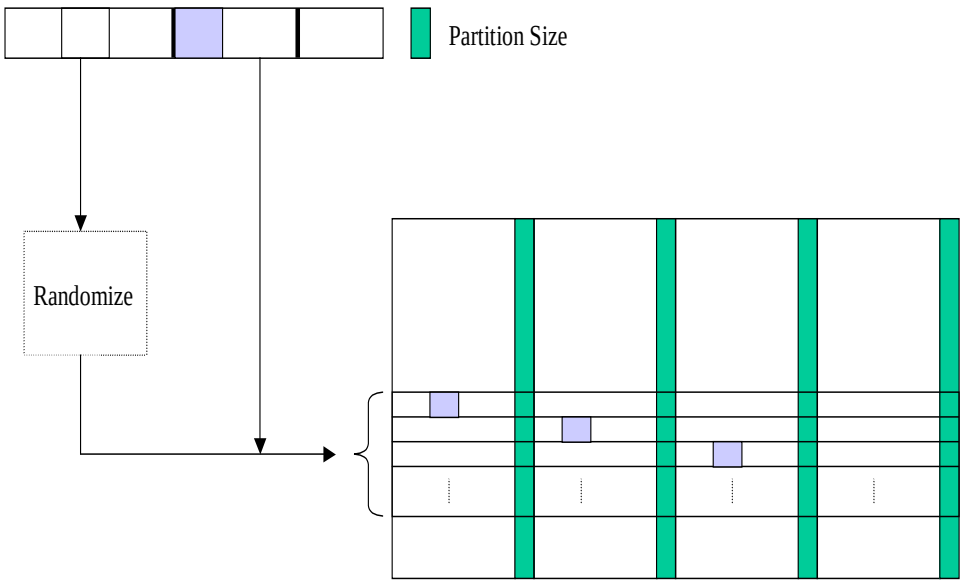


Figure 5-4: Cache Tiles for Cache Partitioning

is replaced. When a cache line from other group is chosen, the priority is to use the LRU cache line, without exclusive flag or minimum size violation.

There is a group partitioning information table which stores the information about the current groups. Also, each cache line has group code bits. The LRU replacement logic is modified to incorporate the changes from the LRU replacement algorithm. There are also some state bits to get the group of the cache line on an access and there are two ways to communicate it to the hardware: 1) using data address ranges or 2) load/store annotations. The hardware support for flexible cache partitioning is shown in Figure 5-3.

Cache Tiles

The above approach for partitioning is based on way-based horizontal partitioning, i.e., the partitions are controlled using the number of cache lines for different groups in a cache set. The other way of partitioning is set-based vertical partitioning where number of sets are used to control the partitioning for the cache lines of different groups. If both ways of controlling the partitions are used, then the partitions are called *cache tiles*.

One way to approximate the set-based vertical partitioning is based on the observation that the high order bits in the tag do not change much compared to the low order bits. One simple way to implement the set-based vertical partitioning is to swap some high order bits of the tag with high order bits of the index. For example, take the p tag bits (starting from

some fixed position) and use them as the upper p bits of the index to the cache and store the unused p index bits as part of the tag. The value p is stored for each cache line as well. When the cache lines are read, the tags are matched appropriately. To specify the partition size associated with an address, a load/store instruction with value p is used. The set-based partitioning is illustrated in Figure 5-4. This can be combined with way-based partitioning shown in Figure 5-3. However, there are several issues associated with this approach that need be dealt with to use this approach with disjoint sequences, and we will not explore this further in this thesis.

Multi-tag Sharing

The cache line usage in terms of the number of words in a cache line used before its eviction may vary for different groups. If the cache line usage is low, then the data words of a cache line can be shared by more than one cache line address. The sharing is achieved using multiple tags for the cache line data words. This is in a sense restricting the space allowed for the conflicting cache line addresses that would not use all the space in the cache line. A multi-tag based sharing of data word space of cache lines can be used to achieve sharing among sequences. This approach may use the ideas of *rotating tags*, *differential tags*, and *distinct tags* to reduce the multi-tag overhead and to maintain the same latency of accesses. Some details of these ideas are given in Section A.3

5.5.3 Software Support

The groups for partitioning need to be defined along with the partitioning information. I propose to use disjoint sequences to define the groups and the associated partitioning information.

First, sequences need to be defined that would represent different groups. This can be done using compiler analysis that identifies the variables with disjoint address ranges. A collection of variables can be used as a group and when this group is accessed in the program, it will form a sequence that is disjoint from other sequences.

Second, the partition size information needs to be determined for different groups. This information can be derived using the footprint information and cache usage efficiency for different groups and they can be estimated either by compiler or profile analysis. The size of the partition that is not going to reduce the hit rate is given by the number of live blocks

in the sequence. A group should be assigned the partition size that is not going to increase its miss rate compared to the case where the group were to share the space with the other groups. The miss rate would not increase, if the cost of self-interference is less than the cost of cross-interference in terms of misses.

The above group definition and partition size assignment would approximate the result of the disjoint sequence theorem. This derived information is incorporated into the program using appropriate instructions or annotations to communicate this information to the hardware support logic.

5.6 Hardware Cost

Each cache line will require the number of bits to indicate the code of the cache line. Also, a table of the maximum number of cache lines for a given code. The replacement logic would have to be modified to incorporate the changes from the LRU policy. For n disjoint sequences, $\log_2 n$ bits would be required to indicate the disjoint sequence for a cache line. If there are m cache lines in the cache, then it would require $m \times (\log_2 n)$ additional bits for the whole cache. There are two ways the disjoint sequence of an access can be conveyed to the hardware. One way is that the disjoint sequence information is part of the load/store instructions. In this case, each load/store instruction would require $\log_2 n$ bits to indicate the disjoint sequence. The other way is that the disjoint sequences are indicated as disjoint address ranges. This would require $n \times 64$ bits to store start and end address of for each range. The partitioning table requires $n \times ((\log_2 a) + 1)$ bits for the partitioning information, where a is the associativity of the cache. The number of bits required to store the disjoint sequences information can be reduced using the number of live sequences information. The live sequences information allows the disjoint sequence numbers to be recycled as the program working set changes during the program execution. The live sequences information can be derived using a program analysis. The modification of the replacement logic would require some $\log_2 a$-bit adders and $\log_2 n$-bit comparators and some small logic to read and write the partitioning information table.

For example, consider a 4-way 8K set associative cache with block size of 32 Bytes and 8 disjoint sequences for partitioning. There would be 3 bits of additional state bits with each cache line, so there would be total $256 \times 3 = 768$ additional bits for the whole cache.

The partition information table would require $8 \times 3 = 24$ bits to store the partitioning information. If the disjoint address ranges are used, the address range information would require $8 \times 64 = 2048$ bits, otherwise 3 bits with each load/store instruction would be required to convey the disjoint sequence information.

Chapter 6

Evaluation

In this chapter, we empirically evaluate the techniques presented in the previous chapters. We first evaluate the intelligent cache replacement mechanism in Section 6.1. Next, in Section 6.2, we evaluate how intelligent cache replacement can be used to control cache pollution arising from an aggressive sequential hardware prefetching method. Finally, in Section 6.3, we evaluate static cache partitioning schemes based on a strategy inspired by the disjoint sequence theorem.

6.1 Intelligent Cache Replacement

For our experiments, we compiled some Spec95 benchmarks for a MIPS-like PISA processor instruction set used by the Simplescalar 3.0 [11] tool set. We generated traces for the benchmarks using Simplescalar 3.0 [11] and chose sub-traces (instruction + data) from the middle of the generated trace. We used a hierarchical cache simulator, *hiercache*, to simulate the trace assuming an L1 cache and memory. In our experiments, we measured the L1 hit rate and the performance of some of the Spec95 benchmarks for various replacement policies.

We describe our experiments using the Spec95 Swim benchmark as an example. We chose a set of arrays as candidates for the kill and keep related experiments. The arrays we considered were *u, v, p, unew, vnew, pnew, uold, vold, pold, cu, cv, z, h, psi*. These variables constitute 29.15% of the total accesses. We did the experiment with different associativities of $2, 4, 8$ and cache line sizes of $2, 4, 8$ words and a cache size 16K bytes. The overall L1 hit rate results for the Swim benchmark are shown in Figure 6-1.

In Figure 6-1, the x-axis *a,b* indicates the associativity and the cache line size in words. The column labeled *LRU* shows the hit rate over all accesses (not just the array accesses) with the LRU policy. The column labeled *Kill* shows the hit rate for the Kill+LRU replacement policy. The columns labeled *KK1, KK2, KK3* show the hit rate for the Kill+Keep+LRU replacement policy with the *Flexible Keep* variation. In *KK1*, the array variables *unew, vnew, pnew* are chosen as the keep candidates. In *KK2*, the array variables *uold, vold, pold* are chosen as the keep candidates. In *KK3*, only the array variable *unew* is chosen as the keep candidate. The hit rates of the variables of interest for an associativity of 4 and a cache line size of 8 words are shown in Figure 6-2 for the same columns as described above. The modified program with cache control instructions does not have any more instruction or data accesses than the original program. In Figure 6-2, the columns *%Imprv* show the percentage improvement in hit rate for the variables over the LRU policy.

Figure 6-3 shows the number of Kill (labeled as #Kill) and Conditional Kill (labeled as #Cond Kill) instructions generated corresponding the number of references (labeled as #Ref) for the array variables of the Spec95 Swim benchmark.

The results show that the performance improves in some cases with the use of our software-assisted replacement mechanisms that use kill and keep instructions. The results in Figure 6-2 show that the hit rates associated with particular variables can be improved very significantly using our method. The bold numbers in the `KK1, KK2,` and `KK3` columns in Figure 6-2 indicate the hit rate of the variables that were the only keep variables for these columns. Choosing a particular variable and applying our method can result in an substantial improvement in hit rate and therefore performance for the code fragments where the variable is accessed. For example, for a variable `vnew`, the hit rate for LRU was 37.37, but we could improve it to 75.86 using the Keep method. This is particularly relevant when we need to meet real-time deadlines in embedded processor systems across code fragments, rather than optimizing performance across the entire program.

Figure 6-4(a) shows the overall hit rate and performance for some Spec95 benchmarks. We show hit rate and performance for a single-issue pipelined processor that stalls on a cache miss. The number of cycles are calculated by assuming 1 cycle for instruction accesses and 1 cycle for on-chip memory and a 10 cycle latency for off-chip memory. The columns show hit rate and number of cycles assuming 10 cycles for off-chip memory access and 1 cycle for on-

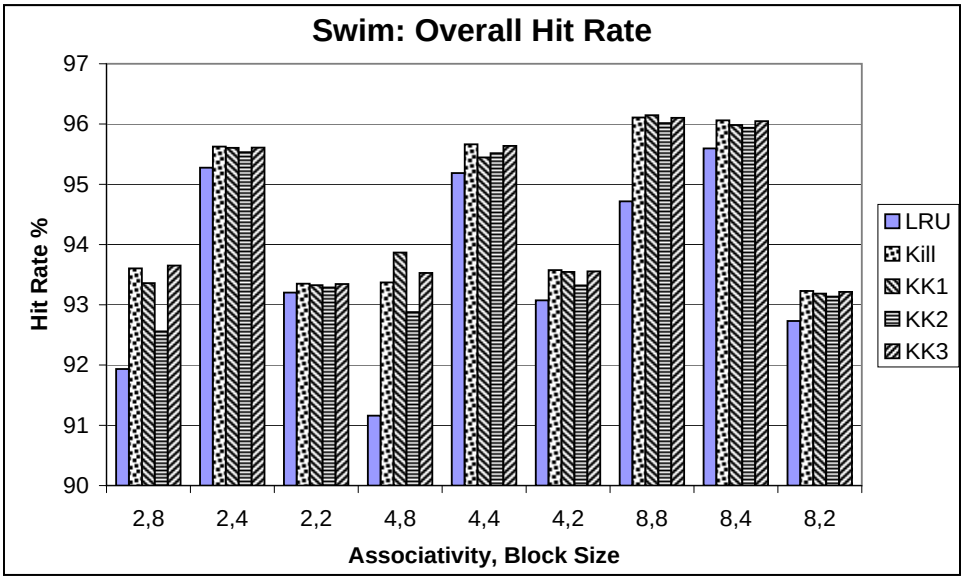


Figure 6-1: Overall Hit Rates for the Spec95 Swim Benchmark (L1 Cache Size 16 KBytes)

chip memory access. The last column shows the performance improvement of Kill+Keep over LRU. Figure 6-4(b) shows the overall worst-case hit rate and performance for the same Spec95 benchmarks. The worst-case hit rate is measured over 10 sets of input data. The columns show hit rate and number of cycles assuming 10 cycles for off-chip memory access and 1 cycle for on-chip memory access. The last column shows the performance improvement of Kill+Keep over LRU.

The programs that do not have much temporal reuse of its data (e.g., some integer benchmarks) do not benefit from kill+LRU replacement in terms of the hit rate improvement, but if the same programs have some data that can benefit by keeping some variables in the cache, then the Kill+Keep strategy can help in improving the hit rates for the keep variables without degrading performance.

6.2 Intelligent Cache Replacement and Prefetching

We first describe the hardware prefetch method we use in this section. Next, we present results integrating trace-based ideal kill with the prefetch method in various ways. Finally, we present results using compiler-inserted kills, and show that significant performance improvements can be gained while providing performance guarantees.

Vars	LRU	Kill	%Imprv	KK1	%Imprv	KK2	%Imprv	KK3	%Imprv
u	85.94	88.57	3.06	86.47	0.62	86.32	0.44	86.67	0.84
v	83.13	88.13	6.01	84.37	1.49	84.20	1.28	86.86	4.49
p	84.24	87.93	4.37	85.42	1.39	84.27	0.03	86.75	2.97
unew	28.67	39.72	38.56	**74.16**	158.68	28.80	0.47	**84.77**	195.71
vnew	37.37	47.17	26.21	**75.86**	102.97	42.83	14.58	43.97	17.64
pnew	31.15	55.51	78.22	**75.28**	141.68	47.31	51.88	48.23	54.84
uold	42.62	50.78	19.15	47.59	11.67	**67.74**	58.95	47.59	11.66
vold	54.92	62.60	13.99	62.44	13.70	**75.05**	36.66	62.48	13.77
pold	47.35	59.25	25.13	55.55	6.41	**71.41**	33.56	55.71	6.65
cu	75.81	79.73	5.54	79.35	5.11	77.96	3.19	79.73	5.54
cv	75.75	82.15	10.28	82.15	10.21	81.45	7.58	82.15	10.28
z	73.81	84.57	14.85	84.50	14.85	78.26	6.12	84.57	14.85
h	74.18	85.53	18.38	85.53	18.38	83.16	12.28	85.53	18.38
psi	92.38	92.84	0.49	92.84	0.49	92.84	0.49	92.84	0.49

Figure 6-2: Hit Rates for the array variables in the Spec95 Swim Benchmark for L1 cache size 16 KB, associativity 4, and cache line size 8 words. The bold numbers in the `KK1`, `KK2`, and `KK3` columns indicate the hit rate of the keep variables for these columns.

Vars	#Ref	#Kill	#Cond Kill
u	28	6	10
v	28	5	12
p	24	2	11
unew	13	6	6
vnew	13	4	8
pnew	13	4	8
uold	13	5	7
vold	13	5	7
pold	13	5	7
cu	15	4	7
cv	15	2	10
z	13	5	5
h	13	4	5
psi	5	0	2

Figure 6-3: Number of Kill Instructions for the array variables in the Spec95 Swim benchmark for L1 cache size 16 KB, associativity 4, and cache line size 8 words

Bench-mark	LRU Hit %	LRU Cycles	KK Hit %	KK Cycles	% Imprv Cycles
tomcatv	94.05	1105082720	95.35	1014793630	8.90
applu	97.88	113204089	97.89	113171529	0.03
swim	91.16	12795379	93.52	11188059	14.37
mswim	95.01	10627767	96.30	9715267	9.39

(a)

LRU Hit %	LRU Cycles	KK Hit %	KK Cycles	% Imprv Cycles
93.96	122920088	95.17	113604798	8.20
97.80	635339200	97.80	635200200	0.02
90.82	100394210	93.19	88016560	14.06
94.92	81969394	96.15	75290924	8.87

(b)

Figure 6-4: Overall Hit Rates and Performance for benchmarks: (a) For a given input (b) Worst case. L1 cache size 16 KB, associativity 4, and cache line size 8 words. We assume off-chip memory access requires 10 processor clock cycles, as compared to a single cycle to access the on-chip cache.

6.2.1 Hardware Prefetch Method

We consider a parameterizable variation on a sequential hardware prefetch method. We integrate this method with Kill + LRU replacement using the strategies of Figure 3-6(c) and 3-6(d).

Upon a cache miss, a block has to be brought into the cache. This block corresponds to new, normal data being brought in. Simultaneously, we prefetch i adjacent blocks and bring them into the cache, if they are not already in the cache. If i is 2, and the block address of the block being brought in is A, then we will prefetch the $A+1$ and $A+2$ blocks. The i^{th} block will be tagged, if it is not already in the cache. This first group of $i+1$ blocks will either replace dead cache blocks or LRU blocks, in that order of preference.

We conditionally prefetch j more blocks corresponding to blocks with addresses $A+i+1,\ \ldots A+i+j$, provided these blocks only replace cache blocks that are dead. This check is performed *before* prefetching.

If there is a hit on a tagged block, we remove the tag from the block, and prefetch the two groups of i and j blocks, as before. The i^{th} and last block in the first group will be

tagged, if it is not already in the cache. We denote this parameterizable method as ((Kill + LRU)$-i$, (Kill)$-j$).

We now give some details pertaining to the hardware implementation of the prefetching method. We have already described how the modified replacement strategy is implemented in Section 4.1. Generating the new addresses for the blocks is trivial. To control the required bandwidth for prefetching, we will only prefetch the second[1] group of j blocks if these blocks will only replace dead cache blocks. Before a prefetch request is made for a block, the cache is interrogated to see if the block is already present in the cache. This involves determining the set indexed by the address of the block to be prefetched. If this set contains the block the prefetch request is not made. Further, for the second group of j blocks, if the set does not contain at least one dead block, the prefetch request is not made. Given the set, it is easy to perform this last check, by scanning the appropriate kill bits.

The bandwidth from the next level of cache to the cache that is issuing the prefetch requests is an important consideration. There might not be enough bandwidth to support a large degree of prefetching ($i+j$ in our case). Limited bandwidth will result in prefetches either not being serviced, or the prefetched data will come in too late to be useful. Checking to see that there is dead space in the second set of j blocks reduces the required bandwidth, and ensures no cache pollution by the second set of prefetched blocks. Of course, for all prefetched blocks we check if there is dead data that can be replaced, before replacing potentially useful data, thereby reducing cache pollution.

6.2.2 Experiments With Ideal Kill

We present results using trace-based ideal kill, where any cache element that will not be accessed before being replaced is killed immediately after the last access to the element that precedes replacement. These results give an upper bound on the performance improvement possible using integrated kill and prefetch methods.

We compare the following replacement and prefetch strategies:

- (LRU, LRU$-i$): Standard LRU with i-block prefetching. When $i = 0$ there is no prefetching done.
- ((Ideal Kill + LRU)$-i$, (Ideal Kill)$-j$): The first block and the first i prefetched

[1]This will be the second group assuming $i > 0$.

blocks will replace either a dead block or an LRU block, but the next j prefetched blocks will only be brought into the cache if they are to replace dead blocks. (cf. Figure 3-6(c) and (d)).

We cannot prove that ((Ideal Kill + LRU)$-i$, (Ideal Kill)$-j$) is better than (LRU, LRU$-i$) for arbitrary i and j because Theorem 5 of Chapter 3 does not hold. For $i = 0$ and arbitrary j Theorem 6 holds. However, it is very unlikely that the former performs worse than the latter for any i, j. Kill + LRU can only result in fewer misses than LRU on normal accesses, and therefore fewer prefetch requests are made in the Kill + LRU scheme, which in turn might result in a miss for Kill + LRU and a hit for LRU later on. Of course, a future miss will result in a similar, but not necessarily identical, prefetch request from Kill + LRU. However, since the prefetch requests in the two strategies have slightly different timing and block groupings, a convoluted scenario might result in the total number of misses for ((Ideal Kill + LRU)$-i$, (Ideal Kill)$-j$) being more than (LRU, LRU$-i$). Our experiments indicate that this does not occur in practice.

We show experimental results for some of the integer and the floating-point benchmarks from the Spec95 (`tomcatv`) and Spec2K (`art`, `bzip`, `gcc`, `mcf`, `swim`, `twolf`, `vpr`) benchmark suites. In Figures 6-5 through 6-12, we show (on the left) the hit rates achieved by ((Ideal Kill + LRU)$-i$, (Ideal Kill)$-j$) and (LRU, LRU$-i$) for $0 \leq i \leq 3$ and $0 \leq i + j \leq 7$. These hit rates correspond to a 4-way set-associative cache of size 16KB with a block size of 32 bytes. We compiled the benchmarks for the Alpha processor instruction set. We generated traces for the benchmarks using Simplescalar 3.0 [11] and chose a 500 million total references (instruction + data) sub-trace from the middle of the generated trace. We used a hierarchical cache simulator, *hiercache*, to simulate the trace assuming a infinite-sized L2 cache. We are interested in gauging the performance of the L1 cache prefetch methods. We also placed a realistic bandwidth constraint on L2 to L1 data transfers.

In each graph, we have five curves. The first curve corresponds to LRU-x, with x varying from 0 to 7. For most benchmarks LRU-x's performance peaks at $x = 1$ or $x = 2$. The next four curves correspond to ((Ideal Kill + LRU)$-i$, (Ideal Kill)$-j$) for i varying from 0 to 3. The ordinate on the x-axis corresponds to the total number of prefetched blocks, i.e., $i + j$. For example, in the ART benchmark Ideal Kill graph, for $i = 3$ and $j = 4$ ($i + j = 7$) we get a 78.1% hit rate.

The benchmarks all show that increasing j for a given i does not hurt the hit rate very

much because we are only prefetching blocks if there is dead space in the cache. Increasing j beyond a certain point does hurt hit rate a little because of increased bandwidth from L2 to L1 when there is a lot of dead data in the cache, but not because of cache pollution. In general, for the benchmarks ((Ideal Kill + LRU)$-i$, (Ideal Kill)$-j$) performs very well for $i = 1$ and $j \geq 0$. We conjectured that it is always (though not provably) better than LRU-i, which is borne out by the results. It is also better in all benchmarks than the no-prefetch case LRU-0.

The results show that the integration of Kill + LRU and an aggressive prefetching strategy provides hit rate improvement, significantly so in some cases. Obviously, the hit rates can be improved further, by using more sophisticated prefetching strategies, at the potential cost of additional hardware. Many processors already incorporate sequential hardware prefetch engines, and their performance can be improved using our methods.

6.2.3 Experiments with Compiler-Inserted Kill

We repeat the above experiments except that instead of using trace-based ideal kill we use compiler-inserted kill as described in Section 4.4.2.

The compiler-inserted kill performs nearly as well as ideal kill. The graphs are nearly identical in many cases. The main differences between the compiler-inserted kill method and the ideal kill method is summarized in Figure 6-13. In the table, the number of kill hints for the trace-based ideal kill is given first. The number of *executions* of Kill load/store and Kill range instructions is given. Note that the executed Kill Range instructions are a very small part of total instructions executed, and their performance impact is negligible. In fact, for these benchmarks, the Kill Range instructions could have been replaced by Kill load/store instructions, with no performance impact. The number of kill hints generated by the compiler is smaller, mainly because the compiler uses a conservative strategy when inserting kill instructions.

6.2.4 Discussion

In Figure 6-14, we give the hit rates corresponding to four important prefetching strategies, LRU-1, LRU-2, ((Ideal Kill + LRU)-1, (Ideal Kill)-1) shown as Ideal(1,1) in the table, and Comp(1,1), the compiler-kill version. Ideal(1,1) is always better than LRU-1, LRU-2 and the no-prefetch case. The same is true for Comp(1,1), except for ART, where Comp(1,1)

is fractionally worse than LRU-2. LRU−1 and LRU-2 are not always better than the no-prefetch LRU-0 case, for example, in SWIM and TOMCATV. The results clearly show the stability that Kill + LRU replacement gives to a prefetch method.

The memory performance numbers assuming a L2 latency of 18 cycles are given in Figure 6-15. Note that this is not overall performance improvement since all instructions do not access memory. The percentage of memory operations is given in the last column of the table.

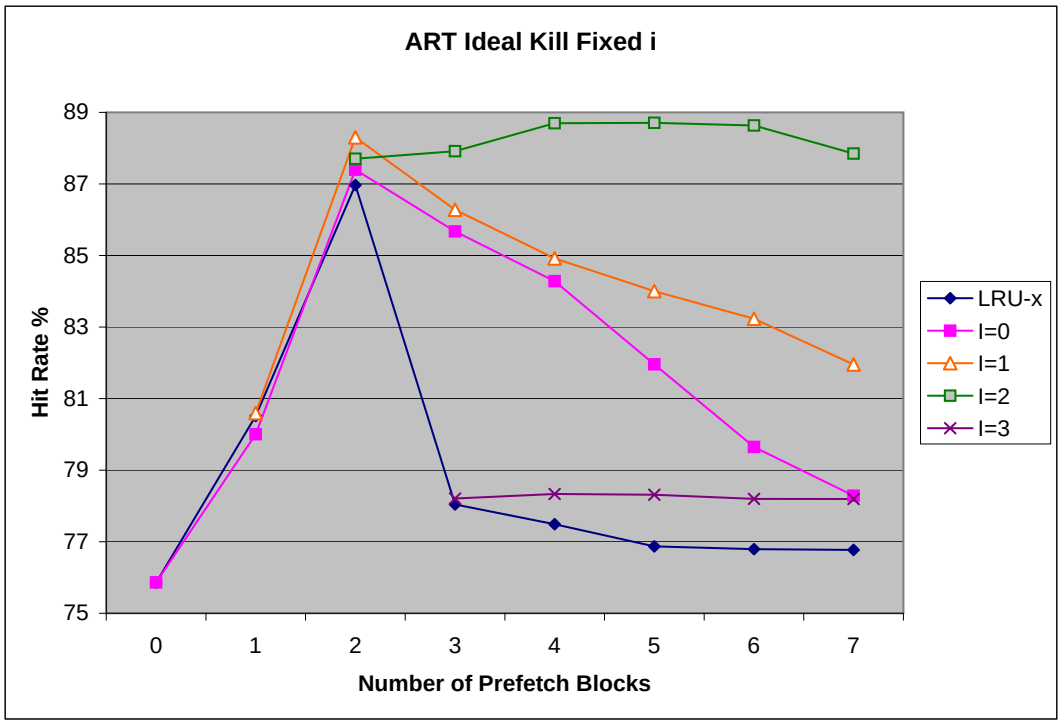
Our purpose here was to show that cache pollution can be controlled by modifying cache replacement, and therefore we are mainly interested in memory performance. The effect of memory performance on overall performance can vary widely depending on the type of processor used, the number of functional units, the number of ports to the cache, and other factors. The empirical evidence supports our claim that modified cache replacement integrated with prefetching can improve memory performance.

It appears that a scheme where 1 or 2 adjacent blocks are prefetched on a cache miss, with 2 blocks being prefetched only in the case where the second block replaces a dead block in the cache improves memory performance over standard LRU-based sequential hardware prefetching. Further, this method is significantly more stable than standard LRU-based sequential hardware prefetching, which might actually hurt performance, in some cases. While we are experimenting with more sophisticated prefetching schemes that may achieve better hit rates and performance to gauge the effect of modified cache replacement, we believe that simple schemes with low hardware overheads are more likely to be adopted in next-generation processors. Our work has attempted to improve both the worst-case and average performance of prefetching schemes.

It can be observed from the Figures 6-5 through 6-12 that prefetching with Kill+LRU gives more stable performance even with the aggressive prefetching where relatively large number of prefetches can be issued on a miss. The stability in presence of prefetching leads to better worst-case performance thus improving cache predictability.

6.3 Cache Partitioning

We evaluate a disjoint sequences-based cache partitioning strategy on a subset of Spec2000 benchmarks. In our experiments, we use two disjoint sequences for partitioning the cache.



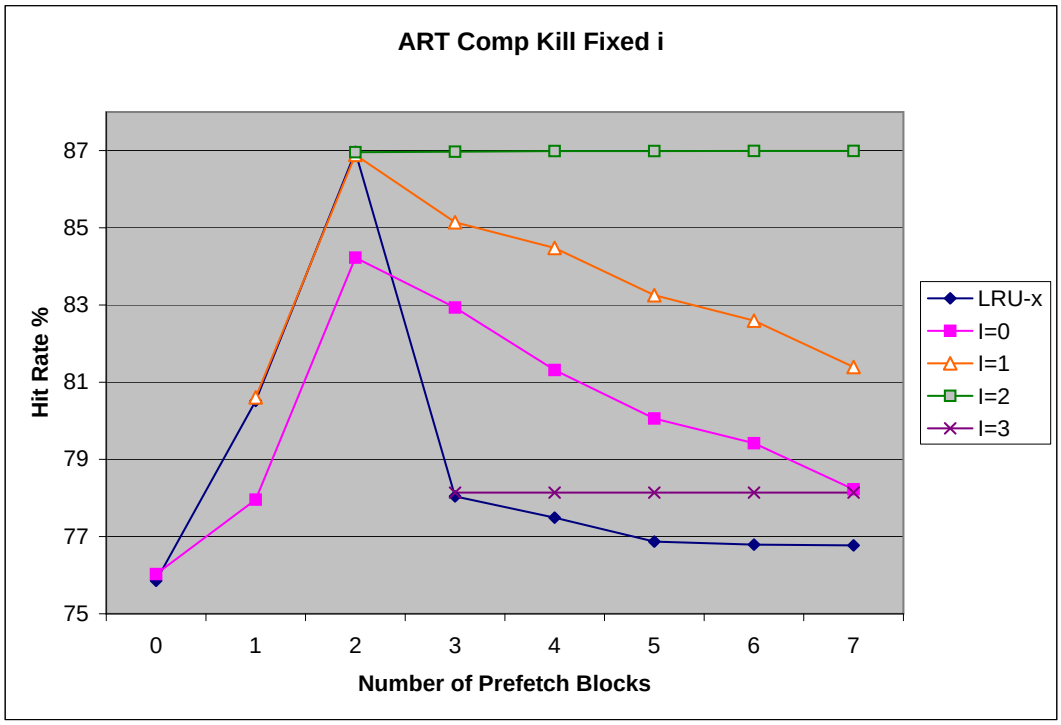
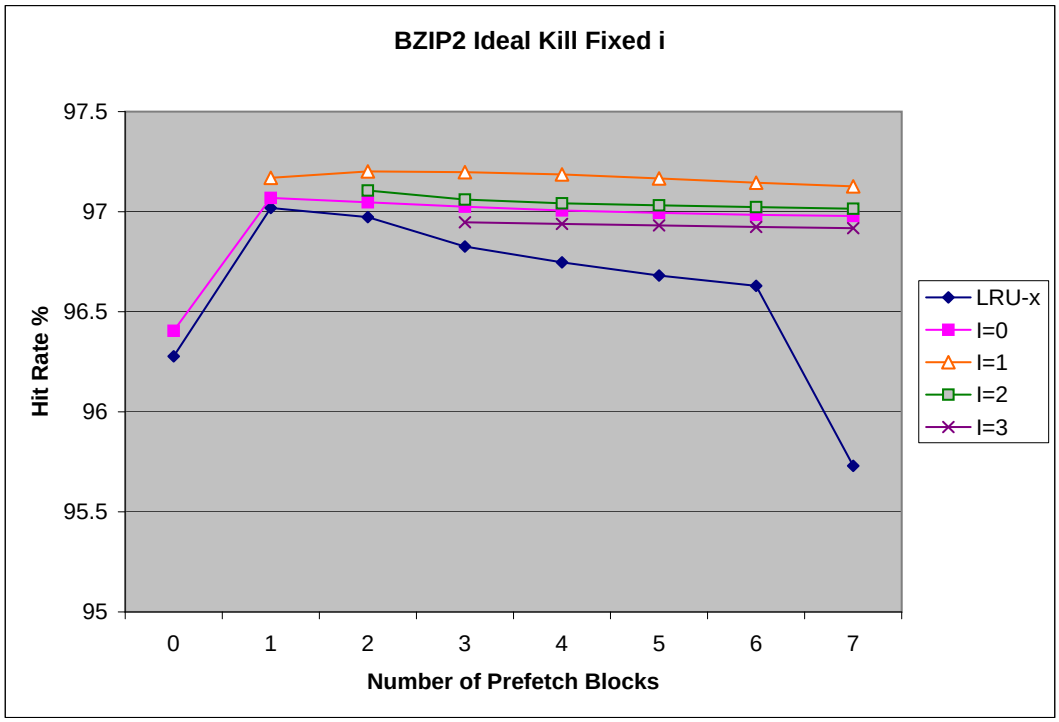


Figure 6-5: ART L1 Hit Rate



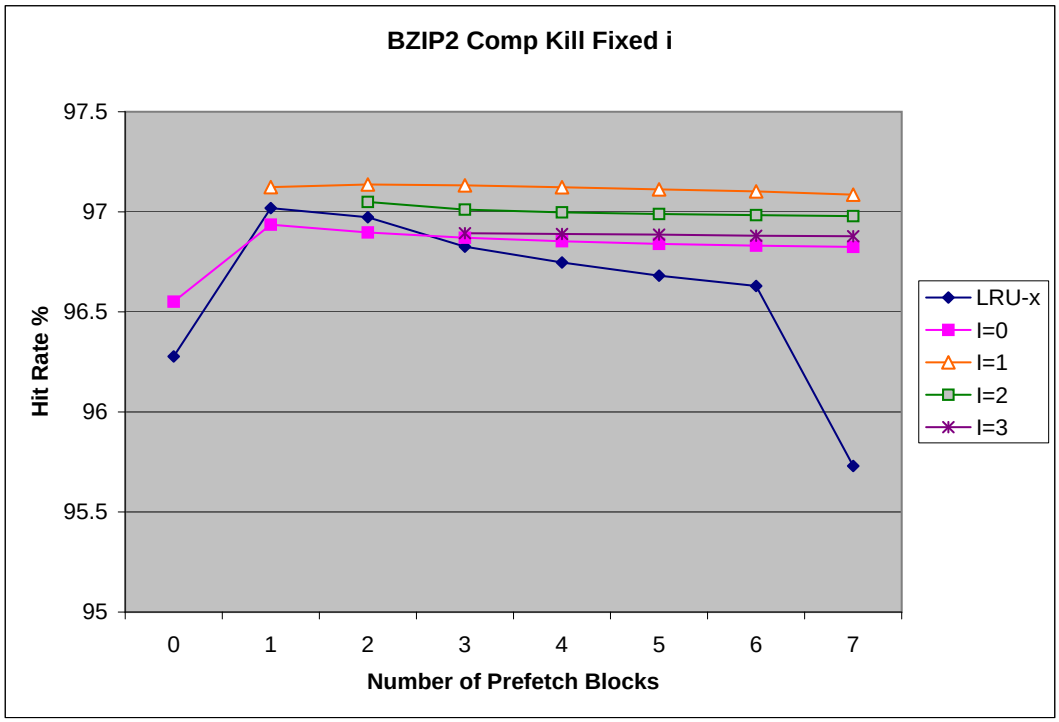
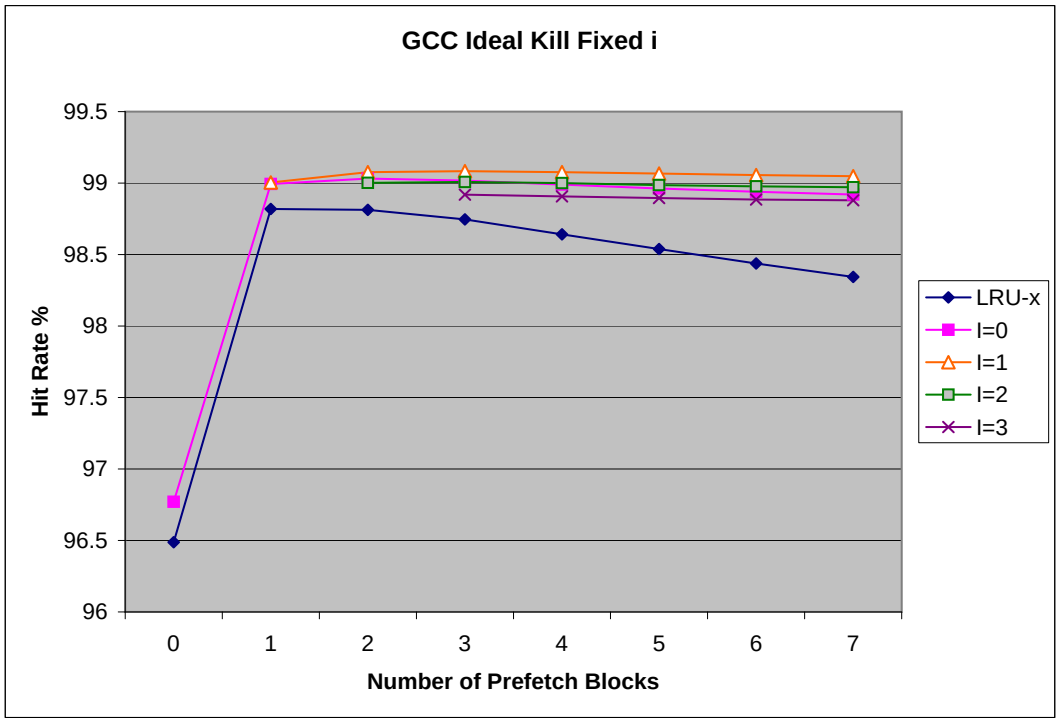


Figure 6-6: BZIP L1 Hit Rate



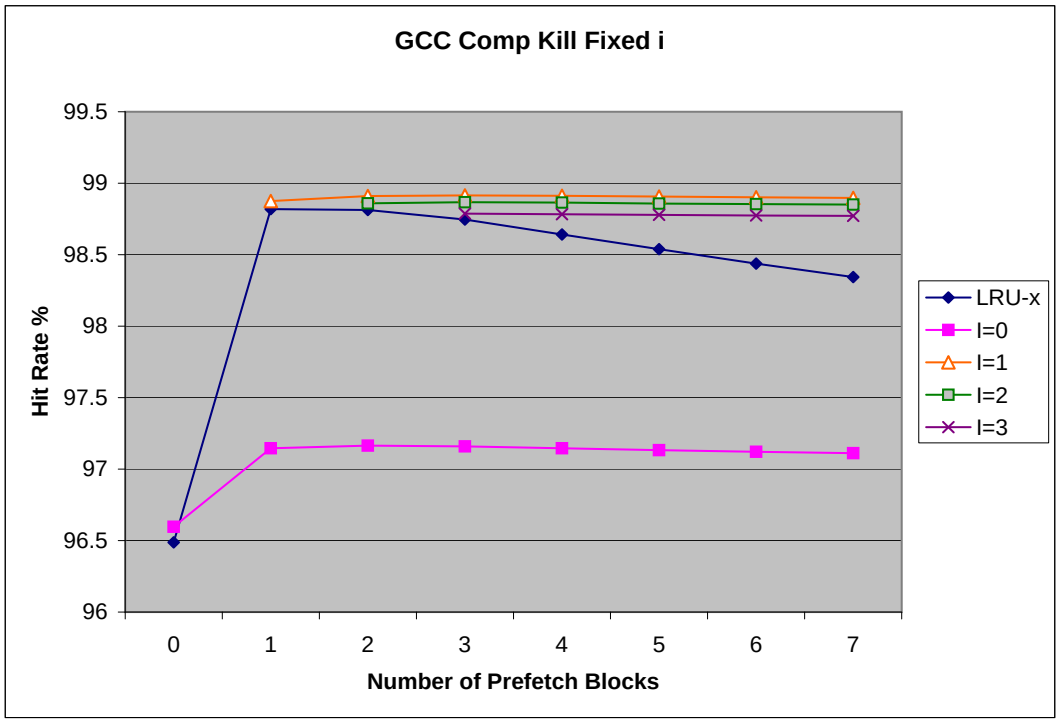
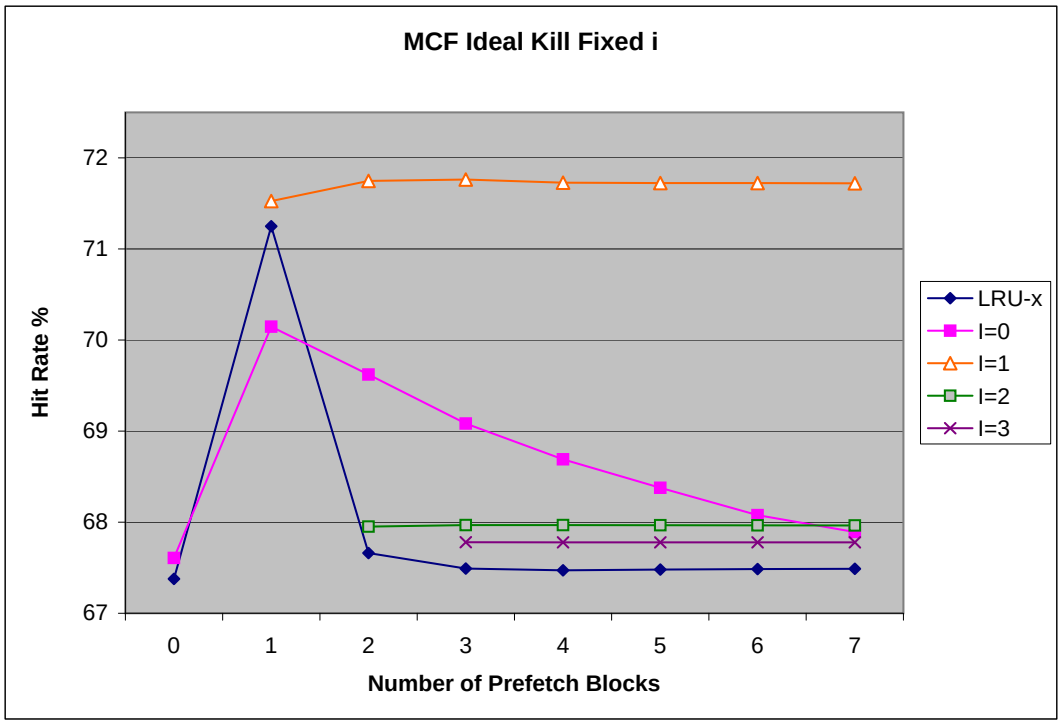


Figure 6-7: GCC L1 Hit Rate



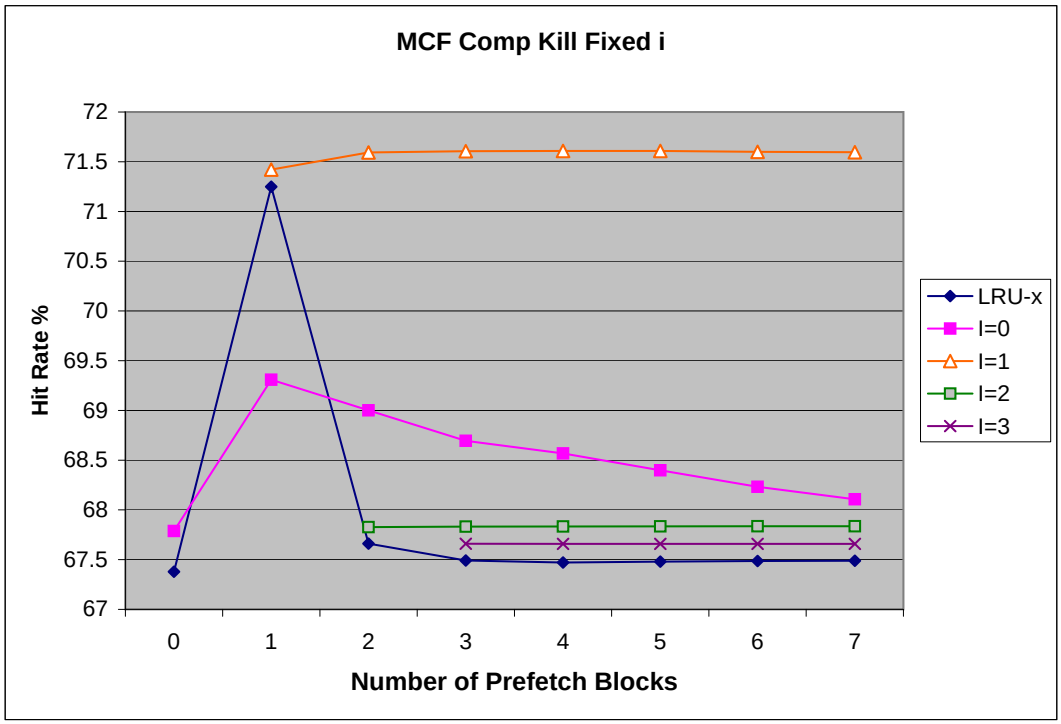


Figure 6-8: MCF L1 Hit Rate

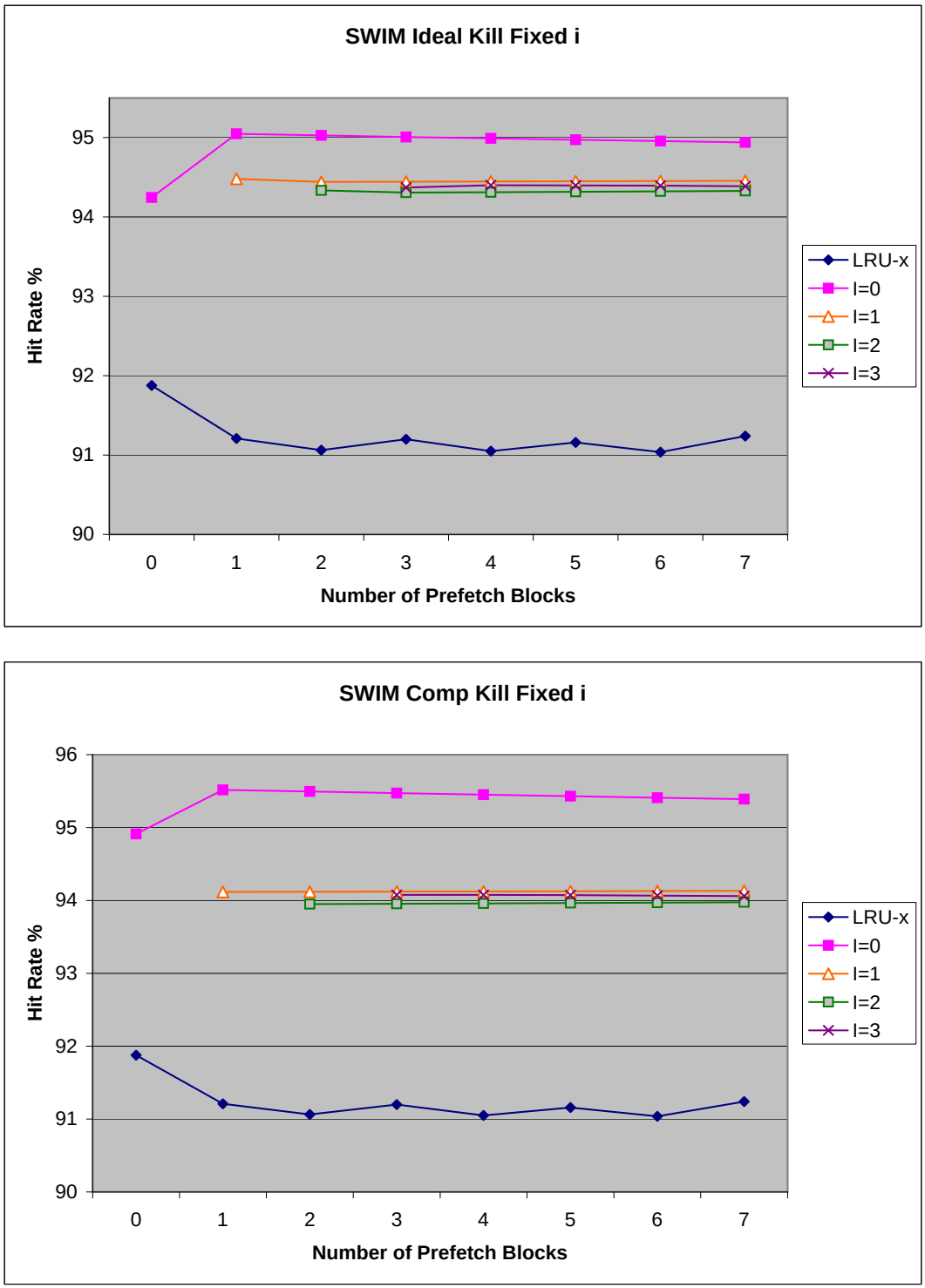


Figure 6-9: SWIM L1 Hit Rate

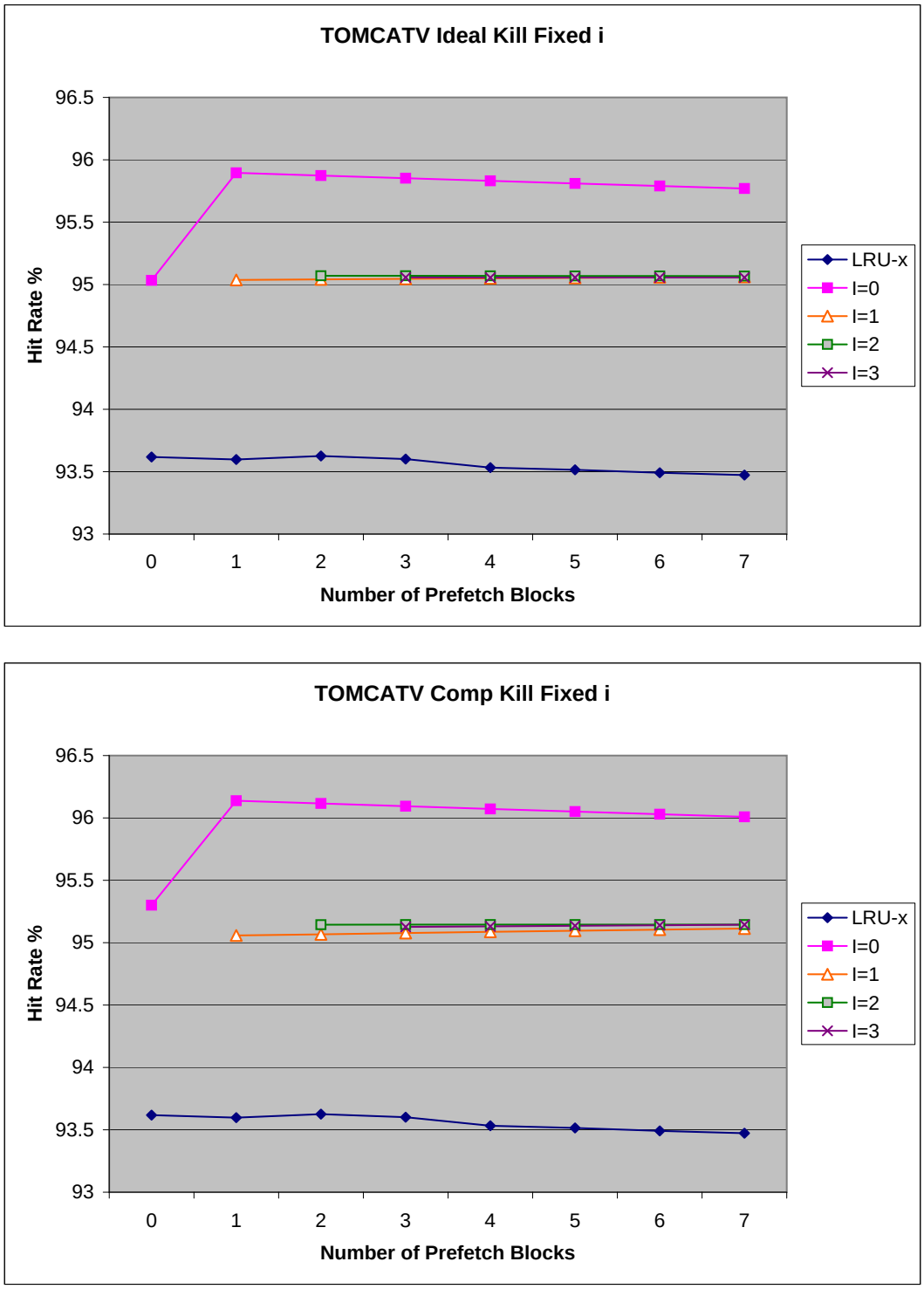
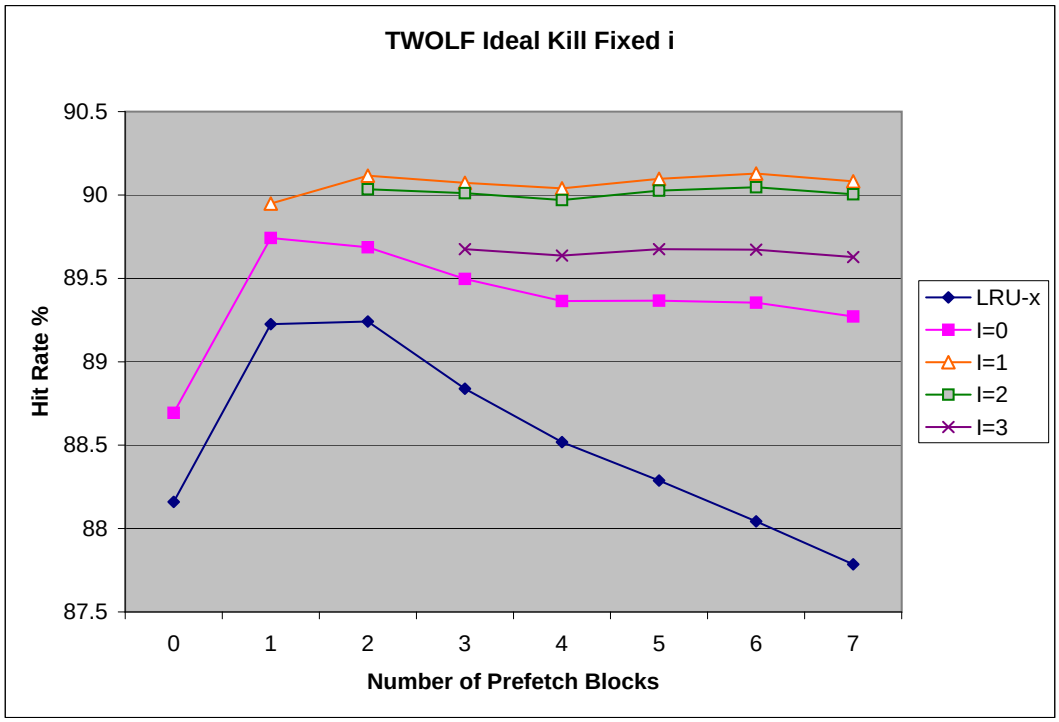


Figure 6-10: TOMCATV L1 Hit Rate



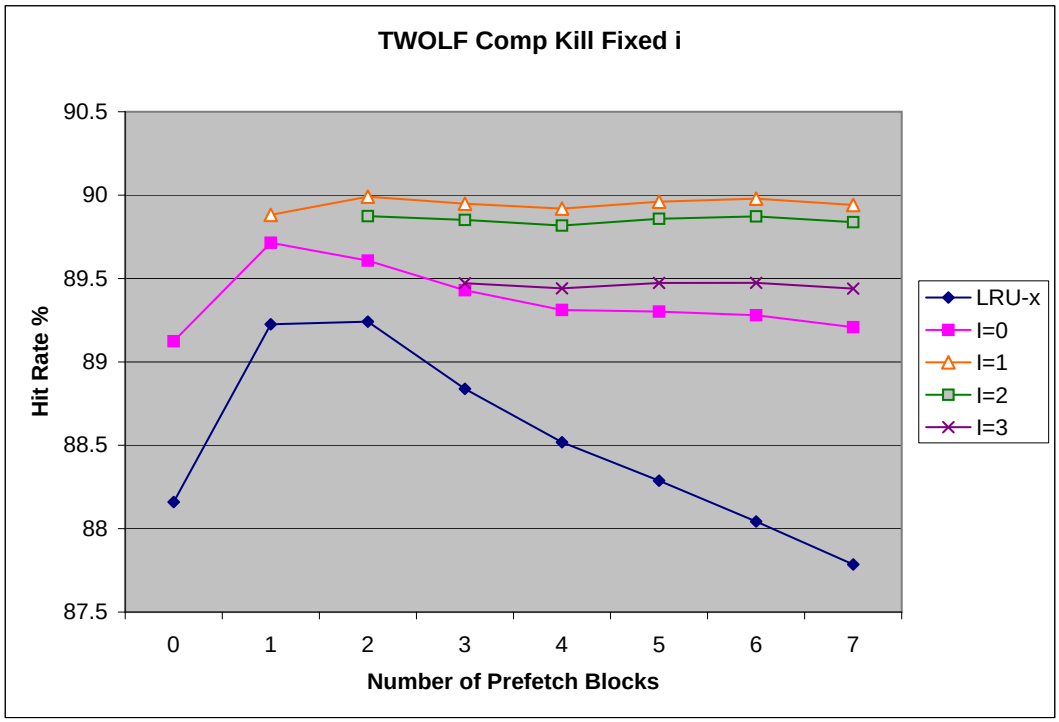
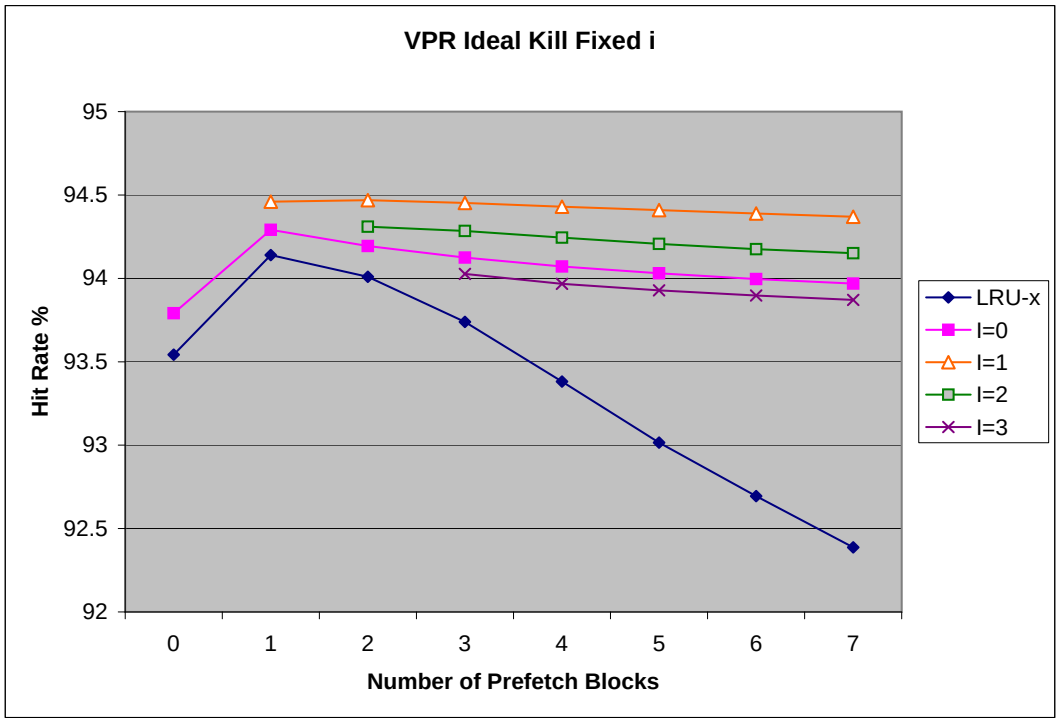


Figure 6-11: TWOLF L1 Hit Rate



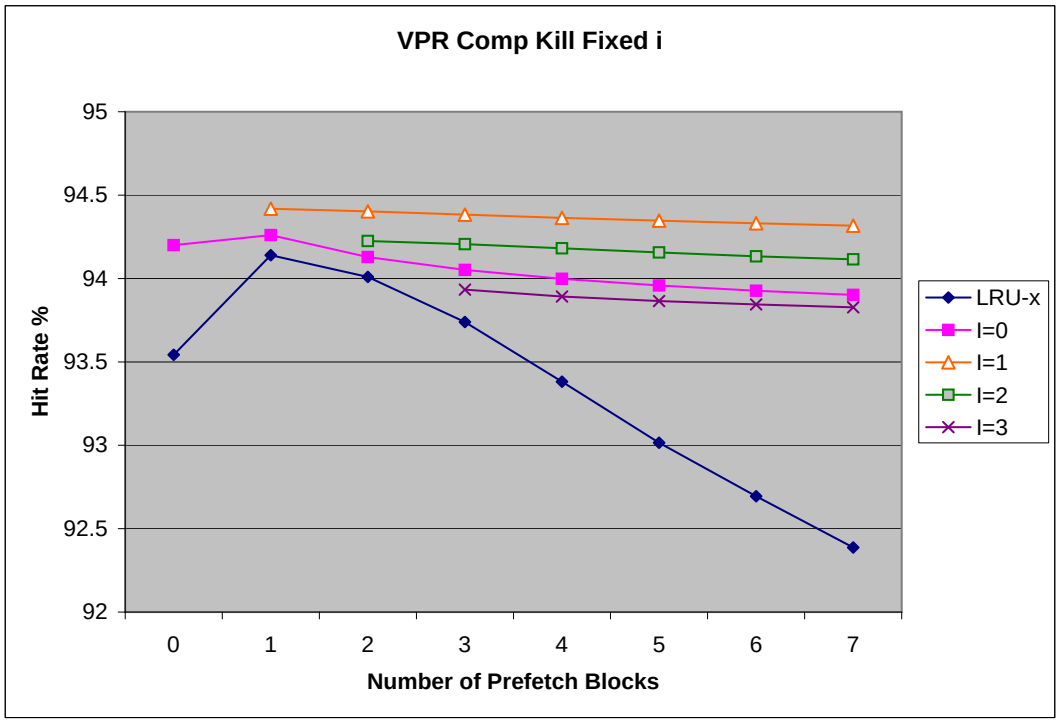


Figure 6-12: VPR L1 Hit Rate

Benchmark	Ideal Kill Hints	Compiler Kill Hints		% of Ideal Kill Hints
		Load-Store	Range	
ART	39943194	20021095	67516	50.29
BZIP	4286636	2562281	195183	64.32
GCC	5385506	1256474	175201	26.58
MCF	49251433	27548733	306489	56.56
SWIM	7657011	5452208	7120	71.30
TOMCATV	9177531	4685160	17544	51.24
TWOLF	15805947	11063292	82631	70.51
VPR	8207570	5208584	37332	63.91

Figure 6-13: Kill Hints for Ideal and Compiler-Kill

Benchmark	LRU-0	LRU-1	LRU-2	Ideal (1,1)	Comp (1,1)
ART	75.85	80.51	86.97	88.30	86.89
BZIP	96.28	97.02	96.97	97.20	97.14
GCC	96.49	98.82	98.81	99.09	98.91
MCF	67.38	71.25	67.66	71.75	71.59
SWIM	91.88	91.21	91.06	94.44	94.12
TOMCATV	93.62	93.60	93.63	95.04	95.07
TWOLF	88.16	89.23	89.24	90.33	89.99
VPR	93.54	94.14	94.01	94.50	94.40

Figure 6-14: LRU-0, LRU-1, LRU-2, Ideal (1,1), and Comp (1,1) L1 Hit Rates

Benchmark	LRU-1	LRU-2	Ideal (1,1)	Comp (1,1)	%MemoryOp
ART	18.37	58.77	70.83	58.11	33.08
BZIP	8.36	7.81	10.64	9.83	23.02
GCC	33.00	32.88	38.25	34.75	30.66
MCF	11.18	0.74	12.80	12.29	30.19
SWIM	-4.55	-5.50	22.42	19.05	18.85
TOMCATV	-0.16	0.06	13.13	13.40	28.65
TWOLF	6.40	6.50	13.95	11.52	26.69
VPR	5.09	3.94	8.39	7.49	25.42
Average	9.71	13.15	23.80	20.80	27.07

Figure 6-15: LRU-1, LRU-2, Ideal (1,1), and Comp (1,1) Memory Performance Improvement with respect to LRU-0 (no prefetching). L2 latency = 18 cycles.

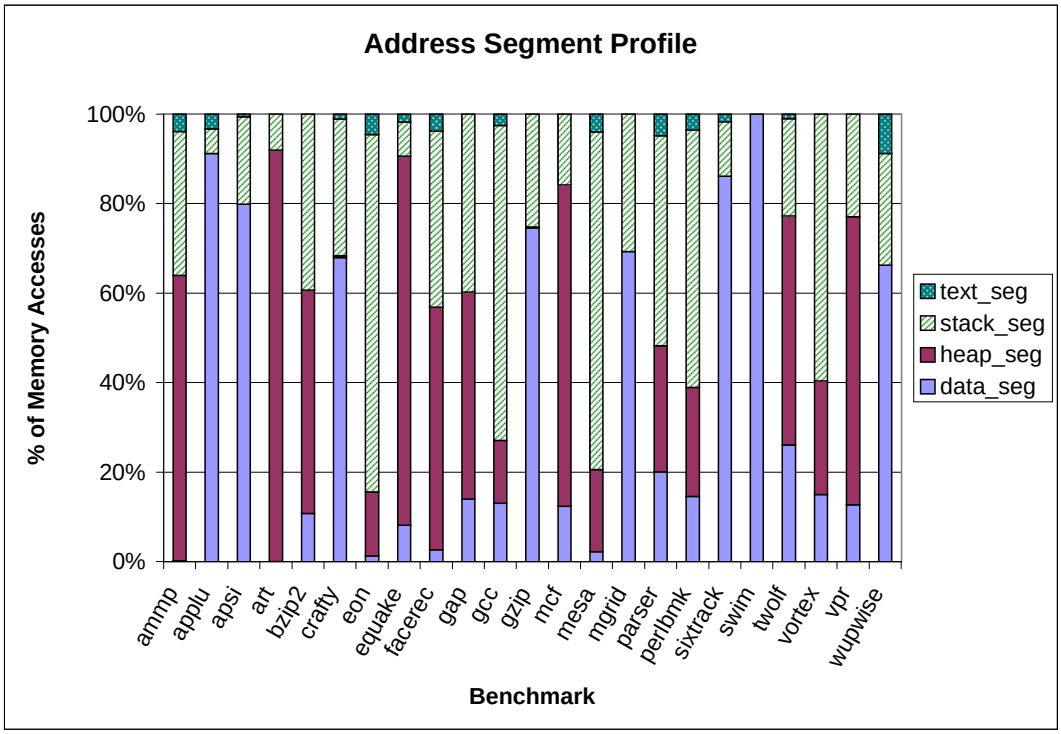


Figure 6-16: Memory Access Address Segment Profile of Spec2000 Benchmarks

We divide the accesses into two disjoint sequences based on the address segments corresponding to the accesses. The data accesses belong to one of the four address segments: `data, heap, stack, text`. These data accesses are categorized into two sequences: *stack_seq* and *non_stack_seq*. The *stack_seq* accesses belong to the stack segment and the *non_stack_seq* accesses belong to the `data, heap, text` segments. The two sequences have disjoint sets of addresses.

The address segment distribution of the data accesses in the Spec2000 benchmarks is shown in Figure 6-16. For example, the benchmark `eon` has 80% *stack_seq* accesses and 20% *non_stack_seq* accesses and the benchmark `mesa` has 76% *stack_seq* accesses and 24% *non_stack_seq* accesses.

We illustrate the cache partitioning strategy using two disjoint sequences with the two benchmarks: `eon` (Spec2000INT) and `mesa` (Spec2000FP). The cache partitioning for these two benchmarks is shown in Figure 6-17. These cache partitioning results provide empirical data for the qualitative cache partitioning example of Figure 5-1. The benchmarks were simulated using an 8K 4-way set-associative cache with 32 bytes blocks. Figure 6-17 shows various misses for `eon` and `mesa` with respect to associativity. The misses for the disjoint

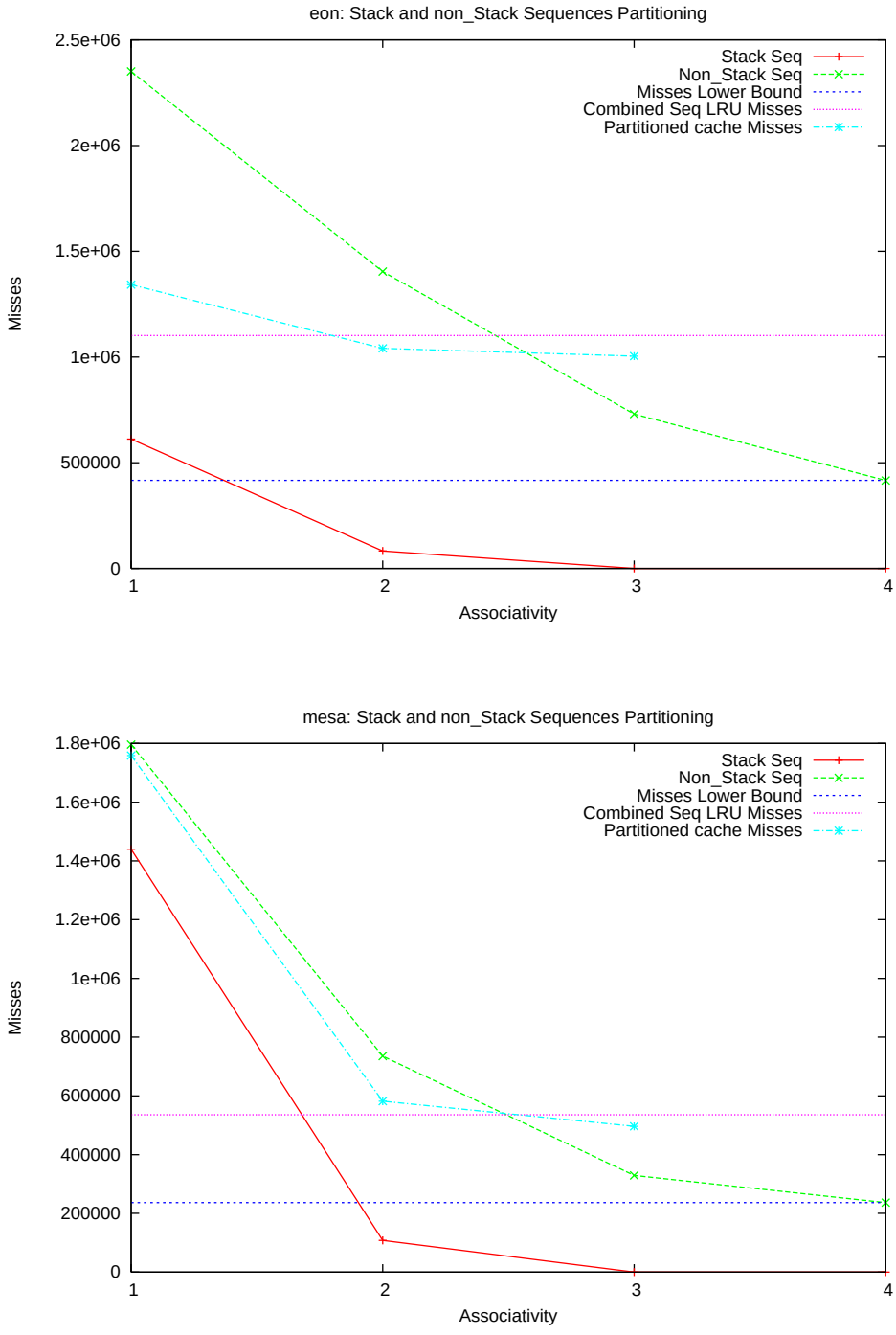


Figure 6-17: Two Sequence-based Partitioning: Misses for eon (Spec2000INT) and mesa (Spec2000FP)

sequences *stack_seq* and *non_stack_seq* for the associativity 1 to 4 (corresponding cache size 2K to 8K) show the individual disjoint sequence misses when these sequences are applied individually to the cache. The number of misses for the *stack_seq* are lower than the misses for the *non_stack_seq* for each associativity point. The number of misses for the 4-way LRU 8K cache is shown as a horizontal line and it represents the misses for the combined sequence. The lower bound for the number of misses for the combined sequence is the sum of the misses of *stack_seq* and *non_stack_seq* at associativity 4.

Any cache partitioning solution should have the total number of misses of the partitioned cache between the two horizontal lines. Figure 6-17 shows three partitioning data points for `eon` and `mesa`. The cache partitioning is implemented using the modified LRU replacement policy as described in 5.5.2. The three cache partitioning points are: (*stack_seq* maximum 1 way, *non_stack_seq* maximum 3 ways), (*stack_seq* maximum 2 ways, *non_stack_seq* maximum 2 ways), and (*stack_seq* maximum 3 ways, *non_stack_seq* maximum 1 way). There are two partitioning points for `eon` and one partitioning point for `mesa` because the number of misses for the partitioned cache is lower than the misses for the combined sequence applied to the cache.

The cache partitioning miss rates for a set of Spec2000 benchmarks based on *stack_seq* and *non_stack_seq* for a 4-way 8K set-associative cache are shown in Table 6.3. The column `LRU` indicates the miss rate for cache without any partitioning. The column `S1_NS3` shows the overall miss rate for the partitioned cache where *stack_seq* has maximum 1 way and *non_stack_seq* has maximum 3 ways. Similarly, the other two partitioning point miss rates are shown in the columns `S2_NS2` and `S3_NS1`. The miss rates for the partitioning points with lower miss rate than LRU are shown in bold case letters. Some benchmarks don't show any improvement with any partitioning point because the granularity of partitioning does not allow the partitioning point miss rate to be lower than the LRU miss rate. Some benchmarks show only marginal improvement with partitioning because the number of *stack_seq* misses are very few compared to *non_stack_seq* misses even though the number of *stack_seq* accesses are not small. Figure 6-18 shows the address segment misses distribution of the Spec2000 benchmarks for 4-way and 8-way 8K caches. For the 4-way 8K cache, the *stack_seq* misses are very small to negligible for all the benchmarks except `eon`, `gcc`, and `mesa`. This significant difference in misses between the two sequences limits the overall miss rate improvement from cache partitioning using the *stack_seq* and *non_stack_seq* sequences.

Benchmark	LRU	S1_NS3	S2_NS2	S3_NS1
ammp	7.97	**7.93**	**7.93**	**7.93**
applu	15.74	15.90	15.90	15.90
apsi	2.72	2.78	2.88	2.88
art	39.80	39.81	39.81	39.81
bzip2	5.12	5.34	5.43	5.43
crafty	7.73	9.73	9.73	9.73
eon	2.36	2.87	**2.23**	***2.15***
equake	17.81	17.86	17.91	17.91
facerec	2.77	**2.76**	**2.76**	**2.76**
gap	1.46	1.60	1.60	1.60
gcc	3.61	4.23	4.49	7.30
gzip	3.89	3.90	3.90	3.90
mcf	34.95	34.96	34.96	34.96
mesa	1.42	4.66	1.54	***1.31***
mgrid	16.30	16.34	16.37	16.38
parser	5.65	6.01	6.01	6.01
perlbmk	0.27	0.29	0.31	0.31
sixtrack	0.92	0.97	0.97	0.97
swim	20.05	**20.04**	**20.04**	**20.04**
twolf	10.58	**10.56**	**10.56**	**10.56**
vortex	2.00	2.18	2.58	2.59
vpr	6.83	6.83	6.83	6.83
wupwise	2.10	2.10	2.10	2.10

Table 6.1: Two Sequence-based Partitioning Results for 4-way 8K Cache

The cache partitioning results for an 8-way 8K set-associative cache are shown in Table 6.3. The `LRU` column shows the miss rates without partitioning and the columns `S1_NS7` through `S7_NS1` show the miss rates of the partitioned cache with *stack_seq* maximum ways $1-7$ (*non_stack_seq* maximum ways $7-1$). For the `mesa` benchmark, the misses of the *stack_seq* are negligible with respect to the *non_stack_seq* sequence in the 8-way 8K cache, resulting in marginal improvement with partitioning for the 8-way 8K cache as compared to the 4-way 8K cache.

Considering the address segment profile of accesses in Figure 6-16, the two sequence-based cache partitioning can also be applied using the `heap` and `data` segment accesses as sequences. When the two-sequence partitioning does not lead to miss rate improvement due to granularity of partitioning or variations within the sequence, then a multi-sequence partitioning algorithm that uses two-sequence partitioning recursively can lead to better partitioning in a hierarchical manner.

Benchmark	LRU	S1_NS7	S2_NS6	S3_NS5	S4_NS4	S5_NS3	S6_NS2	S7_NS1
ammp	7.94	**7.90**	**7.90**	**7.90**	**7.90**	**7.90**	**7.90**	**7.90**
applu	16.16	16.29	16.29	16.29	16.29	16.29	16.29	16.29
apsi	2.55	**2.53**	2.61	2.61	2.61	2.61	2.61	2.61
art	39.80	39.80	39.80	39.80	39.80	39.80	39.80	39.80
bzip2	5.00	5.10	5.16	5.19	5.19	5.19	5.19	5.19
crafty	6.97	7.31	8.64	8.64	8.64	8.64	8.64	8.64
eon	1.69	7.02	2.66	1.73	**1.65**	**1.65**	***1.65***	**1.65**
equake	15.47	15.57	15.58	15.59	15.59	15.59	15.59	15.59
facerec	2.78	4.42	**2.76**	**2.76**	**2.76**	**2.76**	**2.76**	**2.76**
gap	1.45	**1.44**	1.45	1.45	1.45	1.45	1.45	1.45
gcc	3.37	4.66	3.60	3.60	3.95	4.64	5.99	9.50
gzip	3.86	3.88	3.88	3.88	3.88	3.88	3.88	3.88
mcf	34.90	34.90	34.90	34.90	34.90	34.90	34.90	34.90
mesa	0.63	7.26	2.64	1.33	0.74	**0.62**	***0.62***	**0.62**
mgrid	14.77	**14.71**	**14.74**	**14.75**	**14.75**	**14.75**	**14.75**	**14.75**
parser	5.49	5.55	5.73	5.73	5.73	5.73	5.73	5.73
perlbmk	0.25	0.25	0.27	0.27	0.27	0.27	0.27	0.27
sixtrack	0.68	0.80	0.80	0.80	0.80	0.80	0.80	0.80
swim	20.05	**20.04**	**20.04**	**20.04**	**20.04**	**20.04**	**20.04**	**20.04**
twolf	10.01	**9.99**	**9.99**	**9.99**	**9.99**	**9.99**	**9.99**	**9.99**
vortex	1.91	2.77	1.96	2.08	2.15	2.16	2.16	2.16
vpr	6.48	**6.47**	**6.47**	**6.47**	**6.47**	**6.47**	**6.47**	**6.47**
wupwise	1.99	**1.97**	**1.97**	**1.97**	**1.97**	**1.97**	**1.97**	**1.97**

Table 6.2: Two Sequence-based Partitioning Results for 8-way 8K Cache

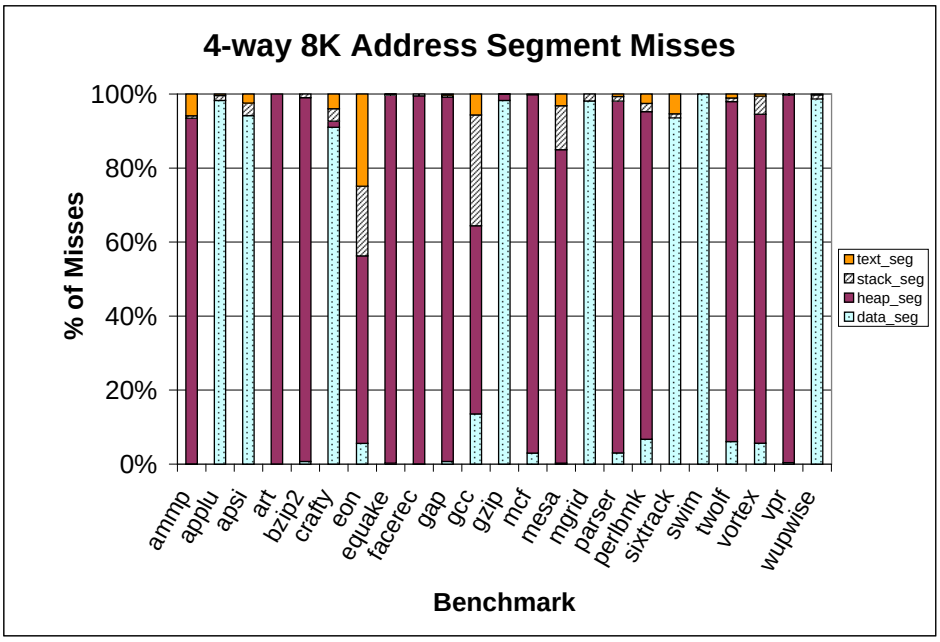
For the two sequence based cache partitioning, a cache partitioning solution has two cache-ways assignments − one for each sequence. The cache partitioning solution leads to lower miss rates than the unpartitioned LRU. In order to find a cache partitioning solution, we use the reuse distance information of each sequence in combination with other parameters to determine the number of cache-ways to assign to each of the two sequences. We obtained the reuse distance profile of *stack_seq* and *non_stack_seq* accesses for the 4-way and 8-way 8K caches with a maximum reuse distance of 256 and computed the weighted average of reuse distance profile for these two sequences. The weighted average of the reuse distance is inversely proportional to the cache partition size assignment. For the `mesa` and `eon` benchmarks, the partition size assignment is `S3_NS1` for the 4-way 8K cache and `S6_NS2` for the 8-way 8K cache. The cache partitioning points determined by the reuse distance-based algorithm show the miss rates in italics in Table 6.3 and Table 6.3. The partitioning points determined by the algorithm that coincide with the simulated bold face partitioning points

Benchmark	In-order			Out-of-order		
	LRU	S3_NS1	Speedup	LRU	S3_NS1	Speedup
eon	0.4913	0.4896	**0.9965**	0.7933	0.7940	**1.0008**
mesa	0.6825	0.6942	**1.0171**	1.2147	1.2591	**1.0365**

Table 6.3: IPC results for two sequence-based partitioning for 4-way 8K Cache

are shown in bold face italics.

In order to get the Instructions Per Cycle (IPC) information, we ran the benchmarks `eon` and `mesa` on the Simplescalar out-of-order simulator. We considered two processor configurations for the instruction issue and execution: in-order without speculation and out-of-order with speculation. The IPC results for the two benchmarks are shown in Table 6.3. These results are for the 4-way 8K cache without partitioning using LRU and with partitioning solution S3_NS1. The LRU columns show the IPC for the unpartitioned cache and the S3_NS1 columns show the IPC for the partition solution S3_NS1. The speedup of the partitioning solution over the LRU is shown the speedup columns. The IPC for `eon` under in-order decreases slightly for S3_NS1 compared to LRU even though the L1 cache miss rates improves for S3_NS1. This decrease in IPC happens because the increase in the misses for the *non_stack_seq* under S3_NS1 is more critical for the IPC. The benchmark `mesa` has 1.71% and 3.65% improvement in IPC for the in-order and out-of-order configurations respectively.



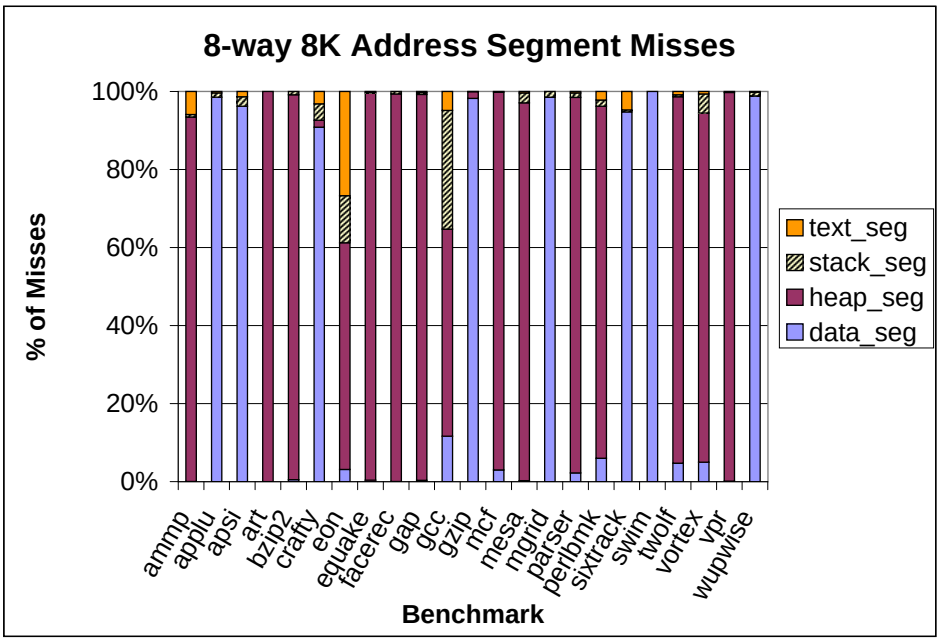


Figure 6-18: Address Segment Misses Profile of Spec2000 Benchmarks

Chapter 7

Related Work

On-chip memory plays an important role in the overall performance of a general-purpose or embedded system. As a result, a major portion of the chip area of modern processors is occupied by on-chip memory (caches, SRAM, ROM). One important component of on-chip memory is a cache. Caches are an integral part of modern processors (e.g., Pentium 4 [37], Alpha 21264 [54], MIPS R10000 [135], and IA-64 [29]) and embedded processors (e.g., ARM9TDMI [1]). Many processors use out-of-order execution to hide cache miss latencies and use miss information/status handling registers (MSHRs) to allow multiple outstanding misses to the cache. Even with these latency hiding techniques, the cache misses can reduce performance. There has been lot of research effort focussed on improving on-chip memory performance using a variety of architectural techniques. Previous work to improve cache performance and predictability that is related to the mechanisms presented in this thesis is described in the following sections.

7.1 Cache Architectures

There are several cache parameters that can be optimized to improve cache performance as well [34]. The effectiveness of direct-mapped caches is studied in [35] and the effect of associativity in caches is described in [36]. In the context of direct-mapped caches, pseudo-associativity was achieved using column-associative caches [2]. The issue bandwidth limitation in caches and proposed solutions are discussed in [9, 10].

Different cache architectures have been proposed either to improve cache performance or to reduce power consumed by the cache. An adaptive non-uniform cache structure is

presented in [55] for wire-delay dominated caches. A modular reconfigurable architecture called smart memories is presented in [71]. The architecture consists of processing tiles where each tile contains local memory, local interconnect, and a processor core. A cool-cache architecture is presented in [118, 119] and a direct-addressed cache architecture is proposed in [129] to reduce cache power.

A comparison of resizable caches that use selective-ways and selective-sets from the energy-delay perspective is presented in [134]. A hybrid selective-sets-and-ways resizable cache is proposed that offers better resizing granularity than both the previous organizations. A cache management framework to handle the heterogeneity in memory access patterns in a CMP platform is described in [40]. It provides for priority classification, assignment, and enforcement of memory access streams. The priority is determined by the degree of locality, latency sensitivity, and application performance needs. The priority enforcement is done through cache allocation, set partitioning, and cache regions. A detailed study of the fairness issue for a shared cache in the CMP architecture is presented in [57]. It proposes fairness metrics that measure the degree of fairness in cache sharing and proposes static and dynamic L2 cache partitioning algorithms that optimize fairness. A reconfigurable cache design that allows the cache SRAM arrays to be dynamically divided into multiple partitions that can be used for different processor activities is presented in [95]. This design was evaluated for instruction reuse in media processing benchmarks.

As described in Chapter 3, my approach is to augment the replacement policy of a set-associative cache to improve cache performance and predictability.

7.2 Locality Optimization

The spatial and temporal locality of memory accesses is used to improve cache performance either in software or in hardware. The software approaches use static locality optimization techniques for loop transformations in the compiler [51]. Some approaches use cache miss analysis to guide the compiler optimizations [16, 31]. An approach that makes use of spatial footprints is presented in [62].

Some approaches use hardware structures to detect and exploit specific data access patterns (e.g., streams) to improve cache performance. In order to improve direct-mapped cache performance, a fully-associative cache with prefetch buffers was proposed in [49]. The

use of stream buffers to improve cache performance was described in [83]. Dynamic access reordering was proposed in [75] and improving bandwidth for streamed references was presented in [76]. A hardware-based approach that uses hardware miss classification was proposed in [24] and a spatial pattern prediction approach was presented in [17]. A locality sensitive multi-module cache design is presented in [101]. The run-time detection of spatial locality for optimization is described in [48].

In [99] the cache miss patterns are classified into four categories corresponding to the type of access pattern – next-line, stride, same-object, or pointer-based transitions. They present a hardware profiling architecture to identify the access pattern which can help in prefetching and other optimizations. In [30] it is shown that locality and miss rates can be associated with and predicted for static memory instructions rather than memory addresses alone based on reuse distance analysis. In [78] spatial regularity is defined for streams and a method to detect streams in an on-line algorithm is described that can be applied in profile-driven optimizations. In [14] a method to compute stack histograms to estimate the number of cache misses in a compiler is illustrated with the locality enhancing transformations(e.g., tiling). The stack histogram is computed with the data dependence distance vectors. A parameterized model of program cache behavior is presented in [139]. The model predicts the cache misses for arbitrary data set sizes given the cache size and associativity. This model uses the reuse distance analysis presented in [28].

In my work, I have focussed on a general method to detect dead data and have not specifically targeted detection of streams. However, the techniques presented to determine dead data, in many cases, will detect short-lived streaming data and mark the data as dead.

7.3 Cache Management

Some current microprocessors have cache management instructions that can flush or clean a given cache line, prefetch a line or zero out a given line [73, 77]. Other processors permit cache line locking within the cache, essentially removing those cache lines as candidates to be replaced [26, 27]. Explicit cache management mechanisms have been introduced into certain processor instruction sets, giving those processors the ability to limit pollution. One such example is the Compaq Alpha 21264 [53] where the new load/store instructions minimize pollution by invalidating the cache-line after it is used.

An approach for data cache locking to improve cache predictability is presented in [125]. In [72] the use of cache line locking and release instructions is suggested based on the frequency of usage of the elements in the cache lines. In [112], active management of data caches by exploiting the reuse information is discussed along with the active block allocation schemes. An approach using dynamic reference analysis and adaptive cache topology is presented in [86]. The approach using dynamic analysis of program access behavior to proactively guide the placement of data in the cache hierarchy in a location-sensitive manner is proposed in [47]. In [33] a fully associative software managed cache is described. An integrated hardware/software scheme is described in [32]. The adaptive approaches use different control mechanisms to improve cache performance. A method based on adaptive cache line size is described in [124]. A method for adaptive data cache management is described in [116]. Cache management based on reuse information is addressed in [97].

A run-time approach that bypasses cache for some accesses is described in [46]. The hardware determines data placement within the cache hierarchy based on dynamic referencing behavior. The theoretical upper bounds of cache hit ratios using cache bypassing are compared with experimental results for integer applications. A mechanism to identify misses as conflict or capacity misses at run-time and using this information in an adaptive miss buffer is presented in [25].

The keep instructions in Chapter 3 bear some similarity to the work on cache line locking [72, 125]. The methods to determine what data to keep used are different. I provide hit rate guarantees when my algorithm is used to insert cache control instructions.

7.4 Replacement Policies

Several replacement policies were proposed for page-based memory systems, though the tradeoffs are different for cache replacement policies. An early-eviction LRU page replacement (EELRU) is described in [107]. Belady's optimal replacement algorithm was presented in [5] and its extension using temporal and spatial locality was presented in [113]. The efficient simulation of the optimal replacement algorithm was described in [110].

Compiler analysis based methods to improve replacement decisions in caches and main memory are presented in [7, 128]. In [132] some modified LRU replacement policies have been proposed to improve the second-level cache behavior that look at the temporal locality

of the cache lines either in an off-line analysis or with the help of some hardware. In [65], policies in the range of Least Recently Used and Least Frequently Used are discussed. The replacement policies which take into account non-uniform miss costs are considered in [44, 45]. An approach that corrects the mistakes of the LRU policy using some hardware support is discussed in [50]. A hardware-based approach to detect dead blocks and correlate them with blocks to prefetch is presented in [64]. An approach that uses time-keeping in clock cycles for cache access interval or cache dead time is presented in [38].

In [103] Keep-me and Evict-me compiler annotations are introduced with memory operations. The annotations are maintained with a cache line and are used on a cache miss. The replacement policy prefers to evict cache lines marked evict-me and retain the cache lines with keep-me annotation if possible. In [127] an aggressive hardware/software prefetching scheme is combined with evict-me based replacement scheme to handle cache pollution caused by data being pushed into L1 by useless prefetches. To handle the non-uniform distribution of memory accesses across different cache sets, a variable-way cache combined with reuse replacement is presented for L2 caches in [93]. The number of tag entries relative to the number of data lines is increased to vary the associativity in response to the program demands. Some adaptive insertion policies are presented in [92] where incoming data block is kept at the LRU position instead of making it an MRU data depending on the working sets of the program. This reduces miss rates for the L2 caches and avoids thrashing data.

7.5 Prefetching

In [121, 123] a summary of some prefetching methods is presented with the design trade-offs in implementing those prefetching strategies. Some of the prefetching approaches are described below.

Hardware Prefetching

In hardware sequential prefetching, multiple adjacent blocks are prefetched on a block access or a miss. A common approach is one block lookahead (OBL) where a prefetch is initiated for the next block when a block is accessed [108]. A variation of the OBL approach uses prefetch-on-miss and tagged prefetch. Cache pollution is a problem if more than one block is prefetched for a block access [91]. Another hardware approach for prefetching uses stride

detection and prediction to prefetch data [18, 19].

Software Prefetching

There have been many proposals for software prefetching in the literature. Many techniques are customized toward loops or other iterative constructs (e.g., [61]). These software prefetchers rely on accurate analysis of memory access patterns to detect which memory addresses are going to be subsequently referenced [68, 69, 79, 82]. The software data prefetching for the HP PA-8000 compiler is presented in [102].

Software has to time the prefetch correctly – too early a prefetch may result in loss of performance due to the replacement of useful data, and too late a prefetch may result in a miss for the prefetched data. I do not actually perform software prefetching in my work – instead software control is used to mark cache blocks as dead as early as possible. This significantly reduces the burden on the compiler, since cycle-accurate timing is not easy to predict in a compiler.

Prefetching with Other Mechanisms

Some approaches combine prefetching with other mechanisms. For example, the dead-block prediction and correlated prefetching is described for direct-mapped caches in [64] and prefetching and replacement decisions have been made in tandem in the context for I/O prefetching for file systems [13, 85]. The work of [63, 64] was one of the first to combine dead block predictors and prefetching. The prefetching method is intimately tied to the determination of dead blocks. Here, I have decoupled the determination of dead blocks from prefetching – any prefetch method can be controlled using the notion of dead blocks. I have used a simple hardware scheme or profile-based analysis to determine dead variables/blocks, and a simple hardware prefetch technique that requires minimal hardware support. The predictor-correlator methods in [64] achieve significant speedups, but at the cost of large hardware tables that require up-to 2MB of storage. A tag-correlating prefetcher (TCP) which uses tag sequence history can be placed at the L2 cache level is presented in [39]. It uses smaller hardware and provides better performance compared to address-based correlating prefetchers. A fixed prefetch block based approach to reduce cache pollution in small caches is presented in [96].

Other Prefetching Approaches

There are many prefetching approaches that target different type of access patterns and use different type of information for prefetching. A prefetching approach based on dependence graph computation is proposed in [3] and an irregular data structure prefetching is considered in [52]. A data prefetch controller which executes a separate instruction stream and cooperates with the processor is presented in [122]. A cooperative hardware-software prefetching scheme called guided region prefetching (GRP) is presented in [126]. This approach uses compiler introduced hints encoded in load instructions to regulate a hardware prefetching engine. An approach to detect strided accesses using profiling and introducing the prefetch instructions in the compiled program is presented in [70].

Finally, my work has not resulted in new prefetch methods, but rather I have focussed on mitigating cache pollution effects caused by aggressive prefetch methods [43].

7.6 Cache Partitioning

Cache partitioning has been proposed to improve performance and predictability by dividing the cache into partitions based on the needs of different data regions accessed in a program. A technique to dynamically partition a cache using column caching was presented in [23]. While column caching can improve predictability for multitasking, it is less effective for single processes. A split spatial/non-spatial cache partitioning approach based on the locality of accesses is proposed and evaluated in [90, 100]. The data is classified into either a spatial or temporal and stored in the corresponding partition. A filter cache is proposed in [98]. It uses two independent L1 caches with different organizations placed in parallel with each cache block containing some reuse information. Another approach that uses a sidebuffer with dynamic allocation is presented in [80].

Another benefit of cache partitioning is that it avoids conflicts between different data regions by mapping them to different regions. An approach to avoid conflict misses dynamically is presented in [6]. There are also approaches for page placement [12, 105], page coloring [8], and data layout [20, 21, 22] to avoid conflict misses. A model for partitioning an instruction cache among multiple processes has been presented [67]. McFarling presents techniques of code placement in main memory to maximize instruction cache hit ratio [74, 115].

An approach to partition the load/store instructions into different cache partitions is presented in [87, 88]. It is evaluated with a direct-mapped cache. In [94] the first level data cache is split into several independent partitions can be distributed across the processor die. The memory instructions are sent to the functional units close to the partition where the data is likely to reside. The goal of this approach is to reduce the latency of large cache arrays. In [117, 120] the memory size requirements of scalars in multimedia systems is explored with the use of this information in cache partitioning. A cache Minimax architecture uses this static predictability of memory accesses to keep the scalars in a small minicache.

In [111] analytical cache models are developed for a multiple process environment and these models are applied to cache partitioning among multiple processes. The optimal cache allocation among two or more processes is studied in [109]. It is shown that the LRU replacement policy is very close to optimal even though in the transient state LRU converges slowly as the allocation approaches the steady-state allocation. It also presents a policy that uses the remaining time quantum for the process and the marginal benefit from allocating more cache space to the running process. In [114] this work is extended with an on-line algorithm for cache partitioning applied to disk caches.

Low Power Caches

There are several cache partitioning approaches that focus on reducing cache power consumption (e.g., [4, 56, 58, 66, 81, 89, 133, 136, 138]). A compiler-directed approach to turn-off parts of the cache containing data not currently in use can lead to some cache energy savings is presented in [137]. A direct addressed cache where tag-unchecked loads and stores are used with the compiler support is presented in [131]. The energy saved in avoiding the tag checks leads to overall energy savings. A combination of the direct addressed cache and software controlled cache line size called Span Cache is presented in [130].

Cache Predictability

In the context of multiprogrammed applications, a cache partitioning strategy is proposed in [104] to improve task scheduling and minimize overall cache size. A process dependent static cache partitioning approach for an instruction cache is presented in [59]. The instruction cache is divided into two partitions where one partition's size depends on the task scheduled to run and the other partition uses LRU.

I have developed a theory for cache partitioning based on disjoint access sequences that can provide guarantees of improved behavior and evaluated it on Spec2000 benchmarks.

7.7 Memory Exploration in Embedded Systems

Memory exploration approaches use different on-chip memory parameters and find appropriate combinations for the desired application domain. Panda, Dutt and Nicolau present techniques for partitioning on-chip memory into scratch-pad memory and cache [84]. The presented algorithm assumes a fixed amount of scratch-pad memory and a fixed-size cache, identifies critical variables and assigns them to scratch-pad memory. The algorithm can be run repeatedly to find the optimum performance point. A system-level memory exploration technique targeting ATM applications is presented in [106]. A simulation-based technique for selecting a processor and required instruction and data caches is presented in [60]. I have focussed on improving cache performance in this thesis.

Chapter 8

Conclusion

8.1 Summary

The on-chip memory in embedded systems may comprise of a cache, an on-chip SRAM, or a combination of a cache and an on-chip SRAM. In this thesis, the problems associated with on-chip cache in embedded systems were addressed to improve cache performance and cache predictability, and to reduce cache pollution due to prefetching.

The problems associated with the cache performance and predictability in embedded systems were addressed using a intelligent replacement mechanism. The theoretical foundation for the cache mechanism was developed along with the hardware and software aspects of the mechanism. Intelligent replacement was evaluated on a set of benchmarks to measure its effectiveness. It improved the performance of the studied benchmarks and improved cache predictability by improving its worst-case application performance over a range of input data. This increased predictability makes caches more amenable for use in real-time embedded systems.

The problem of cache pollution due to prefetching was addressed by integrating the intelligent cache replacement mechanism with hardware prefetching in a variety of ways. I have shown using analysis and experiments with a parameterizable hardware prefetch method that cache pollution, a significant problem in aggressive hardware prefetch methods, can be controlled by using the intelligent cache replacement mechanism. The resultant prefetch strategies improve performance in many cases. Further, I have shown that a new prefetch scheme where 1 or 2 adjacent blocks are prefetched depending on whether there is dead data in the cache or not works very well, and is significantly more stable than standard

sequential hardware prefetching.

The problem of cache performance, cache predictability and cache pollution due to prefetching was also addressed using a cache partitioning approach. A concept of *disjoint sequences* is used as a basis for cache partitioning. I derived conditions that guaranteed that partitioning leads to the same or less number of misses than the unpartitioned cache of the same size. Several partitioning mechanisms were designed which use the concept of disjoint sequences. In particular, the partitioning mechanisms are based on a modified LRU replacement, cache tiles, and multi-tag sharing approach. In addition to the hardware mechanisms, a static partitioning algorithm was developed that used the hardware mechanisms for cache partitioning.

8.2 Extensions

There are several extensions possible for the cache mechanisms described in this thesis. Some of the extensions are described here.

8.2.1 Kill+LRU Replacement for Multiple Processes

The Kill+LRU replacement can be extended to a multi-process environment where the information about process scheduling and process footprints can be used to derive the reuse distance and is subsequently used in determining the dead blocks for a process and marked as killed blocks for Kill+LRU replacement. The operating system may use different types of scheduling methods for processes, e.g., round-robin scheduling, priority-based scheduling. The time slot for which a process runs before the operating system switches to another process may be a parameter to the operating system. For a process, if two accesses to a cache block happen in the same time slot, then the reuse distance would be the same as within the process. But, if the two accesses happen in different time slots of the process, the reuse distance between these two accesses is increased due to the data accessed by the intervening process(es). So, after a time during a time slot, some of the accesses of the current process effectively become the last accesses, and thus the corresponding cache blocks can be marked as killed blocks. This takes into account the information about the next process to execute and its impact on the reuse distance of the current process' accesses. Since different processes may use the same virtual address space, a process number would

need to be kept with the cache blocks in order to identify the cache blocks belonging to other processes. Assuming the cache is not flushed on a context switch, some of the data of the process, if recently run, may still be in the cache and the Kill+LRU replacement can improve on some of these cold misses as well.

8.2.2 Kill+Prefetch at L2 level

The Kill+LRU replacement and its combination with prefetching has been evaluated at the Level 1 (L1) cache level. The Kill+LRU replacement policy can be used in the Level 2 (L2) cache and the prefetching of data from main memory to L2 can be combined with this replacement policy at the L2 level. The L2 cache is accessed when a data access misses in the L1 cache. A block from the L1 cache is evicted and a data block from the L2 cache replaces the evicted block. The L2 cache line size and associativity are bigger than the L1 cache because the effect of L2 cache misses can be significant considering the relatively high memory access latency. Also, the L2 cache may be shared by multiple processors in a shared-memory multiprocessor or by multiple processes in a single processor environment. The L2 cache may have the inclusion property which requires that all the data in L1 cache is also in the L2 cache at all times.

The Kill+LRU replacement at the L2 level works similar to the L1 cache except that (1) the decision to mark an L2 cache block as a killed block is based on the reuse distance condition at the L2 level and (2) an L2 block may be marked as a killed block even if L2 is not accessed on an access, i.e., there is no L1 cache miss. A prefetching approach can be combined with Kill+LRU at the L2 cache level similar to the one for the L1 cache.

8.2.3 Disjoint Sequences for Program Transformations

The concept of disjoint sequences was applied to cache partitioning as discussed in Chapter 5. This concept of disjoint sequences can also be applied to improve performance using program code transformations. The transformations are based on disjoint sequence theorems which show that if we can reorder a program's memory reference stream such that the reordered memory reference stream satisfies a disjointness property, then the transformed program corresponding to the reordered stream is guaranteed to have fewer misses for any cache with arbitrary size or organization so long as the cache uses the LRU replacement policy. We can apply these theoretical results to reorder instructions within a basic block,

to transform loops, to reorder blocks within a procedure, or to reorder procedures within a program so as to improve hit rate for any cache that uses LRU replacement. Based on these theorems, we develop algorithmic methods for program transformation to improve cache performance. The algorithm for transformation uses a disjointness metric to recursively perform the reordering transformation in presence of the dependency constraints. I did some preliminary work on loop transformations using the concept of disjoint sequences [41]. In this work, disjoint sequences were applied for loop distribution transformation. It could be extended to loop fusion, loop interchange, loop tiling, and other loop transformations. Also, the reordering transformation approach can be incorporated into a compiler that targets improved performance.

8.2.4 Profile-based Cache Exploration System

The profile-based evaluation framework can be extended to build a cache exploration system. This system would evaluate a set of cache mechanisms and compare the software/hardware cost and performance trade-off information. Also, the system can evaluate the collective interaction of different mechanisms together and suggest the combination of mechanisms appropriate for the application. We consider the following categories of cache mechanisms for cache memory system exploration: Replacement Policies (LRU and Kill+LRU), Cache Partitioning (way-based, set-based), and Prefetching (combined with Kill+LRU). The memory architecture exploration system would consist of the following components: Configuration/Constraints Language, Assembly/Binary Code Transformation, Software/Hardware Cost Estimator, Simulator, and Optimizer.

Configuration/Constraints Language: The exploration system requires a language to specify the configuration that can be used by the different components. The configuration language should be able to specify the aspects of the cache mechanisms that would be considered by the optimizer. The constraints language specifies the goal constraints for the optimizer.

Assembly Code Transformation: This module is used to transform the assembly/binary code to introduce hints or cache control instructions into the assembly/binary code to evalu-

ate a given configuration. This module is used by the simulator to quantify the performance.

Hardware Cost Estimator: This module takes as input a configuration and estimates the hardware cost of the given configuration. It uses different formulas for logic and memory size estimation. This module uses the transformed assembly/binary code to measure the static software cost, and is also used by the optimizer to obtain the software/hardware cost during the exploration process.

Simulator: This module simulates a given configuration and generates the performance numbers that are used by the optimizer to guide the exploration process. The simulator uses the statically transformed assembly/binary code for simulation.

Optimizer: The optimizer takes the goal constraints and uses an exploration algorithm to explore different cache mechanisms and select the mechanism or a combination of mechanisms that meets the goal constraints.

Bibliography

[1] The ARM9TDMI Technical Reference Manual Rev 3, 2000.

[2] A. Agarwal and S.D. Pudar. Column-Associative Caches: a Technique for Reducing the Miss Rate of Direct-Mapped Caches. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 179–190, 1993.

[3] Murali Annavaram, Jignesh M. Patel, and Edward S. Davidson. Data prefetching by dependence graph computation. In *Proceedings of the 28st Annual International Symposium on Computer Architecture*, pages 52–61, 2001.

[4] Raksit Ashok, Saurabh Chheda, and Csaba Andras Moritz. Cool-mem: Combining statically speculative memory accessing with selective address translation for energy efficiency. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, pages 133–143, San Jose, CA, October 2002.

[5] L.A. Belady. A Study of Replacement Algorithms for a Virtual-Storage Computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[6] Brian K. Bershad, Bradley J. Chen, Denis Lee, and Theodore H. Romer. Avoiding Conflict Misses Dynamically in Large Direct-Mapped Caches. In *ASPLOS VI*, 1994.

[7] Angela Demke Brown and Todd C. Mowry. Taming the Memory Hogs: Using Compiler-Inserted Releases to Manage Physical Memory Intelligently. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI-00)*, pages 31–44, October 2000.

[8] E. Bugnion, J. M. Anderson, T. C. Mowry, M. Rosenblum, and M. S. Lam. Compiler-directed page coloring for multiprocessors. In *Proceedings of the 7th International*

Conference on Architectural Support for Programming Languages and Operating Systems, pages 244–257, Cambridge, MA, October 1996.

[9] D.C. Burger, J.R. Goodman, and A. Kagi. Memory Bandwidth Limitations of Future Microprocessors. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 78–89, Philadelphia, PA, May 1996.

[10] Doug Burger. *Hardware Techniques to Improve the Performance of the Processor/Memory Interface*. PhD thesis, Department of Computer Science, University of Wisconsin at Madison, 1998.

[11] Doug Burger and Todd M. Austin. The simplescalar tool set, version 2.0. Technical report, University of Wisconsin-Madison Computer Science Department, 1997.

[12] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 139–150, San Jose, CA, October 1998.

[13] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. A Study of Integrated Prefetching and Caching Strategies. In *Proceedings of ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 188–197, December 1995.

[14] Calin Cascaval and David A. Padua. Estimating cache misses and locality using stack distances. In *International Conference on Supercomputing*, pages 150–159, San Francisco, CA, USA, June 23-26 2003.

[15] J. P. Casmira and D. R. Kaeli. Modeling Cache Pollution. In *Proceedings of the* 2^{nd} *IASTED Conference on Modeling and Simulation*, pages 123–126, 1995.

[16] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck. Exact analysis of the cache behavior of nested loops. In *Proceedings of the SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 286–297, Snowbird, UT, June 2001.

[17] Chi F. Chen, Se-Hyun Yang, Babak Falsafi, and Andreas Moshovos. Accurate and complexity-effective spatial pattern prediction. In *10th International Symposium on*

High Performance Computer Architecture, pages 276–287, Madrid, Spain, February 14-18 2004.

[18] Tien-Fu Chen. An Effective Programmable Prefetch Engine for On-Chip Caches. In *Proceedings of the 28th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 237–242, Ann Arbor, MI, November 1995.

[19] Tien-Fu Chen and Jean-Loup Baer. Effective Hardware-based Data Prefetching for High-Performance Processors. *IEEE Transactions on Computers*, 44(5):609–623, May 1995.

[20] Trishul M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *Proceedings of the SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 13–24, Atlanta, GA, May 1999.

[21] Trishul M. Chilimbi, Mark D. Hill, and J. R. Larus. Cache-conscious structure layout. In *Proceedings of the SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 1–12, Atlanta, GA, May 1999.

[22] Trishul M. Chilimbi and J. R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *The International Symposium on Memory Management*, pages 37–48, Vancouver, BC, October 1998.

[23] D. Chiou, S. Devadas, P. Jain, and L. Rudolph. Application-Specific Memory Management for Embedded Systems Using Software-Controlled Caches. In *Proceedings of the* 37^{th} *Design Automation Conference*, June 2000.

[24] Jamison Collins and Dean M. Tullsen. Hardware identification of cache conflict misses. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, pages 126–135, 1999.

[25] Jamison D. Collins and Dean M. Tullsen. Runtime identification of cache conflict misses: The adaptive miss buffer. *ACM Transactions on Computer Systems*, 19(4):413–439, November 2001.

[26] Cyrix. Cyrix 6X86MX Processor, May 1998.

[27] Cyrix. Cyrix MII Databook, Feb 1999.

[28] Chen Ding and Yutao Zhong. Predicting Whole-Program Locality through Reuse Distance Analysis. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 245–257, San Diego, CA, USA, June 9-11 2003.

[29] C. Dulong. IA-64 Architecture At Work. *IEEE Computer*, 31(7):24–32, July 1998.

[30] Changpeng Fang, Steve Carr, Soner Onder, and Zhenlin Wang. Reuse-distance-based miss-rate prediction on a per instruction basis. In *The Second ACM SIGPLAN Workshop on Memory System Performance*, Washington, DC, June 8 2004.

[31] S. Ghosh, M. Martonosi, and S. Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 228–239, San Jose, CA, October 1998.

[32] E. H. Gornish and A. V. Veidenbaum. An integrated hardware/software scheme for shared-memory multiprocessors. In *Proceedings of the 1994 International Conference on Parallel Processing*, pages 281–284, St. Charles, IL, 1994.

[33] E. G. Hallnor and S. K. Reinhardt. A fully associative software-managed cache design. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 107–116, Vancouver, Canada, June 2000.

[34] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 2nd edition, 1996.

[35] Mark D. Hill. A Case for Direct-mapped Caches. *IEEE Computer*, 21(12):25–40, December 1988.

[36] Mark D. Hill. Evaluating Associativity in CPU Caches. *IEEE Transactions on Computers*, 38(12):1612–1630, December 1989.

[37] Glenn Hinton, Dave Sager, Mike Upton, Darren Boggs, and Doug Carmean. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, Q1 2001.

[38] Zhigang Hu, Stefanos Kaxiras, and Margaret Martonosi. Timekeeping in the Memory Systems: Predicting and Optimizing Memory Behavior. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, May 2002.

[39] Zhigang Hu, Margaret Martonosi, and Stefanos Kaxiras. Tcp: Tag correlating prefetchers. In *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture*, pages 317–326, Anaheim, CA, February 8-12 2003.

[40] Ravi Iyer. Cqos: A framework for enabling qos in shared caches of cmp platforms. In *International Conference on Supercomputing*, pages 257–266, Saint-Malo, France, June 26-July 1 2004.

[41] Prabhat Jain and Srinivas Devadas. A code reordering transformation for improved cache performance. Comptation Structures Group, Laboratory for Computer Science CSG Memo 436, Massachusetts Institute of Technology, Cambridge, MA, USA 02139, March 2001.

[42] Prabhat Jain, Srinivas Devadas, Daniel Engels, and Larry Rudolph. Software-assisted Cache Replacement Mechanisms for Embedded Systems. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 119–126, November 4-8 2001.

[43] Prabhat Jain, Srinivas Devadas, and Larry Rudolph. Controlling cache pollution in prefetching with software-assisted cache replacement. Comptation Structures Group, Laboratory for Computer Science CSG Memo 462, Massachusetts Institute of Technology, Cambridge, MA, USA 02139, July 2001.

[44] Jaeheon Jeong and Michel Dubois. Optimal replacements in caches with two miss costs. In *Eleventh ACM Symposium on Parallel Algorithms and Architectures*, pages 155–164, Saint Malo, France, June 27-30 1999.

[45] Jaeheon Jeong and Michel Dubois. Cost-sensitive cache replacement algorithms. In *9th International Symposium on High Performance Computer Architecture*, pages 327–336, Anaheim, CA, February 8-12 2003.

[46] Teresa L. Johnson, Daniel A. Connors, Matthew C. Merten, and Wen mei W. Hwu. Run-time cache bypassing. *IEEE Transactions on Computers*, 48(12):1338–1354, December 1999.

[47] Teresa L. Johnson and Wen mei W. Hwu. Run-time Adaptive Cache Hierarchy Management via Reference Analysis. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 315–326, June 1997.

[48] Teresa L. Johnson, M. C. Merten, and Wen mei W. Hwu. Run-time Spatial Locality Detection and Optimization. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 57–64, December 1997.

[49] Norman P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Full-Associative Cache and Prefetch Buffers. In *The 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[50] Martin Kampe, Per Stenstrom, and Michel Dubois. Self-correcting lru replacement policies. In *Proceedings of the 1st Conference on Computing Frontiers*, pages 181–191, Ischia, Italy, 2004.

[51] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. A Matrix-based Approach to Global Locality Optimization Problem. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, pages 306–313, October 1998.

[52] M. Karlsson, F. Dahlgren, and P. Sternstrom. A Prefetching Technique for Irregular Accesses to Linked Data Structures. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, pages 206–217, January 2000.

[53] R. Kessler. The Alpha 21264 Microprocessor: Out-Of-Order Execution at 600 Mhz. In *Hot Chips 10*, August 1998.

[54] Richard E. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2):24–36, March-April 1999.

[55] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An Adaptive, Non-uniform Cache Structure for Wire-delay Dominated On-chip Caches. In *Proceedings of the*

10th International Conference on Architectural Support for Programming Languages and Operating Systems, pages 211–222, October 2002.

[56] S. Kim, N. Vijaykrishnan, M. Kandemir, A. Sivasubramaniam, M. J. Irwin, and E. Geethanjali. Power-aware partitioned cache architectures. In *Proceedings of the 2001 International Symposium on Low Power Electronics and Design (ISLPED'01)*, pages 64–67, Huntington Beach, CA, August 6-7 2001.

[57] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT 2004)*, pages 111–122, Antibes Juan-les-Pins, France, September 29 - October 03 2004.

[58] Soontae Kim, N. Vijaykrishnan, Mahmut Kandemir, Anand Sivasubramaniam, and Mary Jane Irwin. Partitioned instruction cache architecture for energy efficiency. *ACM Transactions on Embedded Computing Systems*, 2(2):163–185, May 2003.

[59] David B. Kirk. Process Dependent Cache Partitioning of Real-Time Systems. In *Proceedings of the IEEE Real Time Systems Symposium*, pages 181–190, 1988.

[60] D. Kirovski, C. Lee, M. Potkonjak, and W. Mangione-Smith. Application-Driven Synthesis of Core-Based Systems. In *Proceedings of the International Conference on Computer-Aided Design*, pages 104–107, November 1997.

[61] A. C. Klaiber and H. M. Levy. An Architecture for Software-controlled Data Prefetching. *SIGARCH Computer Architecture News*, 19(3):43–53, May 1991.

[62] Sanjeev Kumar and C. Wilkerson. Exploiting spatial locality in data caches using spatial footprints. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 357–368, July 2001.

[63] An-Chow Lai and Babak Falsafi. Selective, Accurate, and Timely Self-invalidation Using Last-touch Prediction. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.

[64] An-Chow Lai, Cem Fide, and Babak Falsafi. Dead-Block Prediction & Dead-Block Correlating Prefetchers. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (to appear)*, July 2001.

[65] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In *Proceedings of the international conference on Measurement and modeling of computer systems*, 1999.

[66] Hsien-Hsin S. Lee and Gary S. Tyson. Region-Based Caching: An Enegry-Delay Efficient Memory Architecture for Embedded Processors. In *CASES '00: Proceedings of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 120–127, San Jose, CA, November 17-18 2000.

[67] Y. Li and W. Wolf. A Task-Level Hierarchical Memory Model for System Synthesis of Multiprocessors. In *Proceedings of the* 34^{th} *Design Automation Conference*, pages 153–156, June 1997.

[68] Mikko H. Lipasti, William J. Schmidt, Steven R. Kunkel, and Robert R. Roediger. SPAID: Software Prefetching in Pointer- and Call-intensive Environments. In *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-28)*, pages 231–236, November 1995.

[69] C.-K. Luk and T. C. Mowry. Compiler-based Prefetching for Recursive Data Structures. *ACM SIGOPS Operating Systems Review*, 30(5):222–233, 1996.

[70] Chi-Keung Luk, Robert Muth, Harish Patil, Richard Weiss, P. Geoffrey Lowney, and Robert Cohn. Profile-guided post-link stride prefetching. In *ICS '02: Proceedings of the International Conference on Supercomputing*, pages 167–178, New York, NY, June 22-26 2002.

[71] Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, William J. Dally, and Mark Horowitz. Smart memories: A modular reconfigurable architecture. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 161–171, Vancouver, Canada, June 2000.

[72] N. Maki, K. Hoson, and A Ishida. A Data-Replace-Controlled Cache Memory System and its Performance Evaluations. In *TENCON 99. Proceedings of the IEEE Region 10 Conference*, 1999.

[73] C. May, E. Silha, R. Simpson, H. Warren, and editors. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufmann Publishers, Inc., 1994.

[74] S. McFarling. Program Optimization for Instruction Caches. In *Proceedings of the* 3^{rd} *Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–191, April 1989.

[75] Sally A. McKee, A. Aluwihare, and et. al. Design and evaluation of dynamic access reordering hardware. In *International Conference on Supercomputing*, pages 125–132, May 1996.

[76] Sally A. McKee, R. H. Klenke, and et. al. Smarter Memory: Improving Bandwidth for Streamed References. *IEEE Computer*, 31(7):54–63, July 1998.

[77] Sun Microsystems. UltraSparc User's Manual, July 1997.

[78] Tushar Mohan, Bronis R. de Supinski, Sally A. McKee, Frank Mueller, Andy Yoo, and Martin Schulz. Identifying and exploiting spatial regularity in data memory references. In *Proceedings of the ACM/IEEE SC2003 Conference*, pages 49–59, 2003.

[79] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 62–73, Boston, MA, October 1992.

[80] George Murillo, Scott Noel, Joshua Robinson, and Paul Willmann. Enhancing Data Cache Performance via Dynamic Allocation. In *Project Report*, Rice University, Houston, TX, Spring 2003.

[81] Dan Nicolaescu, Alex Veidenbaum, and Alex Nicolau. Reducing Power Consumption for High-Associativity Data Caches in Embedded Processors. In *Proceedings of the Design, Automation, and Test in Europe Conference (DATE'03)*, pages 1064–1068, Munich, Germany, March 2003.

[82] Toshihiro Ozawa, Yasunori Kimura, and Shin'ichiro Nishizaki. Cache Miss Heuristics and Preloading Techniques for General-purpose Programs. In *Proceedings of the*

28th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-28), November 1995.

[83] Subbarao Palacharla and R.E. Kessler. Evaluating Stream Buffers as a Secondary Cache Replacement. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 24–33, April 1994.

[84] P. R. Panda, N. Dutt, and A. Nicolau. *Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration*. Kluwer Academic Publishers, 1999.

[85] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *Proceedings of* 15^{th} *Symposium on Operating System Principles*, pages 79–95, December 1995.

[86] Jih-Kwon Peir, Yongjoon Lee, and Windsor W. Hsu. Capturing Dynamic Memory Reference Behavior with Adaptive Cache Topology. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, pages 240–250, October 1998.

[87] Peter Petrov and Alex Orailoglu. Performance and power effectiveness in embedded processors - customizable partitioned caches. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(11):1309–1318, November 2001.

[88] Peter Petrov and Alex Orailoglu. Towards effective embedded processors in codesigns: Customizable partitioned caches. In *Proceedings of the Ninth International Symposium on Hardware/Software Codesign*, pages 79–84, Copenhagen, Denmark, 2001.

[89] Gilles Pokam and Francois Bodin. Energy-efficiency Potential of a Phase-based Cache Resizing Scheme for Embedded Systems. In *Proceedings of the 8th IEEE Annual Workshop on Interaction between Compilers and Architectures (INTERACT-8)*, pages 53–62, Madrid, Spain, February 2004.

[90] M. Prvulovic, Darko Marinov, Z. Dimitrijevic, and V. Milutinovic. The split spatial/non-spatial cache: A performance and complexity evaluation. In *IEEE TCCA Newsletters*, pages 8–17, July 1999.

[91] S. Przybylski. The Performance Impact of Block Sizes and Fetch Strategies. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 160–169, Seattle, WA, 1990.

[92] Moinuddin K Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive Insertion Policies for High Performance Caching. In *International Symposium on Computer Architecture*, June 2007.

[93] Moinuddin K. Qureshi, David Thompson, and Yale 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*, Madison, WI, June 2005.

[94] Paul Racunas and Yale N. Patt. Partitioned First-Level Cache Design for Clustered Microarchitectures. In *ICS '03: Proceedings of the International Conference on Supercomputing*, pages 22–31, San Francisco, CA, June 23-26 2003.

[95] Parthasarathy Ranganathan, Sarita Adve, and Norman P. Jouppi. Reconfigurable caches and their application to media processing. In *ISCA '00: Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 214–224, Vancouver, Canada, June 2000.

[96] Pipat Reungsang, Sun Kyu Park, Seh-Woong Jeong, Hyung-Lae Roh, and Gyungho Lee. Reducing Cache Pollution of Prefetching in a Small Data Cache. In *International Conference on Computer Design*, pages 530–533, Austin, TX, 2001.

[97] J. A. Rivers, E. S. Tam, G. S. Tyson, E. S. Davidson, and M. Farrens. Utilizing Reuse Information in Data Cache Management. In *Proceedings of the International Conference on Supercomputing (ICS)*, pages 449–456, July 1998.

[98] Julio Sahuquillo and Ana Pont. The filter cache: A run-time cache management approach. In *25th Euromicro Conference (EUROMICRO '99)*, pages 424–431, 1999.

[99] Suleyman Sair, Timothy Sherwood, and Brad Calder. Quantifying load stream behavior. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture*, pages 197–208, La Jolla, CA, USA, February 2-6 2002.

[100] F.J. Sánchez, A. González, and M. Valero. Software Management of Selective and Dual Data Caches. In *IEEE Computer Society Technical Committee on Computer*

Architecture: Special Issue on Distributed Shared Memory and Related Issues, pages 3–10, March 1997.

[101] Jesus Sanchez and Antonio Gonzalez. A Locality Sensitive Multi-Module Cache with Explicit Management. In *Proceedings of the 1999 International Conference on Supercomupting*, pages 51–59, June 1999.

[102] Vatsa Santhanam, Edward H. Gornish, and Wei-Chung Hsu. Data Prefetching on the HP PA-8000. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 264–273, Denver, CO, June 1997.

[103] Jennifer B. Sartor, Subramaniam Venkiteswaran, Kathryn S. McKinley, and Zhenlin Wang. Cooperative caching with keep-me and evict-me. In *9th Annual Workshop on Interaction between Compiler and Computer Architectures (INTERACT-9)*, pages 46–57, February 13-15 2005.

[104] Sharif M. Shahrier and Jyh-Charn Liu. On predictability and optimization of multiprogrammed caches for real-time applications. pages 17–25, 1997.

[105] Timothy Sherwood, Brad Calder, and Joel Emer. Reducing Cache Misses Using Hardware and Software Page Placement. In *Proceedings of the International Conference on Supercomputing, Rhodes, Greece*, June 1999.

[106] P. Slock, S. Wuytack, F. Catthoor, and G. de Jong. Fast and Extensive System-Level Memory Exploration and ATM Applications. In *Proceedings of 1995 International Symposium on System Synthesis*, pages 74–81, 1997.

[107] Y. Smaragdakis, S. Kaplan, and P. Wilson. Eelru: Simple and effective adaptive page replacement. In *Proceedings of the 1999 ACM Sigmetrics Conference on Measurements and Modeling of Computer Systems*, pages 122–133, May 1999.

[108] A. J. Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[109] Harold S. Stone, Jhon Turek, and Joel L. Wolf. Optimal partitioning of cache memory. *IEEE Transactions on Computers*, 41(9):1054–1068, September 1992.

[110] Rabin A. Sugumar and Santosh G. Abraham. Efficient Simulation of Caches under Optimal Replacement with Applications to Miss Characterization. In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurements and Modeling of Computer Systems*, pages 24–35, May 1993.

[111] G. Edward Suh, Srinivas Devadas, and Larry Rudolph. Analytical cache models with application to cache partitioning. In *15th International Conference on Supercomputing (ICS'01)*, pages 1–12, Sorrento, Italy, June 2001.

[112] Edward S. Tam, Jude A. Rivers, Vijayalakshmi Srinivasan, Gary S. Tyson, and Edward S. Davidson. UltraSparc User's Manual. *IEEE Transactions on Computers*, 48(11):1244–1259, November 1999.

[113] Olivier Temam. An Algorithm for Optimally Exploiting Spatial and Temporal Locality in Upper Memory Levels. *IEEE Transactions on Computers*, 48(2):150–158, February 1999.

[114] Dominique Thiebaut, Harold S. Stone, and Joel L. Wolf. Improving disk cache hit-ratios through cache partitioning. *IEEE Transactions on Computers*, 41(6):665–676, June 1992.

[115] H. Tomiyama and H. Yasuura. Code Placement Techniques for Cache Miss Rate Reduction. *ACM Transactions on Design Automation of Electronic Systems*, 2(4):410–429, October 1997.

[116] Gary Tyson, Matthew Farrens, John Matthews, and Andrew R. Pleszkun. A Modified Approach to Data Cache Management. In *Proceedings of the 28th Annual International Symposium on Microarchitecture Ann Arbor, MI*, November/December 1995.

[117] O. Unsal, Z. Wang, I. Koren, C. Krishna, and C. Moritz. On memory behavior of scalars in embedded multimedia systems, 2001.

[118] Osman S. Unsal, R. Ashok, Israel Koren, C. Mani Krishna, and Csaba Andras Moritz. Cool-cache for hot multimedia. In *Proceedings of the 34th International Symposium on Microarchitecture*, pages 274–283, December 2001.

[119] Osman S. Unsal, R. Ashok, Israel Koren, C. Mani Krishna, and Csaba Andras Moritz. Cool-cache: A Compiler-enabled Energy Efficient Data Caching Framework for Embedded and Multimedia Systems. *ACM Transactions on Embedded Computing Systems, Special Issue on Low Power*, 2(3):373–392, August 2003.

[120] Osman S. Unsal, Israel Koren, C. Mani Krishna, and Csaba Andras Moritz. The minimax cache: An energy-efficient framework for media processors. In *HPCA*, pages 131–140, 2002.

[121] Steven P. Vanderwiel and David J. Lilja. When caches aren't enough: Data prefetching techniques. *IEEE Computer*, pages 23–30, July 1997.

[122] Steven P. VanderWiel and David J. Lilja. A compiler-assisted data prefetch controller. In *International Conference on Computer Design (ICCD '99)*, pages 372–377, Austin, TX, October 10-13 1999.

[123] Steven P. Vanderwiel and David J. Lilja. Data Prefetch Mechanisms. *ACM Computing Surveys*, 32(2):174–199, June 2000.

[124] Alexander V. Veidenbaum, Weiyu Tang, Rajesh Gupta, Alexandru Nicolau, and Xiaomei Ji. Adapting Cache Line Size to Application Behavior. In *International Conference on Supercomputing*, June 1999.

[125] Xavier Vera, Bjorn Lisper, and Jingling Xue. Data cache locking for higher program predictability. In *Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 272–282, 2003.

[126] Zhenlin Wang, Doug Burger, Steven K. Reinhardt, Kathryn S. McKinley, and Charles C. Weems. Guided region prefetching: A cooperative hardware/software approach. In *The 30th Annual International Symposium on Computer Architecture*, pages 388–398, San Diego, CA, June 2003.

[127] Zhenlin Wang, Kathryn S. McKinley, and Doug Burger. Combining cooperative software/hardware prefetching and cache replacement. In *IBM Austin CAS Center for Advanced Studies Conference*, Austin, TX, February 2004.

[128] Zhenlin Wang, Kathryn S. McKinley, Arnold L. Rosenberg, and Charles C. Weems. Using the Compiler to Improve Cache Replacement Decisions. In *Proceedings of the*

2002 International Conference on Parallel Architectures and Compilation Techniques (PACT'02), September 2002.

[129] E. Witchel, S. Larsen, C. S. Ananian, and K. Asanovic. Energy efficient architectures: Direct addressed caches for reduced power consumption. In *Proceedings of the 34th International Symposium on Microarchitecture*, pages 124–133, December 2001.

[130] Emmett Witchel and Krste Asanovic. The span cache: Software controlled tag checks and cache line size. In *Workshop on Complexity-Effective Design, 28th International Symposium on Computer Architecture*, Gothenburg, Sweden, June 2001.

[131] Emmett Witchel, Sam Larsen, C. Scott Ananian, and Krste Asanovic. Direct addressed caches for reduced power consumption. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-34)*, pages 124–133, Austin, TX, December 2001.

[132] W. A. Wong and J.-L Baer. Modified LRU policies for improving second-level cache behavior. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, 1999.

[133] Jun Yang, Jia Yu, and Youtao Zhang. Lightweight Set Buffer: Low Power Data Cache for Multimedia Application. In *ISLPED*, pages 270–273, Seoul, Korea, August 25-27 2003.

[134] Se-Hyun Yang, Michael D. Powell, Babak Falsafi, and T. N. Vijaykumar. Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture*, pages 151–161, La Jolla, CA, USA, February 2-6 2002.

[135] Kenneth C. Yeager. The mips r10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, April 1996.

[136] Chuanjun Zhang, Frank Vahid, and Walid Najjar. A Highly Configurable Cache Architecture for Embedded Systems. In *ISCA '03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 136–146, San Diego, CA, 2003.

[137] W. Zhang, M. Karakoy, M. Kandemir, and G. Chen. A Compiler Approach for Reducing Data Cache Energy. In *ICS '03: Proceedings of the International Conference on Supercomputing*, pages 76–85, San Francisco, CA, June 23-26 2003.

[138] Youtao Zhang and Jun Yang. Low Cost Instruction Cache Designs for Tag Comparison Elimination. In *ISLPED*, Seoul, Korea, August 25-27 2003.

[139] Yutao Zhong, Steven G. Dropsho, and Chen Ding. Miss Rate Prediction across All Program Inputs. In *Proceedings of the 2003 International Conference on Parallel Architectures and Compilation Techniques (PACT'03)*, pages 79–90, September 27-October 1 2003.

Appendix A

Additional Information and Data

In this appendix, we give variant algorithms and additional empirical evaluation on selected algorithms.

A.1 Kill+LRU Replacement and Other Approaches

Kill with LRU and Making Dead Block LRU

In the Kill+LRU replacement, a dead block is marked by setting its kill bit upon the last access to that block. The kill bits are used along with the LRU ordering information to make replacement decisions. In a different approach, dead blocks are handled without the use of kill bits and only with the help of the LRU ordering information. In this approach, a block is killed by making it the LRU block. This approach performs is equivalent to the MRK_LRU variation of the Kill+LRU replacement. Since there is no additional kill bit used with each block, there is no way to distinguish between a dead block at the LRU position and a live block at the LRU position. Moreover, it does not provide any information about the number of dead blocks in the LRU ordering of the blocks which is useful in the Kill+Keep+LRU replacement and Kill+Prefetch+LRU as described later.

Kill with LRU and Split Spatial/Temporal Cache

In the split spatial/temporal cache, the data is sent either to spatial cache or temporal cache. The data is classified either as spatial data or as temporal data. The classification of the data may be done at compile time or done dynamically at run-time. Since the spatial

and temporal caches are separate structures, the combined cache space in these structures may not be used effectively if one type of data causes thrashes its cache structure while the other cache structure is underutilized. Moreover, if the access pattern for one type of data changes to the other type then the data needs to be moved to the other cache structure. The Kill+LRU replacement provides a general mechanism to handle both classes of data. The spatial type of data results in dead blocks and the dead blocks are marked by setting their kill bits. The temporal data is not specifically marked but the temporal type of data benefits by the Kill+LRU replacement because the dead blocks are chosen first for replacement. An additional Keep state is added in the Kill+Keep+LRU replacement which can be used to indicate the temporal type of data. This allows the temporal type of data to be potentially kept longer in the cache.

A.2 OPT Improvement and MRK-based Estimation

One way of estimating the MRK-based miss rate improvement over LRU was described in Section 3.3.1. Another way of estimating the miss rate improvement uses the information from the reuse distance profile and augments the estimation formula in the following way. For each reuse distance $d = a$ to $d = d_{max}$, the contribution is computed the same way as before, but with each step i ($i = a$ to $i = d_{max}$), the weight W_j of the reuse distances $j = d_{max}$ down to $j = a$ are reduced based on weight W_i. This reduction in weights of the higher reuse distances accounts for the replacement of a block by another block with a different reuse distance. The comparison of the MRK-based estimated improvement using the above formula with the OPT improvement over LRU is shown in Figure A-1. The comparison of OPT improvement, the formula in Section 3.3.1, and the formula in this section is shown in Figure A-2. The OPT miss rate improvement over LRU for 2-way and 4-way caches for 100 Million and 1 Billion instructions are shown in Table A.1 and Table A.2 respectively. The tables show the results for three different cache sizes: 8K, 16K, and 32K. The cumulative set reuse distance profiles for the Spec2000 benchmarks for 100 Million and 1 Billion instructions using a 4-way 16K cache are shown in Figure A-3 through Figure A-6.

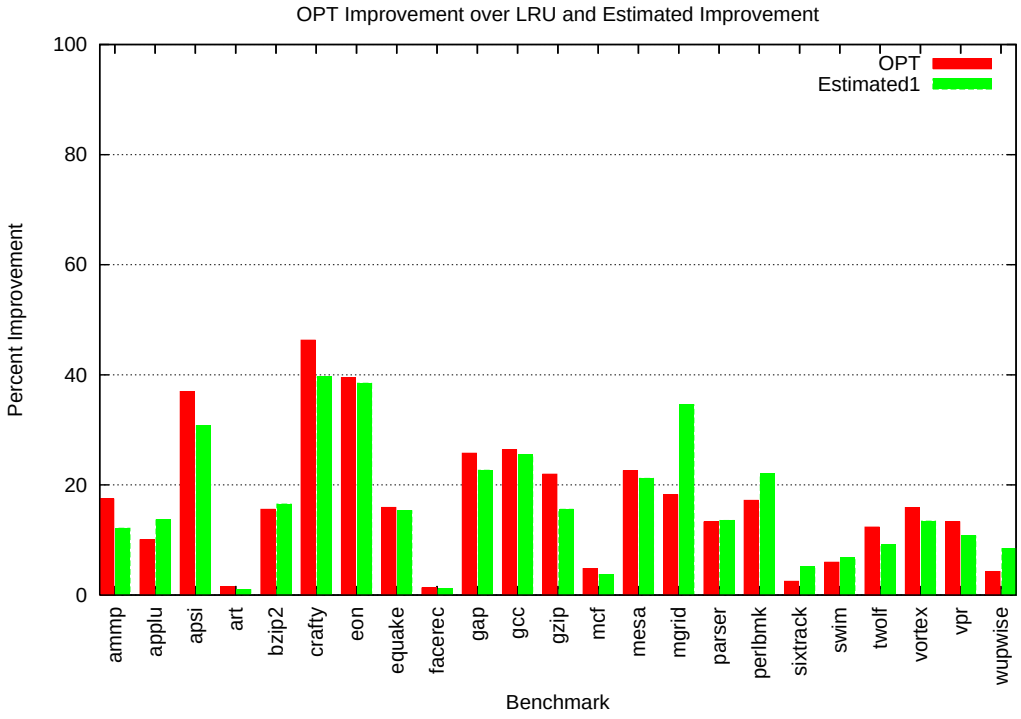


Figure A-1: OPT Improvement and Estimated Improvement Formula 1 Based on Profile data and Uniform Distance Model

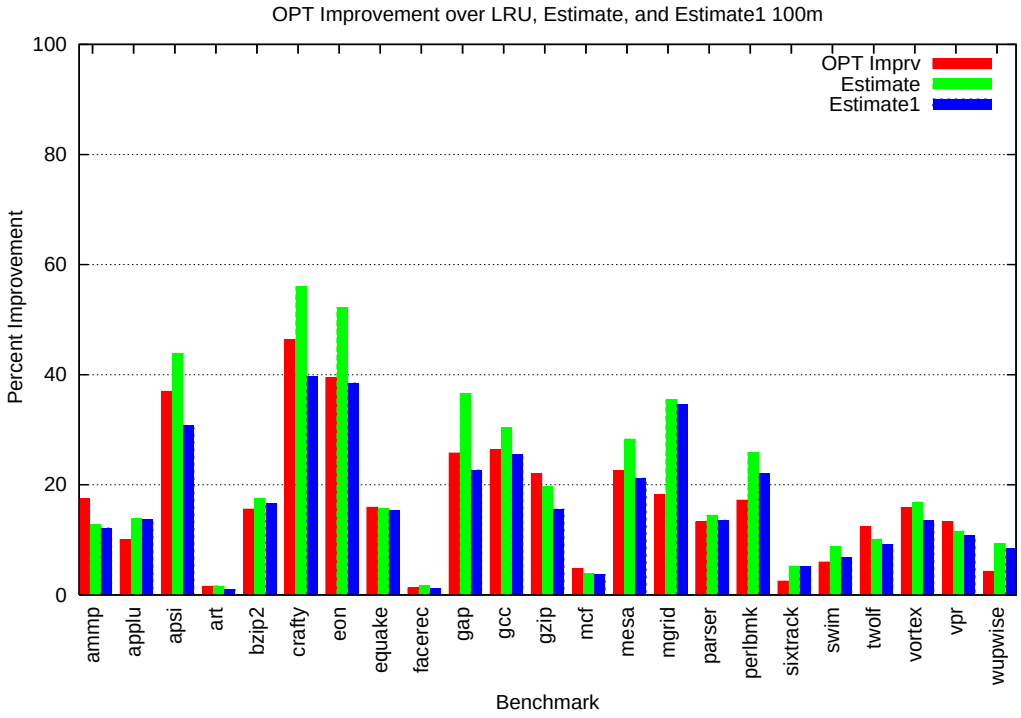


Figure A-2: OPT Improvement and Estimated Improvement Formulas Based on Profile data and Uniform Distance Model

Benchmark	8K 100M	8K 1B	16K 100M	16K 1B	32K 100M	32K 1B
ammp	12.29	12.38	13.90	13.61	14.48	13.69
applu	23.50	100.00	11.47	100.00	2.76	100.00
apsi	27.97	20.94	34.19	20.90	22.47	9.63
art	0.60	0.59	1.07	1.07	2.30	2.30
bzip2	13.89	12.75	13.70	12.75	14.30	13.41
crafty	39.02	38.72	42.10	41.90	41.26	41.42
eon	35.83	36.12	32.58	33.18	34.52	34.93
equake	12.53	12.84	11.22	11.57	9.57	9.95
facerec	4.43	4.67	1.94	2.14	2.03	2.20
gap	14.03	7.36	18.34	6.51	12.44	2.69
gcc	21.09	23.14	20.61	22.47	20.38	22.41
gzip	12.72	14.07	16.40	17.44	21.39	22.00
mcf	2.86	2.62	3.50	3.28	4.05	3.83
mesa	33.42	29.75	20.05	15.52	22.28	15.65
mgrid	17.40	15.83	13.92	9.81	0.56	2.00
parser	14.05	15.04	10.92	11.91	10.92	12.20
perlbmk	17.90	17.73	12.72	13.48	14.44	15.17
sixtrack	36.52	36.56	14.53	14.52	2.51	2.57
swim	0.20	100.00	5.06	100.00	9.83	100.00
twolf	13.10	13.08	11.47	11.44	11.61	11.57
vortex	14.51	14.36	11.96	11.59	16.93	16.49
vpr	16.33	16.64	11.91	12.98	11.26	12.99
wupwise	10.17	13.00	6.39	9.47	3.26	4.36

Table A.1: OPT Miss Rate Improvement Percentage over LRU for 2-way Associative 8K, 16K, 32K Caches

Benchmark	8K 100M	8K 1B	16K 100M	16K 1B	32K 100M	32K 1B
ammp	15.70	15.67	17.45	17.17	18.21	17.31
applu	33.33	100.00	10.03	100.00	0.32	100.00
apsi	27.02	23.54	36.97	22.70	28.82	12.28
art	0.76	0.76	1.56	1.56	3.27	3.27
bzip2	15.39	15.03	15.57	15.16	16.86	16.19
crafty	45.32	44.73	46.29	46.27	42.97	42.58
eon	49.82	48.55	39.49	39.93	38.50	39.62
equake	17.14	17.75	15.91	16.56	15.17	15.85
facerec	13.69	13.71	1.28	1.95	2.54	2.75
gap	11.85	6.77	25.77	10.52	23.24	5.00
gcc	25.39	27.67	26.33	28.40	28.25	29.94
gzip	16.82	18.17	21.95	23.11	28.54	29.37
mcf	3.63	3.39	4.80	4.54	6.34	6.04
mesa	45.21	39.28	22.57	16.68	30.26	20.35
mgrid	26.90	25.53	18.20	12.35	1.25	3.44
parser	14.15	15.19	13.27	14.86	14.33	16.12
perlbmk	19.36	19.33	17.21	17.63	18.48	19.69
sixtrack	36.29	36.29	2.49	2.52	1.16	1.41
swim	2.92	100.00	5.96	100.00	14.60	100.00
twolf	13.29	13.25	12.33	12.31	15.99	15.97
vortex	16.18	15.96	15.89	15.50	23.66	22.93
vpr	15.48	16.28	13.29	14.93	15.13	17.52
wupwise	15.92	17.51	4.28	6.43	6.44	6.30

Table A.2: OPT Miss Rate Improvement Percentage over LRU for 4-way Associative 8K, 16K, 32K Caches

A.3 Multi-tag Sharing

Some details of the ways to reduce the tag overhead in the context of multi-tag sharing approach for partitioning are described below:

- **Rotating tags**: The idea is that when a cache line data space is shared by more than one tag, then the tags effectively rotate to bring out a different tag for comparison. The other tag(s) is(are) driven on the lines that are stored in a buffer for comparison in subsequent cycles. This approach maintains the same associativity logic, but provides the possibility of using more than one tag per cache line. This can effectively increase the associativity without increasing the delay.

- **Differential tags**: In order to reduce the tag overhead, the tags can be stored as differentials with respect to one tag. The tag can be divided into different parts such as [part1, part2, part3, part4] and the differential information can be stored based on these parts. Depending on the space available, more than one part can differ and the corresponding bit masks are kept to construct a different tag. The differential tags are used to minimize the tag space such that more tags can be stored and share the cache line space.

- **Table of Distinct Tags**: This is another way to reduce the overhead of the tags. A table stores distinct tags and the cache line tag space has an index in this table to get the actual tag. But, this will slow down the hit time because of an additional lookup.

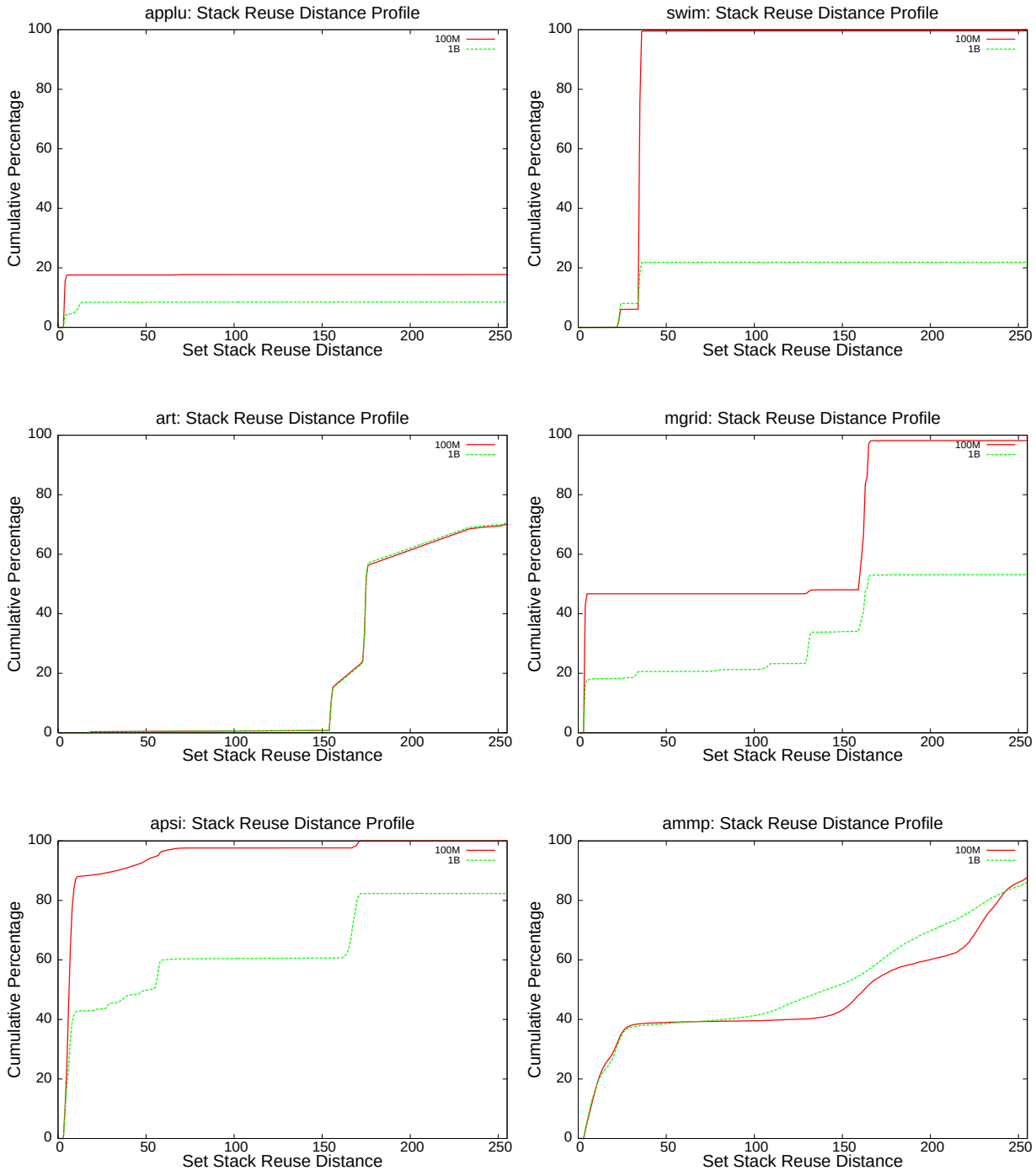


Figure A-3: Cumulative Set Reuse Distance Profiles for Spec2000FP benchmarks (applu, swim, art, mgrid, apsi, ammp) for 100M and 1B instructions using a 4-way 16K Cache

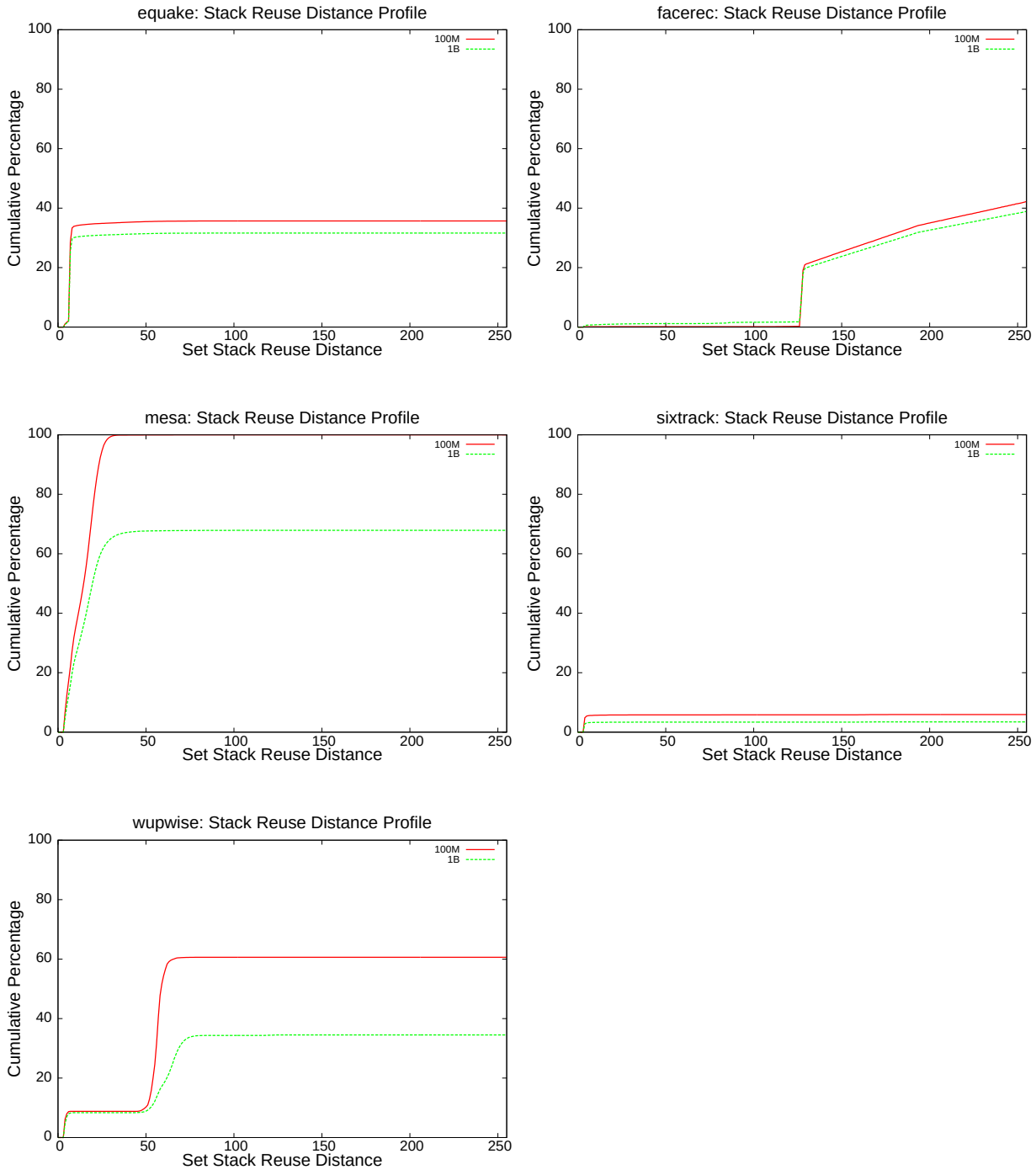


Figure A-4: Cumulative Set Reuse Distance Profiles for Spec2000FP benchmarks (equake, facerec, mesa, sixtrack, wupwise) for 100M and 1B instructions using a 4-way 16K Cache

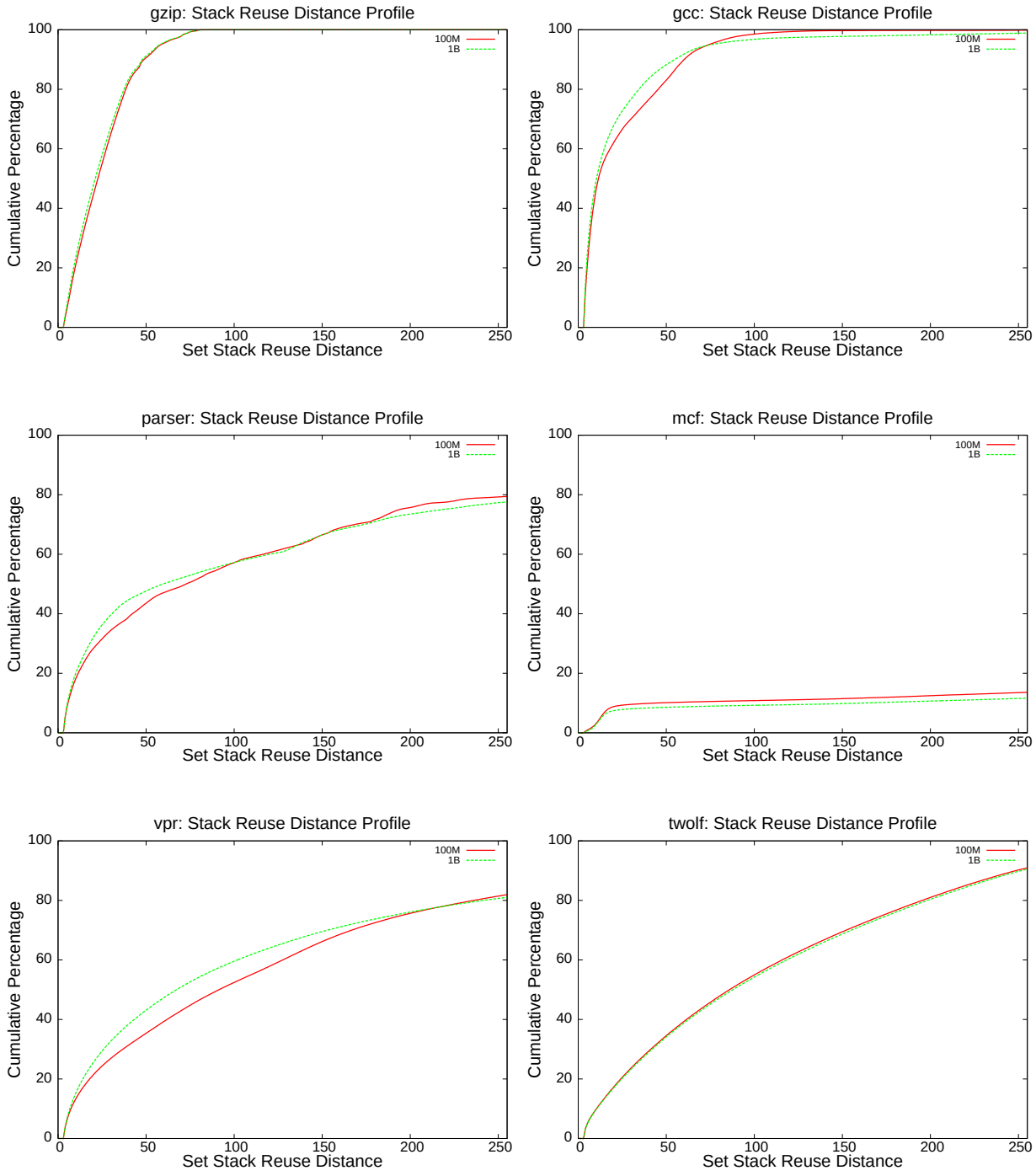


Figure A-5: Cumulative Set Reuse Distance Profiles for Spec2000INT benchmarks (gzip, gcc, parser, mcf, vpr, twolf) for 100M and 1B instructions using a 4-way 16K Cache

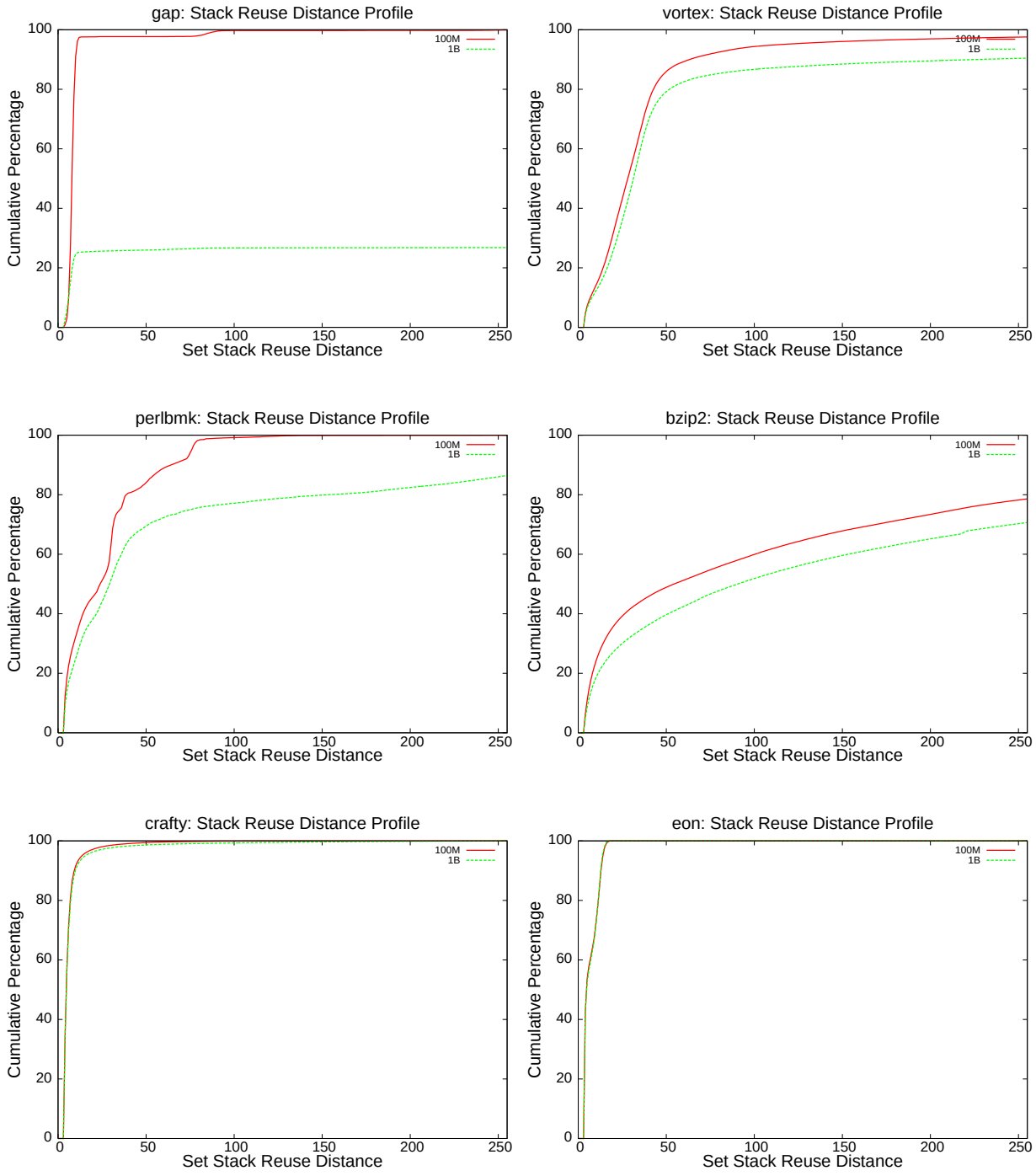


Figure A-6: Cumulative Set Reuse Distance Profiles for Spec2000INT benchmarks (gap, vortex, perlbmk, bzip2, crafty, eon) for 100M and 1B instructions using a 4-way 16K Cache