# Software-assisted Cache Mechanisms for Embedded Systems

by

Prabhat Jain

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 14, 2007

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

Accepted by .....
Arthur C. Smith
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 14, 2007, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

## Abstract

Increasingly embedded systems are 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 and stable prefetching that can lead to predictable cache performance (or cache predictability).

A profile-based analysis, annotation generation, and simulation framework has been implemented to evaluate the cache mechanisms. This profile-based framework can take a compiled benchmark and a set of inputs and evaluate various cache mechanisms and provide a range of possible performance improvement scenerios. The cache mechanisms have been evaluated using this framework to gather the miss rate and IPC and software/harware cost vs. performance tradeoff information. The results show that the proposed cache mechanisms can improve performance at a modest cost.

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. But, the ability to integrate more functionality onto a chip results in increased design complexity. While technologically feasible, it is still impractical and uneconomical to design an entire embedded system in a single custom integrated circuit. Most embedded systems use

both programmable components, Field Programmable Logic, and application-specific integrated circuit (ASIC) logic. The programmable components 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, RAM, application-specific circuitry (ASIC), and peripheral circuitry.

The ASIC logic provides the performance and predictability 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, the software implementations can accommodate late changes in the requirements or design, thus reducing the length of the design cycle. The software can also help in the product evolution – simply changing the software may be enough to augment the functionality for the next generations 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. Problems with embedded memory systems are described in the following sections. I target performance and predictability improvement of on-chip memory through the development of new intelligent hardware mechanisms. 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 equally applicable to general-purpose systems.

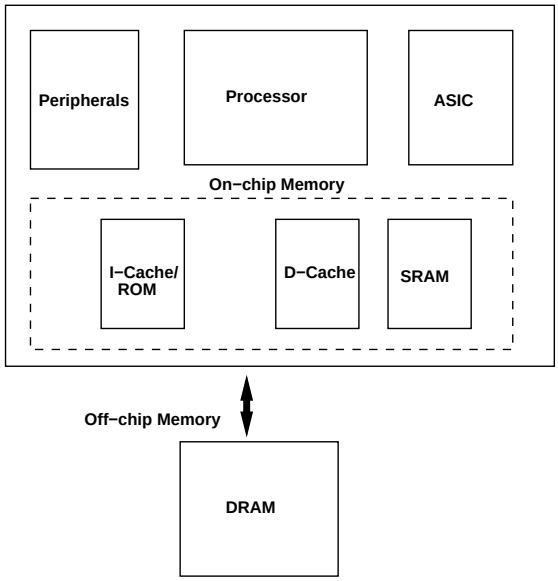


Figure 1-1: $\surd$ Embedded System and On-chip Memory

## 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. There are several problems associated with the use of caches and SRAM when they are used as on-chip memory for an embedded systems. The specific problems addressed in this thesis are described in the next section. My primary focus in this thesis is on the development of intelligent hardware mechanisms for on-chip memory that address these problems.

## 1.3 Research Problems

### 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. The most commonly used replacement strategy is 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 don't use application-specific information in managing their resources. and there is very little software control or assistance with hardware support to manage the cache resources effectively. As a result the caches do not perform well for many 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 several attempts to improve cache performance and my approach draws on previous work but also extends it significantly.

### 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, that 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.

### 1.3.3 Cache Pollution with Prefetching

One performance improvement technique for caches is data prefetching where data that is 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 [183]. 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 [23]. 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 [183]. 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.

## 1.4 Thesis Contributions

I propose to make the on-chip memory (cache and SRAM) intelligent by using new hardware mechanisms. The mechanisms address the different problems I described that are associated with on-chip cache and SRAM. The mechanisms enhance the capabilities of the on-chip memory in terms of better data storage management, computation support, and improved performance and predictability.

The contributions of this thesis consist of three mechanisms to address the problems associated with the use of caches and on-chip SRAM in an embedded system. The specific contributions are as follows:

### 1.4.1 Intelligent Cache Replacement

The LRU cache replacement policy is augmented with additional information to make more intelligent replacement decisions. In particular, a *kill* state and a *keep* state is used along with the LRU information to create an intelligent replacement policy. In this context, the contributions are:

- **Theoretical analysis:** The conditions are derived and proved which guarantee that the number of misses in the intelligent cache replacement mechanism are less than the LRU replacement policy for the fully-associative and set-associative caches. Similar theorems are also proven for the case where an additional *keep* state information is

combined with the intelligent replacement mechanism. Some additional results are derived when prefetching is combined with the intelligent replacement mechanism.

- **Hardware Mechanisms:** Some hardware mechanisms are designed to approximate the conditions to set the kill state information. In particular, the hardware mechanisms based on history signature functions and reuse distance are designed and implemented in a Simplescalar [18] simulator. Kill state information is used to control cache pollution caused by an aggressive hardware prefetch scheme. The intelligent replacement mechanism supported by the hardware mechanisms is evaluated alone and with prefetching on the Spec2000 benchmarks.

- **Software-assisted Mechanisms:** Some software-assisted mechanisms are designed that incorporate application-specific information in the form of hints and control instructions to support the intelligent replacement mechanisms. A 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 alone and with prefetching on the Spec2000 benchmarks.

### 1.4.2 Disjoint Sequences and Cache Partitioning

I propose the concept of *disjoint sequences* that is used to analyze memory access sequences and transform an access sequence into a concatenation of disjoint access sequences. This result is used as a basis for cache partitioning mechanisms and a static cache algorithm to address the problems of cache performance, cache predictability, or cache pollution due to prefetching. The specific contritbutions are:

- **Theorems:** I proved 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 the that cache partitioning leads to the same or less number of misses than the unpartitioned cache of the same size.

- **Partitioning Mechanisms:** Several partitioning mechanisms are designed and explored 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 is developed which uses the hardware mechanisms for cache partitioning. The cache partitioning mechanisms are implemented in the Simplescalar simulator and evaluated with and without prefetching on the Spec2000, Mibench, and Mediabench benchmarks. The static cache partitioning results and more details are presented in Chapter 8.

## 1.5 Thesis Organization

In Chapter 1, the performance bottlenecks of current processors is outlined and research problems are identified and thesis contributions are summarized.

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, the profile-based approach is described in detail. The types of information collected from the profile runs is described, along with the profile-based annotation generation algorithm that uses the profile-based information.

In Chapter 4 and 5, the details of the intelligent cache replacement mechanism are described. In particular, the theorems, the hardware and software assisted mechanisms, implementation of the mechanisms, and evaluation of the intelligent cache replacement mechanism are presented.

In Chapter 6, the combination of the MI-LRU with Keep mechanism and MI-LRU mechanism with prefetching is considered in detail. The Keep algorithm along with the Keep selection and Keep decay options are described as part of the Keep mechanism. The two prefetching schemes considered for the combination with the MI-LRU are described with the hardware required to implement these schemes. The benefits of combining MI-LRU with Keep and prefetching are evaluated in Chapter 9.

In Chapter 7, the new instructions in the context of the proposed cache mechanisms are described in detail. The different fields of the instructions along with the instruction formats are shown for the new instructions. The possible hardware support logic is described for some of the instructions that require the hardware support.

In Chapter 8, the concept of disjoint sequences, a theoretical result on 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 along with the evaluation of mechanisms.

In Chapter 9, the evaluation of the proposed cache mechanisms is described using as subset of Spec2000, Mediabench, and Mibench benchmarks under different scenerios.

In Chapter 10, related work in the context of the intelligent cache replacement mechanism, cache partitioning, and intelligent SRAM is summarized. In Chapter 11 a summary of the thesis is followed by some example scenarios of combinations of mechanisms and 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).

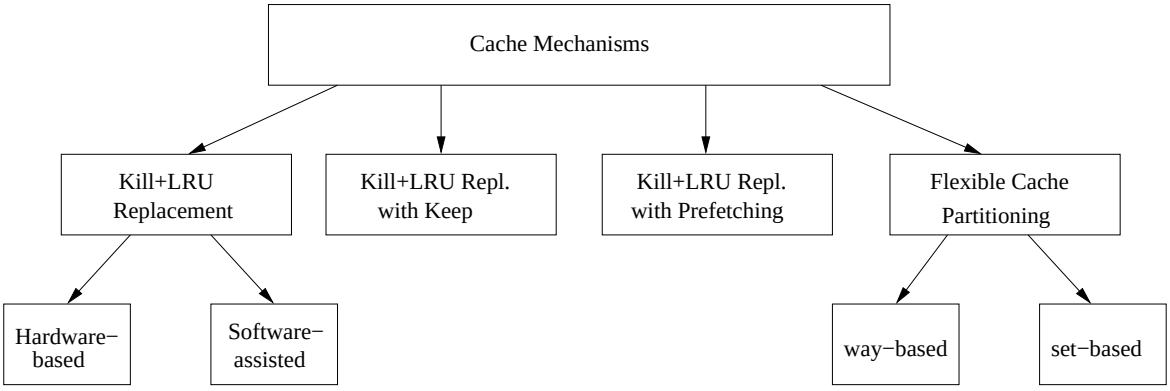


Figure 2-1: $\surd$ Overview of the Mechanisms

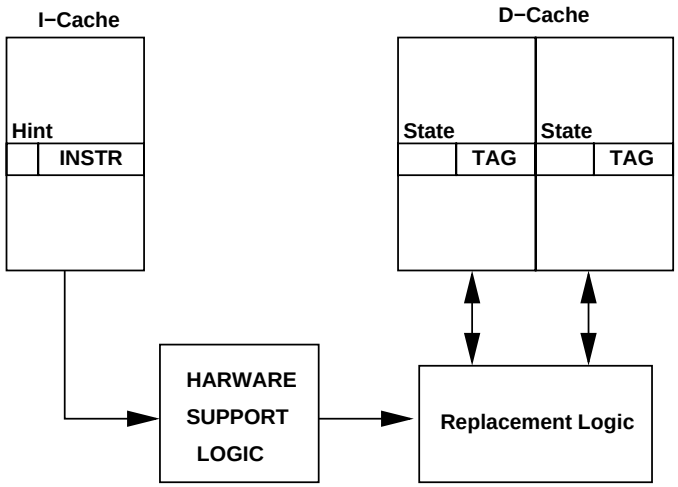


Figure 2-2: $\surd$ Overview of Intelligent Replacement

Each mechanism I propose targets a particular problem associated with using the 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 targetted for the use of on-chip memory as a cache where the intelligent 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

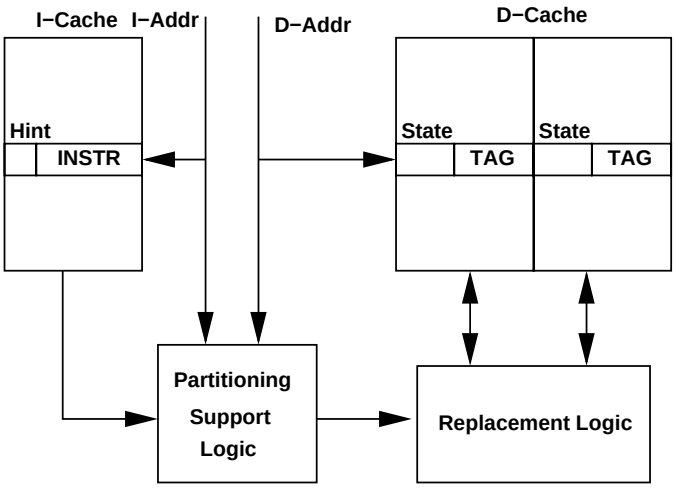


Figure 2-3: $\surd$ Overview of Cache Partitioning

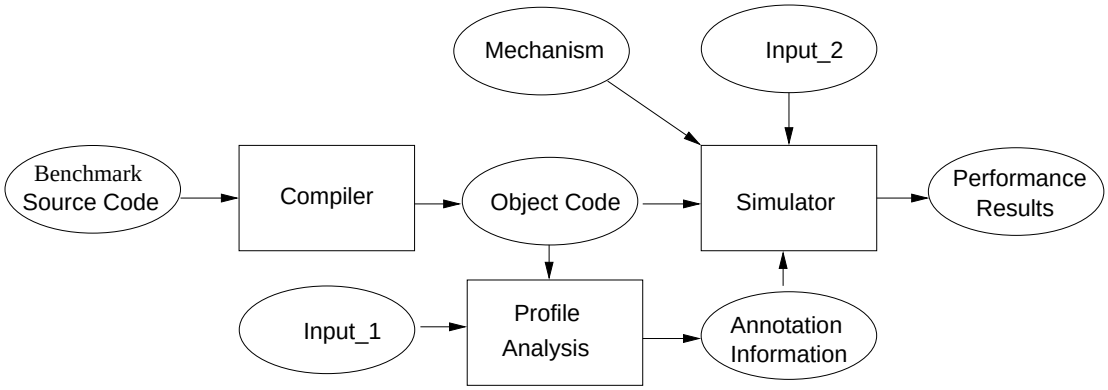


Figure 2-4: $\surd$ Overall Flow

or software-assisted augmentation of the LRU replacement strategy.

The cache pollution due to aggressive hardware prefetching is measured for the LRU replacement and the Intelligent cache replacement is also 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 flexible cache partitioning approach based on disjoint sequences is explored in this thesis. A high-level overview of the cache partitioning is shown in Figure 2-3. The cache partitioning related information is derived using profile-based analysis or dynamically in hardware. The profile-based information is conveyed through some hints or additional instructions.

## 2.2 Evaluation Methodology

### 2.2.1 Overall Flow

The overall flow of information for the intelligent on-chip memory is shown in Figure 2-4. 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 on-chip cache mechanisms.

### 2.2.2 Program Analysis

Two ways of program analysis considered here 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 data. Profile analysis of the program uses a training input data set and gets program related information that may not be available from the 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.

### 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 [18] 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 the Spec2000 benchmarks. The Spec2000 benchmarks consist of the SpecInt2000 - integer benchmarks and SpecFP2000 - 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 alone to evaluate performance and predictability offered by this mechanism. It was also evaluated with prefetching to measure the effectiveness in reducing cache pollution.

# Chapter 3

# ♣ Profile-based Approach

The goal is to provide more information to the cache memory system in making certain cache management decisions. In the profile-based approach the application information is derived through some runs using training or test inputs. The derived application information is used in the cache management in the form of hints, control instructions, and hardware assistance to make cache management decisions. For example, the annotations can provide information to choose signature function and signature table index (hash function) in an approach that uses signatures to predict and augment the cache replacement decisions.

I focus on the profile-based approach instead of a hardware-based approach because collecting information in hardware can take hardware resources, the collected information can be kept for a limited time. If the load/store instruction or cache block for which the information is collected is not used again, the information collection is too late for that instance. In some cases, if the accesses are predictable, the necessary information can be collected for one cache set and used to predict the behavior for the other cache sets associated with the load/store instruction of interest. For the parts of the program that do not have the profile-based information only static information is used in making the cache related decisions.

## 3.1 Profile Analysis

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. I use the profile information derived offline in the profile-based algorithms, but the profile

information can also be derived online in a dynamic profiling hardware. The advantage of offline profile information is that it requires less hardware and gives more leverage in deriving the application specific information.

The profile information to be gathered can be 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 profile-based annotations can lead to small overhead in terms of the static code size, fetch and issue bandwidth, and the instruction cache.

## 3.2 Static Information

The profile-based static information about the program is collected, but this information ideally can be obtained through compiler analysis.

### 3.2.1 Basic Blocks

A basic block is defined as a sequence of non-control instructions that terminate in a control (branch, call, or return) instruction. For the analysis of interest, I collect the load/stores of basic blocks.

The basic blocks whose control points are branches are used in deriving the loop information for the programs.

### 3.2.2 Loop Information

The source-level loop boundaries may not correspond directly to the ones in the compiled program due to compiler optimizations. Also, the annotations are obtained for the compiled program. So, I chose to obtain the loop information from the compiled programs. I derive the loop information from the analysis of the branch targets on the program inputs of interest. This information can also be obtained from the static analysis of the compiled programs.

The branches that don't have any forward branch targets and have only backward branch targets and possibly next_pc target are considered loop branches.

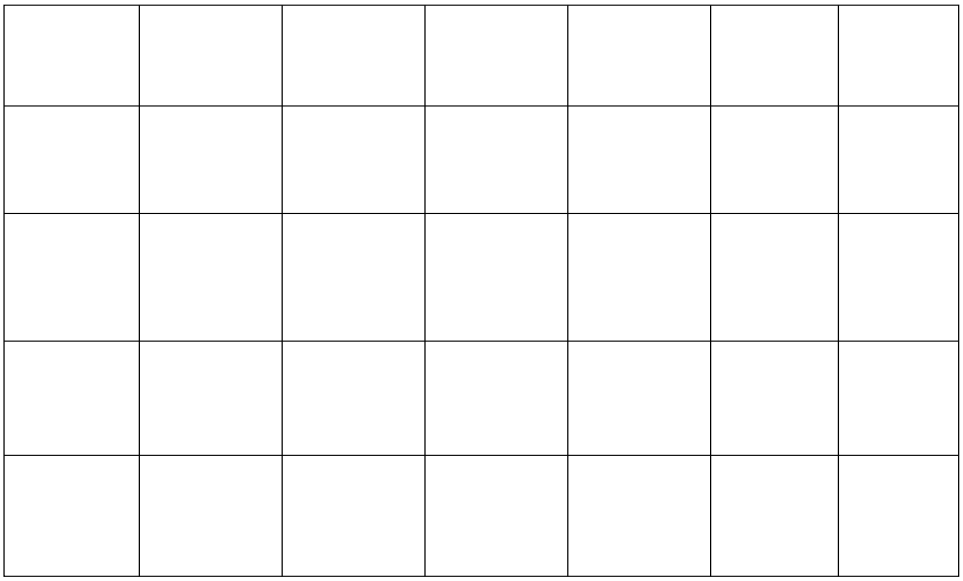

Figure 3-1: □ Basic Block Information

The loop information is used to identify the annotation points for range-based instructions. The annotation points are loop entry and loop exit points.

### 3.2.3 Addressing Modes

The addressing mode of a load/store instruction provides help in the analysis and classification of the accesses.

In the Alpha architecture, there are four address modes that are used. The Global Pointer-based (gp+const) mode uses the gp (r29). The Stack Pointer-based (sp+const) mode uses the sp (r30). The Frame Pointer-based (fp+const) mode uses the fp (r15). The other register based (reg+const) mode uses the integer register. The static load/stores that always access the same address are determined for each addressing mode separately.

### 3.2.4 Address Segment

## 3.3 Derived Information

### 3.3.1 Number of Hits Profile

The blocks with spatial accesses, the number of hits before replacement is relatively small compared to the blocks with temporal accesses. Also, the load/store instructions with spatial accesses the number of hits before replacement is clustered around a small number. This

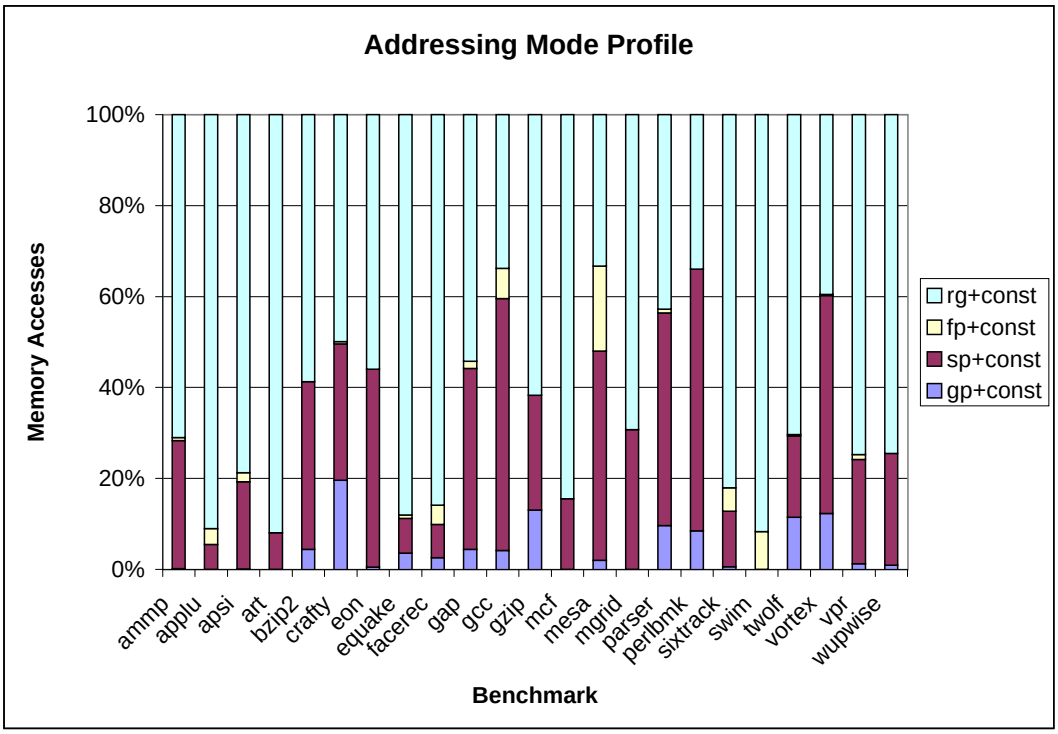


Figure 3-2: Memory Access Addressing Mode Profile of Spec2000 Benchmarks

information may not be used directly, but it provides the information about the cache blocks and the associated load/store instructions that can be targets for profile-based annotations.

### 3.3.2 Reuse Distance Profile

The reuse distance $d$ between the two accesses to the same block $b$ is the number of distinct blocks $d$ between the two accesses. It also determines the increase in hit rate as associativity is increased keeping other cache parameters constant. The reuse distance information can assist in making replacement decisions as shown later.

The reuse distance can be assigned to load/store instructions, data block addresses, or data ranges. I measure the set reuse distance profile for the cache blocks. The reuse distance is measured as a replacement to miss distance for each cache block. The distribution of reuse distances upto a reuse distance $d_{max}$ is measured for the cache blocks. It gives the information about the percentage of blocks falling a given reuse distance category. The cumulative distribution of the reuse distances gives the information about the percentage of blocks with the reuse distance below a given distance.

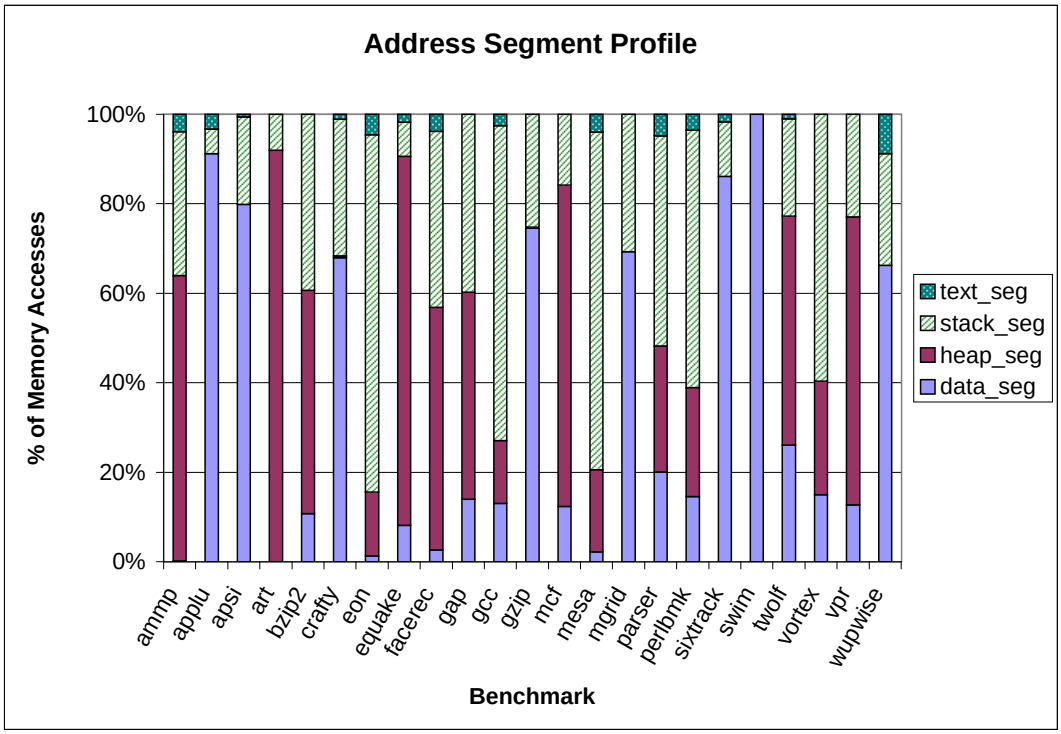


Figure 3-3: Memory Access Address Segment Profile of Spec2000 Benchmarks

### 3.3.3 Active Load/Store Instructions

The number of load/store instructions that miss in the cache is small compared to the total number of load/store instructions. This allows the association of with the load/store instructions for annotations. Some examples of the type of information that can be maintained with the load/store instructions are: stride, reuse distance, number of hits before replacement, etc.

>>> **block accesses Miss-hit-hit..Miss-hit...Miss-hit-hit.....Miss description/diagram**

### 3.3.4 Distinct Load/Store PCs

## 3.4 Load/Store Classification

### 3.4.1 Strided Accesses

A stride $s$ is the difference $Addr_x - Addr_y$ between two consecutive addresses $Addr_x$ and $Addr_y$ accessed by a load/store instruction. Stride $s = s_{i+1}$ is registered, if it is encountered twice in a row, i.e., $s_{i+1} == s_i$, where $s_{i+1} = Addr_{i+1} - Addr_i$ and $s_i = Addr_i - Addr_{i-1}$.

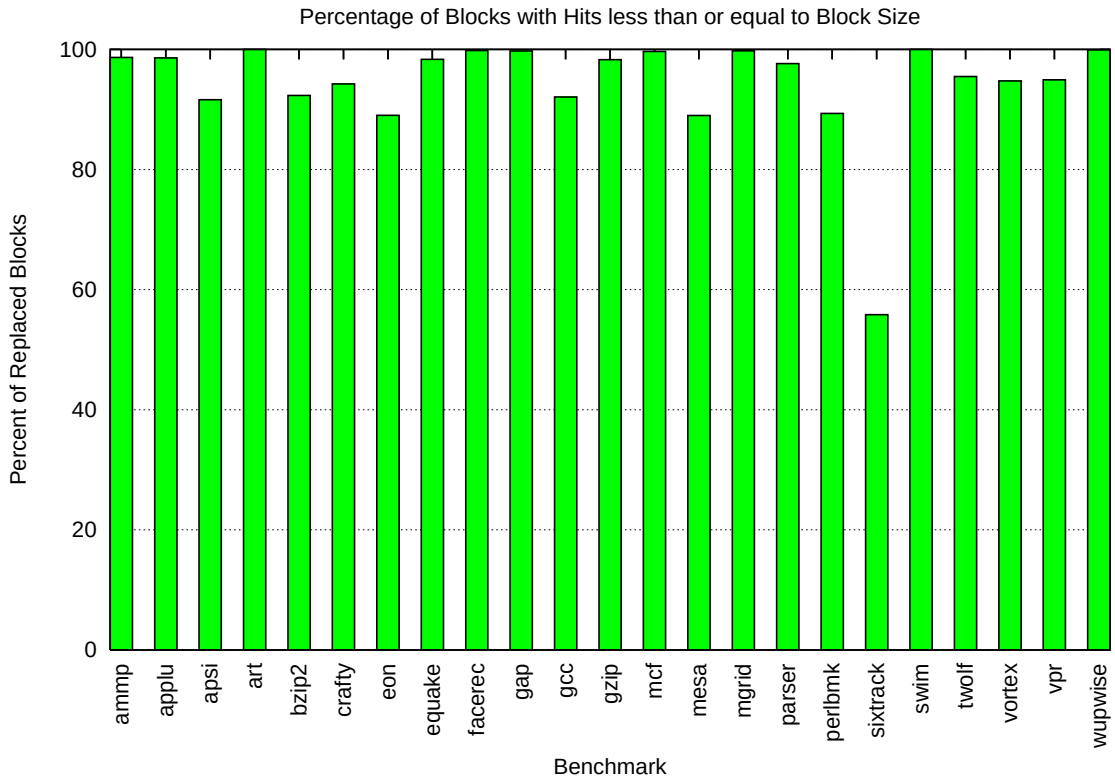


Figure 3-4: Percentage of blocks with number of hits less than or equal to block size before replacement of Spec2000 Benchmarks

### 3.4.2 Pointer Accesses

### 3.4.3 Static Accesses

## 3.5 Profile-based Algorithm

The goal of the annotation selection algorithm is to use the profile-based information to select the annotations such that they result in low static and dynamic overhead. The instructions associated with the selected annotations are assigned at the annoation points such as an instruction address or an loop boundary.

The algorithm uses the information about the disjoint ranges (load/store based or address range based) and it decides when to use a range-based instruction. One way to choose range-based instruction is when load/store-based regular expressions cannot be reduced to single regular expression then the algorithm looks for a range-based single regular expression.

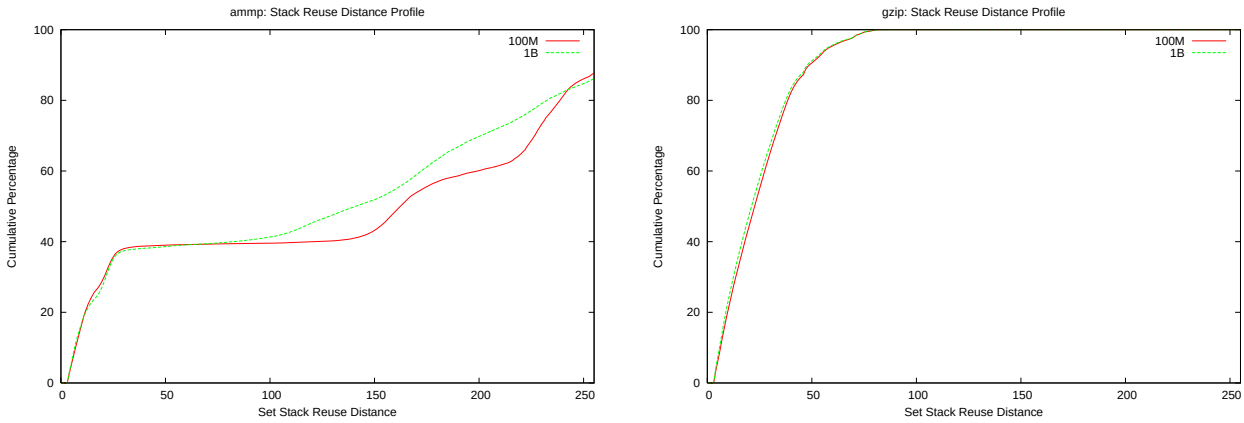


Figure 3-5: ☐ Reuse Distance Profile

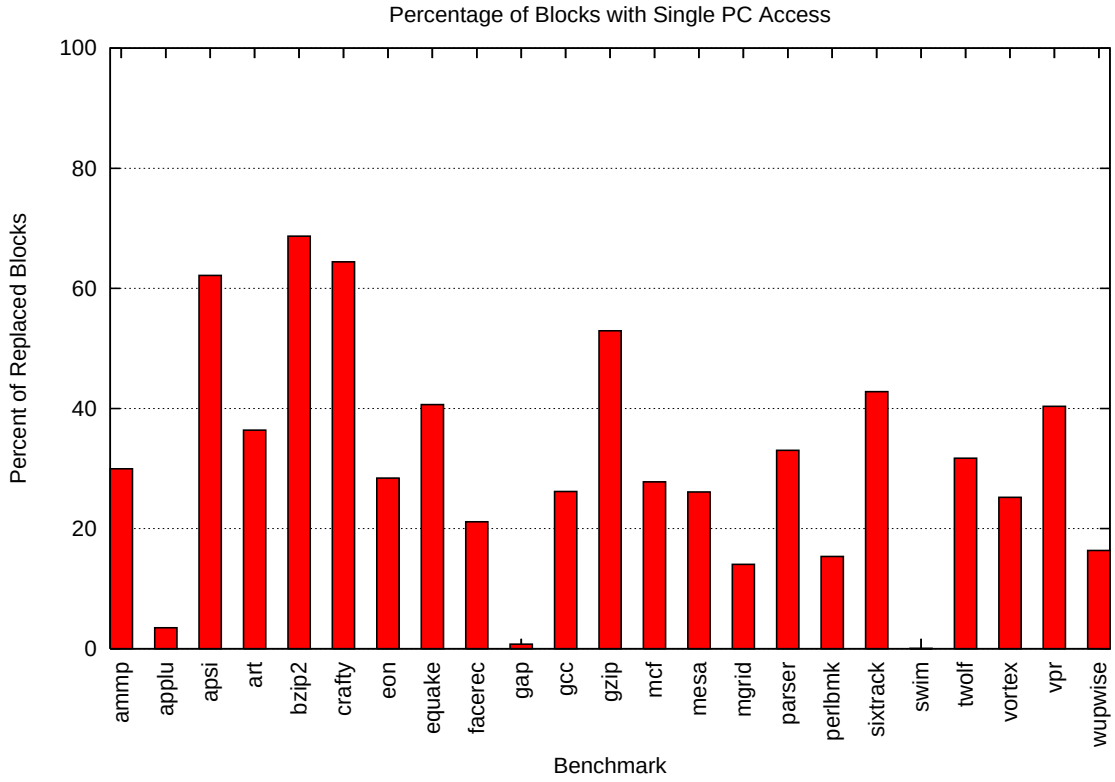


Figure 3-6: Percentag of replaced blocks with accesses only by a sinlge PC

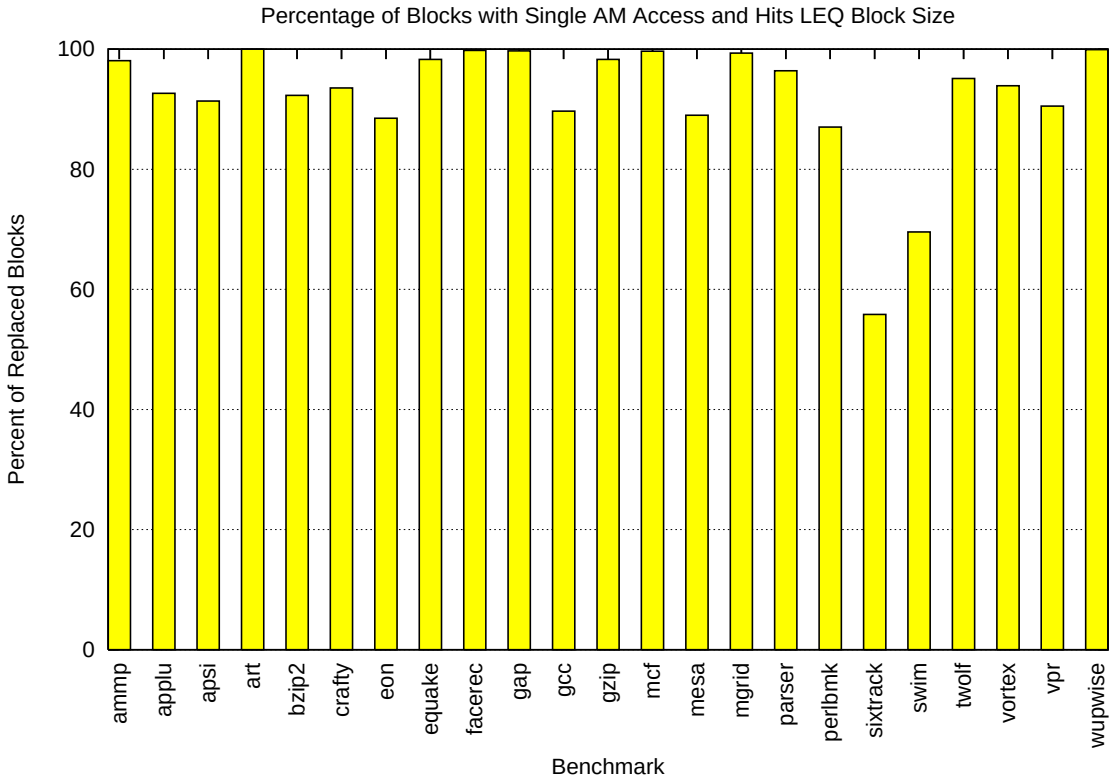


Figure 3-7: Percentag of replaced blocks with Single AM accesses and Hits less than or equal to Block Size

---

1: Determine Load/store information
2: Choose $Maximal\_subset$
3: **if** One Reg Expr **then**
4: Assign Replacement Condition
5: **end if**

---

Figure 3-8: □ Profile-based KILL Algorithm

## 3.6 Partial Regular Expressions

### 3.6.1 Distinct Load/Store PCs

### 3.6.2 Control Point Counts

The control point counts for each distinct load/store PC accessing a block are measured, if a control point exists for the load/store PC accessing the block.

On a miss, for the block being replaced, the control point count is copied along with the regular expression. For the incoming block, the start value of the number of times control point is executed initialized as the current value of the number of times control point has been executed so far. The control point count is initialized to 1 to be updated later on subsequent possible hit(s).

On a hit, if the load/store PC is already part of the list of distinct PCs, then the control point count is computed using the start value and the current value of the number of times control point has been executed so far. If the load/store PC is a new distinct PC accessing the block, the same action is taken as for the miss above for the incoming block.

### 3.6.3 Closest Loop Counts

The closest loop counts are measured in the same way as the control point counts are measured as described above.

## 3.7 Replacement Condition Filtering

### 3.7.1 Set-associative Distance

`mgrid` benefits from filtering the regular expressions.

### 3.7.2 Fully-associative Distance

### 3.7.3 VC-based Distance

## 3.8 Replacement Condition Selection

### 3.8.1 Subset Removal

### 3.8.2 Max Times

### 3.8.3 One Regular Expression

## 3.9 Annotation Generation

In the annotation approach, first a set of cache instructions are chosen and either a static or profile based program analysis is used to identify the annotation points and then appropriate cache instructions are introduced at these annotation points in the program.

### 3.9.1 Instruction Choices

### 3.9.2 Instruction Assignment

### 3.9.3 Annotation Points

# Chapter 4

# MI-LRU Replacement Mechanism

## 4.1 Introduction

The LRU replacement policy is the most common replacement policy and it is the basis of the intelligent cache replacement mechanism which offers miss rate improvement over the LRU replacement 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) [10] 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 4-1 shows the miss rates for the Spec2000FP and Spec2000INT benchmarks respectively using the LRU and OPT replacement policies. The miss rates were obtained

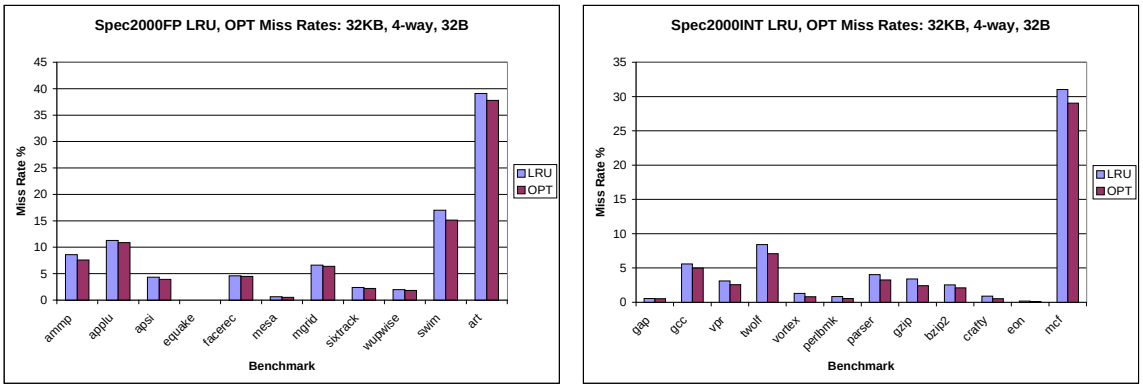


Figure 4-1: $\surd$ LRU and OPT replacement misses for Spec2000FP and Spec2000INT benchmarks using sim-cheetah

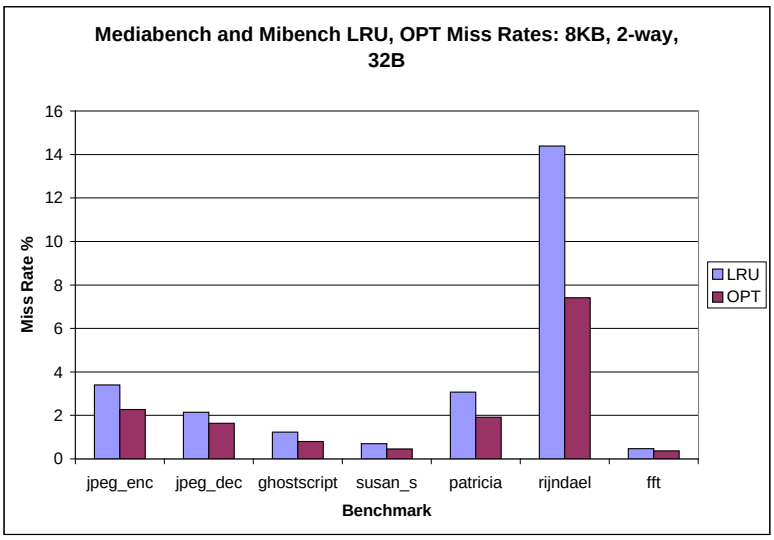


Figure 4-2: LRU and OPT Miss Rates for Mediabench and Mibench

using the Sim-cheetah simulator [168, 18]. For simulation one billion instructions were simulated after skipping the first two billion instructions. The results show that for some benchmarks there is 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 published in [66]. First, the basic concepts of dead block, killing a block, and keeping a block are presented in Section 4.2. Then, the intelligent replacement mechanisms based on these concepts are presented in Section 4.3. The theoretical results for the intelligent replacement mechanism are presented in Section 4.4. The implementation of the intelligent replacement mechanism using hardware and software-assisted approaches is presented in Section 5.1. The experimental results based on the application address traces is presented in Section **??** and ongoing work on experiments based on the mechanisms are listed in Section **??**.

## 4.2 Basic Concepts

### 4.2.1 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.

### 4.2.2 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.

### 4.2.3 Keep a Block

In some cases it may be desirable have 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 $b$.

## 4.3 Mechanisms

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

### 4.3.1 ♣ 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

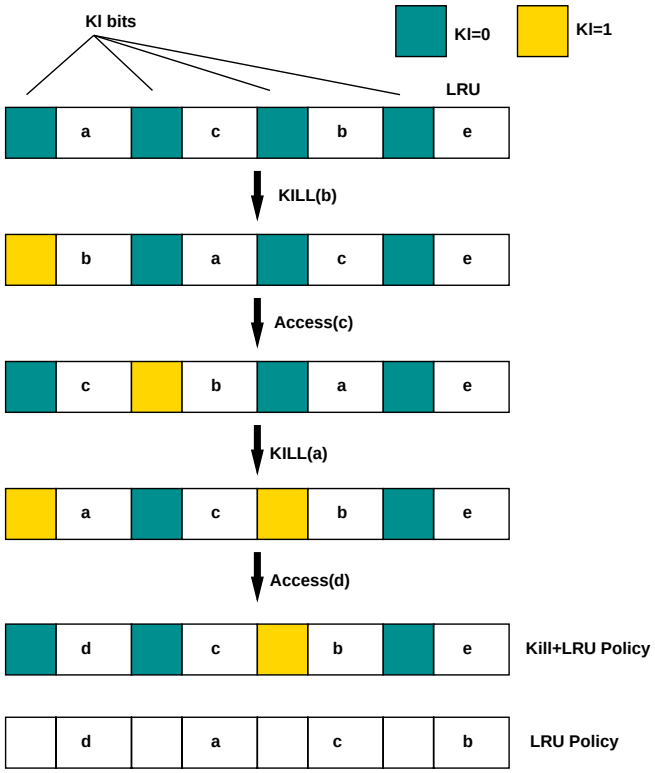


Figure 4-3: $\surd$ Kill+LRU Replacement Policy

are considered: (1) the least recent element that has its $K_l$ bit set is chosen to be replaced; (2) the most recent element that has its $K_l$ bit set is chosen to be replaced. The $K_l$ bit is reset when there is a hit on an element with its $K_l$ 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 4-3 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 element $a$ is chosen for replacement and the new cache state is $\{d, c, b, e\}$. If LRU replacement policy were used, the LRU element

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

### Kill with LRU and OPT Policy

**Theorem 1** *If the reuse distance among the killed blocks is the same, then the MRK_LRU (Most Recently Killed) variation (2) of the Kill+LRU replacement policy is equivalent to the OPT replacement.*

### MRK-LRU based OPT Miss Rate Estimation

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 coressponding 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 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 in Figure 4-4. 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

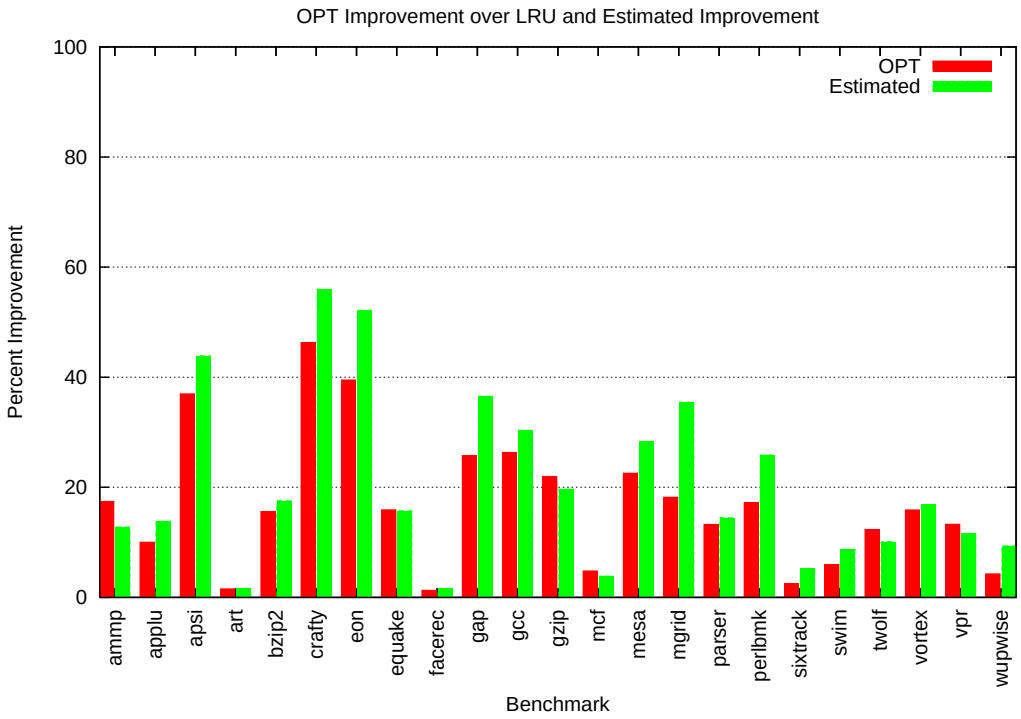


Figure 4-4: OPT Improvement and Estimated Improvement Based on Profile data and Uniform Distance Model

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 replacements choices. The miss-rate estimate using another formula is shown in the Appendix Figure **??**. 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.

### 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 another 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.

### 4.3.2 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 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.

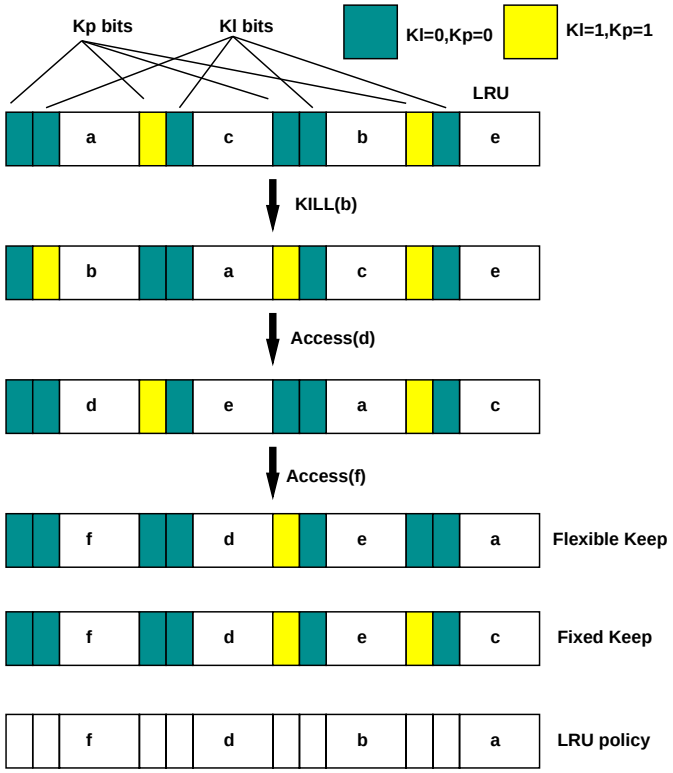


Figure 4-5: $\surd$ Kill+Keep+LRU Replacement Policy

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 4-5 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. Some of the ways to identify the keep data are described in Section 6.1.

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 accomodate different data that needs to be kept in the SRAM at the same time.

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 superset 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 4.4.2 has the weakest condition necessary for the Fixed Keep variation of Kill+Keep+LRU to have better hit rate than the LRU replacement.

### 4.3.3 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. There are four different scenarios shown in Figure 4-6.

**(a)**: Figure 4-6(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, $\phi$), 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 $\phi$ indicates that there is no prefetched data.

**(b)**: Figure 4-6(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 4-6(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 4.3.1. This method is denoted as (Kill + LRU, Kill + LRU).

**(d)**: Figure 4-6(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 kill bit set. Further, when the prefetched data replaces dead data, the LRU order of

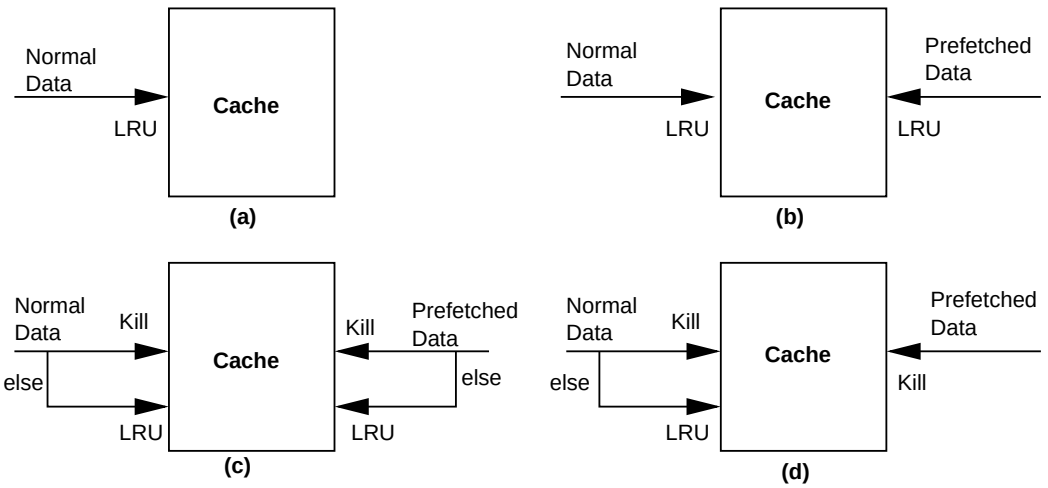


Figure 4-6: $\surd$ Integrating Prefetching with Modified LRU Replacement

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.

## 4.4 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 better than the LRU policy. Also, the theorems for prefetching with LRU and Kill+LRU are presented here.

### 4.4.1 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 2** *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 $C(LRU, t+1) \preceq_0 C(KIL, t+1)$. ■

### 4.4.2 Kill+Keep+LRU Theorem

**Theorem 3** *(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 3(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 3(b). ■

**Set-Associative Caches**

Theorem 2 and Theorem 3 can be generalized to set-associative caches.

**Theorem 4** *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 2, 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 4. ■

**Theorem 5** *(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 4. ■

### 4.4.3 Kill+LRU with Prefetching Theorems

The combination of prefetching with LRU and Kill+LRU replacement as described in Section 4.3.3 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 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 better than (LRU, $\phi$) for any prefetching strategy. Note that (Kill + LRU, Kill + LRU) could actually perform worse than (LRU, $\phi$) (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 6** *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 4 directly. ■

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

**Theorem 7** *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, $\phi$).*

*Proof:* (Sketch) The prefetch strategy (Kill + LRU, Kill) results in two access streams, the normal accesses and the prefetched blocks. The no-prefetch (LRU, $\phi$) strategy has only the normal access stream. Invoking Theorem 4, 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 5

# MI-LRU Implementation

## 5.1 Implementation

As described earlier, 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.

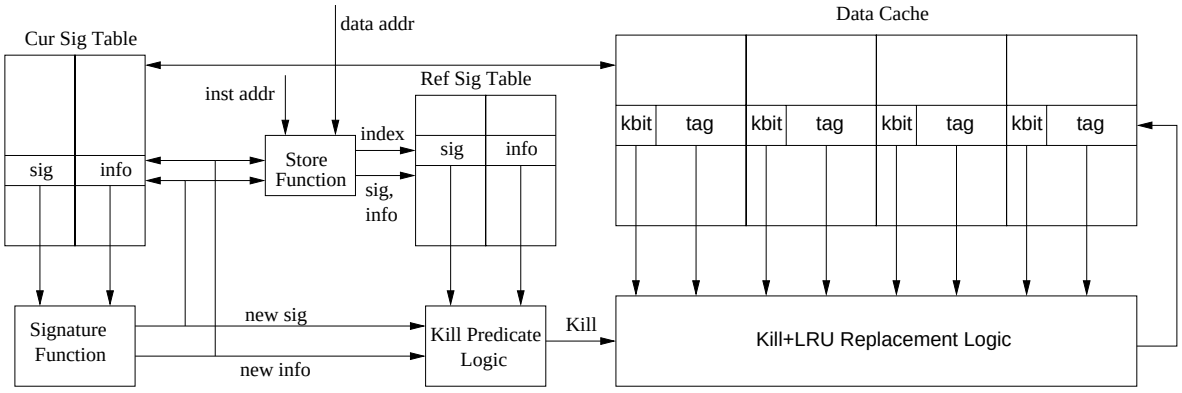


Figure 5-1: $\surd$ Hardware-based Kill+LRU Replacement

### 5.1.1 Intuition behind Signatures

The hardware-based approach needs to build the kill predicates using some hardware support and use them 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 signature is used to predict 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: $\{PC_1, PC_2, ..., PC_n\}$ 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 $Ty$ is compared to $S_x$ and if there is a match, then $b$ is marked as 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 $b_x$ can be used to predict the dead block for block $b_y$. This relies on the same history information based signatures for multiple blocks. So, even if the access sequence for a block does not repeat the history information from one block can be applied to other blocks.

### 5.1.2 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.

**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 5-2: $\surd$ Signature-based Hardware Kill Predicates

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 5-1 and the pseudo-code for the hardware kill predicates algorithm based on signatures is shown in Figure 5-2. 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 $T$.

#### 5.1.2.1 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 XORed 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 XORed 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.
- **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.

#### 5.1.2.2 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 number of load/stores that miss in the data cache bring in a new data cache block into the data cache and 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 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 5-3 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

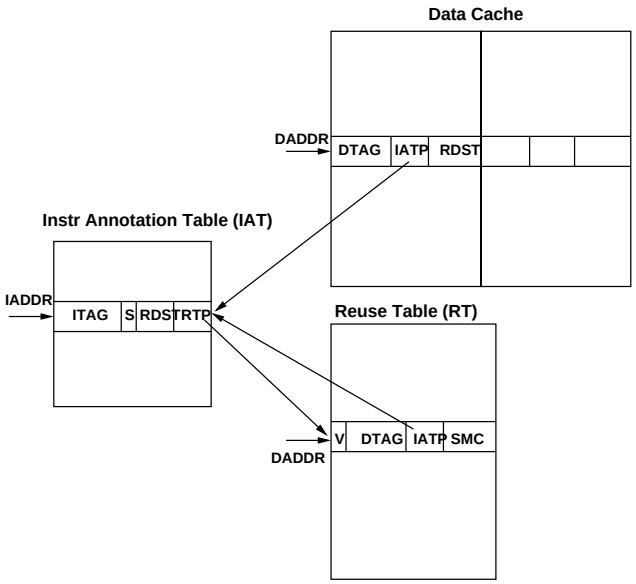


Figure 5-3: $\surd$ Hardware for Reuse Distance Measurement

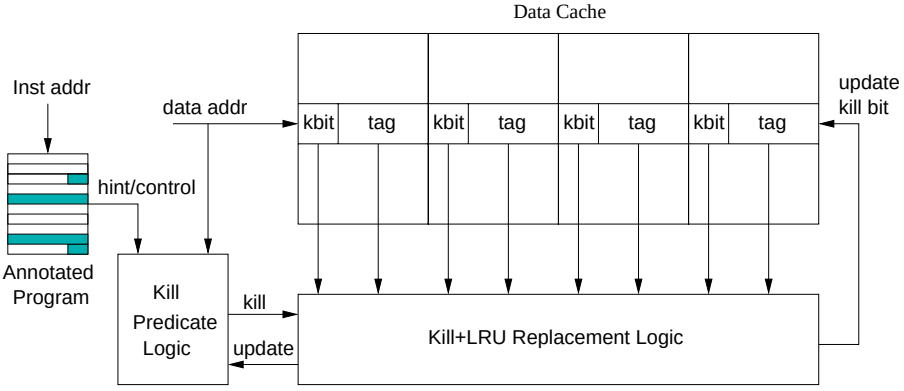


Figure 5-4: $\surd$ Software-assisted Kill+LRU Replacement Hardware

`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.

### 5.1.3 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 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 5-4.

#### 5.1.3.1 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 don't 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 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, set the kill bit of the accessed cache block.

- 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.

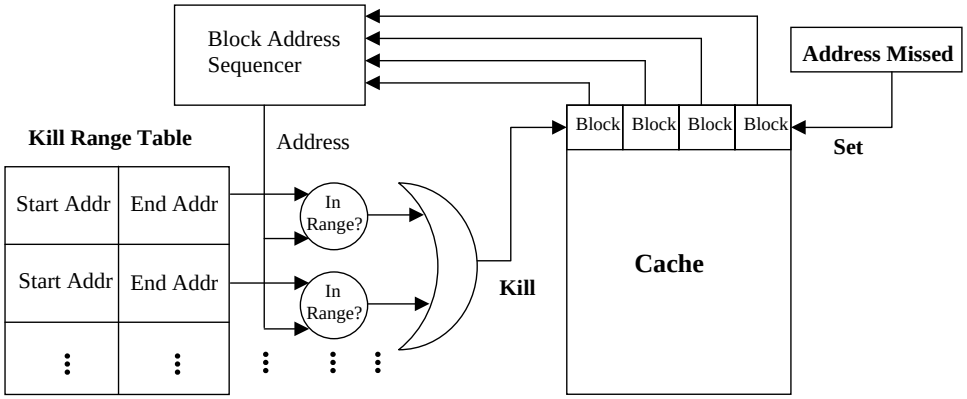


Figure 5-5: $\surd$ Kill Range Implementation Hardware

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 similar to the above instruction.

### Keep Range and Cache Line Locking

The use of 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 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.

#### Kill Range Implementation

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

### 5.1.3.2 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 compile 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. But, the compiler does not have the information about the data layout. 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 the reuse distance $d$ 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 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.

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 can approximate some aspects of the compiler algorithm using some hardware structures.

Since the compiler algorithm is not implemented, a profile-based algorithm is used to implement the software-assisted Kill+LRU replacement. The profile-based algorithm can

collect input data independent information (e.g., type of load/store instruction, loop begin and end markers) or collect data dependent information (e.g., number of blocks between two accesses). In the following profile-based algorithm, the static information about the program is derived using a profile run and used as 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 update the annotation table dynamically.

# Chapter 6

# ♣ Combining MI-LRU with Keep and Prefetching

## 6.1 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 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.

1: Determine the KEEP candidate variables
2: Choose the KEEP length of variables
3: Assign KEEP hints or KEEP instructions

Figure 6-1: □ Profile-based KEEP Algorithm

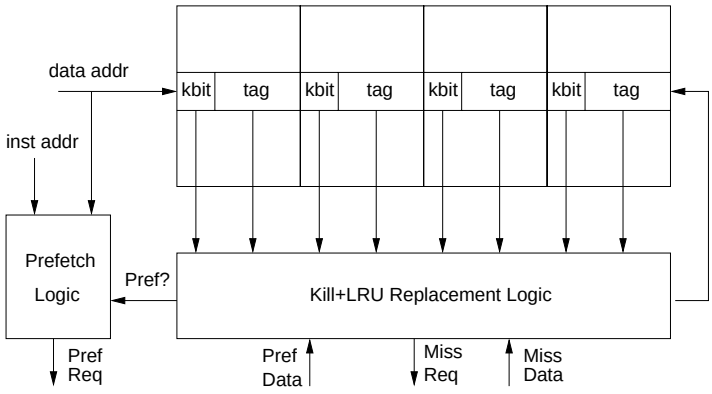


Figure 6-2: √ Prefetching with Kill+LRU Replacement

### 6.1.1 Profile-based Keep Algorithm

### 6.1.2 Keep Candidates Selection

### 6.1.3 Keep Decay Options

### 6.1.4 Keep Instructions

## 6.2 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 6-2. Two prefetching approaches are considered in combination with the Kill+LRU replacement policy as described in the following sections.

### 6.2.1 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.

I 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.

### 6.2.2 Predictor-based Prefetching

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.

### 6.2.3 Prefetch Correlation with Kill+LRU Replacement

The use of hardware-based prefetching or predictor-based prefetching with Kill+LRU replacement decouples the prefetching approach with 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 prfetch 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 7

# ♣ New Instructions

## 7.1 Type of Cache Control Commands

There can be several types of cache control commands with some associated bits in the cache and may be some hardware to implement these commands. The cache control commands are described below.

- Kill commands: these commands allow the arrays or data structures to be marked for replacement. These commands allow the augmentation of the LRU state to kill the data that is not going to be used for some time and the cache can be updated quickly. It is like modifying the LRU state of the cache and accelerating the adaptation of the LRU for the cache.
- Partial kill commands: these commands allow parts of the arrays or variables to be marked for replacement
- Keep command: this command allows a certain variable to be kept in the cache unless released.
- Keep for limited time: this command allows a certain variable to be kept for a limited time. A counter in the set can be decremented to account for the accesses in the set.
- Prefetch range commands: prefetch a range of addresses

The summary of new instructions is given in Table 7.1.

| Instruction | Description |
|---|---|
| `ldkl` | Kill on every load |
| `stkl` | Kill on every store |

Table 7.1: New Instructions

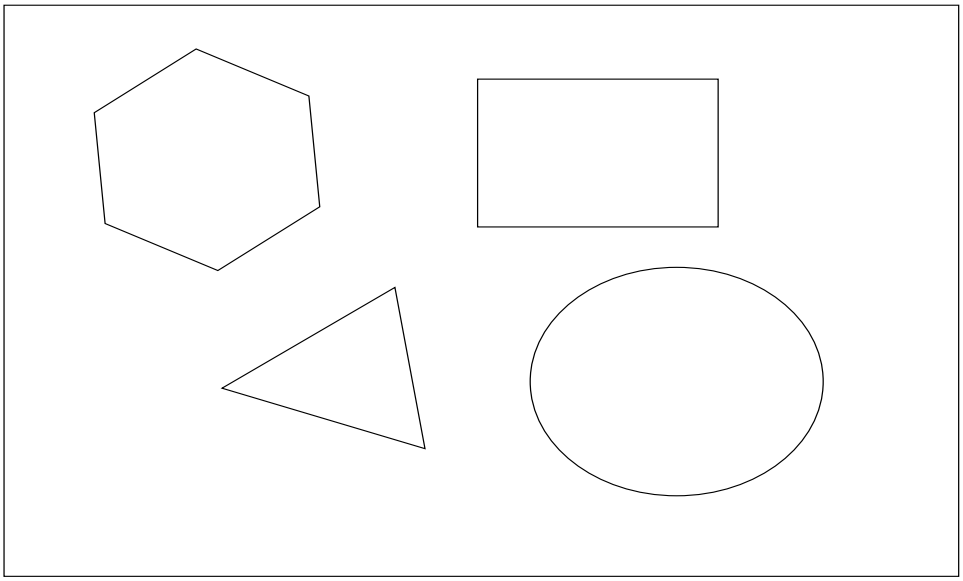

Figure 7-1: □ Hardware Condition-based Trigger Implementation

## 7.2 Instruction Descriptions

## 7.3 Hint-based Instructions

## 7.4 Condition-based Instructions

## 7.5 Range-based Instructions

The trigger PC address can be specified as a pc-relative offset from the setup instruction PC.

The advantage of specifying ranges using explicit address ranges is that the load/store instructions accessing the ranges don't need to contain the range related information. The data symbol table would be useful in determining the static data ranges, but this information is not available in the compiled benchmarks. So, the static address range information can be derived using the profile-based runs of the benchmarks. The address ranges can be derived from the page table scan of the accessed pages and collecting the minimum page number

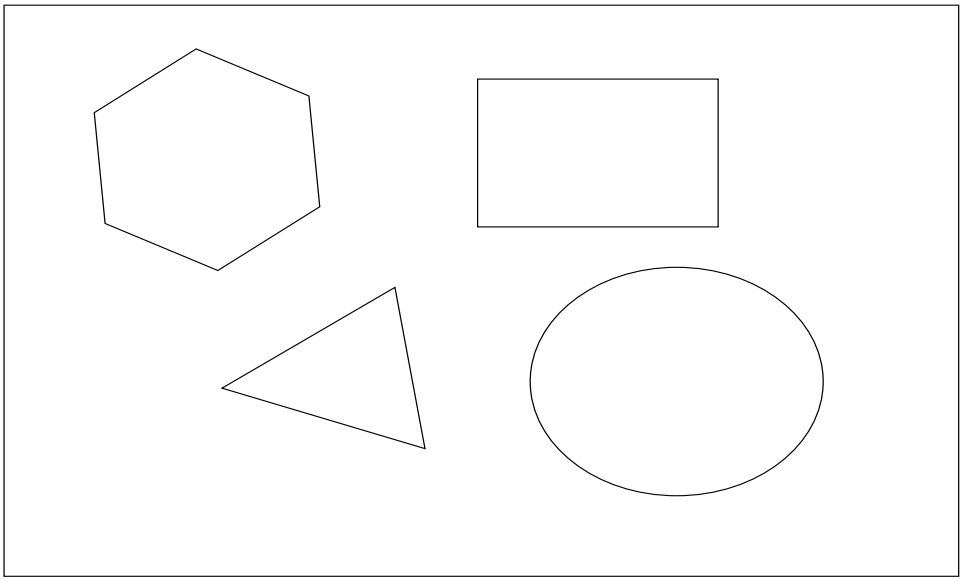

Figure 7-2: □ Range Block Count-based Implementation

and maximum page number for the range list.

The arrays or variables that need to be marked for replacement can be indicated using a cache control instruction that gives the address range that needs to be marked for replacement. This information can be kept in a hardware associative table and whenever the cache is accessed the address is compared with the range of addresses currently under KILL state. If a matching range is found, then a bit is set in the cache line associated with the address indicating that the cache line is marked for replacement. This is like a lazy kill operation because the kill bit of the cache lines for a given address range are updated as the cache lines are accessed.

## 7.6 Run-time Support

### 7.6.1 Modified Code

### 7.6.2 Annotation Cache

In the software-assisted mechanisms, some information is communicated from the software through annotations of the existing instructions or by additional instructions. These additional communication mechanisms may cause execution time overhead. An annotation cache can load the annotations corresponding to the instructions separately and reduce the execution time overhead because these annotations are separated from the instruction

stream.

I will look into the use of annotation cache for software-assisted mechanisms.

## 7.7 ALPHA Implementation Options

### 7.7.1 Unused Fields

### 7.7.2 Unused Opcodes

I experimented with introducing unused opcodes to indicate additional instructions.

### 7.7.3 PAL codes

# Chapter 8

# Disjoint Sequences and Cache Partitioning

## 8.1 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.

## 8.2 Theoretical Results

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

### 8.2.1 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 $\phi$ 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 $\phi$ 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.

### 8.2.2 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 $0 \leq i \leq p-1$.

### 8.2.3 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 $S$. For a direct-mapped cache I showed that for any time $t$, $0 \leq t \leq N$, $C(S,t) \cap R_1 \subseteq C(R_1^{'},t)$ and for a fully-associative cache I showed 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 showed 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 $R_1@R_2$.

### 8.2.4 Effect of Memory Line Size > One 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.

If the elements in sequences are aligned for a cache line size $L$, then the theorem holds for any line size $\leq L$.

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 $l' > l$.

## 8.3 Partitioning with Disjoint Sequences

The notion of disjoint sequences can is 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 the 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 ertain conditions as described in Section 8.3.2.

### 8.3.1 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 8.1. If the

| 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 8.1: $\surd$ Non-disjoint Sequences Example

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.

### 8.3.2 Two Sequence Partitioning Example

Given two sequences, s1 and s2, with the number of accesses n1 and n2 if the number of misses of these sequences is m together for cache size C. If used individually, the number of misses are m1 and m2. I have m1 + m2 $\leq$ m. Now if the 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

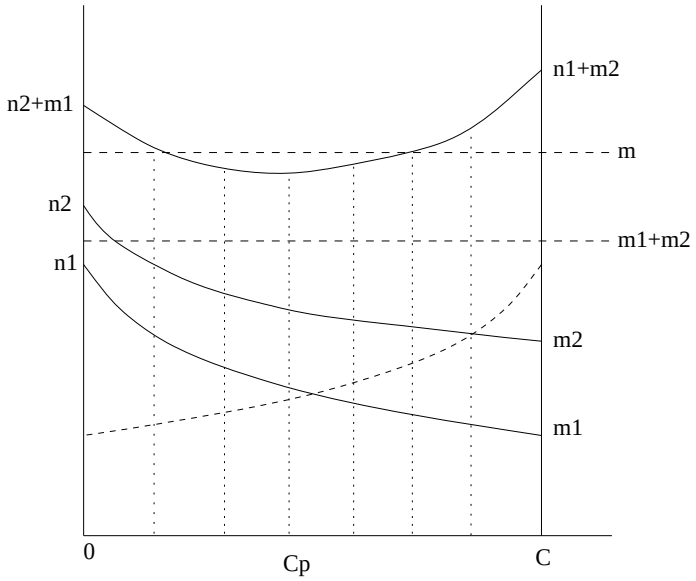


Figure 8-1: $\surd$ Two Sequence Partitioning

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 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.

### 8.3.3 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 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.

## 8.4 Information for Partitioning

The information about the sequences can be the footprint for a sequence or the cache usage efficiency for the sequence.

### 8.4.1 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.

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

### 8.4.2 Sequence Footprints

Profile information can be used to estimate the relative footprint sizes of the sequences.

### 8.4.3 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
- 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.

## 8.5 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 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 compared to the case where the group was sharing the whole cache space with the other sequences.

### 8.5.1 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

```
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 8-2: $\sqrt{}$ Clustering Algorithm

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.

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 8-2.

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),
```

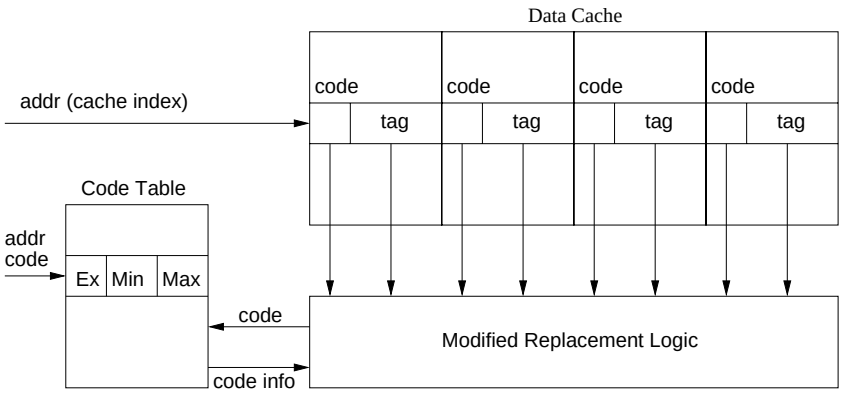


Figure 8-3: $\surd$ Hardware Support for Flexible Partitioning

```
   {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:

```
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)
```

### 8.5.2 Hardware Support

#### Modified LRU Replacement

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

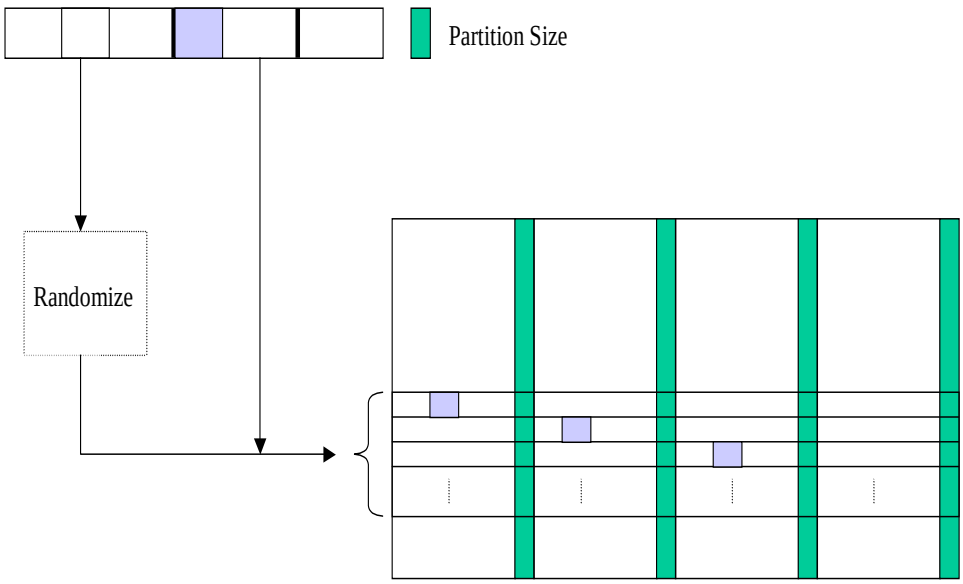


Figure 8-4: Cache Tiles for Cache Partitioning

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: using data address ranges or load/store annotations.

### Cache Tiles

The above approach for partitioning is based on 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 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 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 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. There are several issues associated with this approach that need be dealt with to use this approach with disjoint sequences.

#### 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 addresses. 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.

### 8.5.3 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, the sequences need to be defined that would represent different groups. This can be done using the compiler analysis that would identify 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 which would be 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

1: Determine Disjoint IDs
2: Determine Partition Sizes
3: Assign Partition Sizes

Figure 8-5: □ Profile-based Partitioning Algorithm

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.

## 8.6 Partitioning and Prefetching

The partitioning approaches are combined with hardware-based sequential or stride prefetching prefetching to evaluate the impact of partitioning on reducing cache pollution due to prefetching.

## 8.7 Experiments

## 8.8 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.

# Chapter 9

# ♣ Evaluation

The hardware-based and profiled-based Kill+LRU without prefetching and with prefetching were evaluated using a selection from the Spec2000, Mediabench, and Mibench benchmarks. First, the compiled benchmarks from the three benchmarks suites were simulated using `sim-cheetah` to get LRU miss rate and OPT miss rate numbers. The LRU and OPT miss rate numbers gave an upper bound of the potential miss rate improvement possible using Kill+LRU replacement over LRU. Tweleve benchmarks from the Spec2000 suite were selected for our experiments. Some of the benchmarks from the Mediabench and Mibench suites could not be compiled or run successfully in the Simplescalar simulators and therefore could not be used for the experiments. In addition, some of the Mediabench and Mibench benchmarks had very low miss rates and they were not chosen for the experiments. The Spec2000 benchmarks are shown in Table 9.2.1 and Mediabench and Mibench benchmarks are shown in Table 9.2.2 and the chosen benchmarks are marked with an '*'.

• Describe 2b1b. and why Simpoint-based fastforward and number of instructions. Standard Multiple 100M

## 9.1 Simulation Framework

### 9.1.1 Profile-based Analysis

### 9.1.2 Annotation Support

* GENERATING and Incorporating profile-based information in simulation - collect info from profile run, read into tables and apply during another run. - look at pcstat in sim-

| SpecINT 2000 | |
|---|---|
| gzip | Compression |
| vpr | Placement and routing |
| gcc | C compiler |
| mcf | Combinatorial optimization |
| crafty | Chess |
| parser | Word processing |
| eon | Computer visualization |
| perlbmk | PERL programming language |
| gap | Group theory interpreter |
| vortex | Object-oriented database |
| bzip2 | Compression |
| twolf | place and route simulator |
| **SpecFP 2000** | |
| wupwise | Physics |
| swim | Shallow water modeling |
| mgrid | Multi-grid solver |
| applu | Partial differential equations |
| mesa | 3-D graphics library |
| art | Image recognition |
| equake | Seismic simulation |
| facerec | Image processing |
| ammp | Computational chemistry |
| sixtrack | High energy nuclear physics |
| apsi | Pollutant distribution |

Table 9.1: $\surd$ Benchmarks and Descriptions

outorder

## 9.2 Benchmarks

BENCHMARKS: SPECFP not important to embedded world SpecFP benchmarks for embedded caches aren't too plausible benchmarks matching embedded system focus larger set of benchmarks: mediabench and then pick from them. Mediabench problem is that everything fits in the cache. Maybe run expts on smaller caches.

| Mediabench Benchmarks | |
|---|---|
| adpcm | ADPCM audio coding |
| g721 | CCITT voice compression |
| jpeg | Lossy compression for still images |
| ghostscript | Postscript interpreter |
| **Mibench Benchmarks** | |
| patricia | A Patricia trie data structure |
| fft | Fast Fourier Transform |
| susan | Image recognition package |

Table 9.2: $\surd$ Mediabench and Mibench Benchmarks and Descriptions

### 9.2.1 Spec2000

### 9.2.2 Multimedia and Mibench

Some of the Mediabench and Mibench benchmarks could not be compiled successfully for simulation and some of the benchmarks compiled successfully but could not be simulated due to problems with system calls, libraries, and/or tables. The complete list of the Mediabench and Mibench benchmarks with the issues is shown in Table **??**.

## 9.3 Inputs

The Spec2000 benchmarks have `test`, `train`, and `ref` inputs available for simulation. I generated the `eio` traces for the Spec2000 benchmarks. For `perlbmk` benchmark `eio` traces could not be generated for the `test` and `train` inputs.

The `eio` traces were generated for the full length of the `test` and `train` inputs. Since the test and train inputs are used to derive the profile-based information, the start pc of the eio traces of the SP1 and SP2 was identified and used to fast-forward until it was encountered the first time.

| Benchmark-ref input | SP2: Simpoint | Weight | SP1: Simpoint | Weight |
|---|---|---|---|---|
| ammp-ref | 372 | 0.122334 | 776 | 0.461188 |
| applu-ref | 830 | 0.101392 | 1873 | 0.208515 |
| apsi-ref | 478 | 0.198607 | 2792 | 0.488039 |
| art-110 | 56 | 0.157914 | 140 | 0.272760 |
| bzip2-source | 359 | 0.213083 | 679 | 0.281967 |
| crafty-ref | 818 | 0.182404 | 1773 | 0.250154 |
| eon-cook | 638 | 0.182350 | 78 | 0.208574 |
| equake-ref | 1124 | 0.364969 | 1011 | 0.487386 |
| facerec-ref | 11 | 0.366304 | 1467 | 0.492008 |
| gap-ref | 83 | 0.114282 | 1837 | 0.561764 |
| gcc-200 | 723 | 0.120550 | 101 | 0.352161 |
| gzip-graphic | 260 | 0.221781 | 470 | 0.353885 |
| mcf-ref | 495 | 0.302261 | 219 | 0.316809 |
| mesa-ref | 491 | 0.215482 | 1827 | 0.285771 |
| mgrid-ref | 646 | 0.171907 | 1352 | 0.213524 |
| parser-ref | 630 | 0.190925 | 1468 | 0.241640 |
| perlbmk-splitmail | 251 | 0.216495 | 939 | 0.664820 |
| sixtrack-ref | 168 | 0.168808 | 2991 | 0.221467 |
| swim-ref | 57 | 0.100518 | 202 | 0.158969 |
| twolf-ref | 122 | 0.143948 | 1123 | 0.166146 |
| vortex-one | 360 | 0.178187 | 846 | 0.395038 |
| vpr-route | 265 | 0.211733 | 351 | 0.240282 |
| wupwise-ref | 2195 | 0.410728 | 2721 | 0.431894 |

Figure 9-1: SPEC2000 Multiple Standard 100M Simpoints: SP1 and SP2

### 9.3.1 Train

### 9.3.2 Test

## 9.4 Application Phases

The points in the benchmarks for simulation are chosen based on the simulation points information available for Spec2000 benchmarks using the Simpoint 3.0 algorithm [160].

The Spec2000 benchmarks were simulated for 100 million instructions after fast-forwarding the (early simpoint - 1)*100 million instructions. The weights associated with the simpoints for the benchmarks were used to determine the early simpoint to choose for simulation. The highest weight simpoint was chosen for each benchmark.

### 9.4.1 Simpoint Phase1

### 9.4.2 Simpoint Phase2

## 9.5 Simulation Length

### 9.5.1 100 Million

### 9.5.2 1000 Million

## 9.6 Processor Types

### 9.6.1 Superscalar Inorder

### 9.6.2 Superscalar Out-of-order

## 9.7 Cache Configurations

### 9.7.1 Cache Size

### 9.7.2 Associativity

### 9.7.3 Block Size

- what does it mean when hit rate changes with block size one way or the other and how do the schemes perform for it

## 9.8 MI-LRU Evaluation

### 9.8.1 Hardware-based Kill+LRU Experimental Results

The selected benchmarks were simulated using a simple pipelined processor configuration with no speculation and in-order execution and IL1 and DL1 caches. This configuration gives the baseline miss rate and IPC numbers for the hardware-based Kill+LRU mechanisms. The DL1 miss rate for the selected benchmarks using different hardware-based Kill+LRU approaches are shown Table 9.8. It turns out that the hardware-based Kill+LRU does not lead to any miss rate improvement for the benchmarks. The Table 9.8 shows the Kill prediction accuracy and coverage for the benchmarks.

| Processor | |
|---|---|
| Clock rate | 2GHz |
| Reorder Buffer | 128 entries |
| Load/Store Queue | 128 entries |
| Issue width | 8 instr per cycle |
| **Cache Hierarchy** | |
| L1 Data Cache | Size: 32KB<br>Associativity: 4-way<br>Block Size: 32 bytes<br>Hit latency: 1 cycle<br>MSHRs: Unlimited |
| L1 Instr Cache | Size: 32KB<br>Associativity: 4-way<br>Block Size: 32 bytes<br>Replacement: LRU<br>Hit latency: 1 cycle |
| L2 Data Cache | Size: 1MB<br>Associativity: 4-way<br>Block Size: 64 bytes<br>Replacement: LRU<br>Hit latency: 12 cycles<br>MSHRs: Unlimited |
| L2 Instr Cache | Size: 1MB<br>Associativity: 4-way<br>Block Size: 64 bytes<br>Replacement: LRU<br>Hit latency: 12 cycles |
| Memory | Latency: 70 cycles |

Table 9.3: $\surd$ System Configuration for Experiments

| Benchmark | LRU | ISF2 | ISF3 | DBSF2 | DBSF3 | KP2 | KP3 | KPB2 | KPB3 |
|---|---|---|---|---|---|---|---|---|---|

Table 9.4: $\surd$ Hardward-based Kill+LRU Miss rates for Spec2000, Mediabench, and Mibench Benchmarks. Spec2000 results based on 16KB, 4-way, 32-byte blocks DL1 and Mibench and Mediabench results based on 8KB, 4-way, 32-byte blocks DL1.

| Benchmark | LRU | XROT_D | XROT_I | RDIST |
|---|---|---|---|---|

Table 9.5: $\surd$ Hardward-based Kill+LRU Miss rates for Spec2000, Mediabench, and Mibench Benchmarks. Spec2000 results based on 16KB, 4-way, 32-byte blocks DL1 and Mibench and Mediabench results based on 8KB, 4-way, 32-byte blocks DL1.

| Benchmark | Kpred_Accuracy | Kpred_Coverage |
|---|---|---|

Table 9.6: $\surd$ Hardward-based Kill+LRU Prediction Accuracy and Coverage for Spec2000, Mediabench, and Mibench Benchmarks

### 9.8.2 Analysis

The KILL operation is not very effective on the benchmarks because the number of iterations in a loop is greater than the advantage the KILL operation can give. The KILL operation gives advantage only when a cache line is marked for replacement when it is not the LRU cache line and the LRU data that would otherwise be replaced is used in the near future.

## 9.9 MI-LRU+Keep Evaluation

## 9.10 MI-LRU+Prefetching Evaluation

## 9.11 Static Partitioning Evaluation

## 9.12 Profile-based Instructions Overhead

### 9.12.1 Static Instruction Overhead

### 9.12.2 Dynamic Instruction Overhead

## 9.13 SW/HW Cost *vs.* Performance Trade-off Analysis

COST: cost and what happens to good ones? made worse by new addition or extra hardware could have been used elsewhere to make them work.

### 9.13.1 Hardware Table Sizes

### 9.13.2 Block-associated Sizes

ANALYSIS: analyze results why works and does not work what works and what does not work. document that too.

# Chapter 10

# 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). There has been lot of research effort focussed on improving on-chip memory performance using a variety of architectural techniques. Previous work that is related to the mechanisms presented in this thesis is described in this chapter. In particular, related work for cache performance and predictability and on-chip SRAM usage is described.

Caches are an integral part of modern processors (e.g., Pentium 4 [62], Alpha 21264 [81], MIPS R10000 [201], and IA-64 [50]) 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. Several different approaches to improve cache performance and some approaches to improve cache predictability are summarized below.

## 10.1 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 [131]. 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 [163]. A simulation-based technique for selecting a processor and required instruction and data caches is presented in [90].

I have focussed on independently improving on-chip SRAM performance and cache performance in this thesis.

## 10.2 Cache Architectures

There are several cache parameters that can be optimized to improve cache performance as well [58]. The effectiveness of direct-mapped caches is studied in [60] and the effect of associativity in caches is described in [61]. 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 [16, 17].

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 [83] for wire-delay dominated caches. A cool-cache architecture is presented in [178, 179] and a direct-addressed cache architecture is proposed in [194] to reduce cache power.

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

## 10.3 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 [75]. Some approaches use cache miss analysis to guide the compiler optimizations [54, 26]. An approach that makes use of spatial footprints is presented in [92].

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 [73]. The use of stream buffers to improve cache performance was described in [129]. Dynamic access reordering was proposed in [114] and improving bandwidth for streamed references was presented in [115]. A hardware-based approach that uses hardware miss classification was

proposed in [37] and a spatial pattern prediction approach was presented in [27]. A locality sensitive multi-module cache design is presented in [150]. The run-time detection of spatial locality for optimization is described in [72].

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.

## 10.4 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 [112, 117]. Other processors permit cache line locking within the cache, essentially removing those cache lines as candidates to be replaced [40, 41]. 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 [80] 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 [185]. In [110] 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 [170], 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 [133]. 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 [71]. In [57] a fully associative software managed cache is described. An integrated hardware/software scheme is described in [55]. The adaptive approaches use different control mechanisms to improve cache performance. A method based on adaptive cache line size is described in [184]. A method for adaptive data cache management is described in [175]. Cache management based on reuse information is addressed in [145].

The keep instructions in Chapter **??** bear some similarity to the work on cache line locking [110, 185]. 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.

## 10.5 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 [164]. Belady's optimal replacement algorithm was presented in [10] and its extension using temporal and spatial locality was presented in [171]. The efficient simulation of the optimal replacement algorithm was described in [168].

Compiler analysis based methods to improve replacement decisions in caches and main memory are presented in [191, 14]. A hardware-based approach to detect dead blocks and correlate them blocks to prefetch is presented in [94]. An approach that uses time-keeping in clock cycles for cache access interval or cache dead time is presented in [63]. In [197] 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 [98], policies in the range of Least Recently Used and Least Frequently Used are discussed.

## 10.6 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 [165]. A variation of the OBL approach uses prefetch-on-miss and tagged prefetch. Cache pollution is a problem if more than one block is prefetch for a block access [139]. Another hardware approach for prefetching uses stride detection and prediction to prefetch data [28, 30]. Some approaches combine prefetching with other mechanisms. For example, the dead-block prediction and correlated prefetching is described for direct-mapped caches in [94] and prefetching and replacement decisions have been made in tandem in the context for I/O prefetching for file systems [21, 132].

The work of [93, 94] 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

[94] achieve significant speedups, but at the cost of large hardware tables that require up-to 2MB of storage.

There have been many proposals for software prefetching in the literature. Many techniques are customized toward loops or other iterative constructs (e.g., [91]). These software prefetchers rely on accurate analysis of memory access patterns to detect which memory addresses are going to be subsequently referenced [120] [106] [105] [128].

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.

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 [5] and an irregular data structure prefetching is considered in [77].

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.

## 10.7 Cache Partitioning

Cache partitioning has been proposed to improve performance and predictability by diving 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 [36]. 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 [149, 138]. The data is classified into either a spatial or temporal and stored in the corresponding partition.

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 [11]. There are also approaches for page placement [20, 159], page coloring [15], and data layout [32, 33, 35] to avoid conflict misses. A model for partition-

ing an instruction cache among multiple processes has been presented [101]. McFarling presents techniques of code placement in main memory to maximize instruction cache hit ratio [113, 174].

I have developed a theory for cache memory partitioning based on disjoint access sequences that can provide guarantees of improved behavior when exploited. While techniques that exploit this theory have been developed, they have not been fully implemented and evaluated at the time of writing this thesis.

## New References

comparing phase detection techniques [46] identifying and exploiting spatial regularity [118] load stream behavior [148] computation regrouping [136] dynamic dead-instruction [19] locality through reuse distance analysis [49] using stack distances [22] miss prediction across all program inputs [208]

Spatial characteristics of load instructions [202] Spec CPU2000 performance [59] Characterization of Spec2000 benchmark suite [39] Cache profiling and spec benchmarks [96] Output value locality of Spec and Mediabench [79] Memory behavior of Spec2000 benchmark suite [147] d-tlb behavior of SPEC CPU2000 benchmarks [76]

Mibench benchmark suite [56] MinneSpec benchmarks [103] Mediabench benchmarks [97] A detailed analysis of Mediabench [13]

Off-chip memory traffic measurements [44] The filter cache [88] on-chip stack memory organization [111] Allocation of global data objects [162] Cool-mem [7] minimax cache [180] memory behavior of scalars [177] dynamic allocation [124] lightweight set buffer [199] tag comparison elimination [207] highly configurable cache architecture [203] reducing cache pollution of prefetching [144] managing leakage energy in cache hierarchies [100] static pattern predictor [187] phase-based cache resizing [137] memory design exploration [161] region-based caching [99] profiling tools for hardware/software partitioning [169] reducing high-associativity cache power [125] path-based hardware loop prediction [42] multimedia application characteristics [6] hardware loop unrolling [43] characterization of embedded applications [155] analysis of cache performance of multimedia [198] memory system performance of multimedia [166] characterization of multimedia on general purpose architecture [86]

enabling QoS in shared caches [65] process dependent cache partitioning [89] improving disk hit-ratios through cache partitioning [173] two level partitioning and loop scheduling [192] timing analysis of data caches [193]

direct addressed scratchpad memories [31] probabilistic miss equations [53] reconfigurable caches for media processing [143] compiler approach reducing data cache energy [206] compile-time analysis of cache behavior [186] optimal partitioning of cache memory [167] footprints in the cache [172] dynamic programming algorithm for partitioning [153] customizable partitioned caches [134] towards effective embedded processors [135] reducing cache conflicts [102] eliminating address translation bottleneck [25] paged cache [24] partitioned first-level cache design [142] compiler support for software-based cache partitioning [123] process-dependent partitioning strategy for cache memories [45] power aware partitioned cache architecture [84] partitioned instruction cache for energy efficiency [87] span cache [195] direct addressed cache [196]

v-way cache: demand based associativity [140] cooperative caching with keep-me and evict-me [152] reuse distance based miss rate prediction [52] fair cache sharing and partitioning in chip mutiprocessor [85] combining cooperative hardware/software prefetching with replacement [190] dynamic memory allocation for scratch-pad based embedded systems [176] phase-based miss rate prediction [158] locality phase prediction [157] transition phase classification and prediction [95] data prefetching on the HP PA-8000 [151]

selective cache ways [4] design space optimization of embedded memory systems via remapping [141] bridging processor memory performance via data remapping [130]

flexcache: a framework for flexible compiler generated data caching [119] heterogeneous memory management for embedded systems [8] compiler support for scalable and efficient memory systems [9] cam-tag cache resizing [205] way memoization in instruction caches [108] highly-associative caches for low-power processors [204]

enhancing memory level parallelism via recovery-free value prediction [209]

tag correlating prefetchers [64] software and hardware data prefetching schemes [29] guided region prefetching [189] compiler-assisted data prefetch controller [182] profile-guided post-link stride prefetching [107] dynamic hot data stream prefetching [34] filtering superfluous prefetches [104] data prefetching techniques [181] data prefetch mechanisms survey [183]

exploiting choice in resizable cache design to optimize energy-delay [200] smart memories - a modular reconfigurable architecture [109]

LIRS replacement policy [69] self-correcting lru replacement policies [74] cost-sensitive cache replacement algorithms [68] optimal replacements with two miss costs [67] compiler enhancement of global cache reuse [47] reuse distance analysis [48] reuse distance as a metric for cache behavior [12]

dynamical adjustment of block replacement algorithms [188] kora-2 cache replacement policy [82] neural networks-based adaptive cache replacement [126] cache decay to reduce leakage power [78] modeling live and dead lines in cache memory systems [116] optimality proof of LRU-K page replacement algorithm [127] page replacement for general caching problems [3]

efficient management of memory hierarchies in embedded dram systems [154] run-time cache bypassing [70] runtime identification of cache conflict misses: adaptive miss buffer [38] filter cache management approach [146]

predictability and optimization of multiprogrammed caches [156] understanding correlation profiling [122] predicting data cache misses through correlation profiling [121] tight bound on task interference in instruction caches [51]

# Chapter 11

# Conclusion

## 11.1 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. The problems associated with on-chip cache in embedded systems are addressed to improve cache performance, cache predictability, 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 static partitioning algorithm was developed that used the hardware mechanisms for cache partitioning. The cache partitioning mechanisms were implemented in the Simplescalar simulator and evaluated with and without prefetching on the Spec2000, Mibench, and Mediabench benchmarks as described in Chapter 8.

## 11.2 Extensions

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

### 11.2.1 Kill+LRU Replacement for Multiple Processes

The Kill+LRU replacement can be extended to multi-process environment where the information about the process scheduling and process footprints can be used to derive the reuse distance and used in determining the dead blocks for a process and marked as killed blocks for Kill+LRU replacement.

### 11.2.2 Disjoint Sequences for Loop Transformations

The concept of disjoint sequences can be applied to improve performance using program code transformations. I did some work on loop transformations using the concept of disjoint sequences [**?**]. In this work, disjoint sequences were applied for loop distribution transformation. It can be extended to loop fusion, loop interchange, loop tiling, and other loop transformations to improve program performance.

### 11.2.3 Hardware-based Cache Mechanisms

### 11.2.4 Profile-based Memory Exploration System

### 11.2.5 Kill+Prefetch at L2 level

When the L1 block is evicted, the L2 block can be marked to be killed. This will work when the evicted block was the last use. or the reuse distance is L1+L2 unique blocks. The block from L2 cannot be evicted until it is evicted from L1 to maintain the inclusion property.

# 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] Susanne Albers, Sanjeev Arora, and Sanjeev Khanna. Page replacement for general caching problems. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Descrete Algorithms*, pages 31–40, Baltimore, MD, 1999.

[4] David H. Albonesi. Selective cache ways: On-demand cache resource allocation. In *Proceedings of the 32nd Annual ACM/IEEE Symposium on Microarchitecture*, pages 248–259, Haifa, Israel, 1999.

[5] 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.

[6] Navneet Aron, Honggo Wijaya, Arjun Singh, and Varun Malhotra. Study of Multimedia Application Characteristics. April 17 2003.

[7] 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.

[8] Oren Avissar, Rajeev Barua, and Dave Stewart. Heterogeneous Memory Management for Embedded Systems. In *CASES '01: Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 34–43, Atlanta, GA, November 16-17 2001.

[9] Rajeev Barua, Walter Lee, Saman Amarasinghe, and Anant Agarwal. Compiler support for scalable and efficient memory systems. *IEEE Transactions on Computers, Special Issue on Memory Systems*, 50(11):1234–1247, November 2001.

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

[11] 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.

[12] Kristof Beyls and Erik H. D'Hollander. Reuse distance as a metric for cache behavior. In *Proceedings of the IASTED International Conference Parallel and Distributed Computing and Systems*, pages 617–622, Anaheim, CA, August 21-24 2001.

[13] Benjamin Bishop, Thomas P. Kelliher, and Mary Jane Irwin. A Detailed Analysis of Mediabench. In *Proceedings of the 1999 IEEE Workshop on Signal Processing Systems (SiPS)*, pages 448–455, Taipei, Taiwan, October 20-22 1999.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] J. Adam Butts and Guri Sohi. Dynamic Dead-Instruction Detection and Elimination. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, pages 199–210, San Jose, CA, USA, October 2002.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] Yen-Jen Chang and Feipei Lai. Paged cache: An efficient partition architecture for reducing power, area and access time. In *Proceedings of IEEE Asia Pacific Conference on Circuits and Systems*, pages 473–478, Singapore, December 2002.

[25] Yen-Jen Chang, Feipei Lai, and Chanq-Jang Ruan. Cache design for eliminating address translation bottleneck and reducing the tag area cost. In *Proceedings of the 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'02)*, pages 334–339, Freiburg, Germany, September 16-18 2002.

[26] 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.

[27] 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.

[28] 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.

[29] Tien-Fu Chen and Jean-Loup Baer. A performance study of software and hardware data prefetching schemes. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 223–232, Chicago, IL, 1994.

[30] 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.

[31] Allen Cheng, Chen Ma, Pracheeti Nagarkar, and Hongtao Zhong. Direct addressed scratchpad memories to reduce power consumption. EECS 583 Course Project, Department of EECS, University of Michigan, Winter 2003.

[32] 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.

[33] 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.

[34] Trishul M. Chilimbi and Martin Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Berlin, Germany, June 17-19 2002.

[35] 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.

[36] 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.

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

[38] 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.

[39] R. Cooksey and D. Grunwald. Characterization of the spec2000 benchmark suite, 2001.

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

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

[42] Marcos R. de Alba and David R. Kaeli. Path-based Hardware Loop Prediction.

[43] Marcos R. de Alba and David R. Kaeli. Characterization and Evaluation of Hardware Loop Unrolling. April 17 2003.

[44] Pepijn de Langen and Ben Juurlink. Off-Chip Memory Traffic Measurements of Low-Power Embedded Systems. In *Proceedings of ProRISC Workshop on Circuits, Systems and Signal Processing*, pages 351–358, 2002.

[45] Yannick Deville. A process-dependent partitioning strategy for cache memories. *ACM SIGARCH Computer Architecture News*, 21(1):26–33, March 1993.

[46] Ashutosh S. Dhodapkar and James E. Smith. Comparing Program Phase Detection Techniques. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 217–227, December 2003.

[47] Chen Ding and Ken Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. *Journal of Parallel and Distributed Computing*, 64(1):108–134, January 2004.

[48] Chen Ding and Yutao Zhong. Reuse distance analysis. Technical Report UR-CS-TR-741, Computer Science Department, University of Rochester, Rochester, NY, February 2001.

[49] 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.

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

[51] Harry Dwyer and John Fernando. Establishing a Tight Bound on Task Interference in Embedded System Instruction Caches. In *CASES '01: Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 8–14, Atlanta, GA, November 16-17 2001.

[52] 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.

[53] Basilio B. Fraguela, Ramon Doallo, and Emilio L. Zapata. Probabilistic miss equations: Evaluating memory hierarchy performance. *IEEE Transactions on Computers*, 52(3):321–336, March 2003.

[54] 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.

[55] 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.

[56] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *In IEEE 4th Annual Workshop on Workload Characterization (WWC-4)*, December 2001.

[57] 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.

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

[59] John L Henning. Spec CPU2000: Measuring CPU Performance in the New Millennium. *IEEE Computer*, 33(7):28–35, July 2000.

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

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

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

[63] 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.

[64] 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.

[65] 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.

[66] 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.

[67] 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.

[68] 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.

[69] Song Jiang and Xiaodong Zhang. Lirs: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 31–42, Marina Del Rey, CA, 2002.

[70] 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.

[71] 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.

[72] 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.

[73] 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.

[74] 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.

[75] 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.

[76] Gokul B. Kandiraju and Anand Sivasubramaniam. Characterizing the d-TLB Behavior of SPEC CPU2000 Benchmarks. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 129–139, June 2002.

[77] 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.

[78] Stefanos Kaxiras, Zhigang Hu, and Margaret Martonosi. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 240–251, Goteborg, Sweden, June 2001.

[79] Baris M. Kazar, Joshua J. Yi, and David J. Lilja. A characterization of the output value locality of the spec95, spec2000, and mediabench benchmark suites. Minnesota supercomputing institute, University of Minnesota, Twin Cities, MN.

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

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

[82] H. Khalid and M. S. Obaidat. Kora-2: A new cache replacement policy and its performance. In *The 6th IEEE International Conference on Electronics, Circuits and Systems*, pages 265–269, Pafos, Cyprus, September 5-8 1999.

[83] 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.

[84] 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.

[85] 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.

[86] Seongwoo Kim and Arun K. Somani. Characterization of an extended multimedia benchmark on a general purpose microprocessor architecture. Technical Report DCNL-CA-2000-002, Electrical and Computer Engineering Department, Iowa State University, 2000.

[87] 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.

[88] Johnson Kin, Munish Gupta, and William H. Mangione-Smith. The Filter Cache: An Energy Efficient Memory Structure. In *Proceedings of the International Symposium on Microarchitecture*, pages 184–193, December 1997.

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

[90] 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.

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

[92] 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.

[93] 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.

[94] 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.

[95] Jeremy Lau, Stefan Schoenmackers, and Brad Calder. Transition phase classification and prediction. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, San Francisco, CA, February 12-16 2005.

[96] Alvin R. Lebeck and David A. Wood. Cache Profiling and the SPEC Benchmarks: A Case Study. *IEEE Computer*, 27(10):15–26, October 1994.

[97] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems. In *Proceedings of the International Symposium on Microarchitecture*, pages 330–335, 1997.

[98] 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.

[99] 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.

[100] Lin Li, Ismail Kadayif, Yuh-Fang Tsai, N. Vijaykrishnan, Mahmut Kandemir, Mary Jane Irwin, and Anand Sivasubramaniam. Managing Leakage Energy in Cache Hierarchies. *Journal of Instruction-Level Parallelism*, 5, April 2003.

[101] 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.

[102] Zhiyuan Li. Reducing cache conflicts by partitioning and privatizing shared arrays. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT'99)*, pages 183–190, Newport Beach, CA, October 12-16 1999.

[103] AJ KleinOsowski David J. Lilja. MinneSPEC: A New SPEC Benchmark Workload for Simulation-based Computer Architecture Research. *Computer Architecture Letters*, 1, June 2002.

[104] Wei-Fen Lin, Steven K. Reinhardt, Doug Burger, and Thomas R. Puzak. Filtering superfluous prefetches using density vectors. In *International Conference on Computer Design (ICCD '01)*, pages 124–132, Austin, TX, September 23-26 2001.

[105] 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.

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

[107] 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.

[108] Albert Ma, Michael Zhang, and Krste Asanovic. Way memoization to reduce fetch energy in instruction caches. In *Workshop on Complexity-Effective Design, 28th International Symposium on Computer Architecture*, Gothenburg, Sweden, June 2001.

[109] 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.

[110] 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.

[111] Mahesh Mamidipaka and Nikhil Dutt. On-chip Stack based Memory Organization for Low Power Embedded Architectures. In *Proceedings of the Design, Automation, and Test in Europe Conference (DATE'03)*, pages 1082–1087, Munich, Germany, March 2003.

[112] 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.

[113] 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.

[114] 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.

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

[116] Abraham Mendelson, Dominique Thiebaut, and Dhiraj K. Pradhan. Modeling live and dead lines in cache memory systems. *IEEE Transactions on Computers*, 42(1):1–14, January 1993.

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

[118] 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.

[119] Csaba. A. Moritz, Matthew Frank, and Saman Amarasinghe. Flexcache: A framework for compiler generated data caching. In *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[120] 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.

[121] Todd C. Mowry and Chi-Keung Luk. Predicting data cache misses in non-numeric applications through correlation profiling. In *MICRO-30: Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 314–320, Research Triangle Park, North Carolina, December 1-3 1997.

[122] Todd C. Mowry and Chi-Keung Luk. Understanding why correlation profiling improves the predictability of data cache misses in nonnumeric applications. *IEEE Transactions on Computers*, 49(4):369–384, April 2000.

[123] Frank Mueller. Compiler support for software-based cache partitioning. *ACM SIGPLAN Notices*, 30(11):125–133, November 1995.

[124] 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.

[125] 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.

[126] Mohammad S. Obaidat and Humayun Khalid. Estimating neural networks-based algorithm for adaptive cache replacement. *IEEE Transactions on Systems, Man, and Cybernetics–Part B: Cybernetics*, 28(4):602–611, August 1998.

[127] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. An optimality proof of the lru-k page replacement algorithm. *Journal of the ACM*, 46(1):92–112, January 1999.

[128] 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.

[129] 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.

[130] Krishna V. Palem and Rodric M. Rabbah. Bridging processor and memory performance in ilp processors via data-remapping. Technical Report CREST-TR-01-002, GIT-CC-01-014, Georgia Institute of Technology, Center for Research on Embedded Systems and Technology, June 2001.

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

[132] 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.

[133] 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.

[134] 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.

[135] 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.

[136] Venkata K. Pingali, Sally A. McKee, Wilson C. Hseih, and John B. Carter. Computation regrouping: Restructuring programs for temporal data cache locality. In *International Conference on Supercomputing*, pages 252–261, New York, NY, USA, June 22-26 2002.

[137] 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.

[138] 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.

[139] 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.

[140] 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.

[141] Rodric M. Rabbah, Krishna V. Palem, Vincent J. Mooney III, Pinar Korkmaz, and Kiran Puttaswamy. Design space optimization of embedded memory systems via data remapping. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems*, pages 28–37, Berlin, Germany, 2002.

[142] 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.

[143] 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.

[144] 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.

[145] 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.

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

[147] Suleyman Sair and Mark Charney. Memory behavior of the spec2000 benchmark suite. Technical Report RC-21852, IBM T. J. Watson Research Center, October 2000.

[148] 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.

[149] 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.

[150] 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.

[151] 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.

[152] 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.

[153] John E. Sasinowski and Jay K. Strosnider. A dynamic programming algorithm for cache/memory partitioning for real-time systems. *IEEE Transactions on Computers*, 42(8):997–1001, August 1993.

[154] Ashley Saulsbury, Su-Jaen Huang, and Fredrik Dahlgren. Efficient management of memory hierarchies in embedded dram systems. In *ICS '99: Proceedings of the International Conference on Supercomputing*, pages 464–473, Rhodes, Greece, June 20-25 1999.

[155] Assia Djabelkhir Andre Seznec. Characterization of Embedded Applications for Decoupled Processor Architecture. In *Proceedings of the IEEE 6th Annual Workshop on Workload Characterization*, Austin, TX, October 2003.

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

[157] Xipeng Shen, Yutao Zhong, and Chen Ding. Locality Phase Prediction. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*, pages 165–176, Boston, MA, USA, October 7-13 2004.

[158] Xipeng Shen, Yutao Zhong, and Chen Ding. Phase-based miss rate prediction across program inputs. In *The 17th International Workshop on Languages and Compilers for Parallel Computing*, West Lafayette, IN, September 2004.

[159] 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.

[160] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, October 2002.

[161] Wen-Tsong Shiue and Chaitali Chakrabarti. Memory Design and Exploration for Low Power, Embedded Systems. *Journal of VLSI Signal Processing - Systems for Signal, Image, and Video Technology*, 2001.

[162] Jan Sjodin, Bo Froderberg, and Thomas Lindgren. Allocation of Global Objects in On-Chip RAM. In *Workshop on Compiler and Architecture Support for Embedded Computing Systems*, Washington, DC, USA, December 4-5 1998.

[163] 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.

[164] 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.

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

[166] Sohum Sohoni, Zhiyong Xu, Rui Min, and Timing Hu. A Study of Memory System Performance of Multimedia Applications. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 206–215, Cambridge, MA, June 16-20 2001.

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

[168] 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.

[169] Dinesh C. Suresh, Walid A. Najjar, Frank Vahid, Jason R. Villarreal, and Greg Stitt. Profiling Tools for Hardware/Software Partitioning of Embedded Applications. In *Proceedings of the 2003 ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '03)*, pages 189–198, San Diego, CA, June 11-13 2003.

[170] 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.

[171] 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.

[172] Dominique Thiebaut and Harold S. Stone. Footprints in the cache. *ACM Transactions on Computer Systems*, 5(4):305–329, November 1987.

[173] 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.

[174] 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.

[175] 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.

[176] Sumesh Udayakumaran and Rajeev Barua. Compiler-Decided Dynamic Memory Allocation for Scratch-Pad Based Embedded Systems. In *CASES '03: Proceedings of the 2003 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 276–286, San Jose, CA, October 30 - November 2 2003.

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

[178] 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.

[179] 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.

[180] 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.

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

[182] 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.

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

[184] 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.

[185] 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.

[186] Xavier Vera and Jingling Xue. Efficient compile-time analysis of cache behavior for programs with if statements. In *Fifth International Conference on Algorithms and Architectures for Parallel Processing*, pages 396–407, Beijing, China, October 23-25 2002.

[187] Kugan Vivekanandarajah, Thambipillai Srikanthan, Christopher Clarke, and Saurav Bhattacharya. Static Pattern Predictor (SPP) Based Low Power Instruction Cache Design. In *Embedded Systems and Applications*, pages 210–215, 2003.

[188] Udo Walterscheidt and Ravi Pendse. Dynamical adjustment of block replacement algorithms. In *Proceedings of the 30th Southeastern Symposium on System Theory*, pages 544–548, Morgantown, WV, March 8-10 1998.

[189] 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.

[190] 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.

[191] 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.

[192] Zhong Wang, Michael Kirkpatrick, and Edwin Hsing-Mean Sha. Optimal two level partitioning and loop scheduling for hiding memory latency for dsp applications. In *Design Automation Conference*, pages 540–545, Los Angeles, CA, June 2000.

[193] Randall T. White, Frank Mueller, Christopher A. Healy, David B. Whalley, and Marion G. Harmon. Timing analysis for data caches and set associative caches. In *3rd IEEE Real-Time Technology and Applications Symposium (RTAS '97)*, pages 192–202, Montreal, Canada, June 9-11 1997.

[194] 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.

[195] 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.

[196] 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.

[197] 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.

[198] Zhiyong Xu, Sohum Sohoni, Rui Min, and Timing Hu. An Analysis of Cache Performance of Multimedia Applications. *IEEE Transactions on Computers*, 53(1):20–38, January 2004.

[199] 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.

[200] 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.

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

[202] Joshua J Yi, Resit Sendag, and David J Lilja. The spatial characteristics of load instructions. Laboratory for Advanced Research in Computing Technology and Compilers ARCTiC 02-10, University of Minnesota, Twin Cities, MN, October 2002.

[203] 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.

[204] Michael Zhang and Krste Asanovic. Highly-associative caches for low-power processors. In *Kool Chips Workshop, 33rd International Symposium on Microarchitecture*, Monterey, CA, December 2000.

[205] Michael Zhang and Krste Asanovic. Fine-grain cam-tag cache resizing using miss tags. In *Proceedings of the 2002 International Symposium on Low Power Electronics and Design (ISLPED'02)*, Mnoterey, CA, August 12-14 2002.

[206] 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.

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

[208] 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.

[209] Huiyang Zhou and Thomas M. Conte. Enhancing Memory Level Parallelism via Recovery-Free Value Prediction. In *ICS '03: Proceedings of the International Conference on Supercomputing*, pages 326–335, San Francisco, CA, June 23-26 2003.