Analyzing the Behavior of Loop Nests in the Memory Hierarchy: Methods, Tools, and Applications

by
Erin Parker

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2004

Approved by:

Siddhartha Chatterjee, Advisor

Alvin Lebeck, Reader

Frank Mueller, Reader

Jan Prins, Reader

Jack Snoeyink, Reader

David Plaisted

ABSTRACT

ERIN PARKER: Analyzing the Behavior of Loop Nests in the Memory Hierarchy: Methods, Tools, and Applications.
(Under the direction of Siddhartha Chatterjee.)

Processor speeds are improving at a much faster rate than the speeds of accessing main memory. As a result, data access time dominates the execution times of many programs. Understanding the behavior of programs executing in a memory hierarchy is therefore an important part of improving program performance. This dissertation describes an analytical framework for understanding the behavior of loop-oriented programs executing in a memory hierarchy. The framework has three components: 1) an alternative classification of cache misses that makes it possible to obtain the exact cache behavior of a sequence of program fragments by combining the cache behavior of the individual fragments; 2) the use of Presburger arithmetic to model data access patterns and describe events such as cache misses; and 3) algorithms exploiting the connection among Presburger arithmetic, automata theory, and graph theory to produce exact cache miss counts.

The analytical framework presented in this dissertation goes beyond existing analytical frameworks for modeling cache behavior: it handles set-associative caches, data cache and translation lookaside buffer (TLB) misses, imperfect loop nests, and nonlinear array layouts in an exact manner. Experiments show both the framework's value in the exploration of new memory system designs and its usefulness in guiding code and data transformations for improved program performance.

ACKNOWLEDGMENTS

I thank my thesis advisor Sid Chatterjee for his unwavering guidance and support. Without his encouragement, the work of this dissertation would not have been possible. In particular, his persistence and attention even after leaving Chapel Hill in 2001 are greatly appreciated.

I thank my committee members Alvy Lebeck, Frank Mueller, Dave Plaisted, Jan Prins, and Jack Snoeyink for their advice and for being so generous with their time.

I thank the entire UNC Department of Computer Science for making Sitterson Hall such an enjoyable place to work and learn. Especially, I thank the administrative and technical staff of the computer science department, whose assistance has been invaluable. I thank my UNC classmates and friends for making the last five years a lot of fun. Tom B, Shelby, Dave, Zac, Kimberly, and Josh, your friendship has made it all worthwhile.

I thank the Department of Energy's High-Performance Computer Science Fellowship program, which has sponsored me for the last four years.

Finally, I thank my family for their amazing support and understanding. I could not do without the limitless love and encouragement of my father Walt, mother Barbara, sister Adrienne, sister Amy, and husband Tom. These people deserve more than just their names on this page. I hope that in real life, I can do enough to thank them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

As advances in processor speeds continue to outstrip those of main memory access, the processor-memory performance gap grows every year [66, 138]. The time required for accessing data from main memory is one to two orders of magnitude larger than the time required for operations on the data. **Cache memory**, one or more levels of fast memory placed between the processor and main memory, attempts to reduce the average data access time by capturing the most frequently referenced data close to the processor. In general, hardware alone does not solve the the problem of poor memory performance, because not all programs use cache memory effectively. Therefore, even with the addition of caches, the execution time of many programs is dominated by the time spent accessing data from main memory.

By modifying a program's data access patterns or data layout (*i.e.*, the manner in which program variables map to memory locations), it is possible to use cache memory more effectively and improve the program's performance. Either a programmer can make code and data layout changes "by hand", or a compiler can do so automatically. In transforming a program, it is critical to know what changes will lead to improved performance, or better yet, what changes will lead to the best possible performance. The challenge of transforming a program to reduce its memory access time is understanding the behavior of the program executing in the presence of cache memory. This dissertation presents a static framework that produces the exact cache behavior of a program, given virtually any configuration of cache memory. The remainder of this chapter further motivates the work of analyzing cache behavior (Section 1.1), reviews a variety of methods for determining cache behavior (Section 1.2), lists the contributions of this thesis (Section 1.3), points out the limitations of analysis framework presented here (Section 1.4), and provides a roadmap for the rest of the document (Section 1.5).

1.1 Motivation

Processor and main memory, or dynamic random access memory (DRAM), technologies are constantly improving, but at different rates. From 1980 to 1986, processor frequency improved

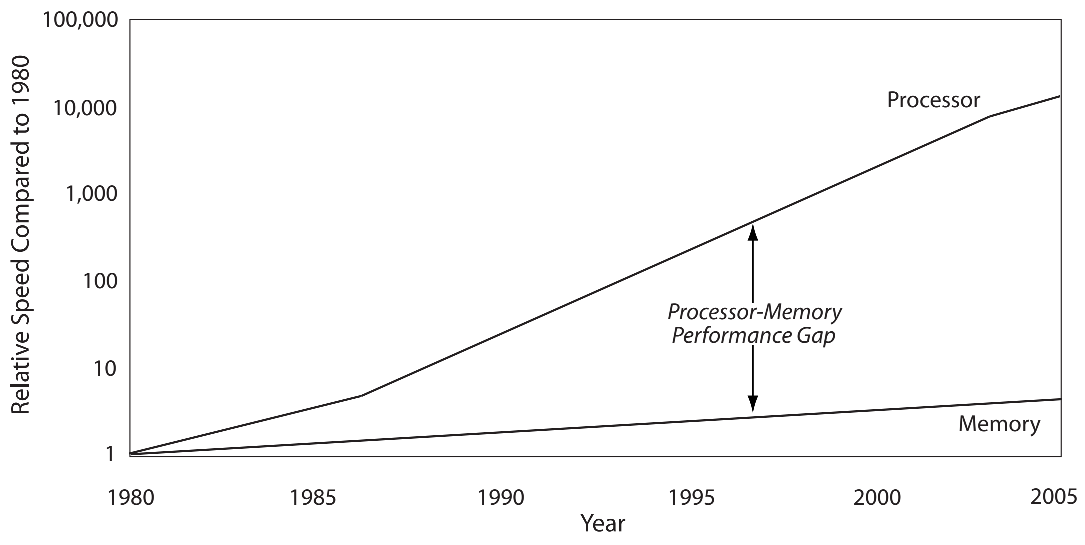


Figure 1.1: Processor speed and speed of accessing main memory plotted over time, with speed in 1980 as a baseline.

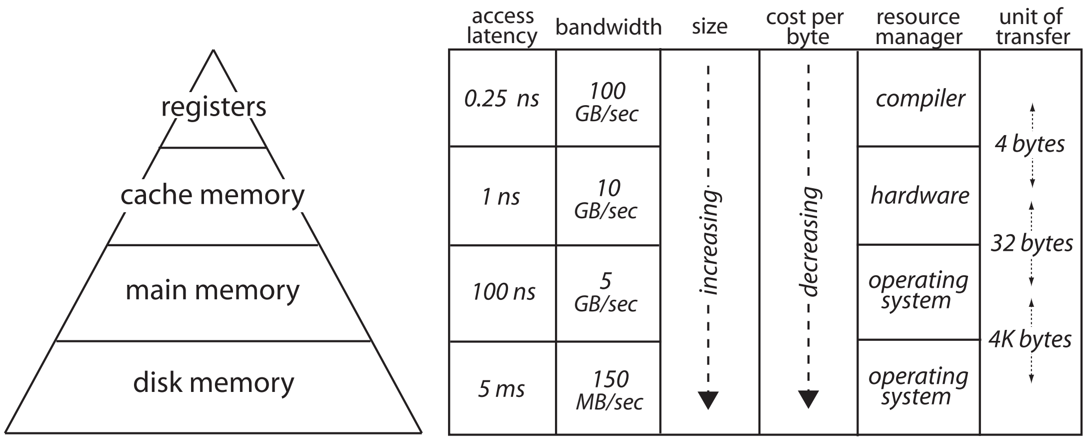
by about 35% per year, while from 1987 to 2002, processor frequency improved by about 55% per year [66]. Since 2003, the rate of improvement in processor frequency has slowed down to about 35% per year [53]. DRAM performance (*i.e.*, the latency of accessing data) has improved steadily at about 7% per year [66]. The diverging rates of improvement have formed the processor-memory performance gap. Figure 1.1 shows how processor and memory performance have improved over time relative to their performance in 1980, with the processor-memory performance gap clearly evident. For more and more programs, memory access time determines execution time, and this trend will continue unless the performance gap begins to close.

In an ideal system, memory would be infinitely large and any data required by a program would be immediately available to the processor. Given the physical constraints of a computer, such an ideal is not possible. As early as 1946, Burks, Goldstine, and von Neumann [28] suggested a hierarchy as a way of arranging memory:

> We are therefore forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible.

In 1962, Kilburn *et al.* [79] proposed automatic management of two levels of memory and demonstrated it in the Atlas computer. A few years later, Wilkes [130] wrote the first paper describing the concept of a cache. The first commercial computer with cache memory, the IBM 360/85 [87], followed shortly. Today almost all computer systems have cache memory, and most have multilevel caches [8, 71, 126].

Figure 1.2 shows the hierarchy of four kinds of memory typically found in systems today. A



access latency	bandwidth	size	cost per byte	resource manager	unit of transfer
0.25 ns	100 GB/sec	increasing	decreasing	compiler	4 bytes
1 ns	10 GB/sec			hardware	32 bytes
100 ns	5 GB/sec			operating system	4K bytes
5 ms	150 MB/sec			operating system	

Figure 1.2: The memory hierarchy pyramid and typical values of relevant parameters [66]. Smaller memories are faster to access and more expensive per byte. Data contained in upper level i is also contained in lower level $i + 1$.

relatively small register file located in the processor core is the most quickly-accessible memory. Disk memory is far from the processor and the slowest to access. Five parameters describe the levels of a uniprocessor memory hierarchy: speed (indicated by both access latency and bandwidth), size, cost per byte, manager of the memory level, and the unit of transferring data between levels. The table in Figure 1.2 gives typical parameter values for each memory level [66]. Moving up the hierarchy, speed and cost per byte increase, as size decreases. Also, an upper level of memory is usually a subset of lower levels (*i.e.*, data contained in upper level i is also contained in lower level $i+1$). Given the speed of the memory hierarchy's upper levels, it would be best to access all data from there, but the expense of fast memory forces the upper levels to be small in size. These levels, therefore, cannot hold all of the data. As a result, some of the data is discarded from the upper levels before the program has finished using it. A program is said to have good **locality of reference** if it can reuse data while it is still in the upper levels of memory, which requires that the use and reuse of data be close together in time. The property of programs to reuse data and instructions they have used recently is called **locality** [66], and locality of reference refers to locality in the context of data.

The job of cache memory is to hold frequently referenced data close to the processor. A cache miss occurs when a program requests data that is not found in cache memory. **Miss penalty** is the cost of a cache miss, *i.e.*, the time required to access missed data from the lower levels of memory. The following equations [66] illustrate the impact of cache misses and miss penalty on the execution time of a program.

$$\text{execution time} = (\text{CPU clock cycles} + \text{memory stall cycles}) \times \text{clock cycle time} \tag{1.1}$$

$$\text{memory stall cycles} = \text{number of cache misses} \times \text{miss penalty} \tag{1.2}$$

It is clear that more cache misses and/or larger miss penalties slow the overall execution time of a program by worsening its memory performance. A good measure of memory performance is **average memory access time**, the average time spent accessing memory per request, as given by the following equation [66].

$$\text{average memory access time} = \text{hit time} + (\text{miss rate} \times \text{miss penalty}) \qquad (1.3)$$

Hit time is the time to access memory from cache, and miss rate is the fraction of cache accesses resulting in a miss. Attempts in software to improve the memory performance of programs (*e.g.*, via transformations in code and data layout) cannot change hit time or miss penalty, but can shorten the average memory access time by reducing the number of cache misses. In other words, improving a program's locality of reference allows execution of the program to better utilize caches.

Efficient programs have good **locality of reference**, both temporal, by accessing recently referenced data together, and spatial, by accessing data with memory addresses near recently referenced data [114]. Caches exploit locality of reference in a program by capturing blocks of memory containing recently referenced data close to the processor. A program must have well-crafted code and data organization in order to achieve good locality of reference, which is a burden on the programmer, since the relationship between writing a program and its resulting locality of reference is often not straightforward. Locality of reference is often elusive and brittle: as an application evolves, seemingly insignificant modifications to its code or to the machine platform can lead to dramatic changes in its locality of reference.

There are two ways of transforming a program to improve its locality of reference: altering the order in which the program accesses data, and altering the data layout. Loop transformations [135] change the ordering of operations in loop-oriented programs in a way that preserves the semantics of the original program while producing a data access sequence with better locality of reference. Loop interchange [133], loop skewing [135], and iteration-space tiling [76, 132, 134] are examples of loop transformations. Although the issues of transformation legality and code generation for transformed loops are well-defined, methods for predicting the performance benefit of transforming a particular loop are less understood. Existing methods for evaluating the potential benefit of loop transformations usually offer only approximate or heuristic findings [40, 84, 93, 101, 108, 109, 132].

Recently, researchers have given some attention to transforming the layout of program data [7, 32, 33, 35, 36, 57, 84, 108, 109, 119, 131]. Data transformations modify the layout of data in memory in order to improve the locality of reference for the program's data access sequence. Array copying [84, 119], array padding [7, 108, 109], and nonlinear array layouts [32, 33, 57, 131] are examples of well-known data transformations. Careful placement of data in C-language structures, either by reordering fields within a single structure or by packing the frequently-accessed fields of several structures in the same cache line [35, 36], is another example of an effective data transformation. Many code and data transformations have parameters whose values determine the behavior of the transformed program, such as tile sizes (for iteration-space tiling) and pad sizes (for array padding). As in the case for loop transformations, predictions of performance improvement due to a data transformation are often approximations. Rarely are code and data layout transformations beneficial when applied blindly. At issue are which transformations to apply, in which order to apply them, and how to set transformation parameters.

In attempting to improve a program's locality of reference, the goal of code and data transformations is to increase the number of times that the program accesses data from cache memory rather than from main memory. When successful, the average memory access time of the program is reduced. Another approach to dealing with large memory access latency is to tolerate, or hide, it. Techniques such as nonblocking (or lock-up free) caches [83], software-controlled prefetching [94, 95, 96], hardware-controlled prefetching [101, 114], stream buffers [70], and speculative loads [110] overlap data access with operations on previously-accessed data, hiding the large latency of accessing memory. These techniques work by reducing the observed latency for a group of memory accesses, exploiting concurrency at the hardware level. All of these methods attempt to cover up the problem of a large average memory access time and do not attempt to reduce it. Prefetching, stream buffers, and speculative loads potentially access more data from memory that is needed by the program, creating more contention from memory resources and possibly worsening the memory performance bottleneck. When effective, techniques for reducing the average memory access time are preferable over those that tolerate it. Extensive research has looked at the use of program transformations for reducing average memory access time, as the next section discusses.

1.2 Determining Cache Behavior

Much research has been devoted to guiding the application of code and data transformations, and most of the work focuses on loop-oriented programs. Improving the memory performance of loops has a significant impact on scientific programs, which spend the majority of their execution time in loops. Scientific programs tend to operate on large amounts of data, of which the cache can hold only a small amount. If scientific programs are not written in a manner that leads to good locality of reference, most of the data is displaced from cache before it can be reused. As a result, scientific programs are often the target of code and data transformations.

Early techniques for guiding loop and data transformations [7, 58, 84, 108] target specific loops, cache memory configurations, and/or loop transformations. Lam *et al.* [84] present a model for approximating the number of cache misses for tiled matrix multiplication and give an algorithm for selecting tile size, a parameter for iteration-space tiling. Fricker *et al.* [58] develop a model for approximating the number of cache misses for tiled matrix-vector multiplication executing in a specific cache memory. Bacon *et al.* [7] introduce an algorithm for selecting pad sizes, a parameter for array padding. Rivera and Tseng [108] put forth techniques for applying inter- and intra-array padding transformations. Although the work has resulted in useful insights on the effects of specific transformations, it may not be possible to generalize these insights to apply to all programs.

Program simulation [46, 85, 91] is a well-established method for gauging memory performance. Entire programs can be simulated and there are no limits on cache memory configurations or program transformations. Simulating the execution of a program allows the effects of back-end compiler phases, such as instruction scheduling and register allocation, to be reflected in the cache behavior of the program. MemSpy [91] simulates the execution of a program to provide cache miss rates and causes for particular code and data objects. Similarly, CProf [85] is a cache performance profiler that uses simulation to identify source code and data structures with poor cache behavior. The Dinero [46] simulator reports the behavior of one or more cache designs, given as input a list of the memory references that a program makes during execution. However, there is a potential disadvantage to simulating execution of a program: the simulation running time greatly exceeds that of the original program, in general. Dynamic instrumentation [43] and hardware counters [25, 51] also capture the memory referencing behavior of a program, and have the same advantages and potential disadvantage as simulation.

Static analysis [31, 34, 54, 62, 122] gathers information from the source code of a program and from the configuration of an underlying memory system to determine the exact or approximate cache behavior without actually running or simulating the program. In particular, for loops, static analysis, whose complexity relates to the static structure of the loops, has the potential to be faster than simulation, whose complexity relates to the iteration count of the loops and is proportional to the execution time of the loops. Static analysis may be especially useful in profiling the cache activity of programs with very long execution times for which it is undesirable to do explicit simulation. Just as knowledge of the low-level, machine-specific arrangement of memory accesses is an advantage for simulation, the lack of such knowledge is a disadvantage for static analysis. Static analysis must make simplifying assumptions about the scheduling of instructions, about what data resides in registers, and so on. Memory behavior information determined at compile time via static analysis of a program can be used in a compiler to select program transformations.

Despite the benefits of static cache behavior analysis, there currently exists no compile-time framework for understanding cache behavior that is flexible, exact, and fast. Chapter 9 discusses several existing frameworks for analyzing memory behavior. The following are six weaknesses common to some or all of these frameworks.

- **Approximating cache behavior.** Simplifications in modeling the behavior of a program executing in a cache lead to an approximation of the program's actual cache behavior. Using sampling to estimate the number of cache misses provides a good understanding of a program's cache behavior when there are many cache misses occurring uniformly throughout the program. When the cache misses are few and sparse throughout the program, only an exact model of cache behavior guarantees an accurate picture of a program's cache performance.

- **Modeling only fully-associative caches.** The mapping of data from main memory to cache depends on the organization of the cache. For data caches, the organization is typically set-associative, sometimes direct-mapped, and almost never fully-associative. Furthermore, frameworks that model only fully-associative caches usually require that memory blocks contain only one data element. A cache model with such parameters is not representative of the caches in today's systems.
- **Modeling data caches only, ignoring misses in the translation lookaside buffer (TLB).** A data cache stores most frequently referenced data close to the processor, while a TLB stores memory address translations. The misses in both the data cache and TLB determine a program's memory performance. Modeling a program's behavior in a data cache and TLB is more complete than modeling its behavior in a data cache alone.
- **Considering only perfectly-nested loops.** A loop nest is a loop containing one or more other loops, and the nesting may be perfect or imperfect. Considering only perfect loop nests restricts the programs whose cache behavior can be analyzed.
- **Considering loop nests in isolation of each other.** Modeling the cache behavior of loop nests out of the context of the rest of the program is inexact. Not taking into account the contents of the cache before a loop nest begins execution leads to an overestimation of the actual cache miss count for the loop nest.
- **Handling only canonical array layout functions (*i.e.* row- and column-major).** Row- and column-major are the standard ways of mapping arrays to memory. However, the ability to evaluate the potential benefit of nonstandard data layouts is useful.

This dissertation describes an analytical framework for understanding the behavior of loop nests executing in a memory hierarchy. The framework has three components:

1) an alternative classification of cache misses that makes it possible to obtain the exact cache behavior of a sequence of program fragments by combining the cache behavior of the individual fragments (see Chapter 3);
2) the use of Presburger arithmetic [102, 103, 117] to model data access patterns and describe events such as cache misses (see Chapter 4); and

3) algorithms exploiting the connection among Presburger arithmetic, automata theory, and graph theory to produce exact cache miss counts (see Chapter 5).

The cache analysis framework presented in this dissertation addresses the six weaknesses outlined above. Given a program consisting of a sequence of loop nests and the configuration of the underlying cache memory, the framework produces the number of cache misses incurred by the program, indicating the program's cache behavior. The framework produces the exact number of cache misses incurred by a sequence of loop nests. The framework models cache memories of any organization, including direct-mapped, set-associative, and fully-associative caches. The framework handles cache memories of virtually any configuration, allowing the counting of both data cache misses and TLB misses. The framework produces cache miss counts for arbitrarily-nested loops. The framework gives the actual cache miss count incurred by a loop nest sequence, and not merely the sum of cache misses incurred by each individual loop nest in isolation. The framework models the data access patterns of a program whether it uses a standard row- or column-major array layout, or a novel nonlinear array layout based on bit interleavings of the binary expansions of the array coordinates. As this document will show, the cache analysis framework of this dissertation does not suffer from any of the weaknesses outline above.

1.3 Thesis Contributions

The following are statements of the major contributions in this dissertation.

- *A new alternative cache miss classification has advantages over traditional cache miss classification schemes.* Unlike traditional miss classification schemes, the alternative miss classification allows combining of the cache behaviors of individual program fragments to obtain the cache behavior of the sequence of such program fragments. Determining the state of the cache at certain points in program execution is critical to accurately combining the cache behavior of the program fragments in the sequence. The alternative miss classification also permits approximation of the number of cache misses incurred by a program fragment sequence with a small error bound, which avoids the computation of cache state.

- *The cache analysis framework of this dissertation models the behavior of loop nests executing in set-associative caches.* The framework produces the exact cache miss count incurred by a loop nest executing in caches of arbitrary associativity. Assuming a least-recently used (LRU) cache replacement policy, a single pass through the method gives cache miss counts for multiple associativity values.

- *The cache analysis framework of this dissertation models the data access patterns of arbitrarily-nested loops using Presburger arithmetic and exploits connections between*

Presburger arithmetic, automata theory, and graph theory to identify cache misses. The framework employs an exact model of the behavior of arbitrarily-nested loops executing in the presence of set-associative caches, expressed in Presburger arithmetic. To determine the number of cache misses incurred by a loop nest, the framework applies automata-theoretic methods for counting and enumerating Presburger formula solutions to the formulas describing cache behavior. The counting method builds on a connection between automata theory and graph theory to efficiently count solutions. The automata-theoretic methods are not specific to cache behavior formulas and are relevant to many other applications in program analysis, such as load balancing.

- *The cache analysis framework of this dissertation is flexible.* The inherent flexibility of the framework derives from the use of Presburger formulas, as any behavior describable in Presburger arithmetic may be modeled in the framework. As a result, the framework can identify cache misses according to either the new alternative miss classification or traditional miss classification. Moreover, the framework handles the row- and column-major array layout functions, as well as nonlinear array layout functions expressible in Presburger arithmetic.

- *The cache analysis framework of this dissertation is a tool for improving program behavior and exploring the space of memory design.* The framework can be used to investigate the effect of changes in the values of code and data transformation parameters. The framework models caches of virtually any configuration, making it well suited to model data cache and translation lookaside buffer (TLB) misses in a wide variety of memory systems.

1.4 Limitations

There are some limitations on the loop nests whose memory behavior may be analyzed by the framework presented in this dissertation, pertaining to control structure and data structure.

Control structure. The framework analyzes sequences of nested count-controlled loops (*i.e.*, loops that execute a specified number of times), and the number of times the loops execute must be known at compile time. For example, consider the three loop nests in Figure 1.3. Loop nest **a** is analyzable because it is clear how many times each of the two loops execute. Loop nest **b** is analyzable only if the value of `MAX` is known at compile time. Loop nest **c** is not analyzable since the number of times the loop executes depends on the values stored in array `A`, which are not known at compile time.

The framework can analyze loop nest **a** in Figure 1.3 because the lower and upper bounds on the loop control variables `i` and `j` are either values known at compile time or are linear expressions of the control variables of the outer loops. The lower bound of loop control variable

```
do i = 0, 1000
   do j = 2*i+5, 1000
      x += A[i,3*j+i]
   enddo
enddo
```

a.

```
do i = 0, MAX
   do j = 2*i+5, MAX
      x += A[i,3*j+i]
   enddo
enddo
```

b.

```
do while x < 1000
   x += A[i]
   i += 1
enddo
```

c.

Figure 1.3: Example loop nests: **a.** analyzable by the framework, **b.** possibly analyzable by the framework, **c.** not analyzable by the framework.

`j` is $2 * \texttt{i} + 5$, which is a linear expression of the control variable of the outer loop `i`. If the lower bound of loop control variable `j` were instead $2 * \texttt{i} * \texttt{i} + 5$, loop **a** would not be analyzable because the expression is not linear in `i`. This restriction on lower and upper loop bounds is due to the underlying polyhedral model [47] of loop nests and the use of Presburger arithmetic to model loop nest behavior.

Also, the framework analyzes loop nests that execute on a uniprocessor.

Data structure. The framework of this dissertation analyzes loop nests that access memory via array references.

The framework can analyze loop nest **a** in Figure 1.3 because the expressions for indexing array `A` are linear expressions of the control variables in the loops containing the array reference. The index expressions are `i` and $3 * \texttt{j} + \texttt{i}$. If one of these expressions were instead $3 * \texttt{i} * \texttt{j}$, loop nest **a** would not be analyzable because the expression is not linear in `i` and `j`. As with lower and upper loop bounds, this restriction on array index expressions is due to the polyhedral model and the use of Presburger arithmetic.

If the array reference in loop nest **a** in Figure 1.3 were instead an indirect reference such as `A[B[i,3*j+i]]`, the framework could not analyze the loop nest, because the values stored in array `B` are not known at compile time.

Given these limitations, the analysis framework presented in this dissertation can analyze loop nests typically found in dense matrix computations, such as linear algebra, Fourier and related transforms, and low-level image processing. The framework cannot analyze loop nests that perform sparse matrix computations, as they tend to have indirect array references. For more on the types of loop nests whose memory behavior may be analyzed by the framework of this dissertation, see Section 4.1.

1.5 Organization

The remainder of this dissertation is organized as follows.

Chapter 2 reviews terminology and background material relevant to this dissertation. In particular, this chapter discusses the underlying models of cache memory, loop nests, and array referencing, which are fundamental to the analytical framework presented here. This chapter also discusses Presburger arithmetic, automata theory, and their connection, on which the framework builds to obtain an accurate model of cache behavior.

Chapter 3 introduces a new alternative classification of cache misses that addresses a shortcoming of traditional miss classification schemes. With traditional miss classification schemes, it is not possible to obtain the exact cache behavior of a sequence of program fragments from the cache behavior of the individual fragments. This chapter shows the role of cache state in producing an accurate cache miss count for a sequence of program fragments and provides an option for approximating the number of cache misses in the sequence with a small error bound.

Chapter 4 discusses how to model data access patterns in Presburger arithmetic. First, this chapter introduces the notions of neighborhood and witness, which serve to identify the situations that cause a cache miss. Then, the chapter shows how to express various types of witnesses as formulas of Presburger arithmetic and gives rules for identifying cache misses based on the existence of such witnesses.

Chapter 5 explains how the analysis framework exploits a fundamental connection between Presburger arithmetic and automata theory to count solutions in Presburger formulas. This chapter illustrates how accepting DFA paths encode the formula solutions and gives algorithms for counting and enumerating such paths. The key to efficiently counting the number of accepting paths in a DFA is to treat the DFA as a directed graph. To reveal the number of cache misses incurred by a loop nest, the framework applies the automata-theoretic methods for counting and enumerating Presburger formula solutions to the formulas describing cache behavior given in Chapter 4.

Chapter 6 provides a high-level view of the analysis framework presented in this dissertation and describes the tools used in the implementation of the framework. The framework employs existing tools to extract relevant loop nest parameters from source code, to simplify Presburger formulas describing cache behavior, and to represent a cache behavior formula as a DFA whose accepting paths recognize the formula's solutions.

Chapter 7 extends the analysis framework presented here to handle nonlinear data layouts expressible in Presburger arithmetic. The analysis framework assumes an LRU cache replacement policy. This chapter also considers the first-in first-out (FIFO) cache replacement policy and gives insights on why it cannot be completely modeled by the framework.

Chapter 8 applies and validates the analysis framework of this dissertation on a variety of example programs. Experiments demonstrate the framework's ability to model exactly the behavior of loop nests executing in set-associative caches by giving accurate cache miss counts for virtually any value of associativity for all example programs. This chapter illustrates the flexibility of the framework by giving cache miss counts for data caches and TLBs and by considering example programs with both canonical and nonlinear data layouts. This chapter also shows the usefulness of the framework in several example loop transformations.

Chapter 9 compares the analysis framework presented here to existing work for memory behavior analysis. In particular, this chapter points out how the analysis framework of this dissertation goes beyond related work by addressing the weaknesses outlined in Section 1.2.

Finally, Chapter 10 concludes the dissertation and provides directions for future work.

Chapter 2

Terminology and Notation

This chapter provides background material and defines notation for the analytical framework presented in this dissertation. Section 2.1 describes the static structure and dynamic behavior of cache memory. Section 2.2 gives the underlying model for loop nests, while Section 2.3 gives the underlying model for referencing array variables in loop nests. Section 2.4 reviews Presburger arithmetic and its worst-case complexity. Section 2.5 discusses the connection between Presburger arithmetic and automata theory.

2.1 Cache Basics

A cache is a small, fast memory located between the processor and main memory designed to capture frequently referenced data [64, 66, 67, 104, 105, 114]. This section reviews the static structure of a cache (Section 2.1.1), the dynamic behavior of a cache (Section 2.1.2), and a special type of cache—the translation lookaside buffer (Section 2.1.3).

2.1.1 Static Cache Structure

From a hardware perspective, a large number of parameters characterize a cache [105]. However, it is standard to characterize the primary organization of a cache with the following three parameters: associativity $\mathcal{A}$, blocksize $\mathcal{B}$, and capacity $\mathcal{C}$ [114]. Capacity, expressed in bytes, is the total amount of data that a cache can hold. The seemingly complex organization of a cache is motivated by the need to quickly locate data. On request of a memory byte address, a search of the entire cache is avoided by quickly narrowing the search to a small portion of the cache. Then, all memory in this limited portion of cache is simultaneously checked to locate the requested memory address. A memory byte address[1] is divided in such a way that this limited portion of the cache is easily identified. Figure 2.1 shows how a memory byte address

[1]This address may be either a physical memory address or a virtual memory address. The framework presented in this dissertation assumes that the data cache is virtually indexed. Fortunately, most operating systems employ page coloring [78], which ensures that the cache set indexes of physical and virtual addresses are identical. Therefore, whether it is a physical address or a virtual address is not of concern.

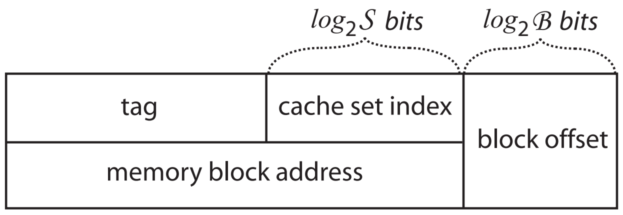


Figure 2.1: The portions of a memory byte address. The index selects the cache set, and the tag disambiguates memory blocks in a set. The block offset locates the desired data within a memory block at the memory block address.

is divided. First, the higher-order bits specify the address of the memory block containing the memory byte address, and the block offset identifies the memory byte address within the memory block. A **memory block** is the unit of mapping main memory to cache. For a cache with blocksize $\mathcal{B}$, each memory block holds $\mathcal{B}$ contiguous bytes of memory. A cache has $\frac{\mathcal{C}}{\mathcal{B}}$ **cache frames**, and each frame either may be occupied by a memory block or may be empty. At no time may two cache frames contain the same memory block. The memory block address is further partitioned, with the higher-order bits specifying a tag and the lower-order bits specifying a cache set index. The cache set index eliminates the need to search the entire cache for a memory address by quickly selecting the cache set (a small portion of the cache) to which a memory block maps. The tag disambiguates memory blocks within the same cache set, and simultaneous checking of the tags for all memory blocks in the cache set rapidly locates the requested memory address or determines its absence from the cache. A **cache set** is the collection of $\mathcal{A}$ frames that a particular memory block may occupy in the cache. A cache set contains one cache frame at each of $\mathcal{A}$ degrees of associativity, and the collection of frames from all sets at each degree of associativity is called a **way**. The number of cache sets, $\mathcal{S}$, is equal to $\frac{\mathcal{C}}{\mathcal{A}\mathcal{B}}$. Figure 2.2 illustrates the cache structure described above. In general, $\mathcal{B}$ and $\mathcal{S}$ are powers of two to allow quick determination of the cache set index and block offset using a bit-level mask (see Figure 2.1).

Of the four cache parameters $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, and $\mathcal{S}$, only three are independent of one another. While associativity $\mathcal{A}$, blocksize $\mathcal{B}$, and capacity $\mathcal{C}$ are considered the main cache parameters, the number of sets $\mathcal{S}$ is critical in mapping data to cache. Notice that the size of the cache set index is $\log_2 \mathcal{S}$. The representation ($\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$; $\mathcal{S}$) denotes a cache with associativity $\mathcal{A}$, blocksize $\mathcal{B}$, capacity $\mathcal{C}$, and $\mathcal{S}$ cache sets. Consistent with the convention that $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ are the main cache parameters, the semicolon signifies that $\mathcal{S}$ is an auxiliary parameter. If $\mathcal{A} = \frac{\mathcal{C}}{\mathcal{B}}$, then $\mathcal{S} = 1$ and a memory block may reside in any frame of the cache; such a cache is designated **fully-associative**. If $\mathcal{A} = 1$, then a memory block must reside in a particular frame; such a cache is designated **direct-mapped** [66]. Figure 2.3 shows five caches with various values of associativity $\mathcal{A}$, fixing capacity $\mathcal{C}$ and blocksize $\mathcal{B}$. Notice how the number of sets $\mathcal{S}$ and the shape of the cache change with the associativity.

	way 0	way 1	$\cdots$	way $\mathcal{A}-1$
set 0	*a cache frame* ($\mathcal{B}$ bytes)	*a cache frame* ($\mathcal{B}$ bytes)	$\cdots$	*a cache frame* ($\mathcal{B}$ bytes)
set 1	*a cache frame* ($\mathcal{B}$ bytes)	*a cache frame* ($\mathcal{B}$ bytes)	$\cdots$	*a cache frame* ($\mathcal{B}$ bytes)
$\vdots$	$\vdots$			$\vdots$
set $\mathcal{S}-1$	*a cache frame* ($\mathcal{B}$ bytes)	*a cache frame* ($\mathcal{B}$ bytes)	$\cdots$	*a cache frame* ($\mathcal{B}$ bytes)

Figure 2.2: Representation of an $\mathcal{A}$-way set-associative cache with blocksize $\mathcal{B}$, $\mathcal{S}$ cache sets, and capacity $\mathcal{C} = \mathcal{A} \cdot \mathcal{B} \cdot \mathcal{S}$.

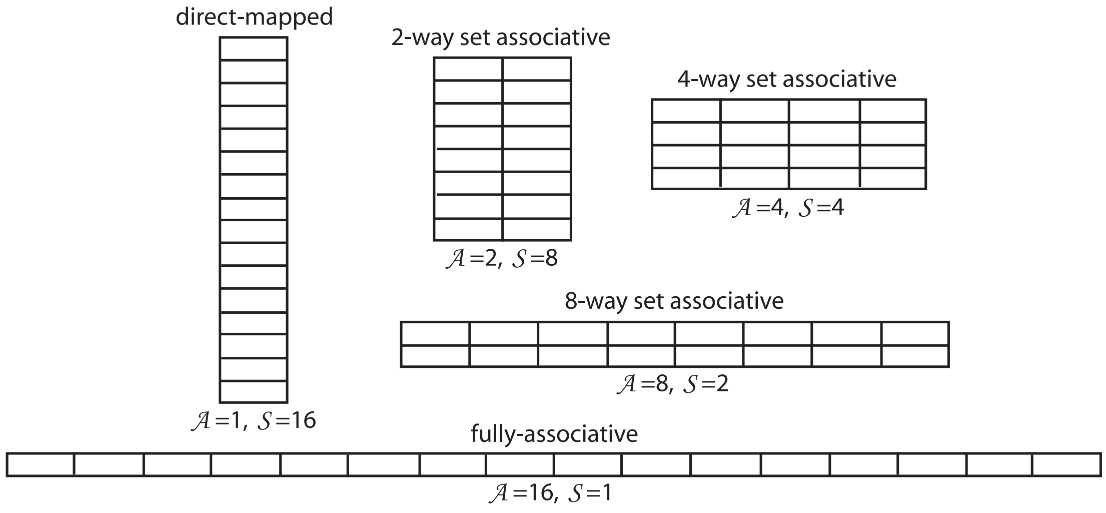

direct-mapped
2-way set associative
4-way set associative
$\mathcal{A}$=4, $\mathcal{S}$=4
$\mathcal{A}$=2, $\mathcal{S}$=8
8-way set associative
$\mathcal{A}$=8, $\mathcal{S}$=2
$\mathcal{A}$=1, $\mathcal{S}$=16
fully-associative
$\mathcal{A}$=16, $\mathcal{S}$=1

Figure 2.3: The different shapes of caches with fixed capacity $\mathcal{C}$ and blocksize $\mathcal{B}$, varying associativity $\mathcal{A}$. Notice how the mapping of data to cache will differ for each shape.

The four cache parameters described above are used to define the mapping[2] of blocks from main memory to cache memory. The *Block* function (with $\mathit{Block}(m) = \lfloor \frac{m}{\mathcal{B}} \rfloor$) associates a memory byte address with a unique memory block address. The *Set* function (with $\mathit{Set}(b) = b \bmod \mathcal{S}$) associates a memory block address with a unique cache set.

For a program executing in a direct-mapped cache, at any point during execution of the program each set of the cache either contains a memory block or is empty. If a program executes in a two-way set-associative cache, each set of the cache contains two distinct memory blocks, one memory block, or is empty at any point during executing of the program. In general,

[2]Other schemes for mapping from main memory to cache have been proposed, such as skewed-associative caches [112, 113] and XOR-based cache placement [63], but the scheme reviewed here is the universally accepted way of mapping blocks of main memory to cache memory in commercial processors.

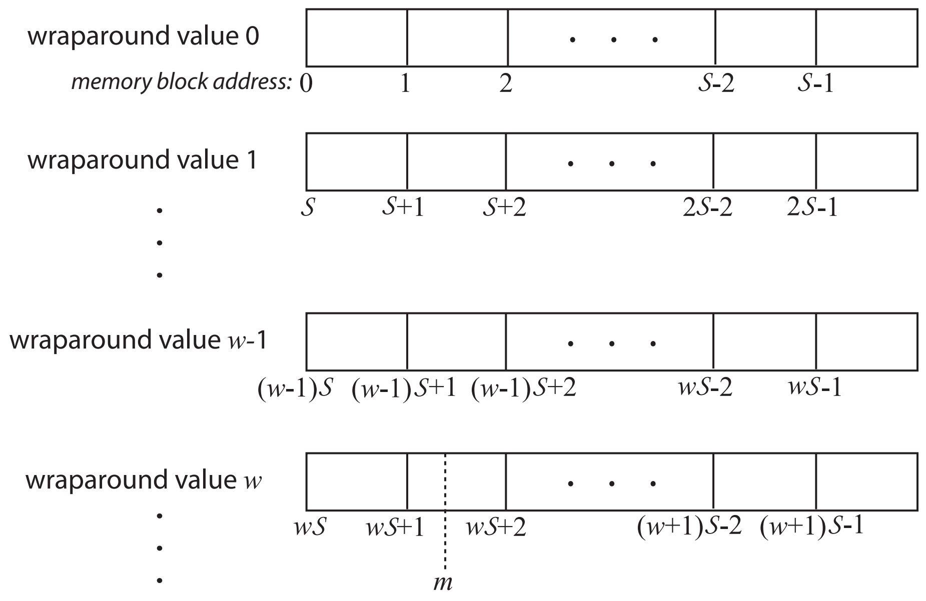

wraparound value 0
memory block address: 0 1 2 S-2 S-1
wraparound value 1
S S+1 S+2 2S-2 2S-1
wraparound value w-1
(w-1)S (w-1)S+1 (w-1)S+2 wS-2 wS-1
wraparound value w
wS wS+1 wS+2 (w+1)S-2 (w+1)S-1
m

Figure 2.4: Wraparound value $w = \mathit{Wrap}(m)$, for memory byte address m. The cache set number $s = 1$ and wraparound value w identify the memory block address for m.

for an $\mathcal{A}$-way set-associative cache, each set of the cache contains one to $\mathcal{A}$ distinct memory blocks or is empty at any point during execution of the program. The **state of the cache** is the collection of memory blocks residing in each cache set at any point during the execution of a program such that the memory blocks in each cache set are ordered by recency of access. Maintaining the order of most recently accessed is convenient for determining which memory blocks are candidates for replacement using the least-recently used (LRU) policy. The state of cache set s, $\mathfrak{C}_s = \{b_1, \ldots, b_n\}$, is the set of memory block addresses resident in cache set s, where $n \in [0, \mathcal{A}]$ is the size of $\mathfrak{C}_s$. The state of cache set s is empty if no memory blocks have yet mapped to cache set s (*i.e.*, $\mathfrak{C}_s = \{\}$ and $n = 0$). The contents of $\mathfrak{C}_s$ have an implicit ordering that indicates the most recently accessed. Memory block address b_i is more recently accessed than b_j if $i < j$. The state of the entire cache is $\mathfrak{C} = \{(s, \mathfrak{C}_s) : 0 \leqslant s < \mathcal{S}\}$. Let $\langle\ \rangle$ be an indexing operator for the state of the entire cache such that $\mathfrak{C}\langle s\rangle = \mathfrak{C}_s$.

Recall from Figure 2.1 that the lower-order bits of a memory block address specify the cache set to which it maps, and the higher-order bits (called the tag) distinguish the memory block from others in the same set. I refer to the tag in a memory address as the **wraparound value**. The *Wrap* function (with $\mathit{Wrap}(m) = \lfloor \mathit{Block}(m)/\mathcal{S} \rfloor$) associates a memory byte address with a unique wraparound value. Figure 2.4 shows the wraparound value w for a memory byte address m, and that all memory byte addresses with a block address equal to $x\mathcal{S}$ up to $(x + 1)\mathcal{S} - 1$ have the same wraparound value x. The intuition of a wraparound value w is that memory

```
$\mathbb{L}_{\mathrm{mm}}$:      do i = 0, $t-1$
            do j = 0, $u-1$
    $S_0$:        c = Z[i,j]
                do k = 0, $v-1$
    $S_1$:            c += X[i,k] * Y[k,j]
                enddo
    $S_2$:        Z[i,j] = c
            enddo
        enddo
```

Figure 2.5: The running example loop nest $\mathbb{L}_{\mathrm{mm}}$.

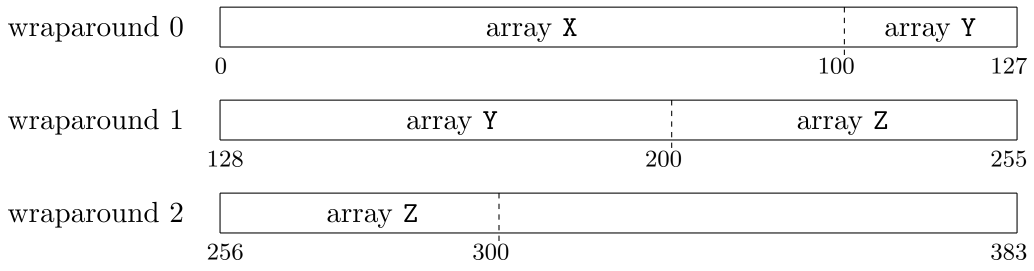


Figure 2.6: Wraparound values for the memory locations accessed by loop nest $\mathbb{L}_{\mathrm{mm}}$.

"wraps around" the cache w times before m maps to set s. Notice that each cache set number and wraparound value pair identifies a unique memory block address, and each memory block address has a unique set number and wraparound value pair. Thus, it is a two-dimensional representation of a memory location (set number, wraparound value), which is an alternative to the traditional one-dimensional representation (memory byte address). In modeling the data access patterns of programs, the framework of this dissertation considers the behavior in each cache set individually and uses wraparound values to specify memory blocks.

In order to demonstrate the concept of a wraparound, consider the loop nest in Figure 2.5, which is the running example used for illustration throughout this document. Loop nest $\mathbb{L}_{\mathrm{mm}}$, which is normalized[3] and language-neutral, performs the matrix multiplication $Z_{t\times u} = X_{t\times v} \cdot Y_{v\times u}$. Suppose that $t = u = v = 20$ and that all arrays contain double-precision elements of 8 bytes each. For an ($\mathcal{A}$, 32, $4096 \cdot \mathcal{A}$; 128) cache, Figure 2.6 shows one way of storing the three arrays in memory. Elements of array X have memory block addresses ranging from 0 to 99 and occupy memory locations with a wraparound value of 0. Some elements in array Y occupy memory locations with a wraparound value of 0, and other elements are in memory locations with a wraparound value of 1. Similarly, some elements of array Z occupy memory locations with a wraparound value of 1, and others with a wraparound value of 2.

[3]In a normalized loop nest, all loops have step size equal to one [135].

Note that the wraparound value of a memory address does not have to be equivalent to its tag. As Chapter 4 discusses, the analytical framework presented in this dissertation uses the distinctness of wraparounds, and not their actual values, to model cache behavior. Therefore, it is always valid to change the starting addresses of all referenced arrays by some multiple of $\mathcal{B}{\cdot}\mathcal{S}$, in effect changing the actual values of all wraparounds but not their distinctness. Suppose that for the running example loop nest $\mathbb{L}_{\mathrm{mm}}$ (with $t = u = v = 20$, 8-byte array elements, and an ($\mathcal{A}$, 32, $4096 \cdot \mathcal{A}$; 128) cache), the starting address of array `X` in memory is 12288, the starting address of array `Y` is 15488, and the starting address of array `Y` is 18688. All elements of array `X` have a wraparound value of 3. Elements of array `Y` have either a wraparound value of 3 or a wraparound value of 4. Elements of array `Z` have either a wraparound value of 4 or a wraparound value of 5. Subtracting $12288 = 3 \cdot 32 \cdot 128$ from each starting array address gives the same wraparound values as in Figure 2.6. Thus, loop nest $\mathbb{L}_{\mathrm{mm}}$ with starting addresses 12288, 15488, and 18688 has the the same cache miss behavior as with the original starting array addresses of 0, 3200, and 6400.

2.1.2 Dynamic Cache Behavior

When the processor requests data at a memory address m, a **cache hit** occurs if $\mathit{Block}(m) \in \mathfrak{C}\langle \mathit{Set}(\mathit{Block}(m))\rangle$. The cache satisfies the memory request after the hit time (*i.e.*, the time required to fetch the requested data). A **cache miss** occurs if $\mathit{Block}(m) \notin \mathfrak{C}\langle \mathit{Set}(\mathit{Block}(m))\rangle$. During a cache miss, the lower levels of memory satisfy the memory request and place memory block $\mathit{Block}(m)$ in cache set $s = \mathit{Set}(\mathit{Block}(m))$, after the miss penalty (see equation (1.3)). A **replacement algorithm** selects which cache frame in set s to update with the new memory block. Ideally, an algorithm should select the cache frame containing the memory block that will be unused for the longest time in the future; this is known as the optimal replacement algorithm [13]. Because the optimal replacement algorithm requires perfect knowledge of the future, it is not implementable in hardware. However, it does serve to gauge the effectiveness of other replacement algorithms. **Least-recently used** (LRU), **first-in first-out** (FIFO), and **random** are three popular replacement algorithms [64]. The LRU algorithm chooses to replace the memory block in set s that has been unused for the longest time. The FIFO algorithm chooses to replace the memory block that has resided in set s for the longest time. The random algorithm chooses randomly among the memory blocks currently residing in set s. Note that for direct-mapped caches, a replacement algorithm is not necessary since each set holds only one memory block. The analytical framework presented in this dissertation assumes a LRU replacement policy.[4] Section 7.2 discusses why the framework does not assume a FIFO

[4]As the cache associativity increases, the number of bits required to implement the LRU algorithm grows exponentially in $\mathcal{A}$ [64]. As a result, LRU replacement is most often approximated (called pseudo-LRU) [64]. The framework of this dissertation assumes true LRU replacement. The possibility of a slight mismatch in a program's cache behavior with true LRU and its behavior with pseudo-LRU is beyond the scope of this dissertation.

replacement policy.

Assuming an LRU replacement policy, the state of the cache changes after a cache hit or miss to preserve the most-recently-accessed ordering and to update the contents of cache state (in the case of a miss).

Definition 2.1 For a memory access a that touches memory block b in cache set s, $\mathfrak{C}\langle s\rangle = \{b_1, \ldots, b_n\}$ becomes $\mathfrak{C}'\langle s\rangle = \mathit{UpdateState}(\mathfrak{C}\langle s\rangle, \mathsf{a})$ where

$$\mathit{UpdateState}(\mathfrak{C}\langle s\rangle, \mathsf{a}) = \begin{cases} \{b, b_1, \ldots, b_{i-1}, b_{i+1}, \ldots, b_n\} & \text{if } b = b_i \text{ hits,} \\ \{b, b_1, \ldots, b_n\} & \text{if } b \text{ misses and } n < \mathcal{A}, \\ \{b, b_1, \ldots, b_{n-1}\} & \text{if } b \text{ misses and } n = \mathcal{A}. \end{cases} \tag{2.1}$$

Traditional cache miss categories are helpful in capturing the cause of a cache miss. A **compulsory miss** (or cold miss) occurs on the first access to a memory block because the block cannot already be resident in cache. Compulsory misses cannot be avoided, except with prefetching. A **capacity miss** occurs when accessing a memory block that was previously in cache but was replaced because the cache could not contain all of the memory blocks needed to execute a program. Capacity misses are due to the restricted size of the cache. A **conflict miss** occurs when accessing a memory block that was previously in a cache set but was replaced because too many other memory blocks mapped to the same cache set. Conflict misses are due to too many memory blocks mapping to a fraction of the cache. One way to compare capacity and conflict misses is to consider associativity. Capacity misses occur independent of the associativity of a cache (*i.e.*, a capacity miss in an LRU set-associative cache with capacity $\mathcal{C}$ also misses in an LRU fully-associative cache with capacity $\mathcal{C}$). Conflict misses depend on associativity (*i.e.*, a conflict miss in an LRU set-associative cache with capacity $\mathcal{C}$ hits in an LRU fully-associative cache with capacity $\mathcal{C}$). This classification of cache misses is known as the 3C model [67]. Because both capacity and conflict misses describe accesses to memory blocks that have resided in cache but were replaced, they are often grouped together as **replacement misses**. I refer to this classification of cache misses as the compulsory-replacement miss classification.

2.1.3 A Special Cache—the TLB

The cache described above is a data cache, storing the contents of the most frequently referenced blocks of main memory close to the processor. This work also considers another type of cache—the translation lookaside buffer (TLB). The TLB is a cache of memory address translations. TLB miss penalties are often larger than data cache miss penalties, contributing significantly to poor program performance.

Many programs operate on more data than main memory can hold, with the remaining data contained in secondary storage (*i.e.*, hard disk). Virtual memory is a technique for automatically managing these two levels of memory so that processes, such as an executing program, need not be aware of whether the required data is in main memory or in secondary storage [66]. When a program needs data, the processor produces a virtual memory address for the data. The address space of virtual memory is divided into pages, and each virtual address is split into a virtual page number and an offset within the page. A page table gives the physical page number corresponding to a virtual page number, which combines with the page offset to give the complete physical address. The physical memory address indicates the actual memory location of the requested data.

The relationship between main memory and disk memory is similar to the relationship between cache and main memory. While a memory block is the unit of mapping to cache, the unit of mapping to main memory is a page. A memory block not found in the cache incurs a cache miss, while a page not found in main memory incurs a page fault. The critical difference in the two relationships is the placement of memory blocks in cache and the placement of pages in main memory. A page can be placed anywhere in main memory, and the mapping of pages to main memory can change over time. A page table is necessary to keep track of the translations from the virtual addresses of pages in main memory to their physical addresses in disk memory. The placement of a memory block into a particular set of the cache is well defined by the memory byte address, and there is no need for a table to keep track of the mappings from main memory to cache.

Translation from a virtual address to the corresponding physical address is necessary on every access to memory. Because page tables are usually too large to fit in cache, the overhead of address translation is large. To avoid address translation on every memory access, recently-performed address translations are held in a special cache—the TLB. The TLB is a hardware table containing cross-references between the virtual and physical addresses of recently referenced pages of memory. If the physical page number of a desired page is not in the TLB, a delay is incurred while the physical page number is determined by a page table lookup. Thus, the TLB is a special cache that captures the most frequently referenced items (namely, the mapping from virtual memory addresses to physical memory addresses), where the cache index is based on the virtual address. TLBs are highly-associative and page sizes are large (typically 4096 bytes and larger), and this combination makes TLBs difficult to model analytically. A TLB miss occurs when the physical memory page number of accessed data is not found in the TLB, and the penalty for such a miss can be hundreds of processor cycles or more. The analysis framework of this dissertation assumes that the instruction and data TLBs are distinct, and the framework is used to determine the data TLB misses incurred by a program.

2.2 Modeling Loop Nests

The analytical framework of this dissertation focuses on **count-controlled loops** (*i.e.*, loops that execute a specified number of times) because the number of loop iterations is known at

```
do i = $x$ to $y$ by $z$                do i = $\lfloor x/z \rfloor$ to $\lfloor y/z \rfloor$ by 1
   c = A[$f$(i)]                           c = A[$f(z \cdot$i $+\ x \bmod z)$]
enddo                                   enddo

       original loop                         normalized loop
```

Figure 2.7: The normalized loop is a semantically-equivalent version of the original loop, transformed to make step size equal to one.

compile time. In a count-controlled loop, there is a program variable that is set to an initial value and incremented/decremented by a constant amount until reaching a specified value limit. Such a program variable is known as a **loop control variable** (LCV).

It is sufficient in modeling loop execution to restrict analysis to **normalized** loops, in which the step size is equal to one [135]. Normalization simplifies the analysis of loops, but with no loss of information, since transforming a loop to get a step size of one preserves the semantics of the original loop (see Figure 2.7). The initial value of the LCV is its lower bound, and the LCV is incremented by one until reaching the specified value limit that is its upper bound. The initial and limit values of a LCV, being characteristic of all count-controlled loops, are easy to identify. For the outermost loop in the running example loop nest $\mathbb{L}_{\text{mm}}$ in Figure 2.5 on page 17 with $t = 20$, the LCV is `i` with lower bound 0 and upper bound 19.

A **loop nest** is a loop containing one or more other loops. The model of nested loops given here is based on the well-known polyhedral model [47]. The depth, d, of the loop nest indicates the number of nesting levels, numbered 0 for the outermost loop to $d-1$ for the innermost loop. The LCV of the loop at level j, ι_j, has lower bound L_j and upper bound U_j. For each LCV ι_j, L_j and U_j must be affine functions of ι_0 to ι_{j-1}. This is a restriction of the polyhedral model and is common among work for analyzing memory behavior. The set of all valid combinations of LCV values for a loop nest is its **iteration space** $\mathcal{I}$, and $\iota = [\iota_0, \ldots, \iota_{d-1}]^{\mathrm{T}}$ denotes a point in the iteration space (known as an **iteration point**). For loop nests, the relationship of iteration point ι executing before iteration point κ, denoted $\iota \prec \kappa$, is equivalent to the lexicographical ordering of iteration points. Consider again the running example loop nest $\mathbb{L}_{\text{mm}}$ in Figure 2.5 with $t = u = v = 20$. Loop nest $\mathbb{L}_{\text{mm}}$ has depth $d = 3$. The LCVs are $\iota_0 =$ `i` with $L_0 = 0$ and $U_0 = 19$, $\iota_1 =$ `j` with $L_1 = 0$ and $U_1 = 19$, and $\iota_2 =$ `k` with $L_2 = 0$ and $U_2 = 19$. For loop nest $\mathbb{L}_{\text{mm}}$, $[0, 0, 19]^{\mathrm{T}} \prec [0, 1, 0]^{\mathrm{T}}$ expresses the temporal ordering relationship of two example iteration points.

The statements in a loop nest are numbered according to the order in which they are executed by the loop nest. If the innermost loop contains each statement S_h in a loop nest, the loop nest is **perfect**. Otherwise, the loop nest is **imperfect**. It is desirable to model the cache behavior of both perfect and imperfect loop nests, but it is cumbersome to have the modeling technique cope with two different kinds of loop nests. Preprocessing imperfect loop nests into semantically-equivalent perfect loop nests eliminates the need to handle two different cases. Embedding algorithms [2, 3, 76, 77, 86] make it possible to transform an imperfect loop nest into a perfect loop nest. The transformations introduce **guards** on statements, which map statements from their places outside the innermost loop to valid places inside the innermost loop. The guard $G_h(\iota)$ evaluates to *true* if statement S_h executes at iteration point ι, and evaluates to *false* otherwise. Loop nest $\mathbb{L}_{\text{mm}}$ in Figure 2.5 has three statements. Because statements S_0 and S_2 are outside of the innermost loop, loop nest $\mathbb{L}_{\text{mm}}$ is imperfect. Guards $G_0([\mathtt{i},\mathtt{j},\mathtt{k}]^{\mathrm{T}}) = (\mathtt{k} = 0)$ and $G_2([\mathtt{i},\mathtt{j},\mathtt{k}]^{\mathrm{T}}) = (\mathtt{k} = 19)$ allow movement of statements S_0 and S_2 to the innermost loop and then execution only when the appropriate guard is true.

2.3 Modeling Array References

In modeling references to memory, array variables are the primary concern, since referencing arrays accounts for most of the accesses to cache. Operations on scalar variables have negligible impact on average memory access time, as they are typically resident in a register. Modeling references to scalar variables is possible, simply by considering them as one-dimensional arrays of length one.

Using the terminology of Ghosh *et al.* [62], a **reference** is the static instance of an array variable read or write in the source code of a program, while a particular execution of that read or write at runtime is an **access**. Recall the running example loop nest $\mathbb{L}_{\text{mm}}$ in Figure 2.5 on page 17, which makes four references to array variables: R_0 = `Z[i,j]` (the read), R_1 = `X[i,k]`, R_2 = `Y[k,j]`, and R_3 = `Z[i,j]` (the write). The references in each statement are ordered according to the order of operations on their corresponding array variables, and given the ordering among statements, this defines the total ordering of references. An iteration point and reference pair uniquely defines an access. Composing the lexicographical ordering among iteration points and the ordering among references specifies a total order among accesses, denoted as "precedes" and written $\lhd$.

Definition 2.2 Access (ι, R_u) **precedes** access $(\iota', R_{u'})$ if the iteration given by ι occurs before the iteration given by ι', or if ι and ι' specify the same iteration and R_u comes before $R_{u'}$ in the total ordering of references. The precedes relationship is written $(\iota, R_u) \lhd (\iota', R_{u'})$.

For loop nest $\mathbb{L}_{\text{mm}}$ with $t = u = v = 20$, $([0,0,19]^{\mathrm{T}}, R_3) \lhd ([0,1,0]^{\mathrm{T}}, R_0)$ expresses the temporal ordering relationship of two example accesses.

A reference R_u has three components: the name of the array variable referenced, some array $Y^{(j)}$; the indexing function of the reference, E_u; and the statement containing the reference, some statement S_h.[5] The index expression of reference R_u is $E_u \cdot \iota$, at iteration ι. The running example loop nest $\mathbb{L}_{\text{mm}}$ in Figure 2.5 operates on three arrays: $Y^{(0)} = \mathtt{X}$, $Y^{(1)} = \mathtt{Y}$, and $Y^{(2)} =$

[5]It would be straightforward to include a fourth component indicating whether the reference is a read or write. Because the analytical framework of this dissertation treats memory reads and writes the same, knowing the read/write status of a reference is unnecessary.

`Z`, and has four references: $R_0 = (Y^{(2)}, E_0, S_0)$, $R_1 = (Y^{(0)}, E_1, S_1)$, $R_2 = (Y^{(1)}, E_2, S_1)$, and $R_3 = (Y^{(2)}, E_3, S_2)$. Given an iteration point, the index expression determines the exact array element accessed. The index expression for a reference at nesting level k must be an affine function of the LCVs $\iota_0 \ldots \iota_k$, which is an inherent restriction of the polyhedral model [47]. For the four references in loop nest $\mathbb{L}_{\text{mm}}$:

$$E_0 = E_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, E_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \text{and } E_2 = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}.$$

In order to specify the memory addresses of the array elements accessed by a loop nest, it is necessary to capture properties of each array $Y^{(j)}$. The dimensionality of array $Y^{(j)}$ is d_j, and its length in the $(k+1)^{\text{th}}$ dimension is ℓ_k. Byte address μ_j denotes the location in memory where the storage of array $Y^{(j)}$ begins, and the number of bytes required to store each of its elements is β_j. For the running example loop nest $\mathbb{L}_{\text{mm}}$ in Figure 2.5, suppose that each array element is double-precision and requires eight bytes (*i.e.*, β_0=β_1=β_2=8). Then, each array requires 3200 bytes of storage. In order to make the arrays contiguous and adjacent in memory, suppose the starting addresses are $\mu_0 = 0$, $\mu_1 = 3200$, and $\mu_2 = 6400$. The layout function $\mathcal{L}_j$ associated with array $Y^{(j)}$ stipulates the manner in which an array element maps to its place in memory. The two canonical array layouts [37] are row-major and column-major, storing an array by rows and columns, respectively. For a reference $R_u = (Y^{(j)}, E_u, S_h)$ and an iteration point ι, let $E_u(\iota) = [e_0, \ldots, e_{d_j-1}]^{\mathrm{T}}$. The two canonical array layout functions are as follows.

$$\text{Row-maj}(E_u(\iota)) = \sum_{p=0}^{d_j-2} \Big(\prod_{q=p+1}^{d_j-1} \ell_q \Big) e_p + e_{d_j-1} \tag{2.2}$$

$$\text{Col-maj}(E_u(\iota)) = e_0 + \sum_{p=1}^{d_j-1} \Big(\prod_{q=0}^{p-1} \ell_q \Big) e_p \tag{2.3}$$

The element of array $Y^{(j)}$ accessed by reference R_u at iteration point ι has memory byte address $m = \mu_j + \mathcal{L}_j(E_u(\iota)) \cdot \beta_j$. In other words, m is the memory byte address touched by access $\mathsf{a} = (\iota, R_u)$. The *Mem* function associates an access with the unique memory byte address that it touches. For the running example loop nest $\mathbb{L}_{\text{mm}}$ in Figure 2.5, access $\mathsf{a} = ([0,5,12]^{\mathrm{T}}, R_2)$ maps to memory byte address 4096 in the following way:

$$\begin{aligned} \mathit{Mem}(([0,5,12]^{\mathrm{T}}, R_2)) &= \mu_1 + \mathcal{L}_1(E_2([0,5,12]^{\mathrm{T}})) \cdot \beta_1 \\ &= 3200 + \text{Col-maj}(\texttt{Y[12,5]}) \cdot 8 \\ &= 3200 + (12 + 5 \cdot 20) \cdot 8 \\ &= 4096. \end{aligned}$$

2.4 Presburger Arithmetic

The analytical framework presented in this dissertation uses Presburger arithmetic to model data access patterns and describe events such as cache misses. **Presburger arithmetic** is the first-order theory of integers with addition. A Presburger formula consists of affine equality and/or inequality constraints connected via the logical operators $\neg$ (not), $\wedge$ (and), $\vee$ (or), and the quantifiers $\forall$ (for all) and $\exists$ (there exists). A formal grammar for Presburger arithmetic (in Backus-Naur form) is as follows:

$$\begin{aligned}
\langle \textit{formula} \rangle \quad ::= \quad & (\langle \textit{formula} \rangle) \mid \langle \textit{formula} \rangle \wedge \langle \textit{formula} \rangle \mid \langle \textit{formula} \rangle \vee \langle \textit{formula} \rangle \mid \\
& \neg \langle \textit{formula} \rangle \mid \exists \textit{var} : \langle \textit{formula} \rangle \mid \forall \textit{var} : \langle \textit{formula} \rangle \mid \\
& \langle \textit{expression} \rangle \langle \textit{rel-op} \rangle \langle \textit{expression} \rangle
\end{aligned}$$

$$\begin{aligned}
\langle \textit{expression} \rangle \quad ::= \quad & (\langle \textit{expression} \rangle) \mid \langle \textit{expression} \rangle + \langle \textit{expression} \rangle \mid \; - \langle \textit{expression} \rangle \mid \\
& \textit{integer} * \langle \textit{expression} \rangle \mid \textit{var} \mid \textit{integer}
\end{aligned}$$

$$\langle \textit{rel-op} \rangle \quad ::= \quad < \mid > \mid \leqslant \mid \geqslant \mid =$$

To improve the readability of Presburger formulas included in this dissertation, I will use certain obvious shortcuts in writing relationships among expressions such as multiple inequalities (*e.g.*, $0 \leqslant x \wedge x < 100$ is written as $0 \leqslant x < 100$).

For example, the following Presburger formula describes a valid iteration point for the running example loop nest $\mathbb{L}_{\text{mm}}$ in Figure 2.5 on page 17:

$$P(i, j, k; t, u, v) = 0 \leqslant i < t \wedge 0 \leqslant j < u \wedge 0 \leqslant k < v$$

with free integer variables i, j, and k (denoting the LCVs `i`, `j`, and `k`) and symbolic constants t, u, and v (denoting the loop upper bounds). The variables t, u, and v are considered symbolic constants because their values are fixed. From the perspective of Presburger arithmetic, i, j, k, t, u, and v are all variables. However, from the perspective of a loop nest, variables i, j, and k are very different from variables t, u, and v. The semicolon in $P(i, j, k; t, u, v)$ denotes this separation of variables. Given loop nest $\mathbb{L}_{\text{mm}}$ with $t = u = v = 20$, the Presburger formula is

$$P(i, j, k) = 0 \leqslant i < 20 \wedge 0 \leqslant j < 20 \wedge 0 \leqslant k < 20$$

with no symbolic constants. A solution to a Presburger formula is any of the values for each free variable that satisfies the formula. For the example Presburger formula $P(i, j, k)$, $i = 9$, $j = 0$, and $k = 5$ is one solution.

In 1929, Presburger [102, 103, 117] proved that the first-order theory of integers with addition is complete (*i.e.*, every such formula or its negation is true) and decidable. In proving completeness, Presburger provided an algorithm which decides the truth of any given sentence in Presburger arithmetic. Presburger's algorithm is based on quantifier elimination [82]. Quantifier elimination repeatedly transforms a sentence[6] with quantifiers ($\forall$, $\exists$) into an equivalent sentence containing fewer quantifiers until there are no quantifiers left. Any equivalent sentence containing fewer quantifiers is considered a simplification of the original sentence. This technique assumes that the truth of a quantifier-free sentence is easy to determine (*e.g.*, $7 < 3 + 5$ is true). Therefore, the truth value of the resulting quantifier-free sentence is the truth value of the original sentence in Presburger arithmetic.

In 1972, Cooper [41] presented a more efficient algorithm for deciding the truth of Presburger formulas, which is also based on quantifier elimination. Oppen [99] proved that the upper bound on the size of the sentence produced by Cooper's algorithm (after elimination of all variables) and the number of computational steps required is $2^{2^{2^{pn}}}$ for some constant $p > 1$, where n is the length of the original sentence. Reddy and Loveland [107] improved Cooper's quantifier elimination method for classes of Presburger arithmetic restricted to a bounded number of alternations of quantifiers. Their improvement yields an upper bound on the complexity of decidability that is one exponent less than that of full Presburger arithmetic. Similarly, Weispfenning [129] showed that by weakening the concept of quantifier elimination slightly to bounded quantifier elimination, the upper and lower bound for quantifier elimination in Presburger arithmetic can be lowered by exactly one exponential. The complexity of quantifier elimination is related to the number of alternating blocks of $\forall$ and $\exists$ quantifiers in the sentence and to the numerical values and coprimality relationships of the integer constants in the sentence [111].

Fischer and Rabin [52] proved that the cost for every algorithm which decides the truth of Presburger sentences is at least $2^{2^{cn}}$ for some constant c and a sufficiently large sentence length n (*i.e.*, there exists n_0 such that this is true for all $n > n_0$). The upper bound for deciding Presburger arithmetic is established to be one exponent lower than Oppen's bound, using a result from Ferrante and Rackoff [49].

For the purposes of this dissertation, the main points are twofold: that it is possible to eliminate the quantifiers in, and decide the truth of any Presburger formula; but that the complexity is superexponential in the worst case. Considering the undesirable worst-case complexity for Presburger arithmetic, the framework of this dissertation exploits a connection between Presburger arithmetic and automata theory to both count and enumerate the solutions to a Presburger formula, rather than coping with the formula directly. The next section reviews the connection between Presburger arithmetic and automata theory.

[6]A sentence in Presburger arithmetic is a formula that has no free variables.

2.5 Automata Theory

There is a fundamental connection between Presburger arithmetic and automata theory, namely, that there exists a deterministic finite automaton (DFA) recognizing the positional binary representation of the solutions of any Presburger formula.[7] Following standard terminology, a DFA M is a 5-tuple $(S, \Sigma,\ \delta,\ q_0,\ F)$, where

S is a finite set of states,
Σ is a finite set of symbols called the alphabet,
$\delta : S \times \Sigma \rightarrow S$ is the transition function,
$q_0 \in S$ is the start state,
$F \subseteq S$ is a set of final states.

The Presburger-DFA connection is perhaps not surprising, given that DFAs can describe arithmetic on the binary representation of natural numbers. The key point to remember in transitioning from Presburger arithmetic to DFAs is the move from a domain of values (natural numbers) to a domain of representations (positional binary encoding).

Büchi [26, 27] originally proved that a subset of $(\{0, 1\}^n)^*$ is recognizable by a finite state automaton if and only if it is definable in WS1S (Weak Second-order Theory of One Successor). Boudet and Comon [23] build on this result. Because Presburger arithmetic can be embedded in WS1S, there is a DFA recognizing the solutions of a Presburger formula. Boudet and Comon formalize the connection between Presburger arithmetic and automata in the following claim.

Claim 2.1 ***(Boudet and Comon)*** *Let $\phi \equiv Q_n x_n, \ldots, Q_1 x_1 \psi$ be a formula of Presburger arithmetic where quantifier Q_i is either $\exists$ or $\forall$ and ψ is an unquantified formula with variables $x_1, \ldots, x_n, y_1, \ldots, y_m$. There is a deterministic and complete automaton recognizing the solutions of ϕ with at most $O(2^{2^{2^{|\phi|}}})$ states.*

The proof of the claim is constructive, with the construction procedure defined by induction on the structure of ϕ. The base cases are linear equalities and inequalities, for which DFA recognizers are easy to construct [9, 10, 23]. Logical connectives of subformulas utilize closure properties of regular sets under intersection, union, and complementation [68]. Existential quantification is handled by projecting the alphabet and the transition function (producing a nondeterministic finite automaton) followed by determinization and state minimization. Universal quantification exploits the tautology $\forall x \phi \equiv \neg \exists x \neg \phi$.

Wolper and Boigelot [137] explain that Boudet and Comon's proof of Claim 2.1 is incorrect, although the result itself may not be false. Klaedtke [80] rigorously proves that there is a triple

[7]The Presburger formula may have either a finite or an infinite number of solutions.

exponential upper bound on the size of the minimal deterministic word automaton (DWA)[8] for a formula in Presburger arithmetic. The following theorem is the formal statement of the result.

Theorem 2.1 ***(Klaedtke)*** *The size of the minimal DWA for a formula of length n is at most $2^{2^{2^{O(n)}}}$.*

The upper bound on automata size given by Klaedtke applies when the first letter of an integer word solution recognized by an automaton is the most significant bit (MSB). When the first letter of an integer word solution is the least significant bit (LSB), automata size may be exponentially smaller than that of the corresponding MSB-first encoding [80]. The converse, that the automata size for an LSB-first encoding may be exponentially larger than for the corresponding MSB-first encoding, has not been shown. The precise relationship of MSB-first and LSB-first encodings for automata representations of Presburger formulas is unknown, and it is not clear which encoding works best in practice.

Independent of whether the encoding is MSB-first or LSB-first, the cost of constructing automata recognizing Presburger formula solutions is superexponential in the worst case. Therefore, converting Presburger formulas to DFAs does not circumvent the difficulty of identifying solutions to the formulas, as both the complexity of DFA construction and the complexity of Presburger formula decidability are superexponential in the worst case. The large worst-case complexity is better understood in the context of DFAs. In the translation of Presburger formulas to DFAs, this complexity manifests in the number of states in the resulting DFA. Specifically, it is the translation of universal quantifiers and negation that causes the exponential blowup of the automaton [23]. The framework of this dissertation builds on the Presburger-DFA connection, constructing DFA representations of cache behavior formulas and counting (or enumerating) the accepting DFA paths to count (or identify) the formula solutions, as Chapter 5 explains. The potentially large cost of DFA construction does motivate simplifying the cache behavior formula as much as possible before constructing a DFA recognizing the formula solutions, which the framework does.

2.6 Summary

Four parameters specify the organization of cache memory and the mapping of data from main memory to cache: associativity $\mathcal{A}$, blocksize $\mathcal{B}$, capacity $\mathcal{C}$, and the number of sets $\mathcal{S}$. As such, these parameters play a prominent role in modeling cache behavior. The LRU replacement algorithm for determining how to update a cache set on a miss is also reflected in how the framework models cache behavior. Cache state $\mathfrak{C}$ specifies the contents of each cache set at

[8]Klaedtke refers to the DFA recognizing the solutions of a Presburger formula as a DWA because the integer solutions are represented as words.

any point during the execution of a program, which is a key component of the alternative cache miss classification presented in Chapter 3.

The underlying models of loop nests and referencing arrays in loop nests capture information about the array elements accessed by a loop nest and the order in which they are accessed. Preprocessing imperfect loop nests into semantically-equivalent perfect loop nests enables the analysis framework to model the cache behavior of arbitrarily-nested loops.

The framework presented in this dissertation uses Presburger arithmetic, the first-order theory of integers with addition, to express the data access patterns of a loop nest and to describe the cache misses incurred by a loop nest. The worst-case complexity of Presburger arithmetic is superexponential, motivating an approach for dealing with Presburger formula solutions indirectly. The analysis framework of this dissertation builds on fundamental connection between Presburger arithmetic and automata theory to determine the solutions to cache behavior formulas. The worst-case complexity of automata construction is the same as for Presburger arithmetic, but the complexity is better understood and manageable in the context of DFAs.

Chapter 3

A New Classification of Cache Misses

Decomposing a program into fragments and considering each fragment individually is critical for managing the difficulty of analyzing the program's cache behavior. This strategy for simplifying the analysis results in no loss of information if there is a way to combine the cache behaviors of individual fragments meaningfully to obtain the cache behavior of the program. Section 3.1 reviews two existing cache miss classification schemes and demonstrates their insufficiency in combining the cache behaviors of program fragments to obtain the cache behavior of their composition. Section 3.2 introduces a new classification of cache misses that makes it possible to achieve a meaningful combination of the cache behaviors of program fragments. Section 3.3 illustrates the role of cache state in combining the cache behaviors of program fragments. Note that what constitutes a program fragment is purposely vague in this chapter, since combining the cache behaviors of program fragments does not depend on the exact nature of the fragments.

3.1 Composability

When using static analysis to count the number of cache misses incurred by a program, it is simpler to consider each fragment of the program individually than to consider the program as a whole. This divide-and-conquer strategy improves the tractability of the analysis by limiting the amount of cache behavior modeled for any one program fragment. Furthermore, if it is possible to relate the cache misses of each distinct program fragment to the cache misses of the composition of these fragments, this simplification suffers no loss of information. For instance, suppose that a program $\mathbb{P}$ consists of two fragments $\mathbb{F}_1$ and $\mathbb{F}_2$, where $\mathbb{F}_{12}$ is their sequential composition (*i.e.*, $\mathbb{P} \equiv \mathbb{F}_{12} \overset{\text{def}}{=} \mathbb{F}_1;\ \mathbb{F}_2$). "Combining" the cache misses of fragments $\mathbb{F}_1$ and $\mathbb{F}_2$ should be equivalent to the cache misses of $\mathbb{F}_{12}$, and therefore, program $\mathbb{P}$. Otherwise, the simplification of considering each program fragment individually does result in loss of information, and the combination of cache misses incurred by each fragment is only an approximation of the misses incurred by the entire program. Whether or not the combination of cache misses in each program fragment is equivalent to the misses in the entire program depends on the scheme for classifying cache misses.

Definition 3.1 Composability is the property of a cache miss classification scheme such that there is an exact way of relating the cache misses of individual program fragments to the cache misses of the entire program.

Two existing cache miss classification schemes lack the property of composability. The well-known 3C model [67] is the traditional cache miss classification scheme, classifying a cache miss into one of three categories: compulsory, capacity, or conflict (see Section 2.1.2). A compulsory miss is incurred on the first access to a memory block and cannot be avoided. A capacity miss is a miss in an LRU set-associative cache with capacity $\mathcal{C}$ that also misses in an LRU fully-associative cache with capacity $\mathcal{C}$. A conflict miss is a miss in an LRU set-associative cache with capacity $\mathcal{C}$ that hits in an LRU fully-associative cache with capacity $\mathcal{C}$. The OPT model [118] for classifying cache misses has the same three categories as the 3C model. However, in classifying capacity and conflict misses, the OPT model compares caches that use the optimal replacement algorithm [13]. Capacity and conflict misses are often grouped together as replacement misses. Because the distinction between the 3C model and the OPT model is in the classification of capacity and conflict misses, the models are identical in their classification of compulsory and replacement misses.

The compulsory-replacement miss classification schemes can be problematic. First, since the cache miss categories of the 3C model and the OPT model are architectural in nature, they are useful in comparing two caches, but that is not the objective of the work in this dissertation. Second, the compulsory-replacement miss classification schemes lack the vital property of composability. To observe how the compulsory-replacement miss classification schemes fall short, suppose that fragment $\mathbb{F}_i$ incurs $\mathbf{C}_i$ compulsory misses and $\mathbf{R}_i$ replacement misses. By definition, a compulsory miss occurs when a program fragment accesses a memory block for the first time, without regard to whether the memory block is in cache when the fragment begins execution. For that reason, there is no simple way to relate the misses of fragments $\mathbb{F}_1$ and $\mathbb{F}_2$ ($\mathbf{C}_1$, $\mathbf{R}_1$, $\mathbf{C}_2$, $\mathbf{R}_2$) to the misses of their composition $\mathbb{F}_{12}$ ($\mathbf{C}_{12}$, $\mathbf{R}_{12}$). Certainly, $\mathbf{C}_{12} + \mathbf{R}_{12} \neq \mathbf{C}_1 + \mathbf{R}_1 + \mathbf{C}_2 + \mathbf{R}_2$, since the compulsory misses in program fragment $\mathbb{F}_2$, $\mathbf{C}_2$, may not even be misses in the composition $\mathbb{F}_{12}$. There are two keys to combining the cache misses of fragments $\mathbb{F}_1$ and $\mathbb{F}_2$ to get the cache misses of their composition $\mathbb{F}_{12}$—cache state (defined in Section 2.1.1) and a new classification of cache misses.

3.2 Interior-Boundary Miss Classification

Given a fragment of a program $\mathbb{P}$ and a cache configuration ($\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$; $\mathcal{S}$), all memory accesses made by the fragment fall into one of three groups: those guaranteed to hit in cache, those guaranteed to miss in cache, and those that could hit or miss depending on the state of the

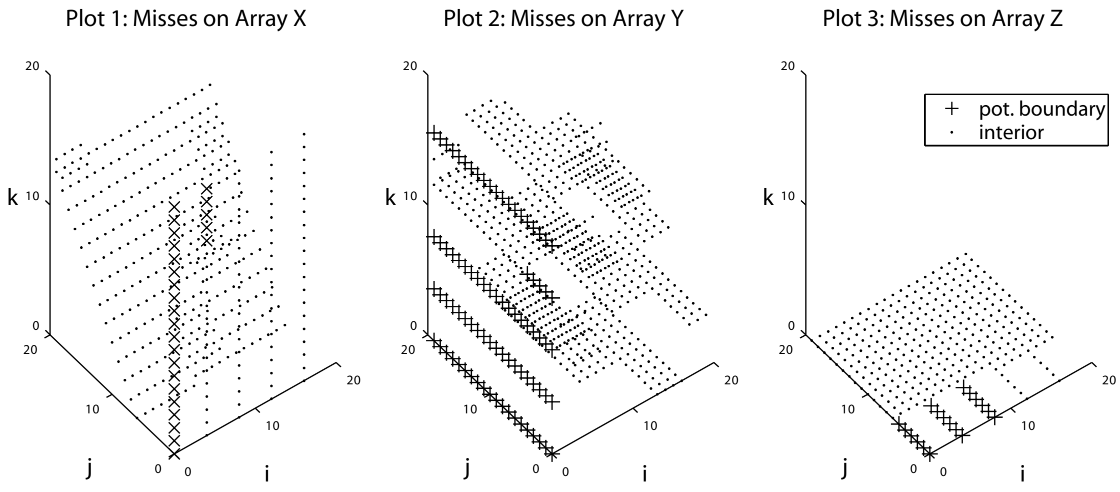


Figure 3.1: 3-D views of the interior and potential boundary misses incurred by loop nest $\mathbb{L}_{\text{mm}}$. Each plot represents the iteration space of $\mathbb{L}_{\text{mm}}$, and a marked iteration point denotes a miss at that point for the corresponding array.

cache at the beginning of program fragment execution. Of course, an access that is guaranteed to hit in cache is simply a cache hit.

Definition 3.2 An **interior miss** is a memory access that is guaranteed to miss in cache, independent of the cache state when a program fragment begins execution.

Definition 3.3 A **potential boundary miss** is a memory access that may hit or miss depending on the cache state when a program fragment begins execution.

Interior misses can be determined by analyzing a program fragment in isolation. The adjective *interior* conveys the assurance of a cache miss despite peripheral factors. Potential boundary misses can be determined by analyzing a program fragment in isolation, but consideration of the cache state is necessary to determine whether a potential boundary miss is actually a cache miss. The adjective *boundary* conveys dependence on matters outside of the program fragment, and *potential* conveys the uncertainty of an actual miss. The method for modeling cache behavior described in Chapter 4 gives a decision procedure for determining which memory accesses are interior misses or potential boundary misses.

Consider the running example loop nest $\mathbb{L}_{\text{mm}}$ of Figure 2.5 on page 17, where $t = u = v = 20$ and arrays $Y^{(0)} = \texttt{X}$, $Y^{(1)} = \texttt{Y}$, and $Y^{(2)} = \texttt{Z}$ are double-precision and linearized in column-major order with starting addresses $\mu_0 = 0$, $\mu_1 = 3200$, and $\mu_2 = 6400$. Figure 3.1 provides three-dimensional views of the interior and potential boundary misses incurred by loop nest $\mathbb{L}_{\text{mm}}$ executing in a (1, 32, 4096; 128) cache. To clearly illustrate the structure of these cache misses, each plot gives the cache misses incurred by a different array of loop nest $\mathbb{L}_{\text{mm}}$. Notice that all of the potential boundary misses incurred by array X are on the boundary of the plot where $\texttt{j} = 0$ (*i.e.*, all such potential boundary misses occur on a first iteration of the `j`-loop). Similarly, all potential boundary misses incurred by array `Y` are on the boundary of the plot where $\texttt{i} = 0$, and all potential boundary misses incurred by array `Z` are on the boundary of the plot where $\texttt{k} = 0$.

Given this interior-boundary classification of cache misses, determination of actual cache misses requires two steps. First, static analysis of a program fragment in isolation identifies the memory accesses that are interior misses and those that are potential boundary misses. Second, using the state of the cache at the beginning of program fragment execution to resolve potential boundary misses identifies such accesses that actually do miss in cache. Resolving a potential boundary miss with the cache state at the beginning of program fragment execution involves comparing the memory block accessed on the potential miss, b, with the memory blocks in the cache state and their ordering by recency of access to determine if memory block b is in the cache at the time of the potential miss. If b is in the cache, the potential boundary miss is a cache hit. If b is not in the cache, the potential boundary miss is a boundary miss.

Let $\mathfrak{P}_s = \{p_1, \ldots, p_m\}$ be the collection of m potential boundary misses incurred by a program fragment in cache set s, ordered by $\lhd$. Let $\{\mathfrak{C}_0 = \mathfrak{C}\langle s\rangle, \mathfrak{C}_i = \mathit{UpdateState}(\mathfrak{C}_{i-1}, p_i) : 1 < i \leqslant m\}$ be the sequence of cache states of set s updated for each potential boundary miss in $\mathfrak{P}_s$ (see equation (2.1) for *UpdateState*). The collection of boundary misses incurred by the program fragment in set s is $\mathfrak{B}_s = \mathit{Resolve}(\mathfrak{C}\langle s\rangle, \mathfrak{P}_s) = \{p_k \in \mathfrak{P}_s : p_k \notin \mathfrak{C}_k\}$. The collection of all potential boundary misses incurred by the program fragment is $\mathfrak{P} = \{\mathfrak{P}_s : 0 \leqslant s < \mathcal{S}\}$, and the collection of all boundary misses incurred by the program fragment is $\mathfrak{B} = \mathit{ResolveAll}(\mathfrak{C}, \mathfrak{P}) = \{\mathfrak{B}_s : 0 \leqslant s < \mathcal{S}\}$.

Definition 3.4 A **boundary miss** is a potential boundary miss that is resolved to be an actual cache miss, given the cache state when a program fragment begins execution.

Notice that determination of potential boundary misses does not require the cache state at the beginning of program fragment execution, but the determination of actual boundary misses does. Clearly, the interior-boundary approach to classifying misses requires mechanisms for deciding the cache state at the beginning of program fragments and resolving potential boundary misses given cache state.

3.3 Cache State

Recall from Section 2.1.1 that the state of the cache, $\mathfrak{C}$, is the collection of memory blocks residing in each cache set at any point during the execution of a program, ordered by recency of access. Suppose that program $\mathbb{P}$ consists of the sequential composition of n fragments, such that $\mathbb{P} \equiv \mathbb{F}_{1n} \overset{\text{def}}{=} \mathbb{F}_1; \mathbb{F}_2; \ldots; \mathbb{F}_{n-1}; \mathbb{F}_n$. Let the state of the cache before execution of fragment $\mathbb{F}_i$ and after execution of fragment $\mathbb{F}_{i-1}$ be $\mathfrak{C}^{(i-1)}$. The cache state when program $\mathbb{P}$ begins execution is $\mathfrak{C}^{(0)}$, and the cache state is $\mathfrak{C}^{(n)}$ when $\mathbb{P}$ ends execution. Cache state

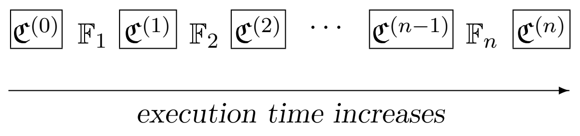

$\mathfrak{C}^{(0)}$ $\mathbb{F}_1$ $\mathfrak{C}^{(1)}$ $\mathbb{F}_2$ $\mathfrak{C}^{(2)}$ $\cdots$ $\mathfrak{C}^{(n-1)}$ $\mathbb{F}_n$ $\mathfrak{C}^{(n)}$
execution time increases

Figure 3.2: The role of cache state in the composition of fragments such that program $\mathbb{P} \equiv \mathbb{F}_{1n} \overset{\text{def}}{=} \mathbb{F}_1; \ldots; \mathbb{F}_n$.

$\mathfrak{C}^{(i)}$ is a reflection of how program fragment $\mathbb{F}_{i-1}$ updates the contents of the cache in cache state $\mathfrak{C}^{(i-1)}$. Therefore, cache state $\mathfrak{C}^{(i)}$ depends on both the behavior of fragment $\mathbb{F}_{i-1}$ and cache state $\mathfrak{C}^{(i-1)}$. Figure 3.2 illustrates how cache state fits into the composition of program fragments in $\mathbb{P}$.

Let $\mathfrak{I}^{(i)}$ be the set of interior misses incurred by a program fragment $\mathbb{F}_i$, and let $\mathbf{I}_i = |\mathfrak{I}^{(i)}|$ be the number of interior misses incurred by fragment $\mathbb{F}_i$. Let $\mathfrak{P}^{(i)}$ be the set of potential boundary misses incurred by a program fragment $\mathbb{F}_i$. Let $\mathfrak{B}^{(i)} = \mathit{ResolveAll}(\mathfrak{C}^{(i-1)}, \mathfrak{P}^{(i)})$ be the set of boundary misses incurred by a program fragment $\mathbb{F}_i$ executing from cache state $\mathfrak{C}^{(i-1)}$, and let $\mathbf{B}_i = |\mathfrak{B}^{(i)}|$ be the number of boundary misses incurred by fragment $\mathbb{F}_i$ executing from cache state $\mathfrak{C}^{(i-1)}$. Let $\mathbb{F}_{1n}$, the sequential composition of n program fragments $\mathbb{F}_1$ through $\mathbb{F}_n$, incur $\mathbf{I}_{1n}$ interior misses and $\mathbf{B}_{1n}$ boundary misses when executing from cache state $\mathfrak{C}^{(0)}$.

Theorem 3.1 *Let $\mathbb{F}_{1n} \overset{\text{def}}{=} \mathbb{F}_1; \mathbb{F}_2; \ldots; \mathbb{F}_n$ be the sequential composition of n program fragments executing from cache state $\mathfrak{C}^{(0)}$. Let each $\mathbb{F}_i$ incur $\mathbf{I}_i$ interior misses and $\mathbf{B}_i$ boundary misses. Then*

$$\mathbf{I}_{1n} + \mathbf{B}_{1n} = \sum_{j=1}^{n} \mathbf{I}_j + \sum_{j=1}^{n} \mathbf{B}_j$$

Proof. The proof is by induction on n, the number of program fragments. The base case, which considers two program fragments $\mathbb{F}_1$ and $\mathbb{F}_2$, is the significant part of the proof. The induction step is trivial.

Because it is independent of cache state, an interior miss incurred by program fragment $\mathbb{F}_1$ or $\mathbb{F}_2$ is also an interior miss in $\mathbb{F}_{12}$. Clearly, a boundary miss incurred by $\mathbb{F}_1$ executing from cache state $\mathfrak{C}^{(0)}$ is also a boundary miss in $\mathbb{F}_{12}$ executing from $\mathfrak{C}^{(0)}$. Let m be a boundary miss incurred by $\mathbb{F}_2$ executing from $\mathfrak{C}^{(1)}$. Either the memory block that m misses on, b, is accessed in $\mathbb{F}_1$ (Case 1) or it is not (Case 2).

Case 1 has four subcases: $b \in \mathfrak{C}^{(0)}$ and $b \in \mathfrak{C}^{(1)}$ (Case 1a), $b \in \mathfrak{C}^{(0)}$ and $b \notin \mathfrak{C}^{(1)}$ (Case 1b), $b \notin \mathfrak{C}^{(0)}$ and $b \in \mathfrak{C}^{(1)}$ (Case 1c), and $b \notin \mathfrak{C}^{(0)}$ and $b \notin \mathfrak{C}^{(1)}$ (Case 1d). For Cases 1a and 1c, there is an access of b in $\mathbb{F}_1$ such that b is not replaced in cache before the end of $\mathbb{F}_1$. For Cases 1b and 1d, b is replaced in cache before the end of $\mathbb{F}_1$ for all accesses of b in $\mathbb{F}_1$. Given all subcases of Case 1, because the behavior of accesses to b in $\mathbb{F}_1$ is known, cache state $\mathfrak{C}^{(1)}$ is not necessary to determine that m is a miss in $\mathbb{F}_{12}$. Thus, m is an interior miss in $\mathbb{F}_{12}$ for Case 1.

Case 2 has three subcases: $b \in \mathfrak{C}^{(0)}$ and $b \in \mathfrak{C}^{(1)}$ (Case 2a), $b \in \mathfrak{C}^{(0)}$ and $b \notin \mathfrak{C}^{(1)}$ (Case 2b), and $b \notin \mathfrak{C}^{(0)}$ and $b \notin \mathfrak{C}^{(1)}$ (Case 2c).[1] For Case 2a, there are no accesses in $\mathbb{F}_1$ that cause b to be replaced in cache, and cache state $\mathfrak{C}^{(0)}$ is necessary to determine that m is a miss in $\mathbb{F}_{12}$. Thus, m is a boundary miss in $\mathbb{F}_{12}$ executing from $\mathfrak{C}^{(0)}$ for Case 2a. For Case 2b, there is an access in $\mathbb{F}_1$ that causes b to be replaced in cache. Because the behavior of the access in $\mathbb{F}_1$ replacing b is known, cache state $\mathfrak{C}^{(1)}$ is not necessary to determine that m is a miss in $\mathbb{F}_{12}$. Thus, m is an interior miss in $\mathbb{F}_{12}$ for Case 2b. For Case 2c, b maps to cache for the first time in $\mathbb{F}_2$, and cache state $\mathfrak{C}^{(0)}$ is necessary to determine that m is a miss in $\mathbb{F}_{12}$. Thus, m is a boundary miss in $\mathbb{F}_{12}$ executing from $\mathfrak{C}^{(0)}$ for Case 2c.

A boundary miss incurred by $\mathbb{F}_2$ executing from $\mathfrak{C}^{(1)}$ may be either a boundary miss or an interior miss in $\mathbb{F}_{12}$ executing from $\mathfrak{C}^{(0)}$, but it is certainly a miss. Therefore, $\mathbf{I}_{12} + \mathbf{B}_{12} = \mathbf{I}_1 + \mathbf{B}_1 + \mathbf{I}_2 + \mathbf{B}_2 = \sum_{j=1}^{2} \mathbf{I}_j + \sum_{j=1}^{2} \mathbf{B}_j$, and the base case is true where $n = 2$ and $\mathbb{P} \equiv \mathbb{F}_{12}$.

Assume for $n = k$ and $\mathbb{P} \equiv \mathbb{F}_{1k}$ that $\mathbf{I}_{1k} + \mathbf{B}_{1k} = \sum_{j=1}^{k} \mathbf{I}_j + \sum_{j=1}^{k} \mathbf{B}_j$. For $n = k+1$ and $\mathbb{P} \equiv \mathbb{F}_{1,k+1}$, let $\mathbb{G}_1 \equiv \mathbb{F}_{1k}$ and let $\mathbb{G}_2 \equiv \mathbb{F}_{k+1}$. Applying the base case to $\mathbb{G}_{12}$ gives $\mathbf{I}_{1,k+1} + \mathbf{B}_{1,k+1} = \mathbf{I}_{ik} + \mathbf{B}_{ik} + \mathbf{I}_{k+1} + \mathbf{B}_{k+1}$. By the induction hypothesis, $\mathbf{I}_{1,k+1} + \mathbf{B}_{1,k+1} = \sum_{j=1}^{k+1} \mathbf{I}_j + \sum_{j=1}^{k+1} \mathbf{B}_j$ for $\mathbb{F}_{1,k+1}$. $\square$

Note that assertions stronger than Theorem 3.1, such as $\mathbf{I}_{1n} = \sum_{i=1}^{n} \mathbf{I}_i$, do not hold in general. It is possible that a boundary miss in program fragment $\mathbb{F}_2$ is an interior miss in $\mathbb{F}_{12}$.

The interior-boundary cache miss classification has an important advantage over its counterpart, the compulsory-replacement miss classification. The boundary misses incurred by a program fragment are bounded from above by the cache footprint of the data structures that it accesses, which is in turn bounded from above by the number of cache frames. Thus, the number of boundary misses for any program fragment is no more than the number of cache frames, as the following lemma establishes.

Lemma 3.1 *The number of boundary misses incurred by a program fragment does not exceed the number of frames in the cache.*

Proof. A potential boundary miss is due to unknown contents of the cache when a program fragment begins execution. When an access of memory block b incurs a potential boundary miss, it becomes known that one cache frame contains b, either because b was already in cache (*i.e.*, the potential boundary miss is actually a cache hit) or because b replaces some other memory block previously in cache (*i.e.*, the potential boundary miss is a boundary miss). Once all frames of the cache contain memory blocks brought to cache (or already in cache) during a potential boundary miss, no subsequent misses can depend on the state of the cache when the program fragment begins execution because the contents of the cache are known. Therefore, a program fragment may not incur any more potential boundary misses than there are frames in the cache. By definition, the number of boundary misses incurred by a program fragment

[1] Notice that the case in which $b \notin \mathfrak{C}^{(0)}$ and $b \in \mathfrak{C}^{(1)}$ is not valid, since it implies that b is accessed in $\mathbb{F}_1$.

is bounded from above by the number of potential boundary misses. Hence, the number of boundary misses incurred by any program fragment is bounded from above by the number of cache frames. □

Typically, the number of cache frames is much smaller than the number of interior misses incurred by the program fragment. Therefore, approximation of the number of cache misses incurred by the composite program $\mathbb{F}_{1n}$ as $\sum_{i=1}^{n} \mathbf{I}_i$ avoids the calculation of cache states, and the error bound on this approximation is small.

Theorem 3.2 *The actual number of cache misses incurred in an* $(\mathcal{A}, \mathcal{B}, \mathcal{C}; \mathcal{S})$ *cache by a program* $\mathbb{P} \equiv \mathbb{F}_{1n}$ *is in the range* $\left[\sum_{i=1}^{n} \mathbf{I}_i, n \cdot \frac{\mathcal{C}}{\mathcal{B}} + \sum_{i=1}^{n} \mathbf{I}_i\right]$.

Proof. Approximation of the number of cache misses incurred by $\mathbb{F}_{1n}$ as $\sum_{i=1}^{n} \mathbf{I}_i$ ignores the number of boundary misses incurred. By Lemma 3.1, a program fragment may incur no more boundary misses than there are frames in the cache. The number of frames in an $(\mathcal{A}, \mathcal{B}, \mathcal{C}; \mathcal{S})$ cache is $\frac{\mathcal{C}}{\mathcal{B}}$. Given that $\mathbb{F}_{1n}$ is a composite of n program fragments, this approximation ignores at most $n \cdot \frac{\mathcal{C}}{\mathcal{B}}$ boundary misses. □

3.4 Summary

This chapter introduces a new classification of cache misses with the composability property, which is critical for combining the cache behaviors of individual fragments of a program to obtain the cache behavior of the entire program. For compulsory-replacement cache miss classification schemes (including the 3C model [67] and the OPT model [118]), there is no simple way of relating the cache misses of each program fragment to the misses of the program because a compulsory miss incurred by a fragment may actually be a cache hit for the entire program.

The interior-boundary classification has two types of cache misses: misses independent of cache state when the program fragment begins execution (called interior misses) and misses dependent on that cache state (called potential boundary misses). Cache state is the glue between individual program fragments, resolving the potential boundary misses that actually do miss in cache (called boundary misses). The interior-boundary cache miss classification scheme overcomes the insufficiency of other miss classification schemes by using the state of the cache when a program fragment begins execution to determine the actual hit/miss status of memory accesses for which this status cannot be determined by considering the fragment alone. Therefore, all cache misses incurred by a program fragment are also misses for the entire program, and the total number of cache misses incurred by program $\mathbb{P}$ is the sum of the cache miss counts for each of the fragments of $\mathbb{P}$ (Theorem 3.1).

Compared to interior misses, determining the boundary misses incurred by a program fragment does require the additional step of resolving potential boundary misses with the cache state when the program fragment begins execution. However, the number of boundary misses is typically much smaller than the number of interior misses incurred by the program fragment. The number of boundary misses for any program fragment is no more than the number of frames in the cache $\frac{\mathcal{C}}{\mathcal{B}}$ (Lemma 3.1), and thus, a program with n fragments has no more than $n \cdot \frac{\mathcal{C}}{\mathcal{B}}$ boundary misses (Theorem 3.2). Therefore, it is possible to approximate the cache behavior of a program by counting only interior misses, omitting the computation of cache state and potential boundary misses, yet yielding an approximation of the total cache miss count with a tight error bound.

Chapter 4

Analyzing Cache Behavior

This chapter presents a method for counting the number of cache misses incurred by a loop nest executing in a cache memory with associativity $\mathcal{A}$, blocksize $\mathcal{B}$, capacity $\mathcal{C}$, and $\mathcal{S}$ sets. The technique applies to cache events that depend on the associativity value of the cache being considered.[1] Fundamental to the technique are the notions of *neighborhood* and *witness*, which serve to identify the situations that may cause a memory access to miss in cache depending on the associativity.

A single pass of the method produces the behavior of a loop nest for multiple cache configurations in which $\mathcal{B}$ and $\mathcal{S}$ are fixed for arbitrary $\mathcal{A}$ and corresponding $\mathcal{C}$. That is, in a single pass the method considers the family of caches $\{(k,\, \mathcal{B},\, k\mathcal{B}\mathcal{S};\, \mathcal{S})\colon\ k \geqslant 1\}$. Varying the number of cache sets and/or the cache blocksize changes the manner in which memory blocks map to cache sets. Therefore, it is not possible to consider the family of caches $\{(k,\, \mathcal{B},\, \mathcal{C};\, \frac{\mathcal{C}}{k\mathcal{B}})\colon\ k \geqslant 1\}$ in one pass, nor is it possible to consider the family of caches $\{(\mathcal{A},\, k,\, \mathcal{C};\, \frac{\mathcal{C}}{k\mathcal{A}})\colon\ k \geqslant 1\}$ in one pass.

The following are the four steps in the method.

1. Define neighborhoods and witnesses for the specific cache event, expressing witnesses in Presburger arithmetic.

2. State and prove a theorem to decide the cache event outcome based on a comparison of the witness count and associativity value $\mathcal{A}$.

3. Represent the witness formula as a DFA and enumerate the solutions.

4. Count the number of witnesses to each memory access and use the theorem stated in step 2 to determine the cache behavior of the access.

Steps 1 and 2 of the method are problem-specific and depend on the cache event being modeled; this chapter addresses steps 1 and 2. Steps 3 and 4 of the method are generic, and work for any type of cache event; the next chapter addresses steps 3 and 4.

[1]As Section 5.4 explains, there are cache events that do not depend on associativity, and a technique for handling such events is given.

Also fundamental to the technique for counting cache misses is a by-set decomposition of cache activity. Decomposing the sequence of memory accesses in a loop nest to consider the activity of each cache set in isolation has the effect of breaking down the analysis problem into a number of smaller, standalone problems. The events in one cache set are independent of the events in another, and an alternate way to view a cache set is as a small, fully-associative cache. This *set-centric* technique for attacking the cache analysis problem is one of the keys that makes my approach tractable, but without loss of any information. A memory access a *maps* to a set s if the array element specified by access a resides at memory byte address m and $s = \mathit{Set}(\mathit{Block}(m))$. Let A_s be the set of accesses in a loop nest that map to cache set s.

The remainder of this chapter is organized as follows. Section 4.1 provides the assumptions about program execution that guide the analysis framework of this dissertation. Section 4.2 introduces the notions of neighborhood and witness. Section 4.3 illustrates how witnesses are expressed as formulas of Presburger arithmetic. Section 4.4 gives theorems to decide cache event outcomes. Throughout this chapter several associativity-dependent cache events are considered—interior miss, replacement miss, potential boundary miss, and cache state.

4.1 Program Execution Model

The analysis framework presented in this dissertation models the cache behavior of a program $\mathbb{P}$, subject to the following assumptions.

- $\mathbb{P}$ consists of a sequence of count-controlled loop nests. (Any program statements outside of a loop may be easily folded into a loop of one iteration.)

- All loops are normalized to have a step size of one.

- For each loop at level j, the upper and lower bounds of loop control variable (LCV) ι_j are affine functions of LCVs $\iota_0 \ldots \iota_{j-1}$.

- The index expression of a reference R (located at nesting depth j) is an affine function of the LCVs belonging to all loops containing the reference $(\iota_0 \ldots \iota_j)$.

- The values of any constants used in the calculation of loop bounds and/or index expressions are known at compile time.

- $\mathbb{P}$ executes on a uniprocessor system with a single-level data cache that has an LRU replacement policy. The execution of $\mathbb{P}$ is in order and there is no prefetching of data. The cache processes data requests one at a time and treats data reads and writes the same, *i.e.*, the framework models a blocking, write-allocate cache with no prefetching.

4.2 Neighborhoods and Witnesses

The key to determining whether a memory access $\mathsf{a} \in \mathsf{A}_s$ misses in a cache is knowing whether b, the memory block requested by a, is resident in cache set s when the access occurs. We show how the presence of b in the cache relates to the number of distinct memory blocks mapping to set s that are accessed in the neighborhood of a.

We call a memory block a **witness** if its presence in the cache may cause a memory access to suffer a cache miss, depending on the associativity value being considered. Notice the asymmetry of the term. A *memory block* is a witness to a *memory access*. The **neighborhood** of an access a limits how much of a loop nest's memory access sequence must be considered to determine whether a is a cache miss. Thus, it is sufficient to look in the neighborhood of a memory access to find the witnesses that may cause it to miss. To determine whether a memory access a is a miss, we avoid considering the entire sequence of memory accesses in two ways: decomposing the sequence by cache set (discussed above) and defining neighborhoods (discussed below).

Neighborhoods and witnesses are named for the type of cache event that they affect. An access incurring an *i*nterior miss has an i-neighborhood and i-witnesses. An access incurring a *r*eplacment miss has an r-neighborhood and r-witnesses. An access incurring a potential *b*oundary miss has a b-neighborhood and b-witnesses. An access affecting the cache *s*tate at the end of program fragment execution has a s-neighborhood and s-witnesses. If yet another type of cache event interests the reader, the corresponding neighborhood and witness terms should be defined in the manner of the examples given here.

Recall from Section 3.2 that an interior miss is a memory access that is guaranteed to miss in cache, independent of the cache state when a program fragment begins execution.

Definition 4.1 The ***i*-neighborhood** of an access $\mathsf{a} \in \mathsf{A}_s$ is the set of all accesses in A_s occurring between a and the most recent access (in the total ordering of accesses, $\lhd$ given in Definition 2.2) touching the same memory block as a; or between a and the beginning of the program fragment, if a is the earliest access to the memory block it touches. Memory block b is an ***i*-witness** to access a if b is touched by an access in the i-neighborhood of a, and if b is distinct from the memory block touched by a.

Recall that replacement misses include capacity and conflict misses from the traditional 3C model [67]. There is a subtle difference between the i-neighborhood and r-neighborhood definitions: an i-neighborhood includes accesses occurring between access a and the beginning of the program fragment, while a r-neighborhood does not.

Definition 4.2 The ***r*-neighborhood** of an access $\mathsf{a} \in \mathsf{A}_s$ is the set of all accesses in A_s occurring between a and the most recent access (in the total ordering of accesses, $\lhd$) touching the same memory block as a. Memory block b is an ***r*-witness** to access a if b is touched by an access in the r-neighborhood of a, and if b is distinct from the memory block touched by a.

Recall that a potential boundary miss is a memory access that may hit or miss in cache, depending on the cache state when a program fragment begins execution.

Definition 4.3 The ***b*-neighborhood** of an access $\mathsf{a} \in \mathsf{A}_s$ is the set of all accesses in A_s occurring between a and the beginning of the program fragment, if a is the earliest access to the memory block it touches. The b-neighborhood of a is empty if a is not the earliest access to the memory block it touches. Memory block b is a ***b*-witness** to access a if b is touched by an access in the b-neighborhood of a, and if b is distinct from the memory block touched by a.

Recall that the cache state at the end of program fragment execution indicates the contents of the cache just as execution of the program fragment completes.

Definition 4.4 The ***s*-neighborhood** of an access $\mathsf{a} \in \mathsf{A}_s$ is the set of all accesses in A_s occurring between a and the end of the program fragment, if a is the latest access to the memory block it touches. The s-neighborhood of a is empty if a is not the latest access to the memory block it touches. Memory block b is an ***s*-witness** to access a if b is touched by an access in the s-neighborhood of a, and if b is distinct from the memory block touched by a.

There are two ways in which multiple memory accesses may touch the same memory block: several memory accesses may touch the same array element, and a memory block may contain multiple array elements. Consider the running example loop nest $\mathbb{L}_{\text{mm}}$ of Figure 2.5 on page 17 where $t = u = v = 20$. Reference R_1 = `X[i,k]` accesses the same array element on each iteration of the `j`-loop. For example, array element `X[3,10]`, and hence the memory block to which it maps, is touched by all memory accesses $\{([3, j, 10]^{\mathrm{T}}, R_1) : 0 \leqslant j < 20\}$. Suppose that the arrays of loop nest $\mathbb{L}_{\text{mm}}$ have a column-major layout and that each memory block holds four array elements (*i.e.*, array elements are 8 bytes each and $\mathcal{B} = 32$ bytes). Given a memory block whose first array element is `X[i,j]`, array elements `X[i+1,j]`, `X[i+2,j]`, and `X[i+3,j]` are in the same memory block. For example, the memory block containing array elements `X[0,10]`, `X[1,10]`, `X[2,10]`, and `X[3,10]` is touched by all memory accesses $\{([i, j, 10]^{\mathrm{T}}, R_1) : 0 \leqslant i \leqslant 3 \wedge 0 \leqslant j < 20\}$.

Figure 4.1 gives an example sequence of memory accesses a, b, c, d, e, and f that map to cache set x. Each type of box around an access denotes a distinct memory block. Notice that accesses a and c touch memory block m, accesses b and e touch memory block n, and accesses d and f touch memory block o. For example, consider access d. Memory blocks m and n are i-witnesses to access d. Access d has no r-witnesses because there is no earlier access to memory block o. Memory blocks m and n are b-witnesses to access d. Memory block n is an s-witness to access d. As another example, consider access e. Memory blocks m and o are i-witnesses to access e. Memory blocks m and o are r-witnesses to access e. Access e has

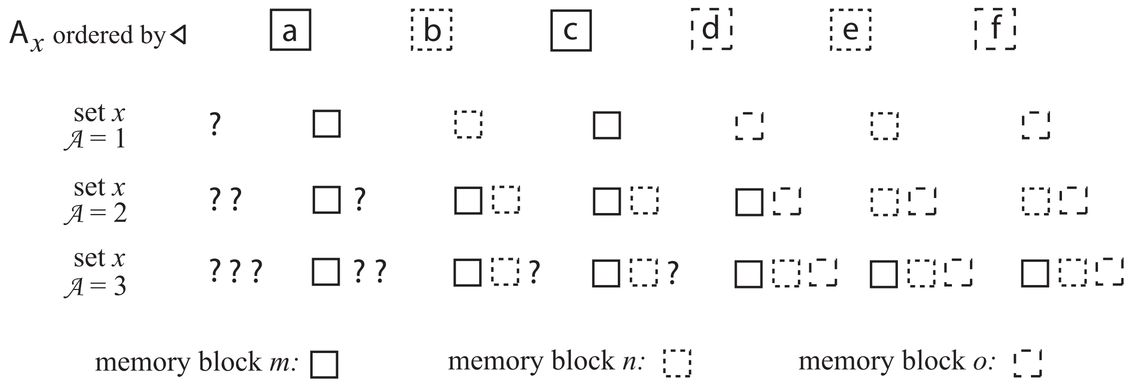

A_x ordered by $\lhd$
a
b
c
d
e
f
set x $\mathcal{A} = 1$
set x $\mathcal{A} = 2$
set x $\mathcal{A} = 3$
memory block m:
memory block n:
memory block o:

Figure 4.1: Example sequence of accesses in A_x and the contents of cache set x after each memory access, given caches with associativity values 1 to 3.

no b-witnesses because it is not the earliest access to memory block n. Memory block o is an s-witness to access e.

4.3 Expressing Witnesses in Presburger Arithmetic

Formulas of Presburger arithmetic express the i-witnesses, r-witnesses, b-witnesses, and s-witnesses to memory accesses executed by a loop nest. Section 4.3.1 presents Presburger formulas to describe basic features of program and memory. Section 4.3.2 gives formulas expressing various types of witnesses using the basic formulas from Section 4.3.1.

4.3.1 Formulas Describing Program and Memory Structure

The following are basic Presburger formulas describing the structure of the program and memory system being modeled. Table 4.1 organizes the Presburger formulas presented in Sections 4.3.1 and 4.3.2, matching formulas descriptions with equation numbers and page numbers.

Valid iteration point. The predicate $\iota \in \mathcal{I}$ describes the condition in which iteration point $\iota = [\iota_0, \ldots, \iota_{d-1}]^T$ belongs to the iteration space $\mathcal{I}$ (defined in Section 2.2).

$$\iota \in \mathcal{I} \overset{\text{def}}{=} \bigwedge_{j=0}^{d-1} (L_j \leqslant \iota_j) \wedge (\iota_j \leqslant U_j) \tag{4.1}$$

Lexicographical ordering of accesses. The predicate $(\iota, R_u) \lhd (\kappa, R_v)$ describes the condition in which the memory access made to reference R_u at iteration point ι precedes (defined in Section 2.3) the memory access made by R_v at iteration point κ. Notice that any access

Formula Description	Eqn. No.	Page No.
$\iota \in \mathcal{I}$	4.1	41
$(\iota, R_u) \lhd (\kappa, R_v)$	4.2	42
$\text{Map}(m, w, s)$	4.3	42
$m\prime = \text{Row-maj}(E_u(\iota))$	4.4	43
$m\prime = \text{Col-maj}(E_u(\iota))$	4.5	43
$((\iota, R_u), e) \in i\text{-witness}(\mathbb{L})$	4.6	43
$((\iota, R_u), e) \in r\text{-witness}(\mathbb{L})$	4.7	44
$((\iota, R_u), e) \in b\text{-witness}(\mathbb{L})$	4.8	46
$((\iota, R_u), e) \in s\text{-witness}(\mathbb{L})$	4.9	46
$(\iota, R_u) \in \text{Earliest}(\mathbb{L})$	4.10	47
$(\iota, R_u) \in \text{Latest}(\mathbb{L})$	4.11	47

Table 4.1: Presburger formulas used to describe cache behavior.

occurring at an iteration point ι precedes memory access (κ, R_v) if $\iota \prec \kappa$. Access to references R_t, such that $t < v$, at iteration point κ also precede memory access (κ, R_v).

$$(\iota, R_u) \lhd (\kappa, R_v) \overset{\text{def}}{=} \tag{4.2}$$
$$\iota \in \mathcal{I} \wedge \kappa \in \mathcal{I} \wedge \Big(\big(\bigvee_{j=0}^{d-1} (\iota_j < \kappa_j \wedge \bigwedge_{k=0}^{j-1} \iota_k = \kappa_k) \big) \vee \big(\bigwedge_{j=0}^{d-1} \iota_j = \kappa_j \wedge u < v \big) \Big)$$

Mapping memory locations to cache sets. Recall the *Block* and *Set* relations from Section 2.1.1. Memory byte address m maps to cache set $s = \mathit{Set}(\mathit{Block}(m))$. The following formula describes this mapping of memory addresses to cache sets, where w is the auxiliary wraparound term (defined in Section 2.1.1), and $w = \mathit{Wrap}(m)$. Recall that $\mathcal{B}$ is the cache blocksize and $\mathcal{S}$ is the number of cache sets, and that the values of $\mathcal{B}$ and $\mathcal{S}$ are fixed. Therefore, the multiplication in the following formula is multiplication by a constant, which is permitted in Presburger arithmetic.

$$\text{Map}(m, w, s) \overset{\text{def}}{=} \mathcal{B} * (w * \mathcal{S} + s) \leqslant m < \mathcal{B} * (w * \mathcal{S} + s) + \mathcal{B} \tag{4.3}$$

Data layouts in memory. Recall the row-major and column-major layout functions from Section 2.3. The following formulas give layouts for reference $R_u = (Y^{(x)}, E_u, S_h)$ and iteration point ι. Recall that $E_u(\iota) = [e_0, \ldots, e_{d_x - 1}]^T$ is the index expression of reference R_u evaluated for iteration point ι. Also recall that ℓ_n is the length of array $Y^{(x)}$ in the $(n+1)^{\text{th}}$ dimension and its value is constant. Therefore, the multiplication in the following formula is allowed in

Presburger arithmetic. The memory byte address of the array element accessed by reference R_u at iteration point ι is $m = \mu_x + m' \cdot \beta_x$.

$$\big(m' = \text{Row-maj}(E_u(\iota))\big) \overset{\text{def}}{=} m' \geqslant 0 \wedge \Big(m' = \big(\sum_{j=0}^{d_x-2} \big(\prod_{k=j+1}^{d_x-1} \ell_k\big) \cdot e_j\big) + e_{d_x-1}\Big) \tag{4.4}$$

$$\big(m' = \text{Col-maj}(E_u(\iota))\big) \overset{\text{def}}{=} m' \geqslant 0 \wedge \big(m' = e_0 + \sum_{j=1}^{d_x-1} \big(\prod_{k=0}^{j-1} \ell_k\big) \cdot e_j\big) \tag{4.5}$$

4.3.2 Formulas Describing Cache Behavior

The formulas presented in Section 4.3.1 fit together to express cache events. The following Presburger formulas capture i-witnesses, r-witnesses, b-witnesses, and s-witnesses.

***i*-witnesses.** Suppose that an access $\mathsf{a} \in \mathsf{A}_s$ touches a memory block b, and that an earlier access $\mathsf{a}' \in \mathsf{A}_s$ touches a memory block b' such that $b \neq b'$. If there does not exist an access a'' between a' and a touching b, then access a' is in the i-neighborhood of a and b' is an i-witness to memory access a. Let access a comprise reference $R_u = (Y^{(x)}, E_u, S_h)$ and iteration point ι, let access a' comprise reference $R_v = (Y^{(y)}, E_v, S_i)$ and iteration point κ, and let access a'' comprise reference $R_w = (Y^{(z)}, E_w, S_j)$ and iteration point ρ. Recall from Section 2.1.1 that another way of representing memory block b' is with the pair (s, e), where b' maps to cache set s with wraparound value e. Because i-witnesses are expressed for a particular cache set s, e represents the distinct memory block b'. The following formula expresses the condition that memory block b' (represented by the wraparound value e) is an i-witness to memory access $\mathsf{a} = (\iota, R_u)$.

$$\begin{aligned}
&\big(((\iota, R_u), e) \in i\text{-witness}(\mathbb{L})\big) \overset{\text{def}}{=} \\
&\quad \iota \in \mathcal{I} \wedge \\
&\quad \Big(\exists d : \text{Map}(\mu_x + \mathcal{L}_x(E_u(\iota)) * \beta_x, d, s) \wedge \\
&\qquad \big(\exists \kappa, v : (\kappa, R_v) \lhd (\iota, R_u) \wedge \text{Map}(\mu_y + \mathcal{L}_y(E_v(\kappa)) * \beta_y, e, s) \wedge \\
&\qquad\quad \neg(\exists \rho, w : (\kappa, R_v) \lhd (\rho, R_w) \lhd (\iota, R_u) \wedge \text{Map}(\mu_z + \mathcal{L}_z(E_w(\rho)) * \beta_z, d, s))\big) \wedge \\
&\qquad \neg(d = e)\Big)
\end{aligned} \tag{4.6}$$

The i-witness formula describes the condition that a memory block identified by wraparound value e is an i-witness to access (ι, R_u). Line 2 of the i-witness formula indicates that access (ι, R_u) occurs at a valid iteration point. Line 3 indicates that access (ι, R_u) maps to cache set s and touches a memory block with wraparound value d. Line 4 says that there is an access

(κ, R_v) occurring before (ι, R_u) that maps to cache set s and touches a memory block with wraparound value e. Line 5 says that there is no access (ρ, R_w) occurring after (κ, R_v) and before (ι, R_u) that maps to cache set s and touches a memory block with wraparound value d (*i.e.*, touches the same memory block as access (ι, R_u)). Line 6 indicates that wraparound values d and e are not equal, which means that the memory blocks they identify are distinct.

In summary, access (κ, R_v) occurs before access (ι, R_u) and each access touches a different memory block (identified by wraparound values e and d, respectively) mapping to the same set. Furthermore, there is no access to the memory block touched by (ι, R_u) occurring between (κ, R_v) and (ι, R_u). By definition, the memory block touched by access (κ, R_v) is an i-witness to access (ι, R_u). Figure 4.2 shows the Presburger formula constructed to describe the i-witnesses of the running example loop nest $\mathbb{L}_{\mathrm{mm}}$ in Figure 2.5 for cache set 0,[2] considering ($\mathcal{A}$, 32, $4096 \cdot \mathcal{A}$; 128) caches with any associativity value $\mathcal{A}$. Notice that the free variables in the Presburger formula of Figure 4.2 are ι_0, ι_1, ι_2, u, and e, where ι_0, ι_1, ι_2, and u represent the memory access in loop nest $\mathbb{L}_{\mathrm{mm}}$ with an i-witness represented by wraparound value e.

***r*-witnesses.** Suppose that an access $\mathsf{a} \in \mathsf{A}_s$ touches a memory block b, and that an earlier access a'' touches the same memory block. If there does not exist an access a''' between a'' and a touching b, then access a'' is the most recent access touching b. Suppose that an access $\mathsf{a}' \in \mathsf{A}_s$ touches a memory block b' such that $b \neq b'$. If a' is between a'' and a, then access a' is in the r-neighborhood of a and b' is an r-witness to memory access a. Let access a comprise reference $R_u = (Y^{(x)}, E_u, S_h)$ and iteration point ι, let access a' comprise reference $R_w = (Y^{(z)}, E_w, S_j)$ and iteration point ρ, let access a'' comprise reference $R_v = (Y^{(y)}, E_v, S_i)$ and iteration point κ, and let access a''' comprise reference $R_t = (Y^{(q)}, E_t, S_g)$ and iteration point ν. The following formula expresses the condition that memory block b' (represented by the wraparound value e) is an r-witness to memory access $\mathsf{a} = (\iota, R_u)$ in loop nest $\mathbb{L}$.

$$\begin{aligned}
&\big(((\iota, R_u), e) \in r\text{-witness}(\mathbb{L})\big) \overset{\text{def}}{=} \qquad\qquad (4.7)\\
&\quad \iota \in \mathcal{I} \wedge\\
&\quad \Big(\exists d : \mathrm{Map}(\mu_x + \mathcal{L}_x(E_u(\iota)) * \beta_x, d, s) \wedge\\
&\qquad \big(\exists \kappa, v : (\kappa, R_v) \lhd (\iota, R_u) \wedge \mathrm{Map}(\mu_y + \mathcal{L}_y(E_v(\kappa)) * \beta_y, d, s) \wedge\\
&\qquad\quad \neg(\exists \nu, t : (\kappa, R_v) \lhd (\nu, R_t) \lhd (\iota, R_u) \wedge \mathrm{Map}(\mu_q + \mathcal{L}_q(E_t(\nu)) * \beta_q, d, s)) \wedge\\
&\qquad\quad (\exists \rho, w : (\kappa, R_v) \lhd (\rho, R_w) \lhd (\iota, R_u) \wedge \mathrm{Map}(\mu_z + \mathcal{L}_z(E_w(\rho)) * \beta_z, e, s))\big) \wedge\\
&\qquad \neg(d = e)\Big)
\end{aligned}$$

The Presburger formula constructed to describe the r-witnesses of the running example loop nest $\mathbb{L}_{\mathrm{mm}}$ is given in Appendix A.

[2]For the purpose of illustration, all examples in Chapters 4 and 5 are kept manageable by showing formulas and DFAs that describe the behavior of just one cache set. The results given in Chapter 8 are for *entire* caches.

$$
\begin{aligned}
&0 \leqslant \iota_0, \iota_1, \iota_2 < 20 \ \wedge \\
&\Big(\exists d : ((\iota_2 = 0 \wedge u = 0 \wedge 32 * (128 * d) \leqslant 6400 + (\iota_0 + 20 * \iota_1) * 8 < 32 * (128 * d + 1)) \vee \\
&\quad (u = 1 \wedge 32 * (128 * d) \leqslant (\iota_0 + 20 * \iota_2) * 8 < 32 * (128 * d + 1)) \vee \\
&\quad (u = 2 \wedge 32 * (128 * d) \leqslant 3200 + (\iota_2 + 20 * \iota_1) * 8 < 32 * (128 * d + 1)) \vee \\
&\quad (\iota_2 = 19 \wedge u = 3 \wedge 32 * (128 * d) \leqslant 6400 + (\iota_0 + 20 * \iota_1) * 8 < 32 * (128 * d + 1))) \wedge \\
&\quad \big(\exists \kappa_0, \kappa_1, \kappa_2, v : 0 \leqslant \kappa_0, \kappa_1, \kappa_2 < 20 \wedge \\
&\qquad (\kappa_0 < \iota_0 \vee (\kappa_0 = \iota_0 \wedge \kappa_1 < \iota_1) \vee \\
&\qquad (\kappa_0 = \iota_0 \wedge \kappa_1 = \iota_1 \wedge \kappa_2 < \iota_2) \vee \\
&\qquad (\kappa_0 = \iota_0 \wedge \kappa_1 = \iota_1 \wedge \kappa_2 = \iota_2 \wedge v < u)) \wedge \\
&\qquad ((\kappa_2 = 0 \wedge v = 0 \wedge 32 * (128 * e) \leqslant 6400 + (\kappa_0 + 20 * \kappa_1) * 8 < 32 * (128 * e + 1)) \vee \\
&\qquad (v = 1 \wedge 32 * (128 * e) \leqslant (\kappa_0 + 20 * \kappa_2) * 8 < 32 * (128 * e + 1)) \vee \\
&\qquad (v = 2 \wedge 32 * (128 * e) \leqslant 3200 + (\kappa_2 + 20 * \kappa_1) * 8 < 32 * (128 * e + 1)) \vee \\
&\qquad (\kappa_2 = 19 \wedge v = 3 \wedge 32 * (128 * e) \leqslant 6400 + (\kappa_0 + 20 * \kappa_1) * 8 < 32 * (128 * e + 1))) \wedge \\
&\qquad (\neg(\exists \rho_0, \rho_1, \rho_2, w : 0 \leqslant \rho_0, \rho_1, \rho_2 < 20 \wedge \\
&\qquad\quad (\rho_0 < \iota_0 \vee (\rho_0 = \iota_0 \wedge \rho_1 < \iota_1) \vee \\
&\qquad\quad (\rho_0 = \iota_0 \wedge \rho_1 = \iota_1 \wedge \rho_2 < \iota_2) \vee \\
&\qquad\quad (\rho_0 = \iota_0 \wedge \rho_1 = \iota_1 \wedge \rho_2 = \iota_2 \wedge w < u)) \wedge \\
&\qquad\quad (\kappa_0 < \rho_0 \vee (\kappa_0 = \rho_0 \wedge \kappa_1 < \rho_1) \vee \\
&\qquad\quad (\kappa_0 = \rho_0 \wedge \kappa_1 = \rho_1 \wedge \kappa_2 < \rho_2) \vee \\
&\qquad\quad (\kappa_0 = \rho_0 \wedge \kappa_1 = \rho_1 \wedge \kappa_2 = \rho_2 \wedge v < w)) \wedge \\
&\qquad\quad ((\rho_2 = 0 \wedge w = 0 \wedge 32 * (128 * d) \leqslant 6400 + (\rho_0 + 20 * \rho_1) * 8 < 32 * (128 * d + 1)) \vee \\
&\qquad\quad (w = 1 \wedge 32 * (128 * d) \leqslant (\rho_0 + 20 * \rho_2) * 8 < 32 * (128 * d + 1)) \vee \\
&\qquad\quad (w = 2 \wedge 32 * (128 * d) \leqslant 3200 + (\rho_2 + 20 * \rho_1) * 8 < 32 * (128 * d + 1)) \vee \\
&\qquad\quad (\rho_2 = 19 \wedge w = 3 \wedge 32 * (128 * d) \leqslant 6200 + (\rho_0 + 20 * \rho_1) * 8 < 32 * (128 * d + 1)))))\big) \\
&\quad \wedge \neg(d = e)\Big)
\end{aligned}
$$

Figure 4.2: Presburger formula describing the i-witnesses of loop nest $\mathbb{L}_{\text{mm}}$ for cache set 0.

***b*-witnesses.** Suppose that an access $\mathsf{a} \in \mathsf{A}_s$ touches a memory block b, and that an earlier access $\mathsf{a}' \in \mathsf{A}_s$ touches a memory block b' such that $b \neq b'$. If there does not exist an access a'' occurring before a touching b (*i.e.*, a is the earliest access to memory block b), then access a' is in the b-neighborhood of a and b' is a b-witness to memory access a. Let access a comprise reference $R_u = (Y^{(x)}, E_u, S_h)$ and iteration point ι, let access a' comprise reference $R_v = (Y^{(y)}, E_v, S_i)$ and iteration point κ, and let access a'' comprise reference $R_w = (Y^{(z)}, E_w, S_j)$ and iteration point ρ. The following formula expresses the condition that memory block b' (represented by the wraparound value e) is a b-witness to memory access $\mathsf{a} = (\iota, R_u)$.

$$\begin{aligned}
&\big(((\iota, R_u), e) \in b\text{-witness}(\mathbb{L})\big) \overset{\text{def}}{=} \qquad (4.8)\\
&\quad \iota \in \mathcal{I} \wedge \\
&\quad \big(\exists d : \mathrm{Map}(\mu_x + \mathcal{L}_x(E_u(\iota)) * \beta_x, d, s) \wedge \\
&\qquad \big(\exists \kappa, v : (\kappa, R_v) \lhd (\iota, R_u) \wedge \mathrm{Map}(\mu_y + \mathcal{L}_y(E_v(\kappa)) * \beta_y, e, s)\big) \wedge \\
&\qquad \neg\big(\exists \rho, w : (\rho, R_w) \lhd (\iota, R_u) \wedge \mathrm{Map}(\mu_z + \mathcal{L}_z(E_w(\rho)) * \beta_z, d, s)\big) \wedge \\
&\qquad \neg(d = e)\big)
\end{aligned}$$

The Presburger formula constructed to describe the b-witnesses of the running example loop nest $\mathbb{L}_{\mathrm{mm}}$ is given in Appendix A.

***s*-witnesses.** Suppose that an access $\mathsf{a} \in \mathsf{A}_s$ touches a memory block b and a later access $\mathsf{a}' \in \mathsf{A}_s$ touches a memory block b' such that $b \neq b'$. If there is no access a'' occurring after a touching b (*i.e.*, a is the latest access to memory block b), then access a' is in the s-neighborhood of a and b' is an s-witness to memory access a. Let access a comprise reference $R_u = (Y^{(x)}, E_u, S_h)$ and iteration point ι, let access a' comprise reference $R_v = (Y^{(y)}, E_v, S_i)$ and iteration point κ, and let access a'' comprise reference $R_w = (Y^{(z)}, E_w, S_j)$ and iteration point ρ. The following formula expresses the condition that memory block b' (represented by the wraparound value e) is an s-witness to memory access $\mathsf{a} = (\iota, R_u)$.

$$\begin{aligned}
&\big(((\iota, R_u), e) \in s\text{-witness}(\mathbb{L})\big) \overset{\text{def}}{=} \qquad (4.9)\\
&\quad \iota \in \mathcal{I} \wedge \\
&\quad \big(\exists d : \mathrm{Map}(\mu_x + \mathcal{L}_x(E_u(\iota)) * \beta_x, d, s) \wedge \\
&\qquad \big(\exists \kappa, v : (\iota, R_u) \lhd (\kappa, R_v) \wedge \mathrm{Map}(\mu_y + \mathcal{L}_y(E_v(\kappa)) * \beta_y, e, s)\big) \wedge \\
&\qquad \neg\big(\exists \rho, w : (\iota, R_u) \lhd (\rho, R_w) \wedge \mathrm{Map}(\mu_z + \mathcal{L}_z(E_w(\rho)) * \beta_z, d, s)\big) \wedge \\
&\qquad \neg(d = e)\big)
\end{aligned}$$

The Presburger formula constructed to describe the s-witnesses of the running example loop nest $\mathbb{L}_{\mathrm{mm}}$ is given in Appendix A.

Earliest. In order to count boundary misses and update cache state, it is necessary to specify the *earliest* and *latest* memory accesses to map to each cache set (in the total ordering of accesses, $\lhd$ given in Definition 2.2). The earliest access to map to a cache set s has 0 b-witnesses and is a potential boundary miss. The latest access to map to a cache set s has 0 s-witnesses and is in the cache state of set s, $\mathfrak{C}\langle s\rangle$, at the end of program fragment execution. Refer to Section 5.3.3 to see how the earliest and latest accesses mapping to each cache set are used to count boundary misses and update cache state.

Suppose that an access $\mathsf{a} \in \mathsf{A}_s$ touches a memory block b. If there does not exist an access $\mathsf{a}' \in \mathsf{A}_s$ occurring before a touching a memory block b', then access a is the earliest memory access in loop nest $\mathbb{L}$ mapping to cache set s. Let access a comprise reference $R_u = (Y^{(x)}, E_u, S_h)$ and iteration point ι, and let access a' comprise reference $R_v = (Y^{(y)}, E_v, S_i)$ and iteration point κ. The following formula expresses the condition that memory access $\mathsf{a} = (\iota, R_u)$ is the *earliest* memory access (in the total ordering of accesses, $\lhd$) to map to cache set s.

$$\begin{aligned}
&\big((\iota, R_u) \in \text{Earliest}(\mathbb{L})\big) \overset{\text{def}}{=} \\
&\quad \iota \in \mathcal{I} \wedge \\
&\quad \Big(\exists d : \text{Map}(\mu_x + \mathcal{L}_x(E_u(\iota)) * \beta_x, d, s) \wedge \\
&\qquad\quad \neg\big(\exists \kappa, v, e : (\kappa, R_v) \lhd (\iota, R_u) \wedge \text{Map}(\mu_y + \mathcal{L}_y(E_v(\kappa)) * \beta_y, e, s)\big)\Big)
\end{aligned} \tag{4.10}$$

Latest. Suppose that an access $\mathsf{a} \in \mathsf{A}_s$ touches a memory block b. If there does not exist an access $\mathsf{a}' \in \mathsf{A}_s$ occurring after a touching a memory block b', then access a is the latest memory access in loop nest $\mathbb{L}$ mapping to cache set s. Let access a comprise reference $R_u = (Y^{(x)}, E_u, S_h)$ and iteration point ι, and let access a' comprise reference $R_v = (Y^{(y)}, E_v, S_i)$ and iteration point κ. The following formula expresses the condition that memory access $\mathsf{a} = (\iota, R_u)$ is the *latest* memory access (in the total ordering of accesses, $\lhd$) to map to cache set s.

$$\begin{aligned}
&\big((\iota, R_u) \in \text{Latest}(\mathbb{L})\big) \overset{\text{def}}{=} \\
&\quad \iota \in \mathcal{I} \wedge \\
&\quad \Big(\exists d : \text{Map}(\mu_x + \mathcal{L}_x(E_u(\iota)) * \beta_x, d, s) \wedge \\
&\qquad\quad \neg\big(\exists \kappa, v, e : (\iota, R_u) \lhd (\kappa, R_v) \wedge \text{Map}(\mu_y + \mathcal{L}_y(E_v(\kappa)) * \beta_y, e, s)\big)\Big)
\end{aligned} \tag{4.11}$$

4.4 Deciding Cache Event Outcomes

The second step of the method for identifying the cache misses incurred by a loop nest is to state and prove a theorem to decide the cache event outcome of a memory access based on a comparison of the number of witnesses to the access and the value of associativity $\mathcal{A}$ being considered. Statements and proofs of theorems to decide the outcome of interior miss (Section 4.4.1), replacement miss (Section 4.4.2), boundary miss (Section 4.4.3), and cache state (Section 4.4.4) events follow.

Recall from Section 2.1 that each set of an ($\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$; $\mathcal{S}$) cache can hold at most $\mathcal{A}$ distinct memory blocks. If a cache set s already contains $\mathcal{A}$ distinct memory blocks upon access of yet another distinct memory block mapping to set s, the least recently used of the original memory blocks is replaced with the new memory block. Comparing the witness count for a memory access $\mathsf{a} \in \mathsf{A}_s$ with the value of $\mathcal{A}$ determines if the memory block accessed by a is among the $\mathcal{A}$ memory blocks in set s; and therefore, the comparison determines the event outcome for access a in a cache with associativity $\mathcal{A}$.

4.4.1 Interior Miss

To determine whether a memory access a touching memory block b incurs an interior miss in a cache with associativity value $\mathcal{A}$, it is necessary and sufficient to know two things: that there are at least $\mathcal{A}$ earlier accesses to distinct memory blocks mapping to the same cache set as b; and that there is no access to b between these earlier accesses and a.

Theorem 4.1 *Memory access a is an interior miss if and only if there are $\mathcal{A}$ or more* i-*witnesses to access a.*

Proof. Suppose that memory access a requests memory block b. If there are $\mathcal{A}$ or more i-witnesses to a, then, by definition, there are $\mathcal{A}$ or more earlier accesses requesting unique memory blocks mapping to the cache set and there is no access to b between these earlier accesses and a. These unique memory blocks fill the frames of the cache set and ensure that b does not reside there, forcing access a to incur an interior miss.

If memory access a is an interior miss, then a incurs a cache miss independent of cache state when the program fragment begins execution. Either a is the earliest access to memory block b (Case 1) or it is not (Case 2). For Case 1, b is guaranteed to be absent from the cache set only if $\mathcal{A}$ or more unique memory blocks are requested between access a and the beginning of the program fragment. By definition, these memory blocks are i-witnesses to memory access a. For Case 2, b is brought to the cache set during access a' (the most recent access requesting b), but it is not resident at the time of access a. Memory block b may be displaced only if $\mathcal{A}$ or more unique memory blocks are requested between accesses a and a'. By definition, these memory blocks are i-witnesses to memory access a. $\square$

Figure 4.1 displays the contents of cache set x before and after each memory access in the example sequence of accesses, for associativity values 1 to 3. For example, consider access e, which has two i-witnesses (memory blocks m and o). Access e is an interior miss for $\mathcal{A} = 1$ and $\mathcal{A} = 2$, but access e is a cache hit for $\mathcal{A} = 3$. This can be seen from the contents of cache set x just before execution of access e. Notice that in the $\mathcal{A} = 3$ case, witnesses do not cause an interior miss.

4.4.2 Replacement Miss

A memory access a touching memory block b incurs a replacement miss in a cache with associativity value $\mathcal{A}$ if a is not the earliest access to memory block b and there are at least $\mathcal{A}$ accesses to distinct memory blocks mapping to the same cache set as b occurring between the most recent access to memory block b and access a.

Theorem 4.2 *Memory access a is a replacement miss if and only if there are $\mathcal{A}$ or more* r-*witnesses to access a.*

Proof. Suppose that memory access a requests memory block b, and let a' be the most recent access touching the same memory block. If there are $\mathcal{A}$ or more r-witnesses to a, then, by definition, there are $\mathcal{A}$ or more unique memory blocks mapping to the same cache set as b between accesses a and a'. These unique memory blocks cause block b, which is in the cache set during access a', to be displaced and force access a to incur a replacement miss.

If memory access a is a replacement miss, then memory block b is brought to the cache set during an access a' (the most recent access requesting b), but b is not resident at the time of access a. Memory block b may be displaced from the cache set only if $\mathcal{A}$ or more unique memory blocks are requested between accesses a and a'. By definition, these memory blocks are r-witnesses to memory access a. □

Consider access c in Figure 4.1, which has one r-witness (memory block n). Access c is a replacement miss for $\mathcal{A} = 1$, but access c is a cache hit for $\mathcal{A} = 2$ and $\mathcal{A} = 3$. This can be seen from the contents of cache set x just before execution of access c. Notice that in the $\mathcal{A} = 2$ and $\mathcal{A} = 3$ cases, the witness does not cause a replacement miss.

4.4.3 Potential Boundary Miss

A memory access a touching memory block b incurs a potential boundary miss in a cache with associativity value $\mathcal{A}$ if a is the earliest access to memory block b and there are fewer than $\mathcal{A}$ earlier accesses to distinct memory blocks mapping to the same cache set as b.

Theorem 4.3 *Memory access a is a potential boundary miss if and only if there are fewer than $\mathcal{A}$* b-*witnesses to a access a.*

Proof. If there are fewer than $\mathcal{A}$ b-witnesses to access a, then fewer than $\mathcal{A}$ distinct memory blocks mapping to cache set s are accessed between a and the beginning of the program fragment. Let X be the set of these fewer than $\mathcal{A}$ distinct memory blocks. At the time of access a, the $\mathcal{A}$ frames of cache set s are filled with memory blocks from the set X and a subset of the original cache blocks contained in the cache state of set s at the beginning of program fragment execution. Access a is a cache hit if it is in the subset of cache state, and it is a boundary miss if it is not. Because the actual miss status of a depends on cache state, it is a potential boundary miss.

If memory access a is a potential boundary miss, then a may hit or miss depending on the cache state when the program fragment begins execution. In order for the actual miss status of a to depend on cache state, at the time of memory access a one or more of the frames of cache set s must contain an original cache block from the cache state at the beginning of program fragment execution. If one or more original cache blocks remain in set s at the time of access a, then fewer than $\mathcal{A}$ distinct memory blocks mapping to set s are accessed between a and the beginning of the program fragment. By definition, these memory blocks are b-witnesses to memory access a. □

Notice that if there are $\mathcal{A}$ or more b-witnesses to a memory access a, then a is an interior miss. This case is already covered above in Theorem 4.1, since i-witnesses and b-witnesses to an access a overlap if a is the earliest access to the memory block that it touches.

Consider access d in Figure 4.1, which has two b-witnesses (memory blocks m and n). Access d is an interior miss for $\mathcal{A} = 1$ and $\mathcal{A} = 2$. Access d is a potential boundary miss for $\mathcal{A} = 3$, and whether access d is an actual boundary miss or a cache hit depends on the memory block represented by ? in the contents of cache set x just before execution of access d. If the mystery memory block is o, access d is a cache hit. If it is any other memory block, access d is a boundary miss. Cache state provides such information.

4.4.4 Cache State

The state of a cache set x at the end of program fragment execution contains the $\mathcal{A}$ memory blocks mapping to set x that are the latest touched (fewer if the program fragment accesses fewer than $\mathcal{A}$ blocks of memory). Figure 4.1 shows the state of cache set x after execution of the example sequence of accesses for associativity values 1 to 3. The state of set x is given by its contents after execution of access f. A memory block b touched by access a belongs to the cache state of set x at the end of program fragment execution if a is the latest access to memory block b and there are fewer than $\mathcal{A}$ later accesses to distinct memory blocks mapping to cache set x.

Theorem 4.4 *The memory block touched by access a is in the cache state at the end of program fragment execution if and only if there are fewer than $\mathcal{A}$ s-witnesses to memory access*

a. *Such a memory block is the k^{th} most recently-accessed of all memory blocks in the cache state at the end of program fragment execution, where $k - 1 < \mathcal{A}$ is the number of* s*-witnesses to access* a.

Proof. Suppose that memory access a touches memory block b. If there are $k-1$ s-witnesses to access a, then $k-1$ distinct memory blocks mapping to cache set s are accessed between a and the end of the program fragment. Let X be the set of these memory blocks. At the end of program fragment execution, cache set s contains the $\mathcal{A}$ memory blocks mapping the set s that are accessed latest in the program fragment. Because set X contains fewer than $\mathcal{A}$ memory blocks, memory block b is among the $\mathcal{A}$ latest accessed and must be in the cache state at the end of program fragment execution. The memory blocks in X are more recently accessed than b and aside from the memory blocks in X, b is the most recently accessed of all the memory blocks in the cache state at the end of program fragment execution. Therefore, b is the k^{th} most recently-accessed of all memory blocks in the cache state at the end of program fragment execution.

If the memory block accessed by a is in the cache state at the end of program fragment execution, then it must be one of the $\mathcal{A}$ memory blocks accessed latest in the program fragment. In order for the memory block accessed by a to be one of the $\mathcal{A}$ latest accessed memory blocks, there must be fewer than $\mathcal{A}$ distinct memory blocks accessed between a and the end of the program fragment. By definition, these memory blocks are s-witnesses to memory access a. □

Consider access e in Figure 4.1, which has one s-witness (memory block o). For $\mathcal{A} = 1$, the memory block n touched by access e is not in the final cache state. For $\mathcal{A} = 2$ and $\mathcal{A} = 3$, memory block n is in the final cache state.

4.5 Summary

This chapter introduces the notions of neighborhood and witness to identify the conditions of a cache event outcome for a memory access. Associativity-dependent cache events include interior misses, replacement misses, potential boundary misses, and whether a memory block is in the cache state at the end of program fragment execution. For other cache events of interest, the reader can define neighborhood and witness terms using the examples provided here as guides. A memory block is a witness to a memory access if its presence in the cache causes the access to incur a cache event, depending on the value of associativity $\mathcal{A}$. Theorems 4.1 to 4.4 serve as rules to decide the outcome of particular cache events based on the witness count of a memory access and the value of associativity $\mathcal{A}$. Given a Presburger formula describing the witnesses to memory accesses executed by a loop nest, it is then sufficient to count the number of witnesses to each memory access and use the rules to determine the cache behavior of the access. The next chapter presents the method for counting witnesses, which first requires enumerating the solutions to the Presburger formula describing witnesses.

Chapter 5

Counting Cache Misses

Given a Presburger formula describing the witnesses for a particular cache event, the goal is to count the number of witnesses to each memory access and determine the cache behavior of the access for an associativity value $\mathcal{A}$. A witness formula describes the witnesses of all memory accesses in a loop nest. Each solution to the formula expresses a memory access and one of its witnesses (or its only witness). To count the witnesses for each memory access, it is necessary to enumerate the solutions of the witness formula, collect the solutions describing a particular memory access, and count such solutions expressing unique witnesses for the access.

Recall from Section 2.4 that computing the integer solutions to a Presburger formula can be a difficult problem, as the complexity of quantifier elimination and decidability for Presburger arithmetic is superexponential in the worst case. Unlike some techniques that attack Presburger formulas directly [38, 106, 125], the approach presented here is to convert a Presburger formula to a different form and then compute its solutions. The key of this indirect approach is to exploit a fundamental connection between Presburger arithmetic and automata theory, namely, that there exists a deterministic finite automaton (DFA) recognizing the positional binary representation of the solutions of any Presburger formula.

Recall from Section 2.5 that the worst-case complexity of constructing the DFA recognizing the solutions of a Presburger formula is superexponential. Therefore, converting Presburger formulas to DFAs does not circumvent the difficulty of computing formula solutions, since the worst-case complexities of automata construction and Presburger arithmetic decidability are the same. However, the worst-case complexity is better understood in the context of DFAs. In the translation of Presburger formulas to DFAs, this complexity manifests in the number of states in the resulting DFA. Specifically, it is the translation of universal quantifiers and negation that causes the exponential blowup of the automaton [23]. Another desirable side effect of exploiting the Presburger arithmetic-DFA connection is that one can obtain the minimal DFA representation of a formula. Given these advantages, the analytical framework of this dissertation builds on the Presburger-DFA connection to compute the solutions of cache behavior formulas.

Recall from Chapter 4 that step 3 of the method for counting the number of cache misses incurred by a loop nest represents the witness formula as a DFA and enumerates its solutions. Section 5.1 explains how to represent a Presburger formula as a DFA, and Section 5.2 shows how to enumerate the accepting paths of a DFA. Step 4 of the method counts the number of witnesses to each memory access and determines the cache behavior of the access by comparing its witness count to associativity value $\mathcal{A}$. Section 5.3 discusses how to count witnesses given the enumeration of formula solutions and how to compare witness counts with associativity values. Finally, Section 5.4 discusses a special case for handling cache events that do not depend on the associativity value of the cache.

5.1 Representing Formulas as DFAs

To illustrate how a DFA represents the solutions of a Presburger formula, Section 5.1.1 explains how accepting DFA paths encode the free variable values that constitute each solution to the formula, Section 5.1.2 gives the DFA recognizing the solutions of an example Presburger formula, and Section 5.1.3 reviews the DFA construction procedure.

5.1.1 Encoding Free Variable Values

Given the DFA representation of a Presburger formula, each accepting path in the DFA encodes free variable values that constitute a solution to the formula. The encoding is the standard binary representation of the integer values of the free variables, *proceeding from least significant bit (LSB) to most significant bit (MSB).*[1] Formally, an encoding of a non-negative integer c is a word $c_m \dots c_0$ over the alphabet $\{0, 1\}$ and $c = \sum_{i=0}^{m} c_i 2^i$. (We show below how to handle the mismatch between this conventional MSB-first representation of integers and the LSB-first encoding by the DFA.) For a fixed word length $m+1$, the encoding of c as $c_m \dots c_0$ is unique, but it is possible to encode c using words of different lengths. For $c = 5$, the word 101 of length 3 and the word 0101 of length 4 both encode c.

A tuple of non-negative integers is encoded by *stacking* their binary representations. In stacking binary representations, it is necessary that each non-negative integer in the tuple is encoded by words of the same length. It is straightforward to make two words the same length without changing the values that they encode. For a v-length word and a w-length word such that $v < w$, attach $w - v$ zeros in the MSBs of the v-length word. Let $c^{(i-1)}$ be the i^{th} non-negative integer in an n-tuple, and let $c_{j-1}^{(i-1)}$ be the j^{th} letter of the word encoding $c^{(i-1)}$. A

[1]The automata-construction procedure [9, 10] used by the framework presented in this dissertation encodes values such that the first letter of an integer word recognized by the automaton is the LSB. It is possible to construct automata such that the MSB is first. As Section 2.5 explains, it is unknown whether an LSB-first or a MSB-first encoding is superior.

stack of letters

$$t_{j-1} = \begin{array}{c} c_{j-1}^{(0)} \\ c_{j-1}^{(1)} \\ \vdots \\ c_{j-1}^{(n-1)} \end{array}$$

represents the j^{th} letter in each word encoding an n-tuple of non-negative integers. The concatenation of letter stacks $t_m \dots t_0$ is the stack of binary representations for the n-tuple, where the length of each binary representation is $m + 1$. The part of a stack which gives the binary representation of the i^{th} non-negative integer in an n-tuple, $c^{(i-1)}$, is referred to as the i^{th} *track* of the stack.

For a DFA $M = (S, \Sigma,\ \delta,\ q_0,\ F)$, let Σ^i be the finite set of i-length strings that are concatenations of the alphabet symbols. For DFAs recognizing the standard binary representations of an n-tuple of integers, the alphabet is

$$\Sigma = \left\{ \begin{array}{cc} x_0 & \\ x_1 & \\ \vdots & : x_j \in \{0, 1\} \\ x_{n-1} & \end{array} \right\}.$$

Let the function $\delta^i : S \times \Sigma^i \to S$ transition from one DFA state to another via an i-length string of alphabet symbols. Let a path of length i from state p to state q, denoted $P_i(p, q)$, be a string $A \in \Sigma^i$ such that $\delta^i(p, A^R) = q$ where string A^R is the *reversal* of string A. (The string reversal is a notational device to resolve the mismatch between the conventional MSB-first representation of numbers and the LSB-first consumption of the encoding by the DFA.)

The algorithms for counting and enumerating accepting DFA paths rely on knowing the set of states reachable from another set of states. Therefore, to easily identify one set of states reachable from another set of states, extend the transition function $\delta : S \times \Sigma \to S$ to $\Delta : 2^S \to 2^S$, defined by $\Delta(S') = \bigcup_{p \in S'} \{\delta(p, a) : a \in \Sigma\}$.

5.1.2 Example DFA

Given a DFA that recognizes the positional binary representation of the solutions of a Presburger formula, each accepting path in the DFA encodes a solution to the formula. Figure 5.1 shows a DFA that recognizes the solutions of the Presburger formula in Figure 4.2, which represents the i-witnesses of the running example loop nest $\mathbb{L}_{\text{mm}}$ in Figure 2.5 on page 17 for cache set 0. Notice that a state numbered 1 is missing from this DFA. State 1 is the garbage state.[2] The garbage state and the edges leading to it have been pruned from this DFA, making

[2]A state that is not part of any accepting path is a garbage state, or dead state. One example of a garbage state is a state that is not a final state and has transitions to itself for all symbols of the alphabet. Another example of a garbage state is any of a strongly connected component of states with transitions among themselves, but no transition to a state outside this component of states.

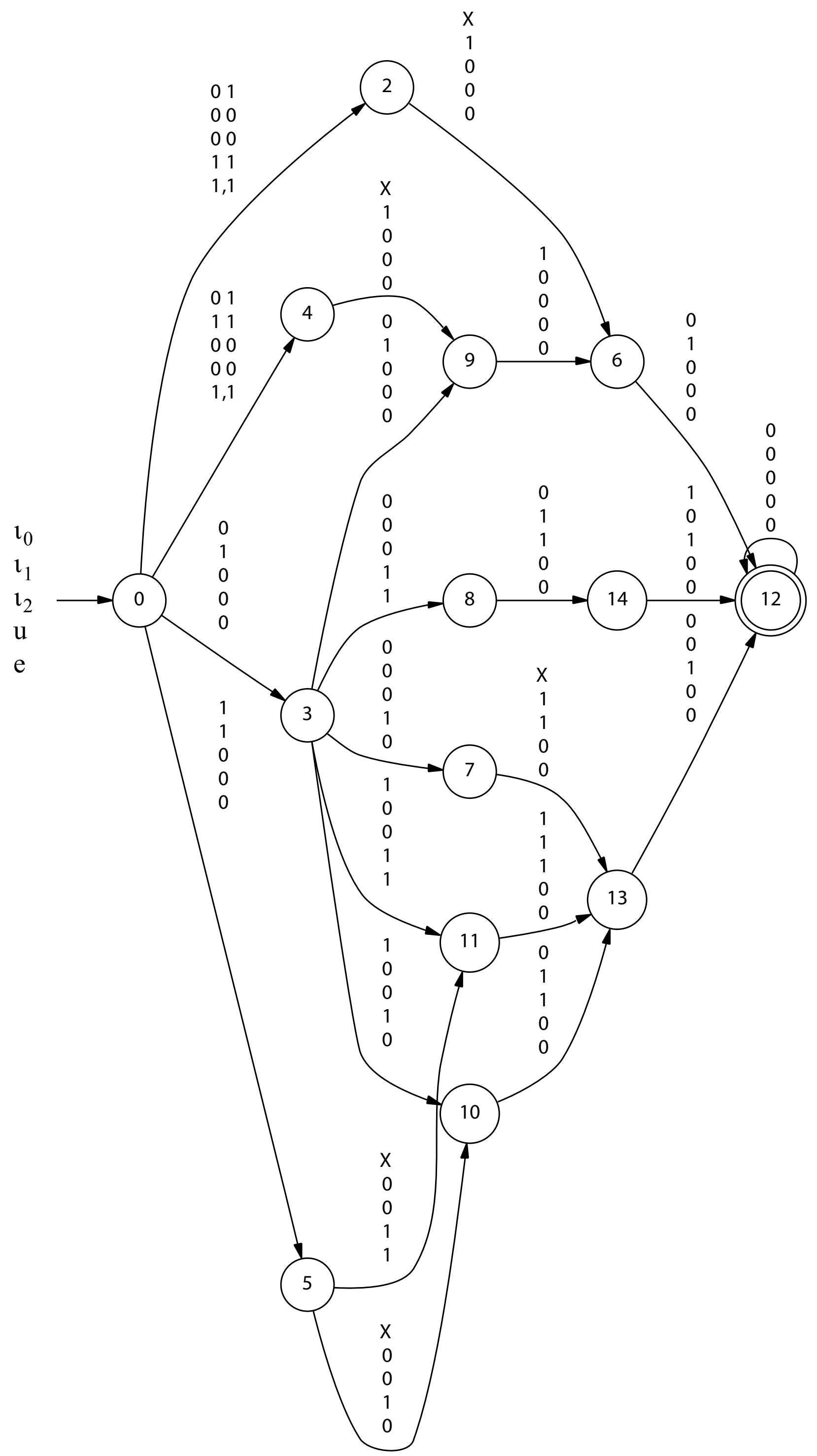


Figure 5.1: DFA recognizing the solutions of the example Presburger formula in Figure 4.2. State 0 is the start state q_0, and the final state denoted with double circles is state 12. A label on each edge is a stack of 0s and 1s corresponding to the binary representations of the non-negative integer values for each free variable ι_0,ι_1,ι_2,u,e (in that order from top to bottom). An X in a stack indicates that either a 0 or 1 is possible in that position.

it less cluttered and easier to read. It is only the accepting paths of a DFA that encode formula solutions; thus, ignoring the states and edges that are not part of any accepting path results in no loss of information.

Recall that the Presburger formula in Figure 4.2 has five free variables: ι_0, ι_1, ι_2, u, and e. For the stacks that label DFA edges, the ordering of binary representations corresponding to the free variable values is not critical to the construction of the DFA. For the DFA in Figure 5.1, the ordering is ι_0, ι_1, ι_2, u, e from top to bottom. Using a different ordering of these free variables produces the same DFA in Figure 5.1, except that the labels reflect the different ordering.

To exemplify the path concept introduced in Section 5.1.1, consider the DFA in Figure 5.1. The accepting paths of the example DFA encode values for the 5-tuple (ι_0,ι_1,ι_2,u,e) that are formula solutions. One path from the start state 0 to state 2, $P_1(0, 2)$, is $[0; 0; 0; 1; 1]$ (where semicolons separate each row of the stack). The path is a stack of letters representing the first letter (*i.e.*, the LSB) of each word encoding a solution for the 5-tuple (ι_0,ι_1,ι_2,u,e). One accepting path of the DFA, $P_4(0, 12)$, is $[0110; 1011; 0000; 0000; 0001]$, which goes through states 4, 9, and 6, proceeding from LSB to MSB. The stack specified by $P_4(0, 12)$ gives the binary representations of the values for the 5-tuple (ι_0,ι_1,ι_2,u,e) that constitute one solution to the formula in Figure 4.2. The first track of $P_4(0, 12)$, 0110, identifies the value corresponding to the first non-negative integer of the tuple (*i.e.*, $\iota_0 = 6$). The second track identifies the values corresponding to the second non-negative integer of the tuple (*i.e.*, $\iota_1 = 11$), and so on. Therefore, the DFA in Figure 5.1 recognizes $\iota_0 = 6$, $\iota_1 = 11$, $\iota_2 = 0$, $u = 0$, and $e = 1$ as one solution to the example formula. To demonstrate the Δ function for identifying one set of states reachable from another set of states, $\Delta(\{6, 12, 13, 14\}) = \{12\}$ in the example DFA of Figure 5.1.

5.1.3 Procedure for Constructing DFAs

The following is the BNF grammar for Presburger arithmetic given in Section 2.4, annotated with a scheme for translating Presburger formulas to DFAs:

$$
\begin{array}{lcll}
\langle \textit{formula} \rangle & ::= & \langle \textit{formula} \rangle \wedge \langle \textit{formula} \rangle & \{\ \langle \textit{DFA} \rangle \cap \langle \textit{DFA} \rangle\ \} \\
 & \mid & \langle \textit{formula} \rangle \vee \langle \textit{formula} \rangle & \{\ \langle \textit{DFA} \rangle \cup \langle \textit{DFA} \rangle\ \} \\
 & \mid & \neg \langle \textit{formula} \rangle & \{\ \langle \overline{\textit{DFA}} \rangle\ \} \\
 & \mid & \exists \textit{var} : \langle \textit{formula} \rangle & \{\ \text{project}(\textit{var}, \langle \textit{DFA} \rangle)\ \} \\
 & \mid & \forall \textit{var} : \langle \textit{formula} \rangle & \{\ \overline{\text{project}(\textit{var}, \langle \overline{\textit{DFA}} \rangle)}\ \} \\
 & \mid & \langle \textit{expression} \rangle \langle \textit{rel-op} \rangle \langle \textit{expression} \rangle & \{\ \langle \textit{LC\_DFA} \rangle\ \}
\end{array}
$$

The base cases are DFAs representing linear equalities and inequalities, denoted $\langle \textit{LC\_DFA} \rangle$ to indicate a DFA representing the solutions of a single linear constraint. Such DFAs are easy to construct from DFAs that describe arithmetic on the binary representation of natural numbers. A finite state machine (FSM) representing the addition of v variables is the basis of all DFA construction for linear equalities and inequalities. An FSM with v states, one for each possible carry value, is sufficient to represent the addition. At any point, the FSM considers the current bit of all v variables $(x_1, \ldots, x_v)$, writing $(k + \sum_{i=1}^{v} x_i) \bmod 2$ to output and moving to the state for carry $\lfloor (k + \sum_{i=1}^{v} x_i)/2 \rfloor$, where k is the carry value at the current state [9].

In handling logical connectives of subformulas, the DFA-construction procedure utilizes closure properties of regular sets under intersection (*i.e.*, $\langle DFA \rangle \cap \langle DFA \rangle$), union (*i.e.*, $\langle DFA \rangle \cup \langle DFA \rangle$), and complementation (*i.e.*, $\overline{\langle DFA \rangle}$) [68]. Existential quantification is handled by projecting the alphabet and the transition function (*i.e.*, $\text{project}(var, \langle DFA \rangle)$). Projection produces a nondeterministic finite automaton, which is followed by determinization and state minimization to get a minimal DFA. Universal quantification exploits the tautology $\forall x \phi \equiv \neg \exists x \neg \phi$.

5.2 Counting and Enumerating Accepting DFA Paths

This section addresses both the counting and enumeration of Presburger formula solutions, given a DFA whose accepting paths encode the solutions to the formula. The method for counting Presburger formula solutions counts the accepting paths of the corresponding DFA, and the method for enumerating solutions is an extension of the counting method. To begin, the key to counting accepting paths is discussed in Section 5.2.1. Section 5.2.2 derives the termination condition for both the counting and enumerating methods, related to DFA path length. Section 5.2.3 presents the algorithm for counting Presburger formula solutions, and Section 5.2.4 presents the algorithm for enumerating Presburger formula solutions.

5.2.1 Treating the DFA as a Graph

The key to the algorithm for counting accepting paths in a DFA is to treat the DFA as a weighted, directed graph. For states p and q, let $wt(p,q) = |\{a \in \Sigma : \delta(p,a) = q\}|$ be the number of alphabet symbols that cause transition from state p to state q. Given DFA $M = (S, \Sigma, \delta, q_0, F)$, define a directed, edge-weighted graph as $G(M) = (V, E, W) = (S, \{(p,q) \mid \exists a \in \Sigma : \delta(p,a) = q\}, \lambda(p,q).wt(p,q))$.

For any choice of path length L, the number of accepting paths of that length in graph $G(M)$ is equivalent to the number of solutions to the Presburger formula represented by DFA M such that the value of each free variable comprising the solution is in the range $[0, 2^L - 1]$. Therefore, the problem of counting the number of solutions represented by a DFA M reduces to a problem of counting the number of accepting paths in the graph $G(M)$. Let $N_i(q) = |P_i(q_0, q)|$ be the number of paths of length i from the vertex q_0 to vertex q. Build up N_i by induction on the path length i as follows.

Theorem 5.1 *For any vertex q and integer $i > 0$, $N_{i+1}(q) = \sum_{e=(p,q)\in E} N_i(p) \cdot W(e)$. If $q = q_0$ then $N_0(q) = 1$, else $N_0(q) = 0$.*
PROOF.

$$\begin{aligned}
\sum_{e=(p,q)\in E} N_i(p) \cdot W(e) &= \sum_{p\in V} N_i(p) \cdot wt(p,q) \\
&= \sum_{p\in V} |\{A = a_1a_2\ldots a_i \in \Sigma^i : \delta^i(q_0, A^R) = p\}| \cdot |\{a_0 \in \Sigma : \delta(p, a_0) = q\}| \\
&= \sum_{p\in V} |\{A' = a_0A = a_0a_1\ldots a_i \in \Sigma^{i+1} : \delta^i(q_0, A^R) = p \ \wedge \ \delta(p, a_0) = q\}| \\
&= |\bigcup_{p\in V} \{A' = a_0A = a_0a_1\ldots a_i \in \Sigma^{i+1} : \delta^i(q_0, A^R) = p \ \wedge \ \delta(p, a_0) = q\}| \\
&= |\{A' = a_0a_1\ldots a_i \in \Sigma^{i+1} : \delta^{i+1}(q_0, A'^R) = q\}| \\
&= N_{i+1}(q).
\end{aligned}$$

The base case of the recurrence is trivial. □

5.2.2 DFA Path Length

It is important to take into account the length of the accepting paths in the DFA. The fundamental reason for this is that the map from values to representations is one-to-many, since any representation of a value may be arbitrarily extended with leading 0s without changing the value that it represents. For example, the DFA in Figure 5.1 recognizes at least two different encodings for the free variable values $\iota_0 = 6$, $\iota_1 = 11$, $\iota_2 = 0$, $u = 0$, and $e = 1$. Two possible paths are $P_4(0,12) = [0110; 1011; 0000; 0000; 0001]$ and $P_5(0,12) = [00110; 01011; 00000; 00000; 00001]$. Despite that these *two* different encodings are recognized by the DFA, they specify only *one* solution to the Presburger formula represented by the DFA. To avoid counting this solution twice, count all solutions identified by accepting paths of the same length (*i.e.*, all values of free variables encoded in a certain number of bits).

In order for counting to be meaningful, the number of solutions to the Presburger formula must be finite. The analysis framework presented here uses Presburger formulas to describe the cache behavior of loop nests that have bounded iteration spaces, and are therefore representable by bounded polytopes. The number of solutions to the cache behavior formulas corresponds to the number of integer points in such polytopes, and the bounded nature of the polytopes ensures a finite number of solutions.

Recall from Section 5.1.1 that accepting paths encode the binary representations of integer values satisfying the Presburger formula represented by the DFA (LSB to MSB). Each integer value has a unique binary representation with the exception of leading 0s. Therefore, a unique formula solution is represented in the DFA by a single accepting path that may be arbitrarily extended with 0s. Let R be the regular expression representing the set of all strings accepted by a DFA that recognizes a Presburger formula P. In order for P to have a finite set of solutions (in the value domain), the Kleene star operator can appear only in limited positions in R.

Lemma 5.1 *Let DFA M recognize Presburger formula P, and let regular expression R represent all accepting paths of M. Formula P has a finite number of solutions if and only if R is of the form 0^*S, where S is a regular expression free of the Kleene star operator.*

Proof. If formula P has a finite number of solutions, then the set of paths accepted by DFA M must represent a finite set of values. The regular expression representing all accepting paths M, R, must represent a finite number of strings, with the exception of any number of 0s at the beginning. Therefore, R is of the form 0^*S, and S must be free of the Kleene star operator.

If regular expression R is of the form 0^*S, then the accepting paths in DFA M are unique except for any number of 0s at the beginning. Therefore, DFA M recognizes a finite set of solution values. □

Lemma 5.1 gives a property of accepting DFA paths that is critical to the algorithm for counting such paths, and it relates the property to the finiteness of Presburger formula solution counts. *Note that this property does not apply to non-accepting paths.* Henceforth, a DFA corresponding to a Presburger formula with a finite number of solutions is called a finite-solution DFA.

Let V_i be the set of vertices reachable from vertex q_0 via paths of length i. To start, q_0 is the only vertex reachable from itself with path length 0 (*i.e.*, $V_0 = \{q_0\}$). In general, the set of vertices reachable from vertex q_0 with path length i is $V_i = \Delta(V_{i-1})$. The following theorem establishes that when $V_i = V_{i-1}$, the sets of vertices reachable from vertex q_0 via paths of length $k \geqslant i$ are identical. In other words, the set of vertices reachable from the starting vertex with paths of a certain length reaches a steady state.

Theorem 5.2 *Given a finite-solution DFA, if $V_i = V_{i-1}$, then $V_k = V_{i-1}$, $\forall k \geqslant i$.*

Proof. By definition, $V_j = \Delta(V_{j-1}), \forall j \geqslant 1$. Applying Δ to both sides of the theorem's premise gives $\Delta(V_i) = \Delta(V_{i-1})$. Applying the definition gives $V_{i+1} = V_i$, and applying the premise gives $V_{i+1} = V_{i-1}$. Now let $V_{i+1} = V_{i-1}$ be the premise and continue as many times as desired to get $V_k = V_{i-1}$, $\forall k \geqslant i$. □

The following theorem relates the condition for the set of vertices reaching a steady state to the number of accepting paths, showing that when $V_i = V_{i-1}$, the number of paths of length i from q_0 to accepting vertices is the same as the number of such paths of length $i-1$.

Theorem 5.3 *Given a finite-solution DFA, if $V_i = V_{i-1}$, then*
$\sum_{q \in F \cap V_i} N_i(q) = \sum_{q \in F \cap V_{i-1}} N_{i-1}(q)$.

Proof. Given the property in Lemma 5.1, every accepting path may have any number of 0s in the MSBs. Therefore, for any vertex q such that $q \in F \cap V_i$, the only edge involving q is (q, q) such that $wt(q, q) = 1$. Finally,

$$\begin{aligned} \sum_{q \in F \cap V_i} N_i(q) &= \sum_{q \in F \cap V_i} \sum_{e=(p,q) \in E} N_{i-1}(p) \cdot W(e) \\ &= \sum_{q \in F \cap V_i} N_{i-1}(q) \cdot wt(q, q) \\ &= \sum_{q \in F \cap V_i} N_{i-1}(q). \end{aligned}$$

□

Given Theorems 5.2 and 5.3, when the set of vertices reaches a steady state ($V_k = V_{i-1}$, $\forall k \geqslant i$), so does the number of accepting paths ($\sum_{q \in F \cap V_k} N_k(q) = \sum_{q \in F \cap V_k} N_{i-1}(q)$, $\forall k \geqslant i$). Notice that set V_k can contain garbage states, but the paths reaching steady state are only the accepting paths. The algorithm for counting accepting paths terminates when $V_i = V_{i-1}$ because it is certain that the number of accepting paths in the DFA has converged.

5.2.3 Counting Accepting Paths

The path-counting algorithm (Algorithm 5.1) counts the number of accepting paths of length L in a directed, edge-weighted graph $G(M)$.

Algorithm 5.1 Counting solutions to Presburger formula.
Input: Finite-solution DFA $M = (S, \Sigma, \delta, q_0, F)$ corresponding to Presburger formula P.
Output: Path length L, number of solutions to formula P such that the value of each free variable is in the range $[0, 2^L - 1]$.
Method:

```
1   Construct the graph $G(M) = (V, E, W)$ from $M$.
2   $V_0 \leftarrow \{q_0\}$
3   $i \leftarrow 0$
4   repeat
5       $i \leftarrow i + 1$
6       $V_i \leftarrow \varnothing$
7       for all $q \in V : \exists p \in V_{i-1} \wedge (p, q) \in E$ do
8           $V_i \leftarrow V_i \cup \{q\}$
9           Calculate $N_i(q)$ using Theorem 5.1.
10      enddo
11  until $V_i = V_{i-1}$
12  $L \leftarrow i - 1$
13  return $L$, $\sum_{q \in F} N_L(q)$
```

For the analysis of Algorithm 5.1, assume a representation of DFA $M = (S, \Sigma, \delta, q_0, F)$ that includes the following components.

1. `states`, a list of all states in the DFA;
2. `final`, a flag for each state p indicating if $p \in F$;
3. `to`, for each state p a list of the states q such that there exists a transition from p to q; and
4. `trans`, a transition table such that an element p, q is the list of alphabet symbols causing transition from state p to state q (note that in general this table is quite sparse).

Notice that Algorithm 5.1 counts the number of accepting paths in a directed graph *without enumerating each accepting path.* Therefore, the cost of the algorithm is sublinear in the number of solutions.

Lemma 5.2 *Algorithm 5.1 counts the number of accepting paths of length L in graph $G(M)$ with worst-case complexity $O(|V|^3)$, where $|V|$ is the number of vertices in $G(M)$.*

Proof. Line 1 of Algorithm 5.1 constructs the directed, edge-weighted graph $G(M)$ from DFA M. The set of vertices V is simply the set of states S. The set of edges E is computed from the `to` lists of all states. The weight matrix W is computed from the table `trans`. Element $W(i, j)$ is the count of all alphabet symbols in the list at `trans`(i, j). The cost of line 1 is $O(|V| + |E|)$.

Lines 7–10 of Algorithm 5.1 consider all vertices q such that there is an edge from a vertex $p \in V_{i-1}$ to vertex q. This step does not require enumeration of all vertices $q \in V$ to check if $(p, q) \in E$. Instead, the `to` list of each vertex $p \in V_{i-1}$ gives all such vertices q. The complexity of lines 7–10 is $O(|E|)$, and the cost clearly depends on the sparsity of table `trans`. At worst there is an edge from every vertex p to every vertex q, making the worst-case complexity of lines 7–10 $O(|V|^2)$. In general, the cost of computation is much smaller.

Finally, line 13 of Algorithm 5.1 requires only a query to the `final` flag of each state, and its cost is $O(|V|)$. The complexity of the entire algorithm depends on the number of times lines 4–11 repeat, which is indicated by the output parameter L. Therefore, the complexity of Algorithm 5.1 is output-sensitive, which is as expected since the structure of $G(M)$ depends on the solutions recognized by M. At worst, one or more accepting paths pass through all vertices of $G(M)$, making $O(|V|)$ the upper bound on L. The bound on the complexity of the entire algorithm is $O(|E| \cdot L)$, which is at worst $O(|V|^3)$. □

Applying the path-counting algorithm (Algorithm 5.1) to the DFA in Figure 5.1, the number of accepting paths is 18 and the path length L is 4.

5.2.4 Enumerating Accepting Paths

Recall that the analysis framework presented in this dissertation determines the cache behavior of a memory access by counting the number of witnesses to the access and comparing the witness count with an associativity value $\mathcal{A}$. To count the witnesses for each memory access, it is necessary to enumerate the solutions of the witness formula. This section presents an extension of the path-counting algorithm (Algorithm 5.1) that enumerates the solutions of a Presburger formula by enumerating the accepting paths in the DFA recognizing the solutions of the formula.

The enumeration algorithm (Algorithm 5.2) enumerates the free variable values encoded by all accepting paths of length L in DFA M, where L is the path length at which the set of states converges (*i.e.*, $S_k = S_L, \forall k > L$). *This algorithm does not depend on the type of cache behavior expressed by the formula.*

Algorithm 5.2 Enumerating solutions to Presburger formula.
Input: Finite-solution DFA $M = (S, \Sigma, \delta, q_0, F)$ corresponding to Presburger formula P.
Output: Path length L, set of solutions to formula P with f free variables (i.e., a set of f-vectors such that the value of each free variable is in the range $[0, 2^L - 1]$).
Method:

```
1   $S_0 \longleftarrow \{q_0\}$
2   for all $p \in S$ do
3       $T_{0,p} \longleftarrow \{[0\ 0 \ldots 0]\}$
4   enddo
5   $i \longleftarrow 0$
6   repeat
7       $i \longleftarrow i + 1$
8       $S_i \longleftarrow \varnothing$
9       for all $q \in S : \exists p \in S_{i-1} \wedge \exists a \in \Sigma : \delta(p, a) = q$ do
10          $S_i \longleftarrow S_i \cup \{q\}$
11          $T_{i,q} \longleftarrow \varnothing$
12          for all $a \in \Sigma : \exists \delta(p, a) = q$ do
13              for all $t \in T_{i-1,p}$ do
14                  $T_{i,q} \longleftarrow T_{i,q} \cup \{t + a * 2^{i-1}\}$
15              enddo
16          enddo
17      enddo
18  until $S_i = S_{i-1}$
19  $L \longleftarrow i - 1$
20  return $L$, $T_{L,q \in F}$
```

Let S_i be the set of states reachable from the start state via paths of length i (*i.e.*, $S_i = \Delta(S_{i-1})$). For a finite-solution DFA, when the set of states reachable from the start state converges (*i.e.*, $S_k = S_{i-1}, \forall k \geqslant i$), the number of accepting paths does as well (see Theorem 5.3). This fact is used as a condition for terminating the enumeration.

Recall that an alphabet symbol $a \in \Sigma$ is a stack of f 0s and 1s, where f is the number of free variables. Let an f-vector, a vector of f non-negative integers, contain the free variable values encoded by a single accepting path. In Algorithm 5.2, each element of the two-dimensional table T is a set of f-vectors. Element $T_{i,p}$ corresponds to the set of free variable values encoded by all paths of length i from vertex q_0 to vertex p. To begin, line 3 assigns to $T_{0,p}$ a set containing a single f-vector of all 0s, for all states p. The output of Algorithm 5.2 is a set of f-vectors, where each f-vector contains values of the f free variables comprising a solution. Because DFA M is minimal,[3] each solution encoded by an accepting path of M is unique.

The enumeration algorithm (Algorithm 5.2) is similar to the path-counting algorithm (Algorithm 5.1). Both terminate when the set of states (or vertices) reachable from q_0 reaches steady state, because it is shown that the enumerations (or number) of accepting paths converges at that point. Unlike Algorithm 5.1, Algorithm 5.2 must consider the value of each alphabet symbol causing transition among states (not just the number of such alphabet symbols). Therefore, the DFA may neither be treated simply as a directed, edge-weighted graph nor experience the cost benefits of ignoring the values of alphabet symbols. Both the worst-case and observed complexity of Algorithm 5.2 demonstrates this.

Lemma 5.3 *Algorithm 5.2 enumerates the free variable values encoded by all accepting paths of length L in DFA M with worst-case complexity* $O((|S| \cdot 2^f \cdot f)^{|S|} \cdot 2^f \cdot |S|^3)$*, where* $|S|$ *is the number of states in M and f is the number of free variables.*

Proof. Lines 9–17 of Algorithm 5.2 consider all states q such that there is a transition from a state $p \in S_{i-1}$ to state q. The `to` list of each state $p \in S_{i-1}$ (of the DFA representation outlined in Section 5.2.3) gives all such states q. In the worst case, it is possible to transition from each state in the DFA to all states in the DFA, making it possible for lines 9–17 to iterate $|S|^2$ times. In general, this quantity is much less.

Lines 12–16 examine all alphabet symbols causing transition from state p to state q. Element (p, q) of the `trans` table gives the list of such symbols. The number of which is $|\Sigma|$ at worst. Note that $|\Sigma| = 2^f$.

Lines 13–15 add to the values encoded by paths of length $i-1$ from q_0 to state p to compute the values of encoded by paths of length i from q_0 to state q. The multiplication in line 14 is an f-vector multiply. In the worst case, the number of paths of length $i-1$ from q_0 to state p is $(|S| \cdot |\Sigma|)^i$. The worst-case complexity of lines 13–15 is $(|S| \cdot |\Sigma| \cdot f)^i$.

The cost of the entire algorithm depends on the number times lines 6–18 repeat, which is

[3]The analysis framework presented here uses automata-construction tools [9, 10, 81] that generate the minimal DFA representation of a given Presburger formula.

indicated by the output parameter L. At worst, one or more accepting paths pass through all states of DFA M, making $O(|S|)$ the upper bound on the value of L. The bound on the complexity of the entire algorithm is $O((|S|{\cdot}|\Sigma|{\cdot}f)^L{\cdot}|\Sigma|{\cdot}|S|^2{\cdot}L)$, which is $O((|S|{\cdot}2^f{\cdot}f)^{|S|}{\cdot}2^f{\cdot}|S|^3)$ at worst. As in the case of the path-counting algorithm (Algorithm 5.1), this bound represents an *extreme* worst case, and in general, the cost of enumeration is much less. $\square$

Using the enumeration algorithm (Algorithm 5.2) to enumerate the solutions encoded by the DFA in Figure 5.1, which represents the i-witnesses of the running example loop nest $\mathbb{L}_{\mathrm{mm}}$ in Figure 2.5 on page 17 for cache set 0, gives $L = 4$ and

$$T_{L,q\in F} = \left\{ \begin{array}{cccccccccccccccccc} 0 & 0 & 1 & 1 & 2 & 2 & 3 & 3 & 4 & 4 & 4 & 5 & 5 & 6 & 6 & 7 & 7 & 8 \\ 5 & 6 & 5 & 6 & 5 & 6 & 5 & 6 & 5 & 11 & 11 & 5 & 11 & 5 & 11 & 5 & 11 & 5 \\ 12, & 0, & 12, & 0, & 12, & 0, & 12, & 0, & 12, & 0\ , & 0\ , & 12, & 0\ , & 12, & 0\ , & 12, & 0\ , & 12 \\ 2 & 1 & 2 & 1 & 2 & 1 & 2 & 1 & 2 & 0 & 0 & 2 & 0 & 2 & 0 & 2 & 0 & 2 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 2 & 1 & 2 & 1 & 2 & 1 & 2 \end{array} \right\},$$

where $T_{L,q\in F}$ is the set of satisfying values for free variables ι_0, ι_1, ι_2, u, and e.

5.3 Counting Witnesses to Determine Cache Behavior

Recall that a witness formula describes the witnesses of all memory accesses in a loop nest and each solution to the formula expresses a memory access and one of its witnesses (or its only witness). Given an enumeration of the witness formula solutions, it is then necessary to collect the solutions describing a particular memory access and count the number of such solutions expressing unique witnesses for the access. Finally, comparison of a memory access's witness count with associativity $\mathcal{A}$ determines the cache event outcome of the access.

This section discusses how to count the number of witnesses for each memory access and how to compare such witness counts with associativity values to determine cache event outcomes. Section 5.3.1 presents the algorithm for counting witnesses to each memory access, given an enumeration of the witness formula solutions. Section 5.3.2 explains how to interpret i-witness and r-witness counts to determine the number of interior misses and replacement misses incurred, respectively. Section 5.3.3 explains how to use the b-witnesses to determine the boundary misses incurred and how use to the s-witnesses to update cache state. The key ideas of this section are counting witnesses and using the theorems stated in Section 4.4 to determine cache behavior.

5.3.1 Counting Witnesses

Each solution to the witness formulas of Section 4.3.2 consists of the free variables that represent a memory access (loop control variables and array reference number) and the free variable to represent a memory block that is a witness to the access. Recall that a wraparound value is sufficient to designate the memory block. The free variable that is the wraparound value is called the *e-value*. Recall that the enumeration algorithm (Algorithm 5.2) enumerates the solutions represented by a minimal DFA. The minimality of the DFA is a result of the automata-construction tools used by the analysis framework presented here. As a result, each enumerated solution is unique, which ensures that the e-values associated with each memory access are distinct.[4] Once Algorithm 5.2 has enumerated all solutions, it is necessary and sufficient to count the number of e-values for each memory access. Counting e-values is necessary because each e-value associated with a particular memory access represents a witness to the access, and the e-value count for each memory access is equivalent to its witness count. Counting e-values is sufficient because the e-values associated with a particular memory access are guaranteed distinct (due to the minimality of the DFA), and there is no need to identify and compare the actual e-values.

The witness-counting algorithm (Algorithm 5.3) counts e-values (and thus, witnesses), by grouping together memory accesses with the same witness count and reporting the number of such accesses for each witness count. *This algorithm does not depend on the type of witnesses being counted.*

Algorithm 5.3 Counting the number of witnesses to each memory access.
Input: Set of solutions X to the original Presburger formula.
Output: The maximum e-value count $emax$ and the histogram N of the number of memory accesses with i distinct e-values for each $i \in [1, emax]$.
Method:

```
$emax \leftarrow 1$
for all $x \in X$ do
    $k \leftarrow \text{Key}(x)$
    $h \leftarrow \text{Find}(k, H)$
    if $h = \text{NULL}$
        Insert $\langle k, 1 \rangle$ into $H$
    else
        $h.\text{val} \leftarrow h.\text{val} + 1$
        $emax \leftarrow \max(emax, h.\text{val})$
    endif
enddo
for all $i = 1$ to $emax$ do
    $N_i \leftarrow 0$
enddo
for all $h \in H$ do
    $N_{h.\text{val}} \leftarrow N_{h.\text{val}} + 1$
enddo
return $emax$, $N$
```

Algorithm 5.3 builds a histogram of witness counts and requires two steps. First, the set of enumerated solutions X condenses to a list of key-value pairs H. For $h \in H$, h.key identifies

[4]In contrast, if the DFA was non-minimal, its enumerated solutions would not be guaranteed unique, and it would not be safe to assume that the e-values associated with each memory access are distinct.

a memory access, and h.val counts the number of distinct e-values for the access. Given $x \in X$, let Key(x) return the associated key. For the example set of solutions enumerated in Section 5.2.4, solution $x = [0; 5; 12; 2; 0]$ has Key$(x) = [0; 5; 12; 2]$. It is not necessary to keep track of the actual e-values because they are guaranteed to be distinct. Let Find(k, H) be a function that returns $h = \langle k, v\rangle$ if H contains h and returns NULL otherwise. The maximum e-value count, $emax$, is dynamically updated during the process of building H. In the analysis framework of this dissertation, H is implemented using hashing so that maintaining, updating, and searching H can be expected to require constant time. Second, the value fields of the entries of H are used to build a histogram N.

The complexity of Algorithm 5.3 is $O(f \cdot |X|)$, where $|X|$ is the size of the solution set for which witnesses are counted and each key has f components.[5] The result of Algorithm 5.3 is a histogram of the number of memory accesses with i distinct e-values (*i.e.*, witnesses) for each value of i from 1 to $emax$. For all $i > emax$, the number of memory accesses with i distinct e-values is 0.

5.3.2 Counting Interior and Replacement Misses

Based on Theorem 4.1, which states that a memory access is an interior miss if and only if there are $\mathcal{A}$ or more i-witnesses to the access, the following gives the number of memory accesses that incur an interior miss for associativity value $\mathcal{A}$, int-misses($\mathcal{A}$), given the output of the witness-counting algorithm (Algorithm 5.3) for i-witnesses.

$$\text{int-misses}(\mathcal{A}) = \sum_{i=\mathcal{A}}^{emax} N_i \tag{5.1}$$

Using the witness-counting algorithm (Algorithm 5.3) to count the number of i-witnesses for the memory accesses encoded by the DFA in Figure 5.1 gives $emax = 2$, $N_1 = 16$, and $N_2 = 1$ (*i.e.*, the maximum e-value count is 2, there are sixteen memory accesses with one witness, and there is one memory access with two witnesses). Therefore, int-misses(1) = 17, int-misses(2) = 1, and int-misses($\mathcal{A}$) = 0 for all $\mathcal{A} \geqslant 3$. Loop nest $\mathbb{L}_{\text{mm}}$ in Figure 2.5 incurs seventeen interior misses in set 0 of a (1, 32, 4096; 128) cache, one interior miss in set 0 of a (2, 32, 8192; 128) cache, and no interior misses in set 0 of all $\{(k, 32, 4096k; 128)\colon\ k \geqslant 3\}$ caches.

The DFA recognizing the r-witnesses of the running example loop nest $\mathbb{L}_{\text{mm}}$ for cache set 0 (*i.e.*, the solutions of the Presburger formula in Figure A.1) is given in Figure A.4. Using the enumeration algorithm (Algorithm 5.2) to enumerate the solutions encoded by the DFA in Figure A.4 gives $L = 4$ and

$$T_{L,q\in F} = \left\{ \begin{array}{ccccccccccccccc} 0 & 1 & 1 & 2 & 2 & 3 & 3 & 4 & 5 & 5 & 6 & 6 & 7 & 7 & 8 \\ 6 & 5 & 6 & 5 & 6 & 5 & 6 & 5 & 5 & 11 & 5 & 11 & 5 & 11 & 5 \\ 0, & 12, & 0, & 12, & 0, & 12, & 0, & 12, & 12, & 0\,, & 12, & 0\,, & 12, & 0\,, & 12 \\ 1 & 2 & 1 & 2 & 1 & 2 & 1 & 2 & 2 & 0 & 2 & 0 & 2 & 0 & 2 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 2 & 1 & 2 & 1 & 2 & 1 & 2 \end{array} \right\}.$$

[5]The original formula contains f free variables.

Based on Theorem 4.2, which states that a memory access is a replacement miss if and only if there are $\mathcal{A}$ or more r-witnesses to the access, the following gives the number of memory accesses that incur a replacement miss for associativity value $\mathcal{A}$, repl-misses($\mathcal{A}$), given the output of the witness-counting algorithm (Algorithm 5.3) for r-witnesses.

$$\text{repl-misses}(\mathcal{A}) = \sum_{i=\mathcal{A}}^{emax} N_i \tag{5.2}$$

Using the witness-counting algorithm (Algorithm 5.3) to count the number of r-witnesses for the memory accesses encoded by the DFA in Figure A.4 gives $emax = 1$ and $N_1 = 15$. Therefore, int-misses(1) $= 15$ and int-misses($\mathcal{A}$) $= 0$ for all $\mathcal{A} \geqslant 2$. Loop nest $\mathbb{L}_{\text{mm}}$ incurs fifteen interior misses in set 0 of a (1, 32, 4096; 128) cache and no interior misses in set 0 of all $\{(k,\ 32,\ 4096k;\ 128)\colon\ k \geqslant 2\}$ caches.

5.3.3 Counting Boundary Misses and Updating Cache State

To enumerate b-witnesses and s-witnesses, it is necessary to identify (not just count) the memory accesses with a certain number of witnesses. Comparison of these memory accesses with the cache state at the beginning of program fragment execution resolves whether potential boundary misses are indeed misses and determines the cache state at the end of program fragment execution. Furthermore, in order to determine boundary misses and cache state, it is necessary to identify the *earliest* and *latest* memory accesses to map to each cache set (in the total ordering of accesses, $\lhd$). The earliest access has zero b-witnesses and is a potential boundary miss. The latest access has zero s-witnesses and is in the cache state at the end of program fragment execution. For Presburger formulas describing such accesses, see Section 4.3.2.

Let Z_s^i be the set of memory accesses in A_s with witness count i. In fact, for each cache set s, set Z_s^i for b-witnesses contains at most one access and set Z_s^i for s-witnesses contains at most one access (both properties are proven below). That is, no two accesses in A_s with non-empty b-neighborhoods may have the same number of b-witnesses. Similarly, no two accesses in A_s with non-empty s-neighborhoods may have the same number of s-witnesses.

Theorem 5.4 *There is exactly one memory access in A_s with* b-*witness count i, and the value of i ranges from 0 to one less than the total number of memory blocks accessed by program fragment $\mathbb{F}$ that map to cache set s.*

PROOF. Any memory access a that has a non-empty b-neighborhood (*i.e.*, a is the earliest access to the memory block that it touches) has a b-witness count $i \geqslant 0$. Let E_s be the set of such accesses in A_s. By definition, all accesses in E_s touch distinct memory blocks. There is some access $\mathsf{a}_1 \in \mathsf{E}_s$ that is the earliest in the total order of all accesses executed by program fragment $\mathbb{F}$. Recall that a memory block b is a b-witness to access a if b is touched by an access in the b-neighborhood of a and b is distinct from the memory block touched by a. Access a_1 has 0 b-witnesses because there are no accesses in A_s between a_1 and the beginning of fragment $\mathbb{F}$ (*i.e.*, there are no accesses in the b-neighborhood of a_1). Similarly, there is some access $\mathsf{a}_2 \in \mathsf{E}_s - \{\mathsf{a}_1\}$ that is the earliest. Access a_2 has one b-witness because only access a_1 occurs between a_2 and the beginning of fragment $\mathbb{F}$. In general, there is some access $\mathsf{a}_k \in \mathsf{E}_s - \{\mathsf{a}_1, \ldots, \mathsf{a}_{k-1}\}$ that is the earliest. Access a_k has $k-1$ b-witnesses because accesses $\mathsf{a}_1, \ldots, \mathsf{a}_{k-1}$ occur between a_k and the beginning of fragment $\mathbb{F}$. If $|\mathsf{E}_s - \{\mathsf{a}_1, \ldots, \mathsf{a}_{k-1}\}| = 1$, then $|\mathsf{E}_s| = k$ and a_k has the largest b-witness count $k-1$.

Let x be the total number of memory blocks accessed by fragment $\mathbb{F}$ that map to cache set s. Because E_s is the set of accesses in A_s that are the earliest among all accesses executed by fragment $\mathbb{F}$ to touch their respective memory blocks, the size of E_s is x. Therefore, for b-witness counts 0 to $x-1$, there is exactly one access in A_s with that number of b-witnesses. $\square$

The fact that an access has a b-witness count i does not imply that it is a boundary miss or even a potential boundary miss. As Section 4.4 explains, if $i \geqslant \mathcal{A}$, then a is an interior miss for a given associativity value $\mathcal{A}$.

Theorem 5.5 *There is exactly one memory access in A_s with* s*-witness count i, and the value of i ranges from 0 to one less than the total number of memory blocks accessed by program fragment $\mathbb{F}$ that map to cache set s.*

Proof. The proof is the same as for Theorem 5.4 with b-witness, b-neighborhood, and earliest replaced by s-witness, s-neighborhood, and latest, respectively. $\square$

To count boundary misses and update cache state, it is necessary to identify the memory accesses with certain witness counts. Given the enumeration of witness formula solutions, The witness-identifying algorithm (Algorithm 5.4) identifies the set of memory accesses with i distinct e-values (*i.e.*, witnesses) for each value of i from 0 to $emax$. As established by Theorems 5.4 and 5.5, for either b-witnesses or s-witnesses, the set of memory accesses with i witnesses contains one memory access. The maximum e-value count, $emax$, is dynamically updated. Let Earliest(s) return the earliest memory accesses mapping to cache set s. Let Latest(s) return the latest memory accesses mapping to cache set s. In the case of b-witnesses, let W_i be the result of Earliest(i). In the case of s-witnesses, let W_i be the result of Latest(i). Let H be a list of key-value pairs as in the witness-counting algorithm (Algorithm 5.3). Let Set(key) be a function that returns the cache set number corresponding to the memory access represented by the key. The result of Algorithm 5.4 is the sets of memory accesses with i distinct e-values, Z_s^i, for each $i \in [0, emax]$ and $s \in [0, \mathcal{S}]$. For all $i > emax$, the set of memory accesses with i distinct e-values is empty. The complexity of the entire algorithm is $O(f \cdot |X|)$, where $|X|$ is the size of the solution set for which witnesses are identified and each key has f components. Notice that the witness-identifying algorithm (Algorithm 5.4) is similar to the witness-counting algorithm (Algorithm 5.3), but witnesses are being identified instead of just being counted.

Algorithm 5.4 Identifying memory accesses with i witnesses.
Input: Set of solutions X to the original Presburger formula.
Output: The maximum e-value count $emax$ and the sets of memory accesses with i distinct e-values Z_s^i, for each $i \in [0, emax]$ and $s \in [0, \mathcal{S}]$.
Method:

```
emax ← 1
for all $x \in X$ do
    $k$ ← Key($x$)
    $h$ ← Find($k, H$)
    if $h$ = NULL
        Insert $\langle k, 1 \rangle$ into $H$
    else
        $h$.val ← $h$.val + 1
        $emax$ ← max($emax, h$.val)
    endif
enddo
for all $i = 0$ to $emax$ do
    for all $s = 0$ to $\mathcal{S}$ do
        $Z_s^i$ ← $\varnothing$
    enddo
enddo
for all $h \in H$ do
    $s$ ← Set($h$.key)
    $Z_s^{h.\mathrm{val}}$ ← $Z_s^{h.\mathrm{val}} \cup \{h.\mathrm{key}\}$
enddo
for all $s = 0$ to $\mathcal{S} - 1$ do
    $Z_s^0$ ← $Z_s^0 \cup \{W_s\}$
enddo
return $emax$, $Z$
```

Boundary Misses

Based on Theorem 4.3, which states that a memory access is a potential boundary miss if and only if there are fewer than $\mathcal{A}$ b-witnesses to the access, the following gives the collection of memory accesses that are potential boundary misses in cache set s for associativity value $\mathcal{A}$, given the output of the witness-identifying algorithm (Algorithm 5.4) for b-witnesses.

$$\text{potential-bnd-misses}(\mathcal{A}) = \bigcup_{i=0}^{\min(\mathcal{A}-1, emax)} Z_s^i \tag{5.3}$$

To determine which of the potential boundary misses are actual cache misses (*i.e.*, boundary misses), it is necessary to consider the cache state at the beginning of program fragment execution.

Recall from Section 2.1.1 that the state of cache set s, $\mathfrak{C}\langle s\rangle$, is the collection of memory blocks residing in s at any point during the execution of a program, ordered by recency of access. It is sufficient to represent each memory block in $\mathfrak{C}\langle s\rangle$ with its wraparound value. The function Size($\mathfrak{C}\langle s\rangle$) returns the number of memory blocks in $\mathfrak{C}\langle s\rangle$. Let $\mathfrak{C}\langle s\rangle$ be extended to represent the state of cache set s for multiple associativity values, which is possible since the ordering of memory blocks by recency of accesess (*i.e.* due to the LRU replacement algorithm) has the stack inclusion property [92].[6] For any particular $\mathcal{A}$, the cache state of set s is the first n memory blocks of $\mathfrak{C}\langle s\rangle$, where $n = \min(\mathcal{A}, \text{Size}(\mathfrak{C}\langle s\rangle))$. For any $\mathcal{A} > \text{Size}(\mathfrak{C}\langle s\rangle)$, cache set s contains $\mathcal{A} - \text{Size}(\mathfrak{C}\langle s\rangle)$ unoccupied frames.

For example, consider $\mathfrak{C}\langle s\rangle = \{0, 10, 4\}$, where the first memory block (with wraparound value 0) is the most-recently accessed, and the last memory block (with wraparound value 4) is the least-recently accessed. Size($\mathfrak{C}\langle s\rangle$) returns 3. For a cache with $\mathcal{A} = 1$, cache set s contains one memory block represented by the pair $(s, 0)$. For a cache with $\mathcal{A} = 2$, set s contains two memory blocks represented by the pairs $(s, 0)$ and $(s, 10)$. For a cache with $\mathcal{A} \geqslant 3$, set s contains three memory blocks represented by the pairs $(s, 0)$, $(s, 10)$, and $(s, 4)$. For caches with $\mathcal{A} > 3$, set s contains $\mathcal{A} - 3$ unoccupied frames.

Recall the *Resolve* function from Section 3.2, which returns the collection of boundary misses incurred by a program fragment in set s given the cache state of s and the collection of potential boundary misses incurred in s. The resolving algorithm (Algorithm 5.5) for resolving potential boundary misses with cache state illustrates the operation of *Resolve*. The algorithm takes as input the cache state of set s at the beginning of program fragment execution, $\mathfrak{C}_s^{\text{in}}$, and the collection of memory accesses with i distinct b-witnesses for cache set s, Z_s^i. The algorithm gives as output the number of boundary misses for associativity $\mathcal{A} = i$, B_i, where i ranges from 1 to $emax' = \max(\text{Size}(\mathfrak{C}_s^{\text{in}}), emax)$. Function Add($\mathfrak{C}_s^{\text{in}}, w$) adds the memory block with wraparound value w to the state of cache set s as the most recently accessed block (*i.e.*, w goes at the beginning of $\mathfrak{C}_s^{\text{in}}$). Function Delete($\mathfrak{C}_s^{\text{in}}, i$) deletes the i^{th} most recently accessed block from $\mathfrak{C}_s^{\text{in}}$. Function GetItem($\mathfrak{C}_s^{\text{in}}, i$) returns the i^{th} most recently accessed block in $\mathfrak{C}_s^{\text{in}}$. Function Wraparound($\mathsf{a}$) returns the wraparound value associated with the memory block touched by access a. The complexity of the entire algorithm is $O(emax \cdot |Z_s^i| \cdot (emax + \text{Size}(\mathfrak{C}\langle s\rangle)))$, where the value of $emax$ is at most the number of wraparounds in the memory footprint of the loop nest (designated W) and the value of $|Z_s^i|$ is 1 (as indicated by Theorems 5.4 and 5.5). In the worse case, the cost of Algorithm 5.5 is $O(W^2 + W \cdot \text{Size}(\mathfrak{C}\langle s\rangle))$. For caches with associativity values $\mathcal{A} > emax'$, the number of boundary misses is $B_{emax'}$. To resolve all potential boundary misses, Algorithm 5.5 is performed for every set $s = 0$ to $\mathcal{S} - 1$.

[6]The inclusion property states that, at a given time, the contents of a memory of size k frames is a subset of the contents of memory of size $k + 1$ frames.

Algorithm 5.5 Resolving Potential Boundary Misses with Cache State.

Input: The cache state of set s at the beginning of program fragment execution $\mathfrak{C}_s^{\text{in}}$, the maximum b-witness count $emax$, the set of memory accesses with i distinct b-witnesses Z_s^i, for each $i \in [0, emax]$

Output: The number of actual boundary misses for cache set s in a cache with $\mathcal{A} = i$, B_i, for each $i \in [1, emax']$.

Method:

```
$emax' \leftarrow \text{Size}(\mathfrak{C}_s^{\text{in}}) + emax$
for all $i = 1$ to $emax'$ do
    $B_i \leftarrow 0$
enddo
for all $i = 0$ to $emax$ do
    for all $z \in Z_s^i$ do
        if $\text{Size}(\mathfrak{C}_s^{\text{in}}) = 0$
            $\text{Add}(\mathfrak{C}_s^{\text{in}}, \text{Wraparound}(z))$
            for all $k = i + 1$ to $emax'$ do
                $B_k \leftarrow B_k + 1$
            enddo
        else
            for all $j = 1$ to $\text{Size}(\mathfrak{C}_s^{\text{in}})$ do
                $t \leftarrow \text{GetItem}(\mathfrak{C}_s^{\text{in}}, j)$
                if $t = \text{Wraparound}(z)$
                    $\text{Delete}(\mathfrak{C}_s^{\text{in}}, j)$
                    $\text{Add}(\mathfrak{C}_s^{\text{in}}, t)$
                    break
                else if $j = \text{Size}(\mathfrak{C}_s^{\text{in}})$
                    $\text{Add}(\mathfrak{C}_s^{\text{in}}, \text{Wraparound}(z))$
                    for all $k = \max(j + 1, i + 1)$ to $emax'$ do
                        $B_k \leftarrow B_k + 1$
                    enddo
                endif
                if $j > i$
                    $B_j \leftarrow B_j + 1$
                endif
            enddo
        endif
    enddo
enddo
return $B$
```

The DFA recognizing the b-witnesses of the running example loop nest $\mathbb{L}_{\mathrm{mm}}$ in Figure 2.5 for cache set 0 (*i.e.*, the solutions of the Presburger formula in Figure A.2) is given in Figure A.5. Using the enumeration algorithm (Algorithm 5.2) to enumerate the solutions encoded by the DFA in Figure A.5 gives $L = 4$ and

$$T_{L,q\in F} = \left\{ \begin{smallmatrix} 4 \\ 11 \\ 0 \\ 0 \\ 0 \end{smallmatrix}, \begin{smallmatrix} 0 \\ 5 \\ 12 \\ 2 \\ 0 \end{smallmatrix}, \begin{smallmatrix} 4 \\ 11 \\ 0 \\ 0 \\ 1 \end{smallmatrix} \right\},$$

where $T_{L,q\in F}$ is the set of satisfying values for free variables ι_0, ι_1, ι_2, u, and e. In the running example, Earliest(0) returns memory access $(0, 0, 0, R_1)$. Using the witness-identifying algorithm (Algorithm 5.4) to identify the memory accesses with certain b-witness counts for loop nest $\mathbb{L}_{\mathrm{mm}}$ in cache set 0 gives $emax = 2$ and

$$Z_0^0 = \left\{ \begin{smallmatrix} 0 \\ 0 \\ 0 \\ 1 \end{smallmatrix} \right\}, \qquad Z_0^1 = \left\{ \begin{smallmatrix} 0 \\ 5 \\ 12 \\ 2 \end{smallmatrix} \right\}, \qquad Z_0^2 = \left\{ \begin{smallmatrix} 4 \\ 11 \\ 0 \\ 0 \end{smallmatrix} \right\},$$

where Z_0^i is the set of memory accesses in A_0 with i distinct e-values for each $i \in [0, 2]$. Suppose that the cache state of set 0 at the beginning of program fragment execution is $\mathfrak{C}_0^{\mathrm{in}} = \{0, 2, 1\}$. Given the result of the witness-identifying algorithm (Algorithm 5.4) for the running example, the resolving algorithm (Algorithm 5.5) returns $emax' = 5$, $B_1 = 0$, $B_2 = 1$, and $B_i = 0, \forall 3 \leqslant i \leqslant 5$. Therefore, loop nest $\mathbb{L}_{\mathrm{mm}}$ incurs no boundary misses in set 0 of a (1, 32, 4096; 128) cache, one boundary miss in set 0 of a (2, 32, 8192; 128) cache, and no boundary misses in set 0 of all $\{(k,$ 32, 4096k; 128): $k \geqslant 3\}$ caches. It may seem strange that a cache with larger associativity and capacity ($\mathcal{A} = 2$, $\mathcal{C} = 8192$) incurs more boundary misses than a cache with smaller associativity and capacity ($\mathcal{A} = 1$, $\mathcal{C} = 4096$) in cache set 0. For $\mathcal{A} = 1$, there is one potential boundary miss that is actually a cache hit. For $\mathcal{A} = 2$, there are two potential boundary misses; one is a cache hit and one is a boundary miss. Furthermore, the memory access that is a boundary miss for $\mathcal{A} = 2$ is still a miss for $\mathcal{A} = 1$—it is an interior miss.

Cache State

In determining the cache state after execution of a program fragment, the objective is to appropriately update the representation of cache state before execution of the program fragment, $\mathfrak{C}_s^{\mathrm{in}}$. The updating algorithm (Algorithm 5.6) determines the cache state of set s at the end of program fragment execution, $\mathfrak{C}_s^{\mathrm{out}}$. The complexity of the entire algorithm is $O(emax \cdot \mathrm{Size}(\mathfrak{C}\langle s\rangle))$, where the value of $emax$ is at most the number of wraparounds in the memory footprint of the loop nest (designated W). In the worse case, the cost of Algorithm 5.6 is $O(W \cdot \mathrm{Size}(\mathfrak{C}\langle s\rangle))$. Algorithm 5.6 is performed for every set $s = 0$ to $\mathcal{S} - 1$.

Algorithm 5.6 Updating Cache State.
Input: The cache state of set s at the beginning of program fragment execution $\mathfrak{C}_s^{\text{in}}$, the maximum s-witness count $emax$, the set of memory accesses with i distinct s-witnesses Z_s^i, for each $i \in [0, emax]$
Output: The cache state of set s at the end of program fragment execution $\mathfrak{C}_s^{\text{out}}$.
Method:

```
$\mathfrak{C}_s^{\text{out}} \leftarrow \mathfrak{C}_s^{\text{in}}$
**for all** $i = emax$ to 0 **do**
    **for all** $z \in Z_{SET}^i$ **do**
        **if** Size($\mathfrak{C}_s^{\text{out}}$) = 0
            Add($\mathfrak{C}_s^{\text{out}}$, Wraparound($z$))
        **else**
            **for all** $j = 1$ to Size($\mathfrak{C}_s^{\text{out}}$) **do**
                $t \leftarrow$ GetItem($\mathfrak{C}_s^{\text{out}}, j$)
                **if** $t =$ Wraparound($z$)
                    Delete($\mathfrak{C}_s^{\text{out}}, j$)
                    Add($\mathfrak{C}_s^{\text{out}}, t$)
                    **break**
                **else if** $j =$ Size($\mathfrak{C}_s^{\text{out}}$)
                    Add($\mathfrak{C}_s^{\text{out}}$, Wraparound($z$))
                **endif**
            **enddo**
        **endif**
    **enddo**
**enddo**
**return** $\mathfrak{C}_s^{\text{out}}$
```

The DFA recognizing the s-witnesses of the running example loop nest $\mathbb{L}_{\text{mm}}$ in Figure 2.5 for cache set 0 (*i.e.*, the solutions of the Presburger formula in Figure A.3) is given in Figure A.6. Using the enumeration algorithm (Algorithm 5.2) to enumerate the solutions encoded by the DFA in Figure A.6 gives $L = 5$ and

$$T_{L,q\in F} = \left\{ \begin{smallmatrix} 3 \\ 19 \\ 0 \\ 1 \\ 1 \end{smallmatrix}, \begin{smallmatrix} 7 \\ 11 \\ 19 \\ 3 \\ 1 \end{smallmatrix}, \begin{smallmatrix} 3 \\ 19 \\ 0 \\ 1 \\ 2 \end{smallmatrix} \right\},$$

where $T_{L,q\in F}$ is the set of satisfying values for free variables ι_0, ι_1, ι_2, u, and e. In the running example, Latest(0) returns memory access $(19, 5, 15, R_2)$. Using the witness-identifying algorithm (Algorithm 5.4) to identify the memory accesses with certain s-witness counts for loop nest $\mathbb{L}_{\text{mm}}$ in cache set 0 gives $emax = 2$ and

$$Z_0^0 = \left\{ \begin{smallmatrix} 19 \\ 5 \\ 15 \\ 2 \end{smallmatrix} \right\}, \qquad Z_0^1 = \left\{ \begin{smallmatrix} 7 \\ 11 \\ 19 \\ 3 \end{smallmatrix} \right\}, \qquad Z_0^2 = \left\{ \begin{smallmatrix} 3 \\ 19 \\ 0 \\ 1 \end{smallmatrix} \right\},$$

where Z_0^i is the set of memory accesses in A_0 with i distinct e-values for each $i \in [0, 2]$. Suppose that the cache state of set 0 at the beginning of program fragment execution is empty, $\mathfrak{C}_0^{\text{in}} = \{\}$. Given the result of the witness-identifying algorithm (Algorithm 5.4) for the running example, the updating algorithm (Algorithm 5.6) returns the cache state of set 0 at the end of the program fragment, $\mathfrak{C}_0^{\text{out}} = \{1, 2, 0\}$.

5.4 Handling Misses Independent of Associativity

For misses that occur independent of the cache's associativity, it is not necessary to apply the neighborhood and witness concepts. In such cases, it is sufficient to describe the cache miss situation as a Presburger formula, represent the formula as a DFA, and count (without enumeration), using the path-counting algorithm (Algorithm 5.1), the accepting paths in the DFA to get the number of solutions.

As examples, Section 5.4.1 presents the case of modeling interior misses in direct-mapped caches, and Section 5.4.2 presents the case of modeling compulsory misses.

5.4.1 Interior Misses in Direct-Mapped Caches

When the cache being considered is direct-mapped (*i.e.*, $\mathcal{A} = 1$), the task of counting interior misses may be accomplished using the method described in Chapter 4. Alternatively, it is possible to count interior misses without enumeration. The following formula expresses the condition that memory access (ι, R_u) incurs an interior miss in a direct-mapped cache.

$$\begin{aligned}
&\big((\iota, R_u) \in i\text{-miss}(\mathbb{L})\big) \overset{\text{def}}{=} \qquad\qquad (5.4)\\
&\quad \iota \in \mathcal{I} \wedge \\
&\quad \Big(\exists d : \text{Map}(\mu_x + \mathcal{L}_x(E_u(\iota)) \cdot \beta_x, d, s) \wedge \\
&\qquad \big(\exists e, \kappa, v : (\kappa, R_v) \lhd (\iota, R_u) \wedge \text{Map}(\mu_y + \mathcal{L}_y(E_v(\kappa)) \cdot \beta_y, e, s) \wedge \\
&\qquad\quad \neg(\exists \rho, w : (\kappa, R_v) \lhd (\rho, R_w) \lhd (\iota, R_u) \wedge \text{Map}(\mu_z + \mathcal{L}_z(E_w(\rho)) \cdot \beta_z, d, s)) \wedge \\
&\qquad \neg(d = e))\Big)
\end{aligned}$$

For any loop nest, applying the path-counting algorithm (Algorithm 5.1) to count the solutions in the Presburger formula given above reveals the number of interior misses in a direct-mapped cache. The Presburger formula constructed to describe the interior misses of the running example loop nest $\mathbb{L}_{\text{mm}}$ in Figure 2.5 on page 17 is given in Figure A.7, and the DFA recognizing the solutions to this formula is given in Figure A.8. Using the path-counting algorithm (Algorithm 5.1) to count the solutions encoded by the DFA in Figure A.8 gives $L = 4$ and 17 accepting paths (*i.e.*, solutions).

5.4.2 Compulsory Misses

The compulsory misses (from the 3C model [67]) incurred by a program are independent of the associativity value of the cache. It is straightforward to describe a compulsory miss situation as a Presburger formula. Any memory access that is the first to request a memory block in the loop nest incurs a compulsory miss. Let reference $R_u = (Y^{(x)}, E_u, S_h)$ at iteration point ι access memory block b_u. Access (ι, R_u) is a compulsory miss if there does not exist a reference $R_v = (Y^{(y)}, E_v, S_i)$ at iteration point κ accessing memory block b_u, such that $(\kappa, R_v) \lhd (\iota, R_u)$ (*i.e.*, (ι, R_u) is the earliest access to memory block b_u). The following formula expresses the condition that memory access (ι, R_u) incurs a compulsory miss.

$$
\begin{aligned}
&\big((\iota, R_u) \in c\text{-miss}(\mathbb{L})\big) \overset{\text{def}}{=} \\
&\quad \iota \in \mathcal{I} \wedge \\
&\quad \Big(\exists d : \text{Map}(\mu_x + \mathcal{L}_x(E_u(\iota)) \cdot \beta_x, d, s) \wedge \\
&\quad \neg\big(\exists \kappa, v : (\kappa, R_v) \lhd (\iota, R_u) \wedge \text{Map}(\mu_y + \mathcal{L}_y(E_v(\kappa)) \cdot \beta_y, d, s)\big)\Big)
\end{aligned}
\tag{5.5}
$$

The Presburger formula constructed to describe the compulsory misses of the running example loop nest $\mathbb{L}_{\text{mm}}$ is given in Figure A.9, and the DFA recognizing the solutions to this formula is given in Figure A.10. Using the path-counting algorithm (Algorithm 5.1) to count the number of solutions encoded by the DFA in Figure A.10, gives $L = 4$ and 3 accepting paths.

5.5 Summary

This chapter shows how to count witnesses for each memory access by first representing the witness formula as a DFA and enumerating the DFA's accepting paths, which encode the formula solutions. The automata-theoretic method for counting and enumerating solutions to Presburger formulas is not specific to formulas describing cache behavior and can be applied to any formula of Presburger arithmetic. To decide whether a memory access suffers an interior miss or a replacement miss, it is sufficient to count its i-witnesses or r-witnesses. To decide whether a memory access suffers a boundary miss or affects cache state, it is necessary to identify its b-witnesses or s-witnesses and compare them with the cache state before the loop nest begins execution.

Chapter 6

Putting It All Together

Chapters 3 to 5 give the theoretical foundations of the analytical framework presented in this dissertation, which models the behavior of loop-oriented programs executing in a memory hierarchy. This chapter provides the high-level view of the framework. Section 6.1 illustrates the framework's method for counting the cache misses incurred by a loop nest and shows how the analysis flows through each step of the method. Section 6.2 discusses the tools used in the implementation of the analysis framework to extract relevant loop nest parameters from source code, to simplify Presburger formulas describing cache behavior, and to represent a cache behavior formula as a DFA whose accepting paths recognize the formula's solutions.

6.1 The Analysis Framework

Figure 6.1 gives a high-level view of the method for modeling cache behavior presented in this dissertation. In modeling the cache behavior of a loop nest $\mathbb{L}$, the figure shows the flow of analysis both for associativity-dependent cache events (the number of interior, replacement, and boundary misses incurred by $\mathbb{L}$, and the cache state after execution of $\mathbb{L}$ for any associativity value $\mathcal{A}$) and for cache events not dependent on associativity (the number of interior misses incurred by $\mathbb{L}$ for direct-mapped caches and the number of compulsory misses incurred by $\mathbb{L}$).

To begin, the method for each cache event is the same. The framework expresses the occurrence of a particular type of event as a formula in Presburger arithmetic using relevant parameters from the loop nest $\mathbb{L}$, and the blocksize $\mathcal{B}$ and number of cache sets $\mathcal{S}$ of the cache being considered. The framework then constructs the DFA recognizing the solutions of the formula[1] according to the fundamental connection between Presburger arithmetic and automata theory (discussed in Sections 2.5 and 5.1).

[1]Simplifying this Presburger formula before converting it to a DFA is critical to the success of the DFA construction, since the original formula is often too complicated for efficient conversion to a DFA.

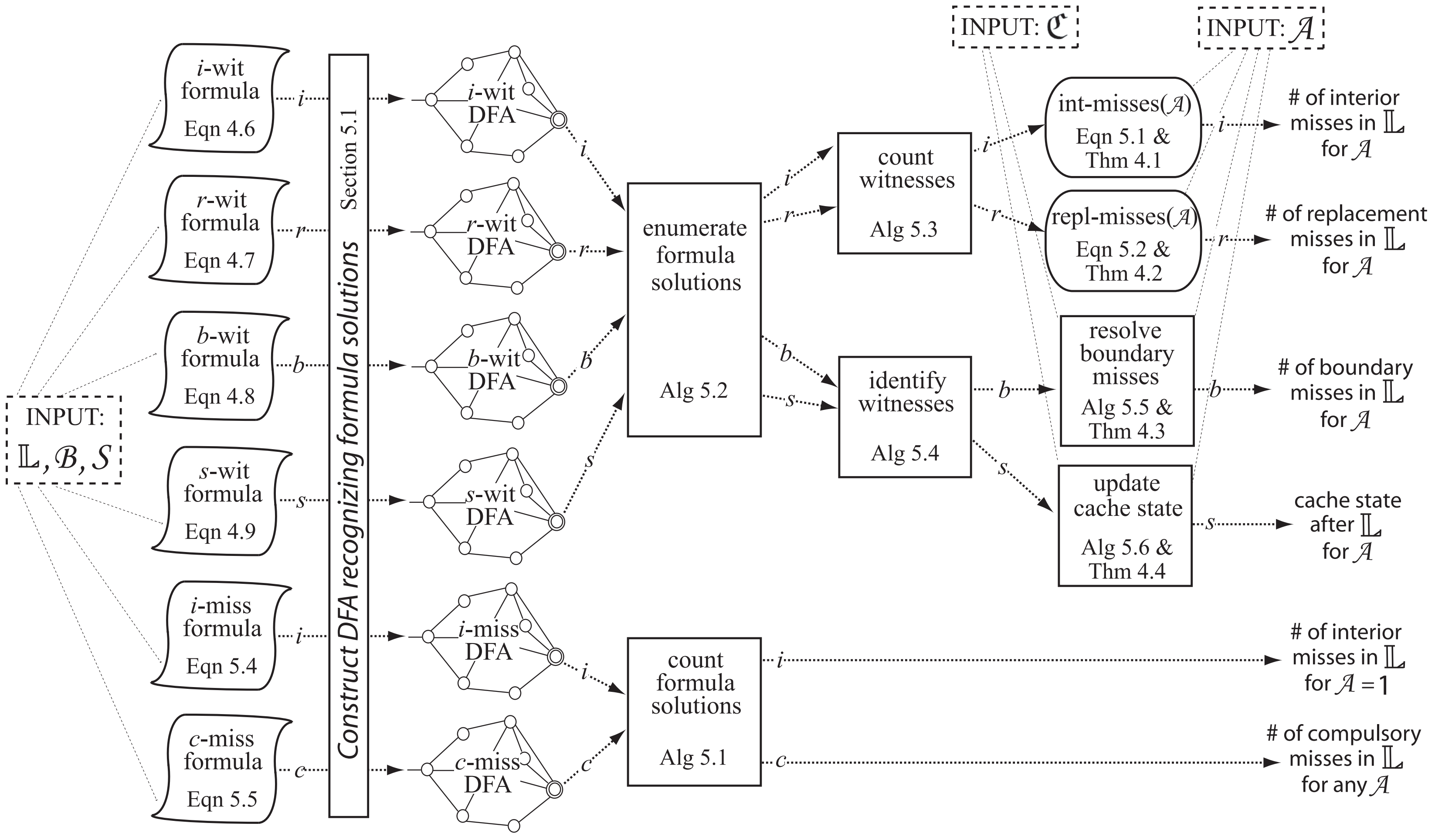

INPUT: $\mathbb{L}, \mathcal{B}, \mathcal{S}$
i-wit formula Eqn 4.6
r-wit formula Eqn 4.7
b-wit formula Eqn 4.8
s-wit formula Eqn 4.9
i-miss formula Eqn 5.4
c-miss formula Eqn 5.5
Construct DFA recognizing formula solutions Section 5.1
i-wit DFA
r-wit DFA
b-wit DFA
s-wit DFA
i-miss DFA
c-miss DFA
enumerate formula solutions Alg 5.2
count formula solutions Alg 5.1
count witnesses Alg 5.3
identify witnesses Alg 5.4
INPUT: $\mathfrak{C}$
INPUT: $\mathcal{A}$
int-misses($\mathcal{A}$) Eqn 5.1 & Thm 4.1
repl-misses($\mathcal{A}$) Eqn 5.2 & Thm 4.2
resolve boundary misses Alg 5.5 & Thm 4.3
update cache state Alg 5.6 & Thm 4.4
# of interior misses in $\mathbb{L}$ for $\mathcal{A}$
# of replacement misses in $\mathbb{L}$ for $\mathcal{A}$
# of boundary misses in $\mathbb{L}$ for $\mathcal{A}$
cache state after $\mathbb{L}$ for $\mathcal{A}$
# of interior misses in $\mathbb{L}$ for $\mathcal{A} = 1$
# of compulsory misses in $\mathbb{L}$ for any $\mathcal{A}$
i
r
b
s
c

Figure 6.1: High-level view of the method for modeling cache behavior presented in this dissertation.

For the DFAs recognizing the solutions of the witness formulas (*i.e.*, the i-wit, r-wit, b-wit, and s-wit DFAs), the framework enumerates the solutions using the enumeration algorithm (Algorithm 5.2), producing the witnesses of all memory accesses in loop nest $\mathbb{L}$. The framework counts the number of witnesses to each memory access for i-witnesses and r-witnesses using the witness-counting algorithm (Algorithm 5.3). Then, given an associativity value $\mathcal{A}$, the number of interior misses incurred by loop nest $\mathbb{L}$ in an ($\mathcal{A}$, $\mathcal{B}$, $\mathcal{A} \cdot \mathcal{B} \cdot \mathcal{S}$; $\mathcal{S}$) cache is computed from the i-witnesses counts of each memory access using equation (5.1) and Theorem 4.1. Similarly, given an associativity value $\mathcal{A}$, the number of replacement misses incurred by loop nest $\mathbb{L}$ in an ($\mathcal{A}$, $\mathcal{B}$, $\mathcal{A} \cdot \mathcal{B} \cdot \mathcal{S}$; $\mathcal{S}$) cache is computed from the r-witnesses counts of each memory access using equation (5.2) and Theorem 4.2.

From the enumeration of solutions recognized by the b-wit and s-wit DFAs, the framework identifies the witnesses to each memory access for b-witnesses and s-witnesses using the witness-identifying algorithm (Algorithm 5.4). Then, given an associativity value $\mathcal{A}$ and the state of the cache when loop nest $\mathbb{L}$ begins execution $\mathfrak{C}$, the framework uses the resolving algorithm (Algorithm 5.5) to determine the potential boundary misses incurred by $\mathbb{L}$, resolves these potential misses with cache state $\mathfrak{C}$, and counts the number of boundary misses incurred by loop nest $\mathbb{L}$ in an ($\mathcal{A}$, $\mathcal{B}$, $\mathcal{A} \cdot \mathcal{B} \cdot \mathcal{S}$; $\mathcal{S}$) cache. Similarly, given an associativity value $\mathcal{A}$ and the state of the cache when loop nest $\mathbb{L}$ begins execution $\mathfrak{C}$, the framework uses the updating algorithm (Algorithm 5.6) to determine the state of the cache after execution of $\mathbb{L}$ for an ($\mathcal{A}$, $\mathcal{B}$, $\mathcal{A} \cdot \mathcal{B} \cdot \mathcal{S}$; $\mathcal{S}$) cache.

Because the interior misses described by the i-miss formula occur in direct-mapped caches, the associativity value is fixed such that $\mathcal{A} = 1$. It is not necessary to apply the neighborhood and witness concepts to count such misses. Therefore, it is sufficient to count (without enumeration) the solutions recognized by the i-miss DFA using the path-counting algorithm (Algorithm 5.1). The solution count gives the number of interior misses incurred by loop nest $\mathbb{L}$ in a (1, $\mathcal{B}$, $\mathcal{B} \cdot \mathcal{S}$; $\mathcal{S}$) cache. Similarly for the compulsory misses described by the c-miss formula, it is not necessary to apply the neighborhood and witness concepts to count such misses, since they occur regardless of the associativity of the cache. The solution count produced by using the path-counting algorithm (Algorithm 5.1) on the c-miss DFA gives the number of compulsory misses incurred by loop nest $\mathbb{L}$ in an ($\mathcal{A}$, $\mathcal{B}$, $\mathcal{A} \cdot \mathcal{B} \cdot \mathcal{S}$; $\mathcal{S}$) cache.

6.2 Implementation

This section describes three tools included in the analysis framework of this dissertation and discusses the leveraging of these tools to implement the theory described in Chapters 4 and 5. Figure 6.2 shows the role of each tool in the division of work.

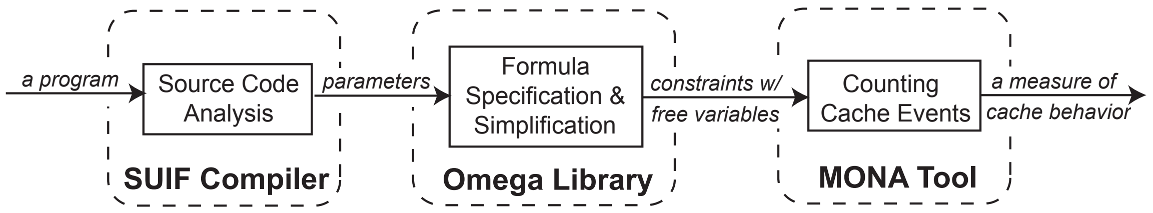


Figure 6.2: Illustration of how tools fit into the analysis framework.

6.2.1 Source Code Analysis

The first phase of the analysis framework presented in this dissertation analyzes the source code of a program, extracting the parameters relevant to its cache activity. To accomplish the source code analysis, the framework examines the intermediate representation (IR) of the source code generated by the SUIF compiler [116]. The compiler translates the source code of a program into an IR that is not tied to either the source language or the target machine language. The SUIF compiler provides both high-level and low-level representations of the source code. Low-level representations consist of sequential instruction lists and facilitate scalar code optimizations, as well as code generation. High-level representations preserve program constructs such as loop nests by means of tree data structures, which are convenient for loop-based analysis of all kinds.

One advantage of the low-level representation is a precise knowledge of each loop nest's memory accesses. The compiler may apply optimizations altering the nature of these accesses, which is not evident in the high-level representation. For example, the compiler may choose to keep a loop-invariant array access in a register over the life of that loop. In spite of this potential shortage of information, the high-level representation is structured, providing concisely the access patterns of each loop nest. Therefore, the analysis framework of this dissertation uses the high-level representation of the source code, despite its limitations.

The SUIF compiler infrastructure consists of a kernel (definition of the IR and functions for accessing and manipulating the IR) and a toolkit of compiler passes built on top of the kernel. The SUIF compiler accepts either C- or Fortran-language source code. The implementation of source code analysis phase is a SUIF compiler pass. The compiler pass invokes SUIF library functions for accessing the high-level IR generated for the given source code.

The compiler pass traverses the IR tree and considers each loop nest individually. Upon encountering a loop in the IR, it extracts the LCV and its bounds from the loop's header nodes. In addition, the compiler pass searches the body of the loop both for other loops and for statements containing array variables. For each array variable referenced, the compiler pass gathers its dimensionality and the length in each dimension from the declaration of the array variable, and determines the byte size of each array element from the base type of the array variable. Given C-language source code, the compiler pass chooses row-major layout for all arrays, and given Fortran-language source code, it chooses column-major layout. At the site of each reference to an array variable, the compiler pass extracts its index expression from the expression tree in the IR. Finally, if the loop body contains other loops, the compiler pass repeats the process above, updating the nesting depth appropriately.

6.2.2 Formula Specification and Simplification

The second phase of the analysis framework presented in this dissertation expresses a cache event as a Presburger formula and simplifies the formula to facilitate DFA construction in the next phase. To build and simplify Presburger formulas, the framework uses the Omega Library. The Omega Library [74, 75] is a set of functions for the manipulation of integer tuple relations and sets described using Presburger formulas. The Presburger formulas in Section 4.3.2 describe the cache events of interest. Parameters extracted from the SUIF compiler pass are used as values for the symbolic constants in these formulas. Given a Presburger formula as input, the Omega Library can provide a simplified version of the formula as output. The simplification process is critical, since the original formulas are often too complicated for efficient translation to DFAs in the next phase.

6.2.3 Counting Cache Events

The last phase of the analysis framework presented in this dissertation counts or enumerates the solutions to a Presburger formula. Interpretation of these results produces the number of cache events occurring during execution of a program. In order to count or enumerate formula solutions, the framework first constructs a DFA recognizing the solutions of the formula and then counts or enumerates the accepting DFA paths. The framework uses the DFA-construction algorithms of Bartzis and Bultan [9, 10] to build the DFAs, and uses the MONA tool to access the DFA when counting or enumerating its accepting paths.

MONA [81] is an automata manipulation tool that also implements decision procedures for WS1S (Weak Second-order theory of One Successor) and WS2S (Weak Second-order theory of Two Successors). The automata are represented by shared, multi-terminal Binary Decision Diagrams. The DFA package includes operations such as union, intersection, complementation, and projection, used to combine DFAs representing linear constraints into a DFA representing a Presburger formula.

Bartzis and Bultan [9, 10] present algorithms for constructing finite automata that represent integer sets satisfying linear constraints. Compared to similar approaches for automata representation [22, 136, 137], the methods of Bartzis and Bultan give bounds on the size of generated automata that are tighter than the established worst-case upper bound for automata construction [23, 80]. The Bartzis and Bultan construction algorithms are used in the analysis framework to represent an affine equality/inequality constraint as a deterministic finite automaton. Using the MONA tool's DFA operations of union, intersection, complementation, and projection, I combine DFA representations of the affine equality/inequality constraints constituting a Presburger formula to get the DFA representation of the formula. Bartzis and Bultan use the MONA tool's DFA package to implement their procedures for constructing DFAs from linear constraints. Implementation of Algorithms 5.1 and 5.2 invoke functions of DFA package to access the generated DFA's state and transition information.

Chapter 7

Extensions to Analysis Framework

The cache analysis framework presented in this dissertation is flexible. As shown in previous chapters, the framework models multiple arbitrarily-nested loops executing in LRU set-associative caches. In addition to the canonical row- and column-major array layouts, the framework also handles nonlinear data layouts expressible in Presburger arithmetic, as Section 7.1 shows. However, there are limits to the analysis framework, in particular for modeling the first-in first-out (FIFO) cache replacement algorithm, as Section 7.2 discusses.

7.1 Nonlinear Data Layouts

As Section 1.1 explains, the arrangement of memory in today's systems is hierarchical. In contrast, for row- and column-major layouts, the linearization of arrays in memory is flat. The intuition of nonlinear array layouts is to arrange array data in a way that more closely matches the hierarchical nature of memory. This intuition has proven to be correct for some numerical codes, in which nonlinear array layouts have better cache performance than the canonical array layouts [32, 33, 57, 131]. Understanding how and when nonlinear array layouts yield better cache performance than canonical array layouts is still an open area of research, which motivates a cache analysis framework that handles such layouts.

The task of an array layout is to map an array element to its location in memory. For an array $Y^{(y)}$ with dimensionality d_y, the layout function takes d_y array coordinates as input and gives a memory offset as output. To get the actual memory address, multiply the offset by the byte size of $Y^{(y)}$'s elements and add the starting address of $Y^{(y)}$ in memory. Canonical array layouts are affine functions of the numerical values of the array coordinates, whereas nonlinear array layout functions interleave bits in the binary expansions of the array coordinates. Specification of canonical layouts using Presburger arithmetic is straightforward, given their simple and affine nature (see Section 2.3). In contrast, the description of nonlinear array layouts based on bit interleaving is complicated, and specifying these layouts using Presburger arithmetic requires some extra effort. The reader should be aware that the following discussion is dense with the details of bit interleaving.

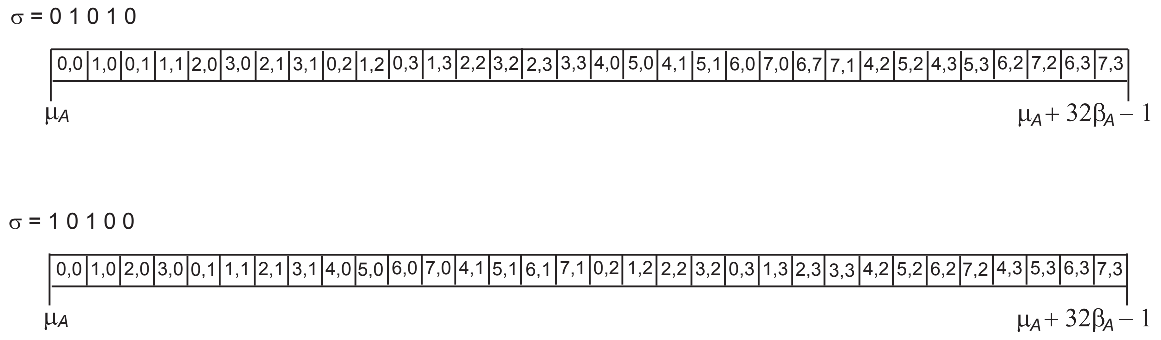


Figure 7.1: Two example arrangements of the data of an 8×4 array A in memory distinguished by different $(3, 2)$-*interleavings*.

To start, it is necessary to model the binary expansion of some non-negative integer c, which is the coordinate for the $(j+1)^{\text{th}}$ dimension of array $Y^{(y)}$ where $j \in [0, d_{y-1}]$. For the length of the array in this dimension, ℓ_j, assume that $\ell_j = 2^{q_j}$. Consequently, the binary expansion of c has q_j bits, with the least significant bit numbered 0 and the most significant bit numbered $q_j - 1$. The binary sequence $c_{q_j-1} \ldots c_0$ is identified with c, where $c = \sum_{i=0}^{q_j-1} c_i 2^i$. The set of all binary sequences of length q_j is denoted as B_{q_j}, and extension of the above identifies B_{q_j} with the interval $[0, 2^{q_j} - 1]$.

Nonlinear layout functions determine the memory address of an array element by **interleaving** the bits of the binary expansions of its array coordinates. There are a variety of ways to interleave such bits. For example, consider two array coordinates $a = 7$ and $b = 1$ for a two-dimensional array with $\ell_0 = 2^3$ and $\ell_1 = 2^2$, such that $a_2a_1a_0 = 111 \in B_3$ and $b_1b_0 = 01 \in B_2$. One interleaving $a_2b_1a_1b_0a_0 = 10111$ produces the memory offset 23, and yet another interleaving $b_1a_2b_0a_1a_0 = 01111$ produces the memory offset 15. A nonlinear array layout function is parameterized according to σ, which describes the order in which bits from the d_y array coordinates are interleaved to linearize the array $Y^{(y)}$ in memory. An $(q_0, ..., q_{d_y-1})$-*interleaving*, σ, is a sequence of length p (where $p = \sum_{i=0}^{d_y-1} q_i$) over the alphabet $\{0, \ldots, (d_y - 1)\}$ containing q_i i's. In the example above, two different σ's define the interleaving of bits, but both are $(3, 2)$-*interleavings* of length 5 over the alphabet $\{0, 1\}$ containing 3 0s and 2 1s.

Given an $(q_0, ..., q_{d_y-1})$-*interleaving* σ, it is necessary to define a map from d_y array coordinates to a memory offset. This is equivalent to defining a map $\Theta : B_{q_0} \times \cdots \times B_{q_{d_y-1}} \to B_p$, which is done in the following way. If $x^{(i)} = x^{(i)}_{q_i-1} \ldots x^{(i)}_1 x^{(i)}_0 \in B_{q_i} \forall i \in [0, d_x - 1]$, then $\Theta(x^{(0)}, \ldots, x^{(d_y-1)})$ is the sequence obtained by replacing the j^{th} u from the right in σ with $x^{(u)}_j$, where Θ is the **mixing function** indexed by σ. Note that $\Theta(0, \ldots, 0) = 0$ for any σ. In the example above, the mixing function $\Theta(111, 01)$ indexed by $\sigma = 01010$ produces the memory offset 23, and indexed by $\sigma = 10100$ produces the memory offset 15. Figure 7.1 illustrates how the layout of an 8×4 array A in memory differs when $\sigma = 01010$ and $\sigma = 10100$ define the interleaving of bits.

Expression of these nonlinear layout functions as Presburger formulas requires the definition of bit values in the binary expansion of a memory offset using Presburger arithmetic. For an array $Y^{(y)}$ whose layout is specified with an $(q_0, \ldots, q_{d_y-1})$-*interleaving* σ, the following $d_y \times p$ matrix $M(\sigma)$ is computed. Letting $g = \sigma_f$, the f^{th} column of $M(\sigma)$ consists of 2^e in the g^{th} position, where σ_f is the e^{th} g from the right, and zeros in every other position. Conceptually, $M(\sigma)$ is a transformation that when applied to the binary expansion of a memory offset o produces the coordinates of the array element at memory address $m = \mu_y + o\beta_y$. For the $(3, 2)$-*interleavings* in the example above,

$$M(01010) = \begin{bmatrix} 4 & 0 & 2 & 0 & 1 \\ 0 & 2 & 0 & 1 & 0 \end{bmatrix} \quad \text{and} \quad M(10100) = \begin{bmatrix} 0 & 4 & 0 & 2 & 1 \\ 2 & 0 & 1 & 0 & 0 \end{bmatrix}.$$

For a reference $R_i = (Y^{(y)}, E_i, S_h)$ and an iteration point ι, the following formula maps E_i and $M(\sigma)$ to a memory offset o. Recall that $p = \sum_{i=0}^{d_y-1} q_i$, and let $O = [o_{p-1}, \ldots, o_0]^T$.

$$\big(o = \text{Interleave}(E_i(\iota), M(\sigma)\big) \overset{\text{def}}{=}$$
$$\exists o_{p-1}, \ldots, o_0 : 0 \leqslant o_{p-1}, \ldots, o_0 \leqslant 1 \wedge o \geqslant 0 \wedge E_i(\iota) = M(\sigma)O \wedge o = \sum_{k=0}^{p-1} o_k 2^k$$

The memory byte address m of the element accessed by reference R_i at iteration point ι is calculated as $m = \mu_y + \mathcal{L}_y(E_i(\iota)) \cdot \beta_y$. By simply using the formula above as the layout function $\mathcal{L}_y$, the cache behavior formulas given in Sections 4.3.2 and 5.4 characterizes the access patterns of an array with such nonlinear layouts. Thus, the analysis framework of this dissertation is capable of determining the cache behavior of loop nests referencing arrays with nonlinear array layouts expressible in Presburger arithmetic.

The Presburger formula constructed to describe the i-witnesses of the running example loop nest $\mathbb{L}_{\text{mm}}$ in Figure 2.5 on page 17 with a nonlinear array layout is given in Figure A.11, and the DFA recognizing the solutions to this formula is given in Figure A.12. In this instance of the running example, $t = u = v = 16$ and all arrays are linearized according to a nonlinear array layout specified by the $(4, 4)$-interleaving $\sigma = 01010011$. Using the enumeration algorithm (Algorithm 5.2) to enumerate the solutions encoded by the DFA in Figure A.12, gives $L = 4$ and

$$T_{L,q\in F} = \left\{\begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 2 & 2 & 3 & 3 & 4 \\ 0, & 15, & 0, & 15, & 0, & 15, & 0, & 15, & 0 \\ 1 & 3 & 1 & 3 & 1 & 3 & 1 & 3 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{matrix}\right\}.$$

Using the witness-counting algorithm (Algorithm 5.3) to count the number of i-witnesses for the memory accesses encoded by the DFA in Figure A.12, gives $emax = 1$ and $N_1 = 9$. Therefore, int-misses$(1) = 9$ and int-misses$(\mathcal{A}) = 0, \forall \mathcal{A} \geqslant 2$. Loop nest $\mathbb{L}_{\text{mm}}$ with the nonlinear array layout incurs nine interior misses in set 0 of a (1, 32, 4096; 128) cache and no interior misses in set 0 of all $\{(k,\ 32,\ 4096k;\ 128)\colon\ k \geqslant 2\}$ caches.

7.2 FIFO Cache Replacement Algorithm

Chapter 4 demonstrates how to model the behavior of a program executing in a cache that uses the LRU cache set replacement algorithm. This section considers whether it is possible to model the first-in first-out (FIFO) cache set replacement algorithm using the cache analysis framework.

In contrast to the LRU cache set replacement algorithm, which evicts from a cache set the memory block that has been unused longest, the FIFO replacement algorithm evicts the memory block that has been resident longest. The **FIFO status** of a cache block is an integer that indicates its position in the ordering of cache blocks to be replaced by incoming cache blocks in the cache set. Let a cache block have a FIFO status of $\mathcal{A}$ if it is the most recent cache block placed in the cache set (and the last to be replaced), where $\mathcal{A}$ is the associativity value of the cache being considered. Let a cache block have a FIFO status of 1 if it is the least recent cache block placed in the cache set (and the first to be replaced).

Let i-witnesses be defined as in Section 4.2 and described as with equation (4.6). For an LRU cache, Theorem 4.1 decides whether a memory access incurs an interior miss based on a comparison of its i-witness count with associativity value $\mathcal{A}$. The following theorem decides whether an access incurs an interior miss in a FIFO cache given a similar comparison of i-witness count and associativity.

Theorem 7.1 *If an access* a *to memory block* b *has* $2\mathcal{A}-1$ *or more* i*-witnesses, then it suffers an interior miss in an* $\mathcal{A}$*-way cache employing a FIFO cache set replacement algorithm.*

PROOF. Either a is the earliest access to memory block b (Case 1) or it is not (Case 2). For Case 1, the existence of $\mathcal{A}$ or more i-witnesses to a ensures that a misses in cache, regardless of cache state before loop nest execution. Therefore, $2\mathcal{A}-1$ i-witnesses certainly ensure an interior miss.

For Case 2, let access a' be the most recent access to memory block b. Just after access a', the cache set to which b maps contains b and $\mathcal{A}-1$ other memory blocks. There may be subsequent accesses to these memory blocks that do not affect the FIFO status of b. Thus, it is possible to have at most $\mathcal{A}-1$ i-witnesses to access a that do not alter the FIFO status of memory block b. Any access in the i-neighborhood of a to a memory block not contained in the cache set just after access a' does cause the FIFO status of memory block b to change. To change the FIFO status of memory block b from $\mathcal{A}$ to 1, $\mathcal{A}-1$ additional i-witnesses are necessary. One more i-witnesses then causes the displacement of b from cache before it is accessed again at access a. Therefore, the maximum number of i-witnesses not affecting the FIFO status of b, $\mathcal{A}-1$; the maximum number of i-witnesses required to change b's FIFO status, $\mathcal{A}-1$; and the i-witness causing displacement of b add up to a total of $2\mathcal{A}-1$ i-witnesses guaranteeing an interior miss. □

Theorem 7.1 has the following implications,

1. *There is no guarantee of the hit/miss status of accesses with fewer than* $2\mathcal{A}-1$ i-*witnesses.* In fact, accesses with anywhere from 1 to $2\mathcal{A} - 2$ i-witnesses may still suffer an interior miss. The actual hit/miss status of such accesses cannot be determined using the cache analysis framework. The difficulty in modeling behavior in a FIFO cache is knowing the FIFO status of a memory block. For all memory blocks contained in a cache set, their FIFO statuses change only during a cache miss in that set. The existence of a cache miss is known in the last phase of the analysis framework, and even then it depends on an associativity value. For all memory blocks contained in a cache set, the LRU statuses change during each memory accesses mapping to that cache set. Such information is built into the formulation of cache behavior, making LRU status more natural to determine analytically than FIFO status. However, it is possible that fundamental changes to the cache analysis framework would allow the modeling of a FIFO replacement algorithm.

2. *One can expect an LRU cache with associativity* $\mathcal{A}$ *to be at least as effective as a FIFO cache with associativity* $2\mathcal{A} - 1$*.* In an LRU cache, accesses with $\mathcal{A}$ or more i-witnesses suffer interior misses, and accesses with fewer than $\mathcal{A}$ i-witnesses are either cache hits or boundary misses. In a FIFO cache, accesses with $2\mathcal{A} - 1$ or more i-witnesses suffer interior misses, and accesses with fewer than $2\mathcal{A} - 1$ i-witnesses are either cache hits, boundary misses, or interior misses.

Therefore, in the cache analysis framework, it is not possible to completely model behavior in a cache employing the FIFO replacement algorithm.

7.3 Summary

The inherent flexibility of the analysis framework presented in this dissertation derives from the use of Presburger formulas. The framework's ability to handle nonlinear array layouts expressible in Presburger arithmetic demonstrates this flexibility. Of course, since row- and column-major layouts are the standard ways of mapping arrays to memory, handling nonlinear array layouts is not critical to determining the cache behavior of program. However, the ability of the analysis framework to evaluate and understand the potential benefit of nonlinear array layouts is useful.

The analysis framework does not have unlimited flexibility. In particular, the assumption that caches employ the LRU replacement algorithm is fundamental to the way in which the framework expresses cache behavior. As a result, it may not be possible to use the analysis framework to model the behavior of caches employing other replacement algorithms.

Chapter 8

Application and Validation

This chapter applies and validates the analysis framework presented in this dissertation by using it to accurately determine the cache behavior of a number of program fragments. In Section 8.1, I consider several cases of isolated, single loop nests and use the framework to determine the number of cache misses incurred in each case. In Section 8.2, I produce the number of cache misses incurred by a sequence of loop nests, employing the interior-boundary miss classification and cache state to precisely compose the cache miss count for each loop nest into a total cache miss count for the entire sequence. In Section 8.3, I use the analysis framework to compute the number of cache misses incurred by a loop nest when the data layout is nonlinear. In Section 8.4, I apply the analysis framework to two loop optimization problems—finding the optimal tilesize for a matrix-vector multiplication loop nest and finding the optimal permutation of loops for a matrix multiplication loop nest. In Section 8.5, I give the number of cache misses incurred by several loop nests that perform aggregate array computations, before and after optimizations based on incrementalization.

Every cache miss count provided here has been validated against the counts produced by a (specially-written) cache simulator. In all cases, the cache miss count generated by the analysis framework is *exactly the same* as the count given by the simulator, so I refrain from giving the simulator's miss counts in addition to the framework's miss counts. All execution times were collected on a 2GHz Intel Xeon processor running Red Hat Linux 7.3.

8.1 Single Loop Nests

First, I demonstrate the analysis framework for single loop nests, isolated from the programs containing them. To indicate the flexibility of the cache analysis techniques, I give compulsory and replacement miss counts for some loop nests and interior and potential boundary miss counts for others. Because I am considering each loop nest isolated from the rest of the program (*i.e.*, cache state before the loop nest begins execution is unknown), the total cache miss count reported may be an overestimate of the actual miss count incurred by the loop nest in the context of a program. Given the compulsory-replacement cache miss classification, the total number of cache misses is overestimated by at most the number of compulsory misses. For a loop nest that accesses N bytes of memory and executes in an $(\mathcal{A}, \mathcal{B}, \mathcal{C}; \mathcal{S})$ cache, the number of compulsory misses incurred is the number of memory blocks it accesses (*i.e.*, $\frac{N}{\mathcal{B}}$ blocks). Given the interior-boundary cache miss classification, the total number of cache misses is overestimated by at most the number of potential boundary misses, which is bounded from above by the number of cache frames (*i.e.*, $\frac{\mathcal{C}}{\mathcal{B}}$ frames), as stated by Lemma 3.1.

I demonstrate the analysis framework on five loop nests, each in data caches and translation lookaside buffers (TLBs). Existing compiler frameworks for analyzing memory behavior (see Chapter 9) have focused on data caches, despite the impact of TLB misses on the performance of a loop nest. The penalty for a TLB miss can be hundreds of processor cycles, and for that reason, examining and reducing the TLB misses incurred by a program can result in significant improvement in its performance. Because the TLB is a cache, I can use the analysis framework to compute the number of TLB misses for a program fragment. The key to modeling the cache activity of a TLB is to set the "cache" blocksize equal to the page size, since that is the granularity of contiguous memory addresses. For example, to model a fully-associative TLB with page size 8192 bytes, I set $\mathcal{B} = 8192$ and $\mathcal{S} = 1$.

Triad loop nest. Consider the following loop nest $\mathbb{L}_{\text{tri}}$.

```
L_tri:  do i = 0, n-1
           R[i] = P[i] + Q[i]
        enddo
```

Suppose that the arrays $Y^{(0)} = \mathtt{P}$, $Y^{(1)} = \mathtt{Q}$, and $Y^{(2)} = \mathtt{R}$ are double-precision (*i.e.*, $\beta_0 = \beta_1 = \beta_2 = 8$ bytes) with starting addresses $\mu_0 = 0$, $\mu_1 = \mu_0 + 8n$, and $\mu_2 = \mu_1 + 8n$.

Table 8.1 gives the results of using the framework to count the number of compulsory and replacement misses incurred by $\mathbb{L}_{\text{tri}}$ for several problem sizes in $(\mathcal{A}, 64, 16384 \cdot \mathcal{A}; 256)$ data caches. Results include the time required for formula expression and simplification, the time required for DFA construction, the time required to enumerate formula solutions and count witnesses, and the number of cache misses for associativity value $\mathcal{A}$. For some problem sizes, the reported time required to count misses is 0.00 seconds. This indicates that the time required is actually less than 0.01 seconds, since the timing granularity is 10 milliseconds. Note that when compulsory miss counts are given, the number is independent of $\mathcal{A}$ and $\mathcal{C}$. *Solution times are for counting misses in all cache sets.*

In Table 8.1, notice that problem sizes $n = 16,380$ and $n = 131,070$ have large DFA-construction times relative to the other problem sizes. This is due to the thrashing[1] caused by

[1]Thrashing occurs when frequently used memory blocks replace each other in cache. The cache blocks are written back to main memory before their reuse is complete, and unnecessary cache misses are the consequence. Thrashing is often caused by an alignment of data structures in memory such that distinct memory blocks map to the same cache set, yet they are accessed repeatedly one after the other.

n	solution time (sec)			# of compulsory misses	# of replacement misses	
	simplify formula	build DFA	count misses			
10,000	0.05	0.12	0.00	3,750	$\mathcal{A} \geqslant 1$	0
16,380	0.08	10.61	0.21	6,143	$\mathcal{A} = 1$	24,571
					$2 \leqslant \mathcal{A} \leqslant 22$	1
					$\mathcal{A} \geqslant 23$	0
100,000	0.06	0.21	0.00	18,750	$\mathcal{A} \geqslant 1$	0
131,070	0.11	27.70	5.50	49,152	$\mathcal{A} = 1$	278,522
					$\mathcal{A} = 2$	147,452
					$3 \leqslant \mathcal{A} \leqslant 190$	2
					$\mathcal{A} \geqslant 191$	0
1,000,000	0.05	0.65	0.00	375,000	$\mathcal{A} \geqslant 1$	0

Table 8.1: Results of using the analysis framework to count the number of compulsory and replacement misses incurred by loop nest $\mathbb{L}_{\text{tri}}$ for several problem sizes in ($\mathcal{A}$, 64, 16384 $\cdot$ $\mathcal{A}$; 256) data caches.

these two problem sizes. The thrashing results in a large amount of cache activity that must be represented by a DFA and leads to relatively long construction times. For the problem sizes without thrashing, there are no replacement misses for any value of $\mathcal{A}$. Loop nest $\mathbb{L}_{\text{tri}}$ has no temporal reuse of data. Therefore, if the spatial reuse can be exploited without conflict, then the only cache misses are compulsory. Notice that the cache miss counts in each table of this chapter are for all $\mathcal{A} \geqslant 1$, because the analysis framework produces miss counts for all values of $\mathcal{A}$ in a single pass. The values of $\mathcal{A}$ found in real caches, such as $\mathcal{A} = 1$, 2, 3, 4, 6, 8, and so on, are the most interesting. However, other values of $\mathcal{A}$ may be useful in considering experimental caches. Furthermore, for every loop nest, there is a value k for which the loop nest incurs no replacement (or interior) misses in $\{(\mathcal{A}, \mathcal{B}, \mathcal{A} \cdot \mathcal{B} \cdot \mathcal{S}; \mathcal{S})\colon \mathcal{A} > k\}$ caches. This maximum associativity value, beyond which the loop nest incurs no such misses, is a useful characteristic of the loop nest.

Table 8.2 gives the results of using the framework to count the number of compulsory and replacement misses incurred by $\mathbb{L}_{\text{tri}}$ for several problem sizes in ($\mathcal{A}$, 8192, 8192 $\cdot$ $\mathcal{A}$; 1) TLBs. In Table 8.2, notice that there are replacement misses for several values of $\mathcal{A}$ for all problem sizes. Even though loop nest $\mathbb{L}_{\text{tri}}$ has no temporal reuse, the spatial reuse cannot always be exploited without conflict because there is only one cache set (*i.e.*, $\mathcal{S} = 1$). Contention for the only cache set results in replacement misses for smaller values of $\mathcal{A}$.

Table 8.3 gives the results of using the framework to count the number of compulsory and replacement misses incurred by $\mathbb{L}_{\text{tri}}$ for $n = 10,000$ in ($\mathcal{A}$, $\mathcal{B}$, 8192 $\cdot$ $\mathcal{A}$; $\mathcal{S}$) caches with varying values of $\mathcal{B}$ and $\mathcal{S}$. The purpose of this variation is to observe how the cache behavior of loop nest $\mathbb{L}_{\text{tri}}$ changes with $\mathcal{B}$ and $\mathcal{S}$, even though the capacity $\mathcal{C}$ of the cache remains the same (*i.e.*, for each value of $\mathcal{A}$, $\mathcal{C}$ remains 8192 $\cdot$ $\mathcal{A}$). In Table 8.3, notice that the replacement miss

n	solution time (sec)			# of compulsory misses	# of replacement misses	
	simplify formula	build DFA	count misses			
1,000	0.07	0.16	0.03	3	$\mathcal{A} = 1$	2,949
					$\mathcal{A} = 2$	2,853
					$\mathcal{A} \geqslant 3$	0
5,000	0.06	8.31	0.12	15	$\mathcal{A} = 1$	14,985
					$\mathcal{A} = 2$	14,985
					$3 \leqslant \mathcal{A} \leqslant 13$	2
					$\mathcal{A} \geqslant 14$	0
10,000	0.07	15.81	0.25	30	$1 \leqslant \mathcal{A} \leqslant 2$	29,970
					$3 \leqslant \mathcal{A} \leqslant 28$	2
					$\mathcal{A} \geqslant 29$	0
50,000	0.07	79.61	2.25	147	$1 \leqslant \mathcal{A} \leqslant 2$	149,853
					$3 \leqslant \mathcal{A} \leqslant 145$	2
					$\mathcal{A} \geqslant 146$	0

Table 8.2: Results of using the analysis framework to count the number of compulsory and replacement misses incurred by loop nest $\mathbb{L}_{\text{tri}}$ for several problem sizes in ($\mathcal{A}$, 8192, $8192 \cdot \mathcal{A}$; 1) TLBs.

count increases as the value of $\mathcal{S}$ decreases, and the compulsory miss count increases as the value of $\mathcal{B}$ decreases. The spatial reuse in $\mathbb{L}_{\text{tri}}$ cannot always be exploited without conflict when there are only a few cache sets (*i.e.*, $1 \leqslant \mathcal{S} \leqslant 4$). Contention for a small number of cache sets causes large numbers of replacement misses for small values of $\mathcal{A}$ when the value of $\mathcal{S}$ is small. Recall that the number of compulsory misses incurred by a loop nest is equal to the number of memory blocks it accesses. Because the total number of memory bytes accessed by loop nest $\mathbb{L}_{\text{tri}}$ is constant, decreasing the blocksize increases the number of memory blocks accessed, and thus, increases the number of compulsory misses. In Table 8.3, notice that solution times are not proportional in any obvious way to the values of $\mathcal{B}$ and $\mathcal{S}$. However, relationships between solution times and cache miss counts are more evident, as discussed with other general observations at the end of this section.

Matrix-vector multiplication loop nest. Consider the following loop nest $\mathbb{L}_{\text{vec}}$.

```
L_vec:  do i = 0, $m-1$
           r = Y[i]
           do j = 0, $n-1$
              r += A[i,j]*X[j]
           enddo
           Y[i] = r
        enddo
```

$\mathcal{B}$	$\mathcal{S}$	solution time (sec)			# of compulsory misses	# of replacement misses	
		simplify formula	build DFA	count misses			
8,192	1	0.07	15.81	0.25	30	$1 \leqslant \mathcal{A} \leqslant 2$	29,970
						$3 \leqslant \mathcal{A} \leqslant 28$	2
						$\mathcal{A} \geqslant 29$	0
4,096	2	0.10	7.45	0.16	59	$\mathcal{A} = 1$	20,549
						$\mathcal{A} = 2$	1,769
						$3 \leqslant \mathcal{A} \leqslant 28$	2
						$\mathcal{A} \geqslant 29$	0
2,048	4	0.09	4.99	0.04	118	$\mathcal{A} = 1$	2,342
						$2 \leqslant \mathcal{A} \leqslant 28$	2
						$\mathcal{A} \geqslant 29$	0
1,024	8	0.08	0.04	0.00	235	$1 \leqslant \mathcal{A} \leqslant 28$	2
						$\mathcal{A} \geqslant 29$	0
512	16	0.09	0.04	0.00	469	$1 \leqslant \mathcal{A} \leqslant 28$	2
						$\mathcal{A} \geqslant 29$	0
256	32	0.09	0.05	0.00	938	$1 \leqslant \mathcal{A} \leqslant 28$	1
						$\mathcal{A} \geqslant 29$	0
128	64	0.13	0.03	0.00	1,875	$\mathcal{A} \geqslant 1$	0
64	128	0.05	0.05	0.00	3,750	$\mathcal{A} \geqslant 1$	0
32	256	0.05	0.12	0.00	7,500	$\mathcal{A} \geqslant 1$	0
16	512	0.08	0.27	0.00	15,000	$\mathcal{A} \geqslant 1$	0
8	1,024	0.04	0.70	0.00	30,000	$\mathcal{A} \geqslant 1$	0

Table 8.3: Results of using the analysis framework to count the number of compulsory and replacement misses incurred by loop nest $\mathbb{L}_{\text{tri}}$ for $n = 10,000$ in ($\mathcal{A}$, $\mathcal{B}$, $8192 \cdot \mathcal{A}$; $\mathcal{S}$) caches with varying values of $\mathcal{B}$ and $\mathcal{S}$.

Suppose that the arrays $Y^{(0)} = \texttt{A}$, $Y^{(1)} = \texttt{X}$, and $Y^{(2)} = \texttt{Y}$ are double-precision (*i.e.*, $\beta_0 = \beta_1 = \beta_2 = 8$ bytes) with starting addresses $\mu_0 = 0$, $\mu_1 = \mu_0 + 8m \cdot n$, and $\mu_2 = \mu_1 + 8n$. Array $Y^{(0)}$ is linearized in column-major order.

Table 8.4 gives the results of using the framework to count the number of interior and potential boundary misses incurred by $\mathbb{L}_{\text{vec}}$ for several problem sizes in ($\mathcal{A}$, 64, $16384 \cdot \mathcal{A}$; 256) data caches. In Table 8.4, notice that the number of potential boundary misses increases with problem size (for $\mathcal{A} \geqslant 1$ if $m = n = 40$ and for $\mathcal{A} \geqslant 2$ otherwise). The reason is that the cache footprint of the memory blocks accessed by loop nest $\mathbb{L}_{\text{vec}}$ increases with problem size. Recall that the potential boundary miss count is equal to this cache footprint. Notice also that the number of interior misses does not necessarily increase with problem size. The three arrays referenced by loop nest $\mathbb{L}_{\text{vec}}$ are stored contiguously (or back-to-back) in memory, and certain problem sizes cause the arrays to be aligned in ways that induce more conflicts, and thus, larger interior miss counts.

Table 8.5 gives the results of using the framework to count the number of interior and potential boundary misses incurred by $\mathbb{L}_{\text{vec}}$ for several problem sizes in ($\mathcal{A}$, 8192, $8192 \cdot \mathcal{A}$; 1) TLBs. In Table 8.5, notice that the number of interior misses essentially increases with problem size. That is, as problem size increases, there are more values of $\mathcal{A}$ with non-zero miss counts and for such values of $\mathcal{A}$ the miss counts increase. The reason is that as the values of m and n increase, the number of distinct memory blocks mapping to the only cache set increases, creating more contention for the set. For example, when $m = n = 40$ two memory blocks accessed by loop nest $\mathbb{L}_{\text{vec}}$ map to the cache set, and when $m = n = 60$ four memory blocks accessed by $\mathbb{L}_{\text{vec}}$ map to the cache set.

Matrix multiplication loop nest. Recall the following loop nest $\mathbb{L}_{\text{mm}}$, used as a running example.

```
L_MM:  do i = 0, $t-1$
          do j = 0, $u-1$
             c = Z[i,j]
             do k = 0, $v-1$
                c += X[i,k] * Y[k,j]
             enddo
             Z[i,j] = c
          enddo
       enddo
```

Suppose that the arrays $Y^{(0)} = \texttt{X}$, $Y^{(1)} = \texttt{Y}$, and $Y^{(2)} = \texttt{Z}$ are double-precision (*i.e.*, $\beta_0 = \beta_1 = \beta_2 = 8$ bytes) and linearized in column-major order with starting addresses $\mu_0 = 0$, $\mu_1 = \mu_0 + 8t \cdot v$, and $\mu_2 = \mu_1 + 8v \cdot u$.

Table 8.6 gives the results of using the framework to count the number of interior and potential boundary misses incurred by $\mathbb{L}_{\text{mm}}$ for several problem sizes in ($\mathcal{A}$, 64, $16384 \cdot \mathcal{A}$; 256)

m	n	solution time (sec)			# of interior misses		# of pot. boundary misses	
		simplify formula	build DFA	count misses				
40	40	0.27	0.00	0.03	$\mathcal{A} \geqslant 1$	0	$\mathcal{A} \geqslant 1$	210
50	40	0.23	0.03	0.04	$\mathcal{A} = 1$	6	$\mathcal{A} = 1$	256
					$\mathcal{A} \geqslant 2$	0	$\mathcal{A} \geqslant 2$	262
50	50	0.65	0.44	0.04	$\mathcal{A} = 1$	918	$\mathcal{A} = 1$	256
					$\mathcal{A} \geqslant 2$	0	$\mathcal{A} \geqslant 2$	325
50	60	0.70	0.59	0.06	$\mathcal{A} = 1$	1,681	$\mathcal{A} = 1$	256
					$\mathcal{A} \geqslant 2$	0	$\mathcal{A} \geqslant 2$	389
60	60	0.53	0.51	0.05	$\mathcal{A} = 1$	355	$\mathcal{A} = 1$	256
					$\mathcal{A} \geqslant 2$	0	$\mathcal{A} \geqslant 2$	465

Table 8.4: Results of using the analysis framework to count the number of interior and potential boundary misses incurred by loop nest $\mathbb{L}_{\text{vec}}$ for several problem sizes in ($\mathcal{A}$, 64, $16384 \cdot \mathcal{A}$; 256) data caches.

m	n	solution time (sec)			# of interior misses		# of pot. boundary misses	
		simplify formula	build DFA	count misses				
40	40	0.33	0.01	0.01	$\mathcal{A} = 1$	2,048	$\mathcal{A} = 1$	1
					$\mathcal{A} \geqslant 2$	0	$\mathcal{A} \geqslant 2$	2
50	40	0.50	0.30	0.01	$\mathcal{A} = 1$	2,091	$\mathcal{A} = 1$	1
					$\mathcal{A} = 2$	127	$\mathcal{A} = 2$	2
					$\mathcal{A} \geqslant 3$	0	$\mathcal{A} \geqslant 3$	3
50	50	0.33	1.30	0.02	$\mathcal{A} = 1$	4,096	$\mathcal{A} = 1$	1
					$\mathcal{A} = 2$	99	$\mathcal{A} = 2$	2
					$\mathcal{A} \geqslant 3$	0	$\mathcal{A} \geqslant 3$	3
50	60	0.60	2.70	0.02	$\mathcal{A} = 1$	4,135	$\mathcal{A} = 1$	1
					$\mathcal{A} = 2$	176	$\mathcal{A} = 2$	2
					$\mathcal{A} = 3$	115	$\mathcal{A} = 3$	3
					$\mathcal{A} \geqslant 4$	0	$\mathcal{A} \geqslant 4$	4
60	60	0.53	3.14	0.02	$\mathcal{A} = 1$	6,144	$\mathcal{A} = 1$	1
					$\mathcal{A} = 2$	179	$\mathcal{A} = 2$	2
					$\mathcal{A} = 3$	178	$\mathcal{A} = 3$	3
					$\mathcal{A} \geqslant 4$	0	$\mathcal{A} \geqslant 4$	4

Table 8.5: Results of using the analysis framework to count the number of interior and potential boundary misses incurred by loop nest $\mathbb{L}_{\text{vec}}$ for several problem sizes in ($\mathcal{A}$, 8192, $8192 \cdot \mathcal{A}$; 1) TLBs.

$t = u = v$	solution time (sec)			# of interior misses		# of pot. boundary misses	
	simplify formula	build DFA	count misses				
20	1.34	0.05	0.03	$\mathcal{A} \geqslant 1$	0	$\mathcal{A} \geqslant 1$	150
30	2.07	0.69	0.04	$\mathcal{A} = 1$	114	$\mathcal{A} = 1$	256
				$\mathcal{A} \geqslant 2$	0	$\mathcal{A} \geqslant 2$	338
40	21.61	32.89	0.12	$\mathcal{A} = 1$	5,262	$\mathcal{A} = 1$	256
				$\mathcal{A} = 2$	88	$\mathcal{A} = 2$	512
				$\mathcal{A} \geqslant 3$	0	$\mathcal{A} \geqslant 3$	600

Table 8.6: Results of using the analysis framework to count the number of interior and potential boundary misses incurred by loop nest $\mathbb{L}_{\text{mm}}$ for several problem sizes in ($\mathcal{A}$, 64, 16384 $\cdot\, \mathcal{A}$; 256) data caches.

data caches. In Table 8.6, notice that the formula-simplification times are quite a bit longer than for loop nest $\mathbb{L}_{\text{tri}}$ in Table 8.1, despite that the problem sizes are much smaller. There are two reasons for this. First, loop nest $\mathbb{L}_{\text{mm}}$ references two-dimensional arrays while $\mathbb{L}_{\text{tri}}$ references one-dimensional arrays. For $t = u = v = 20$, each array of $\mathbb{L}_{\text{mm}}$ has 400 elements. For $n = 1000$, each array of $\mathbb{L}_{\text{tri}}$ has 1000 elements. Second, loop nest $\mathbb{L}_{\text{mm}}$ is more complex than loop nest $\mathbb{L}_{\text{tri}}$. The complexity of a loop nest is related to its nesting depth, the loop bound expressions, the number of references and their index expressions, the number of arrays referenced, and if it is an imperfect loop nest. Loop nest $\mathbb{L}_{\text{tri}}$ has nesting depth 1, is perfectly-nested, and all three references have the same index expression. Loop nest $\mathbb{L}_{\text{mm}}$ has nesting depth 3, is imperfectly-nested, and three of four references have different index expressions. These factors can cause the simplification of a cache behavior formula to be more difficult and require more time. Also notice that for problem size $t = u = v = 20$, there are no interior misses for any value of $\mathcal{A}$ because all array elements referenced by $\mathbb{L}_{\text{mm}}$ fit in cache and never get replaced.

Table 8.7 gives the results of using the framework to count the number of interior and potential boundary misses incurred by $\mathbb{L}_{\text{mm}}$ for several problem sizes in ($\mathcal{A}$, 8192, 8192 $\cdot\, \mathcal{A}$; 1) TLBs. In Table 8.7, notice that for $t = u = v = 40$ the number of potential boundary misses is $\mathcal{A}$ for $1 \leqslant \mathcal{A} \leqslant 5$ and 5 for $\mathcal{A} > 5$. Recall that the number of potential boundary misses incurred by a loop nest is the cache footprint of the memory blocks it accesses, which is bounded from above by the number of cache frames (see Lemma 3.1). For problem size $t = u = v = 40$, $\mathbb{L}_{\text{mm}}$ accesses five 8192-byte pages of memory (or 4800 8-byte array elements). The number of frames in the cache is $\frac{\mathcal{C}}{\mathcal{B}} = \frac{8192 \cdot \mathcal{A}}{8192} = \mathcal{A}$. For $1 \leqslant \mathcal{A} \leqslant 5$, the number of cache frames $\mathcal{A}$ bounds the cache footprint of memory blocks accessed, and the number of potential boundary misses is $\mathcal{A}$. For $\mathcal{A} > 5$, the cache footprint is 5 frames, and the number of potential boundary misses is 5.

$t = u = v$	solution time (sec)			# of interior misses		# of pot. boundary misses	
	simplify formula	build DFA	count misses				
20	0.92	0.06	0.00	$\mathcal{A} = 1$	391	$\mathcal{A} = 1$	1
				$\mathcal{A} \geqslant 2$	0	$\mathcal{A} \geqslant 2$	2
30	1.31	0.57	0.08	$\mathcal{A} = 1$	47,511	$\mathcal{A} = 1$	1
				$\mathcal{A} = 2$	2,015	$\mathcal{A} = 2$	2
				$\mathcal{A} \geqslant 3$	0	$\mathcal{A} \geqslant 3$	3
40	1.54	11.38	0.27	$\mathcal{A} = 1$	117,007	$\mathcal{A} = 1$	1
				$\mathcal{A} = 2$	6,079	$\mathcal{A} = 2$	2
				$\mathcal{A} = 3$	3,638	$\mathcal{A} = 3$	3
				$\mathcal{A} = 4$	279	$\mathcal{A} = 4$	4
				$\mathcal{A} \geqslant 5$	0	$\mathcal{A} \geqslant 5$	5
50	1.86	86.00	0.63	$\mathcal{A} = 1$	242,755	$\mathcal{A} = 1$	1
				$\mathcal{A} = 2$	12,199	$\mathcal{A} = 2$	2
				$\mathcal{A} = 3$	9,848	$\mathcal{A} = 3$	3
				$\mathcal{A} = 4$	8,098	$\mathcal{A} = 4$	4
				$\mathcal{A} = 5$	699	$\mathcal{A} = 5$	5
				$\mathcal{A} = 6$	248	$\mathcal{A} = 6$	6
				$\mathcal{A} = 7$	197	$\mathcal{A} = 7$	7
				$\mathcal{A} \geqslant 8$	0	$\mathcal{A} \geqslant 8$	8

Table 8.7: Results of using the analysis framework to count the number of interior and potential boundary misses incurred by loop nest $\mathbb{L}_{\mathrm{mm}}$ for several problem sizes in ($\mathcal{A}$, 8192, $8192 \cdot \mathcal{A}$; 1) TLBs.

μ_0	μ_1	μ_2	solution time (sec)			# of interior misses		# of pot. boundary misses	
			simplify formula	build DFA	count misses				
0	7,200	14,400	1.25	0.05	0.03	$\mathcal{A} = 1$	82	$\mathcal{A} = 1$	256
						$\mathcal{A} \geqslant 2$	0	$\mathcal{A} \geqslant 2$	338
0	8,000	16,000	0.72	0.06	0.04	$\mathcal{A} = 1$	107	$\mathcal{A} = 1$	232
						$\mathcal{A} \geqslant 2$	0	$\mathcal{A} \geqslant 2$	339
0	16,384	32,768	1.44	53.73	0.08	$\mathcal{A} = 1$	3,595	$\mathcal{A} = 1$	113
						$\mathcal{A} = 2$	2,728	$\mathcal{A} = 2$	226
						$\mathcal{A} \geqslant 3$	0	$\mathcal{A} \geqslant 3$	339
7,200	0	14,400	0.93	0.18	0.04	$\mathcal{A} = 1$	161	$\mathcal{A} = 1$	256
						$\mathcal{A} \geqslant 2$	0	$\mathcal{A} \geqslant 2$	338
7,200	14,400	0	1.13	1.21	0.03	$\mathcal{A} = 1$	162	$\mathcal{A} = 1$	256
						$\mathcal{A} \geqslant 2$	0	$\mathcal{A} \geqslant 2$	338

Table 8.8: Results of using the analysis framework to count the number of interior and potential boundary misses incurred by loop nest $\mathbb{L}_{\text{mm}}$ for $m = n = 30$ in ($\mathcal{A}$, 64, $16384 \cdot \mathcal{A}$; 256) data caches with varying values of μ_0, μ_1, and μ_2.

Table 8.8 gives the results of using the framework to count the number of interior and potential boundary misses incurred by $\mathbb{L}_{\text{mm}}$ for $m = n = 30$ in ($\mathcal{A}$, 64, $16384 \cdot \mathcal{A}$; 256) data caches, where arrays $Y^{(0)} = \mathtt{X}$, $Y^{(1)} = \mathtt{Y}$, and $Y^{(2)} = \mathtt{Z}$ are double-precision (*i.e.*, $\beta_0 = \beta_1 = \beta_2 = 8$ bytes) and linearized in row-major order with varying starting addresses μ_0, μ_1, and μ_2. In Table 8.8, notice that varying array starting addresses in memory can have a dramatic effect on the number of cache misses. The different potential boundary miss counts are an indication of how changes in array starting addresses can affect the cache footprint of the memory blocks accessed by a loop nest. For example, when $\mu_0 = 0$, $\mu_1 = 16,384$ and $\mu_2 = 32,768$ all three arrays map "on top" of one another in cache[2] and occupy only 113 cache sets (113 is the number of memory blocks in each array). When $\mu_0 = 0$, $\mu_1 = 7200$ and $\mu_2 = 14,400$ all three arrays are contiguous in memory and occupy all 256 cache sets. The different interior miss counts are an indication of how changes in array starting addresses can affect the number of conflicts among arrays. In particular, when all three arrays map on top of one another in cache (*i.e.*, $\mu_0 = 0$, $\mu_1 = 16,384$ and $\mu_2 = 32,768$) there are many conflicts leading to a large number of interior misses for $\mathcal{A} = 1$ and 2, as expected.

Matrix multiplication loop nest variation. The matrix multiplication loop nest $\mathbb{L}_{\text{mm}}$ of Figure 2.5 is imperfectly nested because it stores array element `Z[i,j]` in the scalar variable `c`

[2] I say that array $Y^{(p)}$ maps "on top" of array $Y^{(q)}$ in cache when the difference in array starting addresses is a multiple of cache capacity, *i.e.*, $\exists k > 0 : |\mu_p - \mu_q| = k \cdot \mathcal{C}$. The effect is that the i^{th} elements in the linearizations of each array map to the same cache set, for $i \in [0, \min(\prod_{x=0}^{d_p-1} \ell_x, \prod_{x=0}^{d_q-1} \ell_x)$. For $\mu_0 = 0$, $\mu_1 = 16,384$ and $\mu_2 = 32,768$, all three arrays map on top of one another in cache when $\mathcal{A} = 1$, and arrays $Y^{(0)}$ and $Y^{(2)}$ map on top of each other in cache when $\mathcal{A} = 2$.

```
L_mm-var:  do i = 0, (t − 1)/2
              do j = 0, (u − 1)/2
                 c0 = Z[2i,2j]
                 c1 = Z[2i+1,2j]
                 c2 = Z[2i,2j+1]
                 c3 = Z[2i+1,2j+1]
                 do k = 0, v − 1
                    c4 = X[2i,k]
                    c5 = X[2i+1,k]
                    c6 = Y[k,2j]
                    c7 = Y[k,2j+1]
                    c0 = c4 * c6 + c0
                    c1 = c5 * c6 + c1
                    c2 = c4 * c7 + c2
                    c3 = c5 * c7 + c3
                 enddo
                 Z[2i,2j] = c0
                 Z[2i+1,2j] = c1
                 Z[2i,2j+1] = c2
                 Z[2i+1,2j+1] = c3
              enddo
           enddo
```

Figure 8.1: The matrix multiplication variation loop nest $\mathbb{L}_{\text{mm-var}}$.

so that it is register-resident for all iterations of the innermost loop with LCV `k`. A perfectly-nested version of the matrix multiplication loop nest accesses array element `Z[i,j]` during every iteration of the `k`-loop. Storing `Z[i,j]` in a scalar variable is an optimization that prevents any cache misses on the array element during the `k`-loop. As a variation on loop nest $\mathbb{L}_{\text{mm}}$, one can extend this optimization to store several array elements from arrays `X`, `Y`, and `Z` in scalar variables. Consider loop nest $\mathbb{L}_{\text{mm-var}}$ in Figure 8.1 as such a variation on matrix multiplication. One can expect the cache performance of loop nest $\mathbb{L}_{\text{mm-var}}$ to be as good or better than loop nest $\mathbb{L}_{\text{mm}}$ because of the scalar replacement optimization.

In loop nest $\mathbb{L}_{\text{mm-var}}$, the upper bounds on the `i`-loop and `j`-loop may look a bit strange. This is a consequence of loop normalization.[3] In the original matrix multiplication variation loop nest, `i` goes from 0 to $t-1$ in steps of 2, and `j` goes from 0 to $u-1$ in steps of 2. In order to make the step sizes of the `i`-loop and `j`-loop equal to one (a requirement of the cache analysis framework), I performed loop normalization to get loop nest $\mathbb{L}_{\text{mm-var}}$. Suppose that the arrays $Y^{(0)} = \texttt{X}$, $Y^{(1)} = \texttt{Y}$, and $Y^{(2)} = \texttt{Z}$ are double-precision (*i.e.*, $\beta_0 = \beta_1 = \beta_2 = 8$ bytes) and linearized in column-major order with starting addresses $\mu_0 = 0$, $\mu_1 = \mu_0 + 8t \cdot v$, and $\mu_2 = \mu_1 + 8v \cdot u$.

Table 8.9 gives the results of using the framework to count the number of interior misses

[3]Loop normalization transforms all loops in a loop nest into normal form, in which all step sizes are equal to one [135].

$t = u = v$	solution time (sec)			# of interior misses		# of pot. boundary misses	
	simplify formula	build DFA	count misses				
20	36.74	0.05	0.03	$\mathcal{A} \geqslant 1$	0	$\mathcal{A} \geqslant 1$	150
30	70.24	0.74	0.04	$\mathcal{A} = 1$	114	$\mathcal{A} = 1$	256
				$\mathcal{A} \geqslant 2$	0	$\mathcal{A} \geqslant 2$	338
40	83.15	21.16	0.08	$\mathcal{A} = 1$	2,754	$\mathcal{A} = 1$	256
				$\mathcal{A} = 2$	88	$\mathcal{A} = 2$	512
				$\mathcal{A} \geqslant 3$	0	$\mathcal{A} \geqslant 3$	600

Table 8.9: Results of using the analysis framework to count the number of interior and potential boundary misses incurred by loop nest $\mathbb{L}_{\text{mm-var}}$ for several problem sizes in ($\mathcal{A}$, 64, $16384 \cdot \mathcal{A}$; 256) data caches.

$t = u = v$	solution time (sec)			# of interior misses		# of pot. boundary misses	
	simplify formula	build DFA	count misses				
20	6.69	0.02	0.01	$\mathcal{A} = 1$	115	$\mathcal{A} = 1$	1
				$\mathcal{A} \geqslant 2$	0	$\mathcal{A} \geqslant 2$	2
30	8.37	0.43	0.06	$\mathcal{A} = 1$	11,954	$\mathcal{A} = 1$	1
				$\mathcal{A} = 2$	524	$\mathcal{A} = 2$	2
				$\mathcal{A} \geqslant 3$	0	$\mathcal{A} \geqslant 3$	3
40	14.20	23.89	0.12	$\mathcal{A} = 1$	30,559	$\mathcal{A} = 1$	1
				$\mathcal{A} = 2$	4,403	$\mathcal{A} = 2$	2
				$\mathcal{A} = 3$	994	$\mathcal{A} = 3$	3
				$\mathcal{A} = 4$	191	$\mathcal{A} = 4$	4
				$\mathcal{A} \geqslant 5$	0	$\mathcal{A} \geqslant 5$	5
50	19.85	94.37	0.24	$\mathcal{A} = 1$	61,586	$\mathcal{A} = 1$	1
				$\mathcal{A} = 2$	4,719	$\mathcal{A} = 2$	2
				$\mathcal{A} = 3$	2,669	$\mathcal{A} = 3$	3
				$\mathcal{A} = 4$	2,226	$\mathcal{A} = 4$	4
				$\mathcal{A} = 5$	564	$\mathcal{A} = 5$	5
				$\mathcal{A} = 6$	123	$\mathcal{A} = 6$	6
				$\mathcal{A} = 7$	97	$\mathcal{A} = 7$	7
				$\mathcal{A} \geqslant 8$	0	$\mathcal{A} \geqslant 8$	8

Table 8.10: Results of using the analysis framework to count the number of interior and potential boundary misses incurred by loop nest $\mathbb{L}_{\text{mm-var}}$ for several problem sizes in ($\mathcal{A}$, 8192, $8192 \cdot \mathcal{A}$; 1) TLBs.

```
𝕃calc3:  do j = 1, n
            do i = 1, m
               UOLD[i,j] = U[i,j] + a*(UNEW[i,j] - 2*U[i,j] + UOLD[i,j])
               VOLD[i,j] = V[i,j] + a*(VNEW[i,j] - 2*V[i,j] + VOLD[i,j])
               POLD[i,j] = P[i,j] + a*(PNEW[i,j] - 2*P[i,j] + POLD[i,j])
               U[i,j] = UNEW[i,j]
               V[i,j] = VNEW[i,j]
               P[i,j] = PNEW[i,j]
            enddo
         enddo
```

Figure 8.2: Loop nest $\mathbb{L}_{\text{calc3}}$ from the `calc3` subroutine.

incurred by $\mathbb{L}_{\text{mm-var}}$ for several problem sizes in ($\mathcal{A}$, 64, 16384·$\mathcal{A}$; 256) data caches. In Table 8.9, notice that all components of the solution times are larger than for corresponding problem sizes in Table 8.6 for loop nest $\mathbb{L}_{\text{mm}}$. The reason is that loop nest $\mathbb{L}_{\text{mm-var}}$ is more complex than loop nest $\mathbb{L}_{\text{mm}}$. In particular, $\mathbb{L}_{\text{mm-var}}$ makes many more array references than $\mathbb{L}_{\text{mm}}$. Also notice that the optimization does lead to fewer interior misses for problem size $t = u = v = 40$ and associativity value $\mathcal{A} = 1$. Otherwise, the cache misses counts are the same as in Table 8.6.

Table 8.10 gives the results of using the framework to count the number of interior misses incurred by $\mathbb{L}_{\text{mm-var}}$ for several problem sizes in ($\mathcal{A}$, 8192, 8192 · $\mathcal{A}$; 1) TLBs. In Table 8.10, notice that all components of the solution times are larger than for corresponding problem sizes in Table 8.7 for loop nest $\mathbb{L}_{\text{mm}}$, due to the relative complexity of loop nest $\mathbb{L}_{\text{mm-var}}$ (as is the case for the data cache results above). Also notice that the optimization does lead to fewer interior misses in all cases.

Calc3 loop nest. Consider the loop nest $\mathbb{L}_{\text{calc3}}$ from the `calc3` subroutine of `swim.f` in SPECfp95 [115] in Figure 8.2. Suppose that the arrays $Y^{(0)}$ = `U`, $Y^{(1)}$ = `V`, $Y^{(2)}$ = `P`, $Y^{(3)}$ = `UNEW`, $Y^{(4)}$ = `VNEW`, $Y^{(5)}$ = `PNEW`, $Y^{(6)}$ = `UOLD`, $Y^{(7)}$ = `VOLD`, and $Y^{(8)}$ = `POLD` are double-precision (*i.e.*, $\beta_k = 8$ bytes, $\forall k \in [0, 8]$) and linearized in column-major order with starting addresses $\mu_0 = 0$ and $\mu_k = \mu_{k-1} + 8m \cdot n, \forall k \in [1, 8]$.

Table 8.11 gives the results of using the framework to count the number of compulsory and replacement misses incurred by $\mathbb{L}_{\text{calc3}}$ for several problem sizes in ($\mathcal{A}$, 64, 16384 · $\mathcal{A}$; 256) data caches. In Table 8.11, notice that for problem size $m = 128$, $n = 50$ the DFA-construction time is large relative to that of the other problem sizes. The large DFA-construction time is due to the thrashing caused by this problem size. The thrashing results in a large amount of cache activity that must be represented by a DFA and leads to a relatively long construction time. For problem size $m = 750$, $n = 25$ there is some isolated thrashing that leads to a small amount of cache activity. For the problem sizes without thrashing, there are no replacement misses for any value of $\mathcal{A}$. Like loop nest $\mathbb{L}_{\text{tri}}$, loop nest $\mathbb{L}_{\text{calc3}}$ has no temporal reuse of data. Therefore, if the spatial reuse can be exploited without conflict, then the only cache misses

m n	solution time (sec)			# of compulsory misses	# of replacement misses	
	simplify formula	build DFA	count misses			
100 100	2.51	0.38	0.00	11,250	$\mathcal{A} \geqslant 1$	0
128 50	1.30	5.67	0.16	7,200	$\mathcal{A} = 1$	12,000
					$\mathcal{A} \geqslant 2$	0
750 25	4.15	0.82	0.00	21,094	$1 \leqslant \mathcal{A} \leqslant 81$	6
					$\mathcal{A} = 82$	3
					$\mathcal{A} \geqslant 83$	0
500 500	2.50	1.27	0.00	281,250	$\mathcal{A} \geqslant 1$	0
1,000 1,000	1.26	2.25	0.00	1,125,000	$\mathcal{A} \geqslant 1$	0

Table 8.11: Results of using the analysis framework to count the number of compulsory and replacement misses incurred by loop nest $\mathbb{L}_{\text{calc3}}$ for several problem sizes in ($\mathcal{A}$, 64, 16384 $\cdot$ $\mathcal{A}$; 256) data caches.

m n	solution time (sec)			# of compulsory misses	# of replacement misses	
	simplify formula	build DFA	count misses			
20 20	3.26	0.06	0.02	2	$\mathcal{A} = 1$	5,503
					$\mathcal{A} \geqslant 1$	0
30 30	4.54	8.25	0.07	4	$\mathcal{A} = 1$	18,648
					$\mathcal{A} = 2$	11,201
					$\mathcal{A} = 3$	4,159
					$\mathcal{A} \geqslant 4$	0

Table 8.12: Results of using the analysis framework to count the number of compulsory and replacement misses incurred by loop nest $\mathbb{L}_{\text{calc3}}$ for several problem sizes in ($\mathcal{A}$, 16384, 16384$\cdot$$\mathcal{A}$; 1) TLBs.

are compulsory misses.

Table 8.12 gives the results of using the framework to count the number of compulsory and replacement misses incurred by $\mathbb{L}_{\text{calc3}}$ for several problem sizes in ($\mathcal{A}$, 16384, 16384 $\cdot$ $\mathcal{A}$; 1) TLBs. In Table 8.12, notice that even though the problem sizes are smaller than in Table 8.11, the solution times are essentially larger. The reason for this is that in the TLB with only one cache set, there is much contention for the set resulting in a large amount of cache activity, which must be modeled.

General Observations. The following are some observations on all of the results for single loop nests presented in this section.

1. The time required for formula expression and simplification increases as loop nests become more complex, but does not necessarily increase with problem size or cache miss

count. The complexity of a loop nest is related to its nesting depth, the loop bound expressions, the number of references and their index expressions, the number of arrays referenced, and whether it is an imperfect loop nest.

2. The time required for DFA-construction increases with the number of cache misses. This increase is most evident for replacement, interior, and potential boundary misses because the DFAs represents zero or more r-witnesses, i-witnesses, or b-witnesses for each memory access. For compulsory misses, the increase is less evident because such misses occur independent of associativity and the DFAs represent actual misses, and as a result, encode less information than when representing witnesses.

3. The time required to count misses increases with the number of cache misses. Recall that counting replacement, interior, and potential boundary misses involves enumerating the accepting paths in the DFA representing a r-witness, i-witness, or b-witness formula and counting the number of such witnesses to each memory access. To count compulsory misses, one can simply count the number of accepting paths in the DFA representing a compulsory-miss formula (see Section 5.4.2). As a consequence, it requires much less time to count the number compulsory misses in a loop nest than to count other types of cache misses.

8.2 Loop Nest Sequence

To demonstrate the composability of program fragments allowed by the interior-boundary miss classification, I give cache miss counts for the sequence of four loop nests in Figure 8.3, from the `calc3` subroutine of `swim.f` in SPECfp95. Notice that even though $\mathbb{L}_3$ is not a true loop nest, it can be treated as one by including its statements in a loop with one iteration. Suppose that the arrays $Y^{(0)} = \texttt{U}$, $Y^{(1)} = \texttt{V}$, $Y^{(2)} = \texttt{P}$, $Y^{(3)} = \texttt{UNEW}$, $Y^{(4)} = \texttt{VNEW}$, $Y^{(5)} = \texttt{PNEW}$, $Y^{(6)} = \texttt{UOLD}$, $Y^{(7)} = \texttt{VOLD}$, and $Y^{(8)} = \texttt{POLD}$ are double-precision (*i.e.*, $\beta_k = 8, \forall k \in [0, 8]$ bytes) and linearized in column-major order with starting addresses $\mu_0 = 0$ and $\mu_k = \mu_{k-1} + 8(m+1) \cdot (n+1), \forall k \in [1, 8]$.

Tables 8.13 to 8.16 give the results of using the framework to count the number of boundary and interior misses incurred by each loop nest in `calc3` for several problem sizes in ($\mathcal{A}$, 64, $16384 \cdot \mathcal{A}$; 256) data caches. The cache is assumed to be cold at the beginning of the loop nest sequence (*i.e.*, the cache state is empty). Results include the time required for formula expression and simplification, the time required for DFA construction, the time required to enumerate formula solutions and count witnesses, and the number of cache misses for associativity value $\mathcal{A}$. Solution time (for each problem size and loop nest) is the total time for computing interior misses, boundary misses, and cache state. The total number of cache misses incurred by the loop nest sequence is the sum of misses incurred by all of the loop nests. Table 8.17 gives the

```
𝕃₀:  do j = 1, n
        do i = 1, m
           UOLD[i,j] = U[i,j] + a*(UNEW[i,j] - 2*U[i,j] + UOLD[i,j])
           VOLD[i,j] = V[i,j] + a*(VNEW[i,j] - 2*V[i,j] + VOLD[i,j])
           POLD[i,j] = P[i,j] + a*(PNEW[i,j] - 2*P[i,j] + POLD[i,j])
           U[i,j] = UNEW[i,j]
           V[i,j] = VNEW[i,j]
           P[i,j] = PNEW[i,j]
        enddo
     enddo
𝕃₁:  do j = 1, n
        UOLD[m+1,j] = UOLD[1,j]
        VOLD[m+1,j] = VOLD[1,j]
        POLD[m+1,j] = POLD[1,j]
        U[m+1,j] = U[1,j]
        V[m+1,j] = V[1,j]
        P[m+1,j] = P[1,j]
     enddo
𝕃₂:  do i = 1, m
        UOLD[i,n+1] = UOLD[i,1]
        VOLD[i,n+1] = VOLD[i,1]
        POLD[i,n+1] = POLD[i,1]
        U[i,n+1] = U[i,1]
        V[i,n+1] = V[i,1]
        P[i,n+1] = P[i,1]
     enddo
𝕃₃:  UOLD[m+1,n+1] = UOLD[1,1]
     VOLD[m+1,n+1] = VOLD[1,1]
     POLD[m+1,n+1] = POLD[1,1]
     U[m+1,n+1] = U[1,1]
     V[m+1,n+1] = V[1,1]
     P[m+1,n+1] = P[1,1]
```

Figure 8.3: The sequence of four loop nests from the `calc3` subroutine.

$m\ n$	$\mathbb{L}$	solution time (sec)			# of interior misses		# of boundary misses	
		simplify formula	build DFA	count misses				
19 19	$\mathbb{L}_0$	11.78	0.49	0.11	$\mathcal{A} = 1$	180	$\mathcal{A} = 1$	252
					$\mathcal{A} \geqslant 2$	0	$\mathcal{A} \geqslant 2$	432
	$\mathbb{L}_1$	2.22	0.17	0.07	$\mathcal{A} = 1$	19	$\mathcal{A} = 1$	81
					$\mathcal{A} \geqslant 2$	0	$\mathcal{A} \geqslant 2$	0
	$\mathbb{L}_2$	1.44	0.04	0.07	$\mathcal{A} \geqslant 1$	0	$\mathcal{A} = 1$	21
							$\mathcal{A} \geqslant 2$	12
	$\mathbb{L}_3$	0.55	0.01	0.07	$\mathcal{A} \geqslant 1$	0	$\mathcal{A} \geqslant 1$	0

Table 8.13: Results of using the analysis framework to count the number of boundary and interior misses incurred by the loop nest sequence in `calc3` for $m = n = 19$ in ($\mathcal{A}$, 64, $16384 \cdot \mathcal{A}$; 256) data caches.

$m\ n$	$\mathbb{L}$	solution time (sec)			# of interior misses		# of boundary misses	
		simplify formula	build DFA	count misses				
29 29	$\mathbb{L}_0$	21.00	4.18	0.17	$\mathcal{A} = 1$	729	$\mathcal{A} = 1$	256
					$\mathcal{A} = 2$	473	$\mathcal{A} = 2$	512
					$\mathcal{A} = 3$	217	$\mathcal{A} = 3$	768
					$\mathcal{A} \geqslant 4$	0	$\mathcal{A} \geqslant 4$	985
	$\mathbb{L}_1$	5.05	1.42	0.08	$\mathcal{A} = 1$	42	$\mathcal{A} = 1$	143
					$\mathcal{A} \geqslant 2$	0	$\mathcal{A} = 2$	118
							$\mathcal{A} = 3$	55
							$\mathcal{A} \geqslant 4$	0
	$\mathbb{L}_2$	2.15	0.04	0.10	$\mathcal{A} \geqslant 1$	0	$\mathcal{A} = 1$	38
							$\mathcal{A} = 2$	34
							$\mathcal{A} = 3$	34
							$\mathcal{A} \geqslant 4$	19
	$\mathbb{L}_3$	0.49	0.02	0.08	$\mathcal{A} \geqslant 1$	0	$\mathcal{A} \geqslant 1$	0

Table 8.14: Results of using the analysis framework to count the number of boundary and interior misses incurred by the loop nest sequence in `calc3` for $m = n = 29$ in ($\mathcal{A}$, 64, $16384 \cdot \mathcal{A}$; 256) data caches.

m n	$\mathbb{L}$	solution time (sec)			# of interior misses		# of boundary misses	
		simplify formula	build DFA	count misses				
19 99	$\mathbb{L}_0$	11.44	25.89	0.19	$\mathcal{A} = 1$	1,976	$\mathcal{A} = 1$	256
					$\mathcal{A} = 2$	1,720	$\mathcal{A} = 2$	512
					$\mathcal{A} = 3$	1,464	$\mathcal{A} = 3$	768
					$\mathcal{A} = 4$	1,208	$\mathcal{A} = 4$	1,024
					$\mathcal{A} = 5$	952	$\mathcal{A} = 5$	1,280
					$\mathcal{A} = 6$	696	$\mathcal{A} = 6$	1,536
					$\mathcal{A} = 7$	440	$\mathcal{A} = 7$	1,792
					$\mathcal{A} = 8$	200	$\mathcal{A} = 8$	2,032
					$\mathcal{A} \geqslant 9$	0	$\mathcal{A} \geqslant 9$	2,232
	$\mathbb{L}_1$	2.17	6.23	0.14	$\mathcal{A} = 1$	638	$\mathcal{A} = 1$	256
					$\mathcal{A} = 2$	382	$\mathcal{A} = 2$	508
					$\mathcal{A} = 3$	139	$\mathcal{A} = 3$	688
					$\mathcal{A} \geqslant 4$	0	$\mathcal{A} = 4$	597
							$\mathcal{A} = 5$	453
							$\mathcal{A} = 6$	420
							$\mathcal{A} = 7$	379
							$\mathcal{A} = 8$	160
							$\mathcal{A} \geqslant 9$	0
	$\mathbb{L}_2$	1.44	0.04	0.09	$\mathcal{A} \geqslant 1$	0	$\mathcal{A} = 1$	30
							$\mathcal{A} = 2$	28
							$\mathcal{A} = 3$	23
							$4 \leqslant \mathcal{A} \leqslant 7$	18
							$\mathcal{A} = 8$	13
							$\mathcal{A} \geqslant 9$	12
	$\mathbb{L}_3$	0.54	0.01	0.09	$\mathcal{A} \geqslant 1$	0	$\mathcal{A} \geqslant 1$	0

Table 8.15: Results of using the analysis framework to count the number of boundary and interior misses incurred by the loop nest sequence in `calc3` for $m = 19$, $n = 99$ in ($\mathcal{A}$, 64, $16384 \cdot \mathcal{A}$; 256) data caches.

m n	$\mathbb{L}$	solution time (sec)			# of interior misses		# of boundary misses	
		simplify formula	build DFA	count misses				
99 19	$\mathbb{L}_0$	13.16	70.62	0.23	$\mathcal{A} = 1$	1,886	$\mathcal{A} = 1$	256
					$\mathcal{A} = 2$	1,630	$\mathcal{A} = 2$	512
					$\mathcal{A} = 3$	1,374	$\mathcal{A} = 3$	768
					$\mathcal{A} = 4$	1,118	$\mathcal{A} = 4$	1,024
					$\mathcal{A} = 5$	862	$\mathcal{A} = 5$	1,280
					$\mathcal{A} = 6$	606	$\mathcal{A} = 6$	1,536
					$\mathcal{A} = 7$	392	$\mathcal{A} = 7$	1,750
					$\mathcal{A} = 8$	190	$\mathcal{A} = 8$	1,952
					$\mathcal{A} \geqslant 9$	0	$\mathcal{A} \geqslant 9$	2,142
	$\mathbb{L}_1$	2.25	6.19	0.09	$\mathcal{A} = 1$	47	$\mathcal{A} = 1$	115
					$\mathcal{A} \geqslant 2$	0	$\mathcal{A} = 2$	117
							$\mathcal{A} = 3$	94
							$\mathcal{A} = 4$	82
							$\mathcal{A} = 5$	79
							$\mathcal{A} = 6$	75
							$\mathcal{A} = 7$	62
							$\mathcal{A} = 8$	23
							$\mathcal{A} \geqslant 9$	0
	$\mathbb{L}_2$	2.25	0.40	0.11	$\mathcal{A} = 1$	76	$\mathcal{A} = 1$	71
					$\mathcal{A} = 2$	8	$\mathcal{A} = 2$	134
					$\mathcal{A} \geqslant 3$	0	$\mathcal{A} = 3$	139
							$4 \leqslant \mathcal{A} \leqslant 5$	138
							$\mathcal{A} = 6$	105
							$\mathcal{A} = 7$	100
							$\mathcal{A} = 8$	89
							$\mathcal{A} \geqslant 9$	72
	$\mathbb{L}_3$	0.54	0.02	0.10	$\mathcal{A} \geqslant 1$	0	$\mathcal{A} = 1$	6
							$\mathcal{A} = 2$	2
							$\mathcal{A} \geqslant 3$	0

Table 8.16: Results of using the analysis framework to count the number of boundary and interior misses incurred by the loop nest sequence in `calc3` for $m = 99$, $n = 19$ in ($\mathcal{A}$, 64, $16384 \cdot \mathcal{A}$; 256) data caches.

m n	$\mathbb{L}$	solution time (sec)			# of interior misses		# of boundary misses	
		simplify formula	build DFA	count misses				
19 19	$\mathbb{L}_0$	16.65	4.09	0.06	$\mathcal{A} = 1$	7,367	$\mathcal{A} = 1$	1
					$\mathcal{A} = 2$	4,174	$\mathcal{A} = 2$	2
					$\mathcal{A} = 3$	1,139	$\mathcal{A} = 3$	3
					$\mathcal{A} \geqslant 4$	0	$\mathcal{A} \geqslant 4$	4
	$\mathbb{L}_1$	1.77	0.19	0.00	$\mathcal{A} = 1$	64	$\mathcal{A} = 1$	1
					$\mathcal{A} = 2$	63	$\mathcal{A} = 2$	1
					$\mathcal{A} = 3$	29	$\mathcal{A} = 3$	2
					$\mathcal{A} \geqslant 4$	0	$\mathcal{A} \geqslant 4$	0
	$\mathbb{L}_2$	0.87	0.05	0.00	$\mathcal{A} = 1$	75	$\mathcal{A} = 1$	1
					$\mathcal{A} = 2$	74	$\mathcal{A} = 2$	2
					$\mathcal{A} = 3$	73	$\mathcal{A} = 3$	3
					$\mathcal{A} \geqslant 4$	0	$\mathcal{A} \geqslant 4$	0
	$\mathbb{L}_3$	0.51	0.02	0.00	$\mathcal{A} = 1$	3	$\mathcal{A} = 1$	1
					$\mathcal{A} = 2$	2	$\mathcal{A} = 2$	2
					$\mathcal{A} = 3$	1	$\mathcal{A} = 3$	3
					$\mathcal{A} \geqslant 4$	0	$\mathcal{A} \geqslant 4$	0

Table 8.17: Results of using the analysis framework to count the number of boundary and interior misses incurred by the loop nest sequence in `calc3` for $m = n = 19$ in ($\mathcal{A}$, 8192, $8192 \cdot \mathcal{A}$; 1) TLBs.

results of using the framework to count the number of boundary and interior misses incurred by each loop nest in `calc3` for problem size $m = 19$, $n = 19$ in ($\mathcal{A}$, 8192, $8192 \cdot \mathcal{A}$; 1) TLBs.

For comparison, I also give cache miss counts for the sequence of loop nests in `calc3` using the compulsory-replacement classification of misses. Table 8.18 gives the results of using the framework to count the number of compulsory and replacement misses incurred by each loop nest in `calc3` for several problem sizes in ($\mathcal{A}$, 64, $16384 \cdot \mathcal{A}$; 256) data caches. For Tables 8.13 to 8.17 shown above, which give interior and boundary misses, the total number of cache misses incurred by the loop nest sequence is the sum of misses incurred by all of the loop nests. In contrast, the sum of misses incurred by all of the loop nests in Table 8.18 is not necessarily the total number of cache misses incurred by the loop nest sequence. In fact, the sum of misses incurred by all of the loop nests is likely an overestimation of the total number of cache misses incurred by the loop nest sequence. The overestimation is due to compulsory misses incurred by loop nests $\mathbb{L}_1$, $\mathbb{L}_2$, and $\mathbb{L}_3$ that may actually be cache hits. The compulsory-replacement miss classification assumes that the cache is empty at the beginning of each loop nest. In particular for this sequence of loop nests from `calc3`, in which all loop nests access closely related data, assuming an empty cache leads to a misclassification of some cache hits as misses. It is precisely the aim of the interior-boundary classification to avoid overestimating miss counts.

m n	$\mathbb{L}$	solution time (sec)			# of compulsory misses	# of replacement misses	
		simplify formula	build DFA	count misses			
99 99	$\mathbb{L}_0$	2.33	0.39	0.00	11,142	$\mathcal{A} \geqslant 1$	0
	$\mathbb{L}_1$	0.41	3.81	0.00	894	$\mathcal{A} \geqslant 1$	0
	$\mathbb{L}_2$	0.29	0.02	0.00	156	$\mathcal{A} \geqslant 1$	0
	$\mathbb{L}_3$	0.09	0.01	0.00	12	$\mathcal{A} \geqslant 1$	0
127 49	$\mathbb{L}_0$	1.17	5.41	0.16	7,056	$\mathcal{A} = 1$	11,662
						$\mathcal{A} \geqslant 2$	0
	$\mathbb{L}_1$	0.17	0.80	0.00	588	$\mathcal{A} \geqslant 1$	0
	$\mathbb{L}_2$	0.27	0.03	0.01	192	$\mathcal{A} = 1$	444
						$\mathcal{A} \geqslant 2$	0
	$\mathbb{L}_3$	0.12	0.00	0.00	12	$\mathcal{A} \geqslant 1$	0
749 24	$\mathbb{L}_0$	3.75	0.58	0.00	20,256	$\mathcal{A} \geqslant 1$	0
	$\mathbb{L}_1$	1.00	10.22	0.00	184	$\mathcal{A} \geqslant 1$	0
	$\mathbb{L}_2$	0.47	0.22	0.00	1,131	$1 \leqslant \mathcal{A} \leqslant 6$	3
						$\mathcal{A} = 7$	1
						$\mathcal{A} \geqslant 8$	0
	$\mathbb{L}_3$	0.15	0.00	0.00	9	$\mathcal{A} \geqslant 1$	0
499 499	$\mathbb{L}_0$	2.43	1.37	0.00	280,692	$\mathcal{A} \geqslant 1$	0
	$\mathbb{L}_1$	0.41	11.96	0.00	4,494	$\mathcal{A} \geqslant 1$	0
	$\mathbb{L}_2$	0.30	0.27	0.00	756	$\mathcal{A} \geqslant 1$	0
	$\mathbb{L}_3$	0.10	0.01	0.00	12	$\mathcal{A} \geqslant 1$	0
999 999	$\mathbb{L}_0$	1.19	2.46	0.00	1,123,875	$\mathcal{A} \geqslant 1$	0
	$\mathbb{L}_1$	0.15	5.05	0.00	11,988	$\mathcal{A} \geqslant 1$	0
	$\mathbb{L}_2$	0.26	0.89	0.00	1,500	$\mathcal{A} \geqslant 1$	0
	$\mathbb{L}_3$	0.10	0.01	0.00	12	$\mathcal{A} \geqslant 1$	0

Table 8.18: Results of using the analysis framework to count the number of compulsory and replacement misses incurred by the loop nest sequence in `calc3` for several problem sizes in ($\mathcal{A}$, 64, $16384 \cdot \mathcal{A}$; 256) data caches.

Observations. The following are some observations on the results for the `calc3` loop nest sequence presented in this section.

1. In Tables 8.13 and 8.17 the formula-simplification times are roughly the same for corresponding loop nests, despite the differing problem sizes and cache configurations.

2. Loop nest $\mathbb{L}_0$ is the most complicated of the `calc3` sequence with 21 references (compared to 12 for each of the other loop nests), a nesting depth of 2 (compared to 1 for each of the other loop nests), and more complex index expressions. As a consequence, the solution times for $\mathbb{L}_0$ are considerably larger than for the other loop nests in all tables.

3. In Tables 8.13 and 8.17 the number of *actual* boundary misses are given, rather than the number of *potential* boundary misses. Actual boundary misses in loop nest $\mathbb{L}_\mathrm{i}$ are determined by resolving the potential boundary misses in $\mathbb{L}_\mathrm{i}$ against the computed cache state at the end of $\mathbb{L}_\mathrm{i-1}$'s execution (see Section 3.2). The potential boundary misses in $\mathbb{L}_0$ are resolved against an empty cache state.

4. Notice that the problem sizes in Table 8.18 for compulsory and replacement misses are larger than those in Tables 8.13 to 8.17 for boundary and interior misses. Furthermore, the solution times given in Table 8.18 are essentially smaller than those given in Tables 8.13 to 8.17. Because there is no temporal reuse in the loop nests of `calc3`, they incur no replacement misses except in cases of thrashing (see problem sizes $m = 127$, $n = 49$ and $m = 749$, $n = 24$), and the majority of misses are of the compulsory type. As discussed previously, the time required to build DFAs and count misses is usually smaller for compulsory misses, than for other types of misses.

8.3 Nonlinear Data Layouts

To demonstrate the ability of the cache analysis framework to handle nonlinear data layouts (see Section 7.1 for background), I give cache miss counts for the matrix-vector multiplication loop nest $\mathbb{L}_\mathrm{vec}$ where array $Y^{(2)} =$ `A` has a nonlinear data layout. Suppose that the arrays $Y^{(0)} =$ `A`, $Y^{(1)} =$ `X`, and $Y^{(2)} =$ `Y` are double-precision (*i.e.*, $\beta_0 = \beta_1 = \beta_2 = 8$ bytes) with starting addresses $\mu_0 = 0$, $\mu_1 = \mu_0 + 8m \cdot n$, and $\mu_2 = \mu_1 + 8n$. The data of array $Y^{(0)}$ is laid out in memory according to a (m, n)-interleaving σ, which describes the order in which bits from the two array coordinates are interleaved to linearize array $Y^{(0)}$.

Tables 8.19 to 8.22 give the results of using the framework to count the number of potential boundary and interior misses incurred by loop nest $\mathbb{L}_\mathrm{vec}$ for several problem sizes in ($\mathcal{A}$, 64, $16384 \cdot \mathcal{A}$; 256) data caches. Results include the time required for formula expression and simplification, the time required for DFA construction, the time required to enumerate formula solutions and count witnesses, and the number of cache misses for associativity value $\mathcal{A}$.

σ	solution time (sec)			# of interior misses		# of pot. boundary misses	
	simplify formula	build DFA	count misses				
00001111	1.80	0.04	0.00	$\mathcal{A}=1$ $\mathcal{A}\geqslant 2$	33 0	$\mathcal{A}=1$ $\mathcal{A}\geqslant 2$	64 72
00100111	7.91	0.04	0.00	$\mathcal{A}=1$ $\mathcal{A}\geqslant 2$	23 0	$\mathcal{A}=1$ $\mathcal{A}\geqslant 2$	64 72
00011011	15.69	0.04	0.00	$\mathcal{A}=1$ $\mathcal{A}\geqslant 2$	17 0	$\mathcal{A}=1$ $\mathcal{A}\geqslant 2$	64 72
00110011	9.48	0.04	0.00	$\mathcal{A}=1$ $\mathcal{A}\geqslant 2$	18 0	$\mathcal{A}=1$ $\mathcal{A}\geqslant 2$	64 72
01000111	4.28	0.04	0.00	$\mathcal{A}=1$ $\mathcal{A}\geqslant 2$	23 0	$\mathcal{A}=1$ $\mathcal{A}\geqslant 2$	64 72
10000111	3.13	0.04	0.00	$\mathcal{A}=1$ $\mathcal{A}\geqslant 2$	23 0	$\mathcal{A}=1$ $\mathcal{A}\geqslant 2$	64 72
10010011	9.56	0.04	0.00	$\mathcal{A}=1$ $\mathcal{A}\geqslant 2$	18 0	$\mathcal{A}=1$ $\mathcal{A}\geqslant 2$	64 72
11110000	1.90	0.04	0.00	$\mathcal{A}=1$ $\mathcal{A}\geqslant 2$	63 0	$\mathcal{A}=1$ $\mathcal{A}\geqslant 2$	64 72

Table 8.19: Results of using the analysis framework to count the number of interior and potential boundary misses incurred by loop nest $\mathbb{L}_{\text{vec}}$ for $m = n = 16$ and various $(4, 4)$-interleavings σ in ($\mathcal{A}$, 32, $2048 \cdot \mathcal{A}$; 64) data caches.

Observations. The following are some observations on the results for nonlinear data layouts presented in this section.

1. For $m = 2^x$ and $n = 2^y$, an interleaving of x 0s followed by y 1s is equivalent to a row-major layout, and an interleaving of y 1s followed by x 0s is equivalent to a column-major layout. Loop nest $\mathbb{L}_{\text{vec}}$ usually incurs fewer interior misses when array $Y^{(0)}$ has a nonlinear data layout compared to row- and column-major.

2. For each problem size, the interleaving(s) with the fewest interior misses for some value of $\mathcal{A}$ may not have the fewest number for other values of $\mathcal{A}$.

3. For each problem size, all interleavings have the same boundary miss counts, as expected.

4. The time required for formula simplification varies with different interleavings, but tends to increase with the degree of interleaving (*i.e.*, $\sigma = 00110011$ is more interleaved than $\sigma = 10000111$).

5. The time required for DFA-construction does not seem to be related to the degree of interleaving or miss counts.

σ	solution time (sec)			# of interior misses		# of pot. boundary misses	
	simplify formula	build DFA	count misses				
000011111	2.25	0.23	0.00	$\mathcal{A}=1$	182	$\mathcal{A}=1$	64
				$\mathcal{A}=2$	12	$\mathcal{A}=2$	128
				$\mathcal{A}\geqslant 3$	0	$\mathcal{A}\geqslant 3$	140
000111011	17.80	0.17	0.01	$\mathcal{A}=1$	102	$\mathcal{A}=1$	64
				$\mathcal{A}=2$	12	$\mathcal{A}=2$	128
				$\mathcal{A}\geqslant 3$	0	$\mathcal{A}\geqslant 3$	140
010001111	4.78	0.20	0.01	$\mathcal{A}=1$	138	$\mathcal{A}=1$	64
				$\mathcal{A}=2$	12	$\mathcal{A}=2$	128
				$\mathcal{A}\geqslant 3$	0	$\mathcal{A}\geqslant 3$	140
010101011	18.66	0.17	0.01	$\mathcal{A}=1$	104	$\mathcal{A}=1$	64
				$\mathcal{A}=2$	12	$\mathcal{A}=2$	128
				$\mathcal{A}\geqslant 3$	0	$\mathcal{A}\geqslant 3$	140
011111000	4.00	0.24	0.01	$\mathcal{A}=1$	212	$\mathcal{A}=1$	64
				$\mathcal{A}=2$	12	$\mathcal{A}=2$	128
				$\mathcal{A}\geqslant 3$	0	$\mathcal{A}\geqslant 3$	140
100001111	3.61	0.24	0.01	$\mathcal{A}=1$	133	$\mathcal{A}=1$	64
				$\mathcal{A}=2$	17	$\mathcal{A}=2$	128
				$\mathcal{A}\geqslant 3$	0	$\mathcal{A}\geqslant 3$	140
100110011	17.46	0.23	0.00	$\mathcal{A}=1$	100	$\mathcal{A}=1$	64
				$\mathcal{A}=2$	16	$\mathcal{A}=2$	128
				$\mathcal{A}\geqslant 3$	0	$\mathcal{A}\geqslant 3$	140
111110000	2.43	0.39	0.00	$\mathcal{A}=1$	513	$\mathcal{A}=1$	64
				$\mathcal{A}=2$	124	$\mathcal{A}=2$	128
				$\mathcal{A}\geqslant 3$	0	$\mathcal{A}\geqslant 3$	140

Table 8.20: Results of using the analysis framework to count the number of interior and potential boundary misses incurred by loop nest $\mathbb{L}_{\text{vec}}$ for $m = 16$, $n = 32$ and various $(4, 5)$-interleavings σ in ($\mathcal{A}$, 32, $2048 \cdot \mathcal{A}$; 64) data caches.

σ	solution time (sec)			# of interior misses		# of pot. boundary misses	
	simplify formula	build DFA	count misses				
000001111	2.31	0.20	0.00	$\mathcal{A} = 1$	130	$\mathcal{A} = 1$	64
				$\mathcal{A} = 2$	12	$\mathcal{A} = 2$	128
				$\mathcal{A} \geqslant 3$	0	$\mathcal{A} \geqslant 3$	140
000100111	11.53	0.17	0.00	$\mathcal{A} = 1$	108	$\mathcal{A} = 1$	64
				$\mathcal{A} = 2$	12	$\mathcal{A} = 2$	128
				$\mathcal{A} \geqslant 3$	0	$\mathcal{A} \geqslant 3$	140
000110011	16.93	0.17	0.00	$\mathcal{A} = 1$	97	$\mathcal{A} = 1$	64
				$\mathcal{A} = 2$	12	$\mathcal{A} = 2$	128
				$\mathcal{A} \geqslant 3$	0	$\mathcal{A} \geqslant 3$	140
010000111	4.83	0.18	0.00	$\mathcal{A} = 1$	108	$\mathcal{A} = 1$	64
				$\mathcal{A} = 2$	12	$\mathcal{A} = 2$	128
				$\mathcal{A} \geqslant 3$	0	$\mathcal{A} \geqslant 3$	140
011110000	3.92	0.24	0.01	$\mathcal{A} = 1$	187	$\mathcal{A} = 1$	64
				$\mathcal{A} = 2$	12	$\mathcal{A} = 2$	128
				$\mathcal{A} \geqslant 3$	0	$\mathcal{A} \geqslant 3$	140
100000111	3.61	0.16	0.00	$\mathcal{A} = 1$	105	$\mathcal{A} = 1$	64
				$\mathcal{A} = 2$	15	$\mathcal{A} = 2$	128
				$\mathcal{A} \geqslant 3$	0	$\mathcal{A} \geqslant 3$	140
100010011	16.60	0.15	0.01	$\mathcal{A} = 1$	94	$\mathcal{A} = 1$	64
				$\mathcal{A} = 2$	15	$\mathcal{A} = 2$	128
				$\mathcal{A} \geqslant 3$	0	$\mathcal{A} \geqslant 3$	140
111100000	2.40	0.36	0.01	$\mathcal{A} = 1$	477	$\mathcal{A} = 1$	64
				$\mathcal{A} = 2$	50	$\mathcal{A} = 2$	128
				$\mathcal{A} \geqslant 3$	0	$\mathcal{A} \geqslant 3$	140

Table 8.21: Results of using the analysis framework to count the number of interior and potential boundary misses incurred by loop nest $\mathbb{L}_{\text{vec}}$ for $m = 32$, $n = 16$ and various $(5, 4)$-interleavings σ in ($\mathcal{A}$, 32, $2048 \cdot \mathcal{A}$; 64) data caches.

σ	solution time (sec)			# of interior misses		# of pot. boundary misses	
	simplify formula	build DFA	count misses				
0000011111	2.44	0.44	0.01	$\mathcal{A} = 1$	428	$\mathcal{A} = 1$	64
				$\mathcal{A} = 2$	144	$\mathcal{A} = 2$	128
				$\mathcal{A} = 3$	80	$\mathcal{A} = 3$	192
				$\mathcal{A} = 4$	16	$\mathcal{A} = 4$	256
				$\mathcal{A} \geqslant 5$	0	$\mathcal{A} \geqslant 5$	272
0001110011	18.15	0.27	0.01	$\mathcal{A} = 1$	266	$\mathcal{A} = 1$	64
				$\mathcal{A} = 2$	144	$\mathcal{A} = 2$	128
				$\mathcal{A} = 3$	80	$\mathcal{A} = 3$	192
				$\mathcal{A} = 4$	16	$\mathcal{A} = 4$	256
				$\mathcal{A} \geqslant 5$	0	$\mathcal{A} \geqslant 5$	272
0011000111	10.60	0.30	0.01	$\mathcal{A} = 1$	290	$\mathcal{A} = 1$	64
				$\mathcal{A} = 2$	144	$\mathcal{A} = 2$	128
				$\mathcal{A} = 3$	80	$\mathcal{A} = 3$	192
				$\mathcal{A} = 4$	16	$\mathcal{A} = 4$	256
				$\mathcal{A} \geqslant 5$	0	$\mathcal{A} \geqslant 5$	272
0100001111	5.67	1.20	0.01	$\mathcal{A} = 1$	326	$\mathcal{A} = 1$	64
				$\mathcal{A} = 2$	154	$\mathcal{A} = 2$	128
				$\mathcal{A} = 3$	80	$\mathcal{A} = 3$	192
				$\mathcal{A} = 4$	16	$\mathcal{A} = 4$	256
				$\mathcal{A} \geqslant 5$	0	$\mathcal{A} \geqslant 5$	272
0101000111	11.38	0.90	0.01	$\mathcal{A} = 1$	280	$\mathcal{A} = 1$	64
				$\mathcal{A} = 2$	154	$\mathcal{A} = 2$	128
				$\mathcal{A} = 3$	80	$\mathcal{A} = 3$	192
				$\mathcal{A} = 4$	16	$\mathcal{A} = 4$	256
				$\mathcal{A} \geqslant 5$	0	$\mathcal{A} \geqslant 5$	272
1100000111	5.43	0.37	0.00	$\mathcal{A} = 1$	269	$\mathcal{A} = 1$	64
				$\mathcal{A} = 2$	155	$\mathcal{A} = 2$	128
				$\mathcal{A} = 3$	87	$\mathcal{A} = 3$	192
				$\mathcal{A} = 4$	19	$\mathcal{A} = 4$	256
				$\mathcal{A} \geqslant 5$	0	$\mathcal{A} \geqslant 5$	272
1110000011	7.62	0.30	0.01	$\mathcal{A} = 1$	246	$\mathcal{A} = 1$	64
				$\mathcal{A} = 2$	155	$\mathcal{A} = 2$	128
				$\mathcal{A} = 3$	87	$\mathcal{A} = 3$	192
				$\mathcal{A} = 4$	19	$\mathcal{A} = 4$	256
				$\mathcal{A} \geqslant 5$	0	$\mathcal{A} \geqslant 5$	272
1111100000	2.61	0.82	0.03	$\mathcal{A} = 1$	1045	$\mathcal{A} = 1$	64
				$\mathcal{A} = 2$	978	$\mathcal{A} = 2$	128
				$\mathcal{A} = 3$	910	$\mathcal{A} = 3$	192
				$\mathcal{A} = 4$	266	$\mathcal{A} = 4$	256
				$\mathcal{A} \geqslant 5$	0	$\mathcal{A} \geqslant 5$	272

Table 8.22: Results of using the analysis framework to count the number of interior and potential boundary misses incurred by loop nest $\mathbb{L}_{\text{vec}}$ for $m = n = 32$ and various $(5, 5)$-interleavings σ in $(\mathcal{A}, 32, 2048 \cdot \mathcal{A}; 64)$ data caches.

6. The time required to count misses remains basically constant no matter the degree of interleaving or miss counts.

8.4 Loop Transformation

Loop transformations [134] alter the ordering of operations in a loop nest in a way that preserves the semantics of the original loop nest while modifying the way in which it accesses data. The new data access sequence may have better locality of reference than the original, and thus incur fewer cache misses. Certainly, the goal of applying a loop transformation is a reduction in cache misses, but this is not guaranteed. The modified loop nest may incur more cache misses than the original, and it is difficult to predict how a loop transformation will affect cache behavior. I demonstrate how the analysis framework can be used to examine the effects of two loop transformations—loop tiling and loop permutation.

8.4.1 Loop Tiling

Loop tiling (or loop blocking) is a loop transformation that can improve the performance of a loop nest by decreasing the number of cache misses. Loop tiling causes the iterations of one or more loops to be executed in "tiles" or "blocks". Consider the original matrix-vector multiplication loop nest $\mathbb{L}_{\text{orig}}$ and a tiled matrix-vector multiplication loop nest $\mathbb{L}_{\text{tile}}$ with tile size T, both given below. Note that the upper bound on the jj-loop, which may look strange, is a consequence of loop normalization. Selecting the optimal size T for such tiles is a difficult problem [29, 40, 84].

```
L_orig: do i = 0, $m-1$
           r = Y[i]
           do j = 0, $n-1$
              r += A[i,j]*X[j]
           enddo
           Y[i] = r
        enddo
```

```
L_tile: do jj = 0, $(n-1)/T$
           do i = 0, $m-1$
           r = Y[i]
              do j = T*jj, $\min(T*jj+T-1, n-1)$
                 r += A[i,j]*X[j]
              enddo
              Y[i] = r
           enddo
        enddo
```

Suppose that the arrays $Y^{(0)} = \texttt{A}$, $Y^{(1)} = \texttt{X}$, and $Y^{(2)} = \texttt{Y}$ are double-precision (*i.e.*, $\beta_0 = \beta_1 = \beta_2 = 8$ bytes) with starting addresses $\mu_0 = 0$, $\mu_1 = \mu_0 + 8m \cdot n$, and $\mu_2 = \mu_1 + 8n$. Array $Y^{(0)}$ is linearized in column-major order.

Table 8.23 gives the results of using the framework to count the number of interior misses incurred by $\mathbb{L}_{\text{tiled}}$ for $m = n = 60$ in ($\mathcal{A}$, 64, 8192$\cdot\mathcal{A}$; 128) data caches, varying the value of tile size T. Potential boundary misses are not counted because the count is unaffected by tile size.

T	solution time (sec)			# of interior misses	
	simplify formula	build DFA	count misses		
5	2.10	33.51	0.01	$\mathcal{A} = 1$	479
				$\mathcal{A} = 2$	209
				$\mathcal{A} = 3$	81
				$\mathcal{A} \geqslant 4$	0
10	2.41	38.91	0.03	$\mathcal{A} = 1$	509
				$\mathcal{A} = 2$	209
				$\mathcal{A} = 3$	81
				$\mathcal{A} \geqslant 4$	0
20	2.00	71.53	0.03	$\mathcal{A} = 1$	947
				$\mathcal{A} = 2$	211
				$\mathcal{A} = 3$	81
				$\mathcal{A} \geqslant 4$	0
30	1.67	59.43	0.05	$\mathcal{A} = 1$	1,751
				$\mathcal{A} = 2$	244
				$\mathcal{A} = 3$	82
				$\mathcal{A} \geqslant 4$	0
60	0.68	50.33	0.06	$\mathcal{A} = 1$	2,570
				$\mathcal{A} = 2$	380
				$\mathcal{A} = 3$	100
				$\mathcal{A} \geqslant 4$	0

Table 8.23: Results of using the analysis framework to count the number of interior misses incurred by loop nest $\mathbb{L}_{\text{tiled}}$ for $m = n = 60$ in ($\mathcal{A}$, 64, $8192 \cdot \mathcal{A}$; 128) data caches with varying values of tile size T.

For comparison, the following gives the results of using the framework to count the interior misses incurred by $\mathbb{L}_{\text{orig}}$ for $m = n = 60$ in ($\mathcal{A}$, 64, $8192 \cdot \mathcal{A}$; 128) data caches.

solution time (sec)			# of interior misses	
simplify formula	build DFA	count misses		
0.66	49.42	0.04	$\mathcal{A} = 1$	2,570
			$\mathcal{A} = 2$	380
			$\mathcal{A} = 3$	100
			$\mathcal{A} \geqslant 4$	0

Observations. The following are some observations on the results for loop tiling presented in this section.

1. For tile size $T = 60$, the interior miss count of loop nest $\mathbb{L}_{\text{tile}}$ is identical to that of loop nest $\mathbb{L}_{\text{orig}}$ (for $m = n = 60$), as expected.

$\mathbb{L}_{\mathrm{ijk}}$:
```
do i = 0, $t-1$
   do j = 0, $u-1$
      c = Z[i,j]
      do k = 0, $v-1$
         c += X[i,k]*Y[k,j]
      enddo
      Z[i,j] = c
   enddo
enddo
```

$\mathbb{L}_{\mathrm{ikj}}$:
```
do i = 0, $t-1$
   do k = 0, $v-1$
      c = X[i,k]
      do j = 0, $u-1$
         Z[i,j] += c*Y[k,j]
      enddo
   enddo
enddo
```

$\mathbb{L}_{\mathrm{jik}}$:
```
do j = 0, $u-1$
   do i = 0, $t-1$
      c = Z[i,j]
      do k = 0, $v-1$
         c += X[i,k]*Y[k,j]
      enddo
      Z[i,j] = c
   enddo
enddo
```

$\mathbb{L}_{\mathrm{jki}}$:
```
do j = 0, $u-1$
   do k = 0, $v-1$
      c = Y[k,j]
      do i = 0, $t-1$
         Z[i,j] += X[i,k]*c
      enddo
   enddo
enddo
```

$\mathbb{L}_{\mathrm{kij}}$:
```
do k = 0, $v-1$
   do i = 0, $t-1$
      c = X[i,k]
      do j = 0, $u-1$
         Z[i,j] += c*Y[k,j]
      enddo
   enddo
enddo
```

$\mathbb{L}_{\mathrm{kji}}$:
```
do k = 0, $v-1$
   do j = 0, $u-1$
      c = Y[k,j]
      do i = 0, $t-1$
         Z[i,j] += X[i,k]*c
      enddo
   enddo
enddo
```

Figure 8.4: The six permutations of the matrix multiplication loop nest.

2. For all tile sizes $T \in \{5, 10, 20, 30\}$, loop nest $\mathbb{L}_{\mathrm{tile}}$ incurs fewer interior misses than loop nest $\mathbb{L}_{\mathrm{orig}}$ for $1 \leqslant \mathcal{A} \leqslant 3$. For $\mathcal{A} = 1$, tile size $T = 5$ yields the fewest interior misses. For $\mathcal{A} = 2$, tile sizes $T = 5$ or 10 yield the fewest interior misses. For $\mathcal{A} = 3$, tile sizes $T = 5, 10,$ or 20 yield the fewest interior misses.

3. The time required to count interior misses increases with the miss count. Formula-simplification times and DFA-construction times are not necessarily proportional to miss counts or problem size.

8.4.2 Loop Permutation

Loop permutation is a loop transformation that can be useful in improving the performance of a loop nest by decreasing the number of cache misses it incurs. Loop permutation changes the nesting order of some or all loops. Consider the matrix multiplication loop nest $\mathbb{L}_{\mathrm{ijk}}$ in Figure 8.4, which is identical to loop nest $\mathbb{L}_{\mathrm{mm}}$ in Figure 2.5. Array element `Z[i,j]` is reused during every iteration of the innermost loop of $\mathbb{L}_{\mathrm{ijk}}$ (the `k`-loop). Loop nest $\mathbb{L}_{\mathrm{ijk}}$ stores array element `Z[i,j]` in the scalar variable `c` so that it is register-resident for all iterations of the `k`-loop. Interchanging the `j`-loop and the `k`-loop gives loop nest $\mathbb{L}_{\mathrm{ikj}}$ in Figure 8.4. Now, array element `X[i,k]` is reused during every iteration of the innermost loop of $\mathbb{L}_{\mathrm{ikj}}$ and is stored in `c` so that it is register-resident. Other permutations of the matrix multiplication loop nest are written similarly. Figure 8.4 shows all six permutations of the matrix multiplication loop nest: $\mathbb{L}_{\mathrm{ijk}}$, $\mathbb{L}_{\mathrm{ikj}}$, $\mathbb{L}_{\mathrm{jik}}$, $\mathbb{L}_{\mathrm{jki}}$, $\mathbb{L}_{\mathrm{kij}}$, and $\mathbb{L}_{\mathrm{kji}}$. Each permutation has different access patterns and potentially different memory performance.

Suppose that the arrays $Y^{(0)} = \mathtt{X}$, $Y^{(1)} = \mathtt{Y}$, and $Y^{(2)} = \mathtt{Z}$ are double-precision (*i.e.*, $\beta_0 = \beta_1 = \beta_2 = 8$ bytes) and linearized in column-major order with starting addresses $\mu_0 = 0$, $\mu_1 = \mu_0 + 8t \cdot v$, and $\mu_2 = \mu_1 + 8v \cdot u$. Table 8.24 gives the results of using the framework to count the number of interior and potential boundary misses incurred by all six loop permutations for several problem sizes in $(\mathcal{A}, 64, 16384 \cdot \mathcal{A}; 256)$ data caches.

Observations. The following are some observations on the results for loop permutation presented in this section.

1. For each problem size, all six permutations incur the same number of boundary misses. This is as expected, since all loop nests access the same data in memory.

2. For problem size $t = u = v = 20$, there are no interior misses for any value of $\mathcal{A}$ because all array elements referenced by the loop nests fit in cache and never get replaced. Therefore, all six permutations have the same cache miss count. Despite identical problem size and cache miss count, the six permutations require a variety of formula-simplification times because the data access patterns described for each permutation are different.

3. For problem size $t = u = v = 30$ and $\mathcal{A} = 1$, loop nests $\mathbb{L}_{\mathrm{ijk}}$ and $\mathbb{L}_{\mathrm{ikj}}$ have the fewest cache misses, and loop nest $\mathbb{L}_{\mathrm{jki}}$ has the most cache misses. For $\mathcal{A} >= 2$, all six permutations have the same cache miss count. Solution times are not proportional to cache miss count.

4. For problem size $t = u = v = 40$ and $\mathcal{A} = 1$, loop nest $\mathbb{L}_{\mathrm{jki}}$ has the fewest cache misses, and loop nest $\mathbb{L}_{\mathrm{ikj}}$ has the most cache misses. For $\mathcal{A} = 2$, loop nests $\mathbb{L}_{\mathrm{ijk}}$, $\mathbb{L}_{\mathrm{ikj}}$, $\mathbb{L}_{\mathrm{jik}}$, and $\mathbb{L}_{\mathrm{jki}}$ have the smallest cache misses, and loop nest $\mathbb{L}_{\mathrm{kji}}$ has the most cache misses. Solution times are not proportional to cache miss count.

5. No single permutation of the matrix multiplication loop nest will always yield the smallest number of cache misses.

8.5 Aggregate Array Computations

An aggregate array computation is a loop or collection of loops that computes accumulated quantities over elements of arrays. Such computations are commonly found in programs, and

$t = u = v$	$\mathbb{L}$	solution time (sec)			# of interior misses		# of pot. boundary misses	
		simplify formula	build DFA	count misses				
20	$\mathbb{L}_{\text{ijk}}$	1.34	0.05	0.03	$\mathcal{A} \geqslant 1$	0	$\mathcal{A} \geqslant 1$	150
	$\mathbb{L}_{\text{ikj}}$	2.56	0.06	0.04	$\mathcal{A} \geqslant 1$	0	$\mathcal{A} \geqslant 1$	150
	$\mathbb{L}_{\text{jik}}$	0.74	0.01	0.04	$\mathcal{A} \geqslant 1$	0	$\mathcal{A} \geqslant 1$	150
	$\mathbb{L}_{\text{jki}}$	0.71	0.02	0.04	$\mathcal{A} \geqslant 1$	0	$\mathcal{A} \geqslant 1$	150
	$\mathbb{L}_{\text{kij}}$	3.48	0.04	0.03	$\mathcal{A} \geqslant 1$	0	$\mathcal{A} \geqslant 1$	150
	$\mathbb{L}_{\text{kji}}$	0.64	0.02	0.04	$\mathcal{A} \geqslant 1$	0	$\mathcal{A} \geqslant 1$	150
30	$\mathbb{L}_{\text{ijk}}$	2.07	0.69	0.04	$\mathcal{A} = 1$	114	$\mathcal{A} = 1$	256
					$\mathcal{A} \geqslant 2$	0	$\mathcal{A} \geqslant 2$	338
	$\mathbb{L}_{\text{ikj}}$	5.47	0.69	0.05	$\mathcal{A} = 1$	114	$\mathcal{A} = 1$	256
					$\mathcal{A} \geqslant 2$	0	$\mathcal{A} \geqslant 2$	338
	$\mathbb{L}_{\text{jik}}$	1.18	0.23	0.03	$\mathcal{A} = 1$	161	$\mathcal{A} = 1$	256
					$\mathcal{A} \geqslant 2$	0	$\mathcal{A} \geqslant 2$	338
	$\mathbb{L}_{\text{jki}}$	0.75	0.22	0.04	$\mathcal{A} = 1$	349	$\mathcal{A} = 1$	256
					$\mathcal{A} \geqslant 2$	0	$\mathcal{A} \geqslant 2$	338
	$\mathbb{L}_{\text{kij}}$	5.90	0.99	0.04	$\mathcal{A} = 1$	162	$\mathcal{A} = 1$	256
					$\mathcal{A} \geqslant 2$	0	$\mathcal{A} \geqslant 2$	338
	$\mathbb{L}_{\text{kji}}$	1.06	0.30	0.05	$\mathcal{A} = 1$	348	$\mathcal{A} = 1$	256
					$\mathcal{A} \geqslant 2$	0	$\mathcal{A} \geqslant 2$	338
40	$\mathbb{L}_{\text{ijk}}$	21.61	32.89	0.12	$\mathcal{A} = 1$	5,262	$\mathcal{A} = 1$	256
					$\mathcal{A} = 2$	88	$\mathcal{A} = 2$	512
					$\mathcal{A} \geqslant 3$	0	$\mathcal{A} \geqslant 3$	600
	$\mathbb{L}_{\text{ikj}}$	27.40	39.96	0.18	$\mathcal{A} = 1$	20,936	$\mathcal{A} = 1$	256
					$\mathcal{A} = 2$	88	$\mathcal{A} = 2$	512
					$\mathcal{A} \geqslant 3$	0	$\mathcal{A} \geqslant 3$	600
	$\mathbb{L}_{\text{jik}}$	0.95	12.97	0.07	$\mathcal{A} = 1$	2,870	$\mathcal{A} = 1$	256
					$\mathcal{A} = 2$	88	$\mathcal{A} = 2$	512
					$\mathcal{A} \geqslant 3$	0	$\mathcal{A} \geqslant 3$	600
	$\mathbb{L}_{\text{jki}}$	0.55	8.57	0.07	$\mathcal{A} = 1$	952	$\mathcal{A} = 1$	256
					$\mathcal{A} = 2$	88	$\mathcal{A} = 2$	512
					$\mathcal{A} \geqslant 3$	0	$\mathcal{A} \geqslant 3$	600
	$\mathbb{L}_{\text{kij}}$	24.81	37.63	0.19	$\mathcal{A} = 1$	19,360	$\mathcal{A} = 1$	256
					$\mathcal{A} = 2$	107	$\mathcal{A} = 2$	512
					$\mathcal{A} \geqslant 3$	0	$\mathcal{A} \geqslant 3$	600
	$\mathbb{L}_{\text{kji}}$	0.79	7.49	0.08	$\mathcal{A} = 1$	2,904	$\mathcal{A} = 1$	256
					$\mathcal{A} = 2$	153	$\mathcal{A} = 2$	512
					$\mathcal{A} \geqslant 3$	0	$\mathcal{A} \geqslant 3$	600

Table 8.24: Results of using the analysis framework to count the number of interior and potential boundary misses incurred by all six loop permutations of the matrix multiplication loop nest for several problem sizes in ($\mathcal{A}$, 64, $16384 \cdot \mathcal{A}$; 256) data caches.

```
Lps-orig:  do i = 0, n-1
              S[i] = 0                   Lps-opt:  S[0] = A[0]
              do j = 0, i                          do i = 0, n-1
                 S[i] = S[i] + A[j]                   S[i] = S[i-1] + A[i]
              enddo                                enddo
           enddo
```

Figure 8.5: Original partial summation loop nest $\mathbb{L}_{\text{ps-orig}}$ and optimized version $\mathbb{L}_{\text{ps-opt}}$.

the array elements required often overlap, creating redundancy in the overall computation. Liu *et al.* [88, 89] present a method for eliminating such redundancies to improve the performance of the aggregate array computation. The method is based on incrementalization, which updates the values of aggregate array computations from iteration to iteration, instead of starting from scratch in each iteration. The optimization method is designed to reduce the number of operations in the aggregate array computation, and it is of interest how the optimization method effects the cache behavior of an aggregate array computation. I demonstrate how the analysis framework can be used to determine the number of cache misses incurred by an aggregate array computation, before and after applying Liu *et al.*'s optimization method. The following sections consider two aggregate array computations—partial summation and sequence local average.

8.5.1 Partial Summation

Given an array of n elements, the objective is to sum elements 0 to i for each i from 0 to $n-1$. Loop nest $\mathbb{L}_{\text{ps-orig}}$ in Figure 8.5 performs this computation in a straightforward manner, requiring $O(n^2)$ time. Loop nest $\mathbb{L}_{\text{ps-opt}}$ in Figure 8.5 performs the same computation, but in only $O(n)$ time. Notice that $\mathbb{L}_{\text{ps-opt}}$ is not actually a loop nest because statement `S[0] = A[0]` is not contained in a loop. In order to analyze $\mathbb{L}_{\text{ps-opt}}$, I treat its contents as if they are contained in a loop of one iteration.

Suppose that the arrays $Y^{(0)} = \texttt{A}$ and $Y^{(1)} = \texttt{S}$ are double-precision (*i.e.*, $\beta_0 = \beta_1 = 8$ bytes) with starting addresses $\mu_0 = 0$ and $\mu_1 = \mu_0 + 8n$. Table 8.25 gives the results of using the framework to count the number of interior and potential boundary misses incurred by the original loop nest $\mathbb{L}_{\text{ps-orig}}$ for several problem sizes in ($\mathcal{A}$, 64, $32768 \cdot \mathcal{A}$; 512) data caches. Table 8.26 gives the results for the optimized loop nest $\mathbb{L}_{\text{ps-opt}}$.

Observations. The following are some observations on the results for the original and optimized versions of partial summation presented in this section.

1. Loop nest $\mathbb{L}_{\text{ps-opt}}$ makes significantly fewer accesses to arrays `S` and `A` than loop nest $\mathbb{L}_{\text{ps-orig}}$. As a result, $\mathbb{L}_{\text{ps-opt}}$ incurs fewer interior misses than $\mathbb{L}_{\text{ps-orig}}$ in the direct-mapped cache.

n	solution time (sec)			# of interior misses		# of pot. boundary misses	
	simplify formula	build DFA	count misses				
2,000	0.20	0.06	0.17	$\mathcal{A} \geqslant 1$	0	$\mathcal{A} \geqslant 1$	500
3,000	0.34	0.31	0.52	$\mathcal{A} = 1$	30,939	$\mathcal{A} = 1$	512
				$\mathcal{A} \geqslant 2$	0	$\mathcal{A} \geqslant 2$	750
4,000	0.37	0.22	0.70	$\mathcal{A} = 1$	63,439	$\mathcal{A} = 1$	512
				$\mathcal{A} \geqslant 2$	0	$\mathcal{A} \geqslant 2$	1,000
5,000	0.50	2.51	1.61	$\mathcal{A} = 1$	132,721	$\mathcal{A} = 1$	512
				$\mathcal{A} = 2$	226	$\mathcal{A} = 2$	1,024
				$\mathcal{A} \geqslant 3$	0	$\mathcal{A} \geqslant 3$	1,250
6,000	0.35	2.39	3.64	$\mathcal{A} = 1$	516,971	$\mathcal{A} = 1$	512
				$\mathcal{A} = 2$	476	$\mathcal{A} = 2$	1,024
				$\mathcal{A} \geqslant 3$	0	$\mathcal{A} \geqslant 3$	1,500

Table 8.25: Results of using the analysis framework to count the number of interior and potential boundary misses incurred by loop nest $\mathbb{L}_{\text{ps-orig}}$ for several problem sizes in ($\mathcal{A}$, 64, $32768 \cdot \mathcal{A}$; 512) data caches.

n	solution time (sec)			# of interior misses		# of pot. boundary misses	
	simplify formula	build DFA	count misses				
2,000	0.07	0.00	0.18	$\mathcal{A} \geqslant 1$	0	$\mathcal{A} \geqslant 1$	500
3,000	0.08	0.01	0.15	$\mathcal{A} = 1$	238	$\mathcal{A} = 1$	512
				$\mathcal{A} \geqslant 2$	0	$\mathcal{A} \geqslant 2$	750
4,000	0.10	0.02	0.23	$\mathcal{A} = 1$	488	$\mathcal{A} = 1$	512
				$\mathcal{A} \geqslant 2$	0	$\mathcal{A} \geqslant 2$	1,000
5,000	0.13	0.59	0.19	$\mathcal{A} = 1$	738	$\mathcal{A} = 1$	512
				$\mathcal{A} = 2$	226	$\mathcal{A} = 2$	1,024
				$\mathcal{A} \geqslant 3$	0	$\mathcal{A} \geqslant 3$	1,250
6,000	0.18	0.54	0.18	$\mathcal{A} = 1$	988	$\mathcal{A} = 1$	512
				$\mathcal{A} = 2$	476	$\mathcal{A} = 2$	1,024
				$\mathcal{A} \geqslant 3$	0	$\mathcal{A} \geqslant 3$	1,500

Table 8.26: Results of using the analysis framework to count the number of interior and potential boundary misses incurred by loop nest $\mathbb{L}_{\text{ps-opt}}$ for several problem sizes in ($\mathcal{A}$, 64, $32768 \cdot \mathcal{A}$; 512) data caches.

```
                                          $\mathbb{L}_{\texttt{sla-opt}}$:  S[0] = 0
$\mathbb{L}_{\texttt{sla-orig}}$:  do i = 0, $n-k$                    do j = 0, $k-1$
              s = 0                                   S[0] = S[0] + A[j]
              do j = i, i + $k-1$                    enddo
                 s = s + A[j]                        AVE[0] = S[0]/$k$
              enddo                                  do i = 1, $n-k$
              AVE[i] = s/$k$                            S[i] = S[i-1] - A[i-1] + A[i-1+$k$]
           enddo                                        AVE[i] = S[i]/$k$
                                                     enddo
```

Figure 8.6: Original sequence local average loop nest $\mathbb{L}_{\text{sla-orig}}$ and optimized version $\mathbb{L}_{\text{sla-opt}}$.

2. For $\mathcal{A} \geqslant 2$, loop nests $\mathbb{L}_{\text{ps-orig}}$ and $\mathbb{L}_{\text{ps-opt}}$ incur the same number of interior misses.

3. Loop nests $\mathbb{L}_{\text{ps-orig}}$ and $\mathbb{L}_{\text{ps-opt}}$ incur the same number of potential boundary misses, since the cache footprint of the memory blocks accessed by each loop nest is the same.

8.5.2 Sequence Local Average

Given an array of n elements, the objective is to sum elements i to $i+k-1$ for each i from 0 to $n-k$. Loop nest $\mathbb{L}_{\text{sla-orig}}$ in Figure 8.6 performs this computation in a straightforward manner, requiring $O(nk)$ time. Loop nest $\mathbb{L}_{\text{sla-opt}}$ in Figure 8.6 performs the same computation, but in only $O(n)$ time. In order to analyze $\mathbb{L}_{\text{sla-opt}}$, I treat its contents as if they are contained in a loop of one iteration.

Suppose that the arrays $Y^{(0)} = \texttt{A}$, $Y^{(1)} = \texttt{AVE}$, and $Y^{(2)} = \texttt{S}$ are double-precision (*i.e.*, $\beta_0 = \beta_1 = \beta_2 = 8$ bytes) with starting addresses $\mu_0 = 0$, $\mu_1 = \mu_0 + 8n$, and $\mu_2 = \mu_1 + 8(n-k)$. Table 8.27 gives the results of using the framework to count the number of interior and potential boundary misses incurred by the original loop nest $\mathbb{L}_{\text{sla-orig}}$ for several problem sizes in ($\mathcal{A}$, 64, $32768 \cdot \mathcal{A}$; 512) data caches. Table 8.28 gives the results for the optimized loop nest $\mathbb{L}_{\text{sla-opt}}$.

Observations. The following are some observations on the results for the original and optimized versions of sequence local average computation presented in this section.

1. Loop nest $\mathbb{L}_{\text{sla-opt}}$ has fewer operations than loop nest $\mathbb{L}_{\text{sla-orig}}$, but it requires an additional array S. Therefore, the cache footprint of the memory blocks accessed by $\mathbb{L}_{\text{sla-opt}}$ is larger than those accessed by $\mathbb{L}_{\text{sla-orig}}$ for $n = 5,000$ and $\mathcal{A} \geqslant 3$, $n = 15,000$ and $\mathcal{A} \geqslant 8$, and $n = 25,000$ and $\mathcal{A} \geqslant 12$. As a result, loop nest $\mathbb{L}_{\text{sla-opt}}$ incurs more potential boundary misses than $\mathbb{L}_{\text{sla-orig}}$ in these cases.

2. Despite the additional accesses to array S in loop nest $\mathbb{L}_{\text{sla-opt}}$, the reduction in the overall number of array accesses results in fewer interior misses for $n = 5,000$ and $\mathcal{A} = 1$ and for $n = 25,000$ and $\mathcal{A} = 1$. In all other cases, accesses to the extra array S causes more interior misses in $\mathbb{L}_{\text{sla-opt}}$.

n	solution time (sec)			# of interior misses		# of pot. boundary misses	
	simplify formula	build DFA	count misses				
5,000	0.11	16.98	0.24	$\mathcal{A} = 1$	8,114	$\mathcal{A} = 1$	512
				$\mathcal{A} = 2$	102	$\mathcal{A} = 2$	1,024
				$\mathcal{A} \geqslant 3$	0	$\mathcal{A} \geqslant 3$	1,126
15,000	0.10	4.91	0.36	$\mathcal{A} = 1$	3,114	$\mathcal{A} = 1$	512
				$\mathcal{A} = 2$	2,602	$\mathcal{A} = 2$	1,024
				$\mathcal{A} = 3$	2,090	$\mathcal{A} = 3$	1,536
				$\mathcal{A} = 4$	1,578	$\mathcal{A} = 4$	2,048
				$\mathcal{A} = 5$	1,066	$\mathcal{A} = 5$	2,560
				$\mathcal{A} = 6$	554	$\mathcal{A} = 6$	3,072
				$\mathcal{A} = 7$	42	$\mathcal{A} = 7$	3,584
				$\mathcal{A} \geqslant 8$	0	$\mathcal{A} \geqslant 8$	3,626
25,000	0.12	39.38	1.33	$\mathcal{A} = 1$	50,614	$\mathcal{A} = 1$	512
				$\mathcal{A} = 2$	5,102	$\mathcal{A} = 2$	1,024
				$\mathcal{A} = 3$	4,590	$\mathcal{A} = 3$	1,536
				$\mathcal{A} = 4$	4,078	$\mathcal{A} = 4$	2,048
				$\mathcal{A} = 5$	3,566	$\mathcal{A} = 5$	2,560
				$\mathcal{A} = 6$	3,054	$\mathcal{A} = 6$	3,072
				$\mathcal{A} = 7$	2,542	$\mathcal{A} = 7$	3,584
				$\mathcal{A} = 8$	2,030	$\mathcal{A} = 8$	4,096
				$\mathcal{A} = 9$	1,518	$\mathcal{A} = 9$	4,608
				$\mathcal{A} = 10$	1,006	$\mathcal{A} = 10$	5,120
				$\mathcal{A} = 11$	494	$\mathcal{A} = 11$	5,632
				$\mathcal{A} \geqslant 12$	0	$\mathcal{A} \geqslant 12$	6,126

Table 8.27: Results of using the analysis framework to count the number of interior and potential boundary misses incurred by loop nest $\mathbb{L}_{\text{sla-orig}}$ for several problem sizes in ($\mathcal{A}$, 64, $32768 \cdot \mathcal{A}$; 512) data caches.

8.6 Summary

In this chapter, I have applied and validated the analysis framework presented in this dissertation by using it to accurately determine the cache behavior of a number of program fragments, given a variety of cache configurations, data layouts, and cache miss classifications. Every cache miss count provided here matches that of a cache simulator, establishing the correctness of the analysis framework.

The following points summarize the capabilities of the analysis framework and how they have been demonstrated.

1. The results provided in Tables 8.2, 8.5, 8.7, 8.10, 8.12, and 8.17 demonstrate that the framework counts TLB misses, in addition to data cache misses. Furthermore, the framework analyzes cache behavior given any values of the cache configuration parameters

n	solution time (sec)			# of interior misses		# of pot. boundary misses	
	simplify formula	build DFA	count misses				
5,000	2.26	3.28	0.18	$\mathcal{A} = 1$	1,529	$\mathcal{A} = 1$	512
				$\mathcal{A} = 2$	990	$\mathcal{A} = 2$	1,024
				$\mathcal{A} = 3$	90	$\mathcal{A} = 3$	1,536
				$\mathcal{A} \geqslant 4$	0	$\mathcal{A} \geqslant 4$	1,626
15,000	2.26	18.83	0.58	$\mathcal{A} = 1$	6,589	$\mathcal{A} = 1$	512
				$\mathcal{A} = 2$	4,353	$\mathcal{A} = 2$	1,024
				$\mathcal{A} = 3$	3,841	$\mathcal{A} = 3$	1,536
				$\mathcal{A} = 4$	3,329	$\mathcal{A} = 4$	2,048
				$\mathcal{A} = 5$	2,871	$\mathcal{A} = 5$	2,560
				$\mathcal{A} = 6$	2,305	$\mathcal{A} = 6$	3,072
				$\mathcal{A} = 7$	1,793	$\mathcal{A} = 7$	3,584
				$\mathcal{A} = 8$	1,281	$\mathcal{A} = 8$	4,096
				$\mathcal{A} = 9$	769	$\mathcal{A} = 9$	4,608
				$\mathcal{A} = 10$	257	$\mathcal{A} = 10$	5,120
				$\mathcal{A} \geqslant 11$	0	$\mathcal{A} \geqslant 11$	5,376
25,000	1.18	20.54	1.32	$\mathcal{A} = 1$	11,561	$\mathcal{A} = 1$	512
				$\mathcal{A} = 2$	8,103	$\mathcal{A} = 2$	1,024
				$\mathcal{A} = 3$	7,591	$\mathcal{A} = 3$	1,536
				$\mathcal{A} = 4$	7,079	$\mathcal{A} = 4$	2,048
				$\mathcal{A} = 5$	6,567	$\mathcal{A} = 5$	2,560
				$\mathcal{A} = 6$	6,055	$\mathcal{A} = 6$	3,072
				$\mathcal{A} = 7$	5,543	$\mathcal{A} = 7$	3,584
				$\mathcal{A} = 8$	5,031	$\mathcal{A} = 8$	4,096
				$\mathcal{A} = 9$	4,519	$\mathcal{A} = 9$	4,608
				$\mathcal{A} = 10$	4,007	$\mathcal{A} = 10$	5,120
				$\mathcal{A} = 11$	3,495	$\mathcal{A} = 11$	5,632
				$\mathcal{A} = 12$	2,983	$\mathcal{A} = 12$	6,144
				$\mathcal{A} = 13$	2,471	$\mathcal{A} = 13$	6,656
				$\mathcal{A} = 14$	1,959	$\mathcal{A} = 14$	7,168
				$\mathcal{A} = 15$	1,447	$\mathcal{A} = 15$	7,680
				$\mathcal{A} = 16$	935	$\mathcal{A} = 16$	8,192
				$\mathcal{A} = 17$	422	$\mathcal{A} = 17$	8,704
				$\mathcal{A} \geqslant 18$	0	$\mathcal{A} \geqslant 18$	9,126

Table 8.28: Results of using the analysis framework to count the number of interior and potential boundary misses incurred by loop nest $\mathbb{L}_{\text{sla-opt}}$ for several problem sizes in ($\mathcal{A}$, 64, $32768 \cdot \mathcal{A}$; 512) data caches.

(*i.e.*, $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, and $\mathcal{S}$), real or imagined. The framework is well suited to investigate the behavior of programs on experimental, non-existent caches.

2. All results provided in this chapter demonstrate that the framework identifies cache misses according to either the compulsory-replacement classification or the interior-boundary classification. Moreover, given the generality of the technique for expressing and counting cache misses, it is possible to use the framework to determine cache behavior according to yet other miss classifications.

3. The results provided in Tables 8.19 to 8.22 demonstrate that the framework counts cache misses in the presence of data layouts other than the canonical row- and column-major layouts. The framework's flexibility in handling nonlinear data layouts allows investigation into the behavior and potential benefit of experimental modes of placing data in memory.

4. The results provided in Tables 8.13 to 8.18 demonstrate that the numbers of cache misses incurred by single loop nests are combined to accurately give the number of cache misses incurred by the sequence of such loop nests using the interior-boundary miss classification.

5. All results provided in this chapter demonstrate that the framework gives miss counts for multiple caches in one pass. Given actual values for blocksize and the number of cache sets, say $\mathcal{B} = 64$ and $\mathcal{S} = 256$, one pass through the framework determines the number of misses incurred in all caches with capacity $\mathcal{C} = 16,384 \cdot \mathcal{A}$ for all $\mathcal{A} \geqslant 1$.

For some of the results provided in this chapter, the amount of time required by the framework to analyze the cache behavior of a loop nest (or sequence of loop nests) is too large to make such analysis feasible within an optimizing compiler. In particular, the pursuit of exact cache miss counts and flexible models contributes to the large solution times. Use of the analytical framework is best suited for pre-hardware design exploration in which its ability to model experimental caches is critical.

The analysis framework uses the Omega Library [75] to simplify cache behavior formulas and the automata-construction algorithms of Bartzis and Bultan [9, 10] with the MONA tool [81] to construct DFAs recognizing the solutions of cache behavior formulas. The large solutions times required to simplify formulas and build DFAs are due both to limits in the underlying theory of the tools and the lack of robustness in their implementations. Some instances of cache behavior attack weaknesses of the implementation, while others approach the worst-case complexity associated with Presburger arithmetic. Efforts to improve the efficiency of either the Omega Library or the automata-construction tools will certainly improve the efficiency of the analysis framework. Therefore, to some extent, the limitation of large solution times may eventually be removed. However, if any of the large solution times are due to the worst-case complexity of Presburger arithmetic, such a limitation is fundamental to the analysis framework's method for describing cache behavior.

Chapter 9

Related Work

This chapter discusses work related to analyzing memory behavior and compares the work to the framework described in this dissertation. I organize pieces of related work according to the most salient feature of the approach, although some pieces could fall into more than one category.

9.1 Work Based on Reuse Vectors

The Cache Miss Equations (CMEs) framework of Ghosh *et al.* [59, 60, 61, 62] generates linear Diophantine equations that characterize the cache misses of a single loop nest. Ghosh *et al.* use the CMEs to understand the cache behavior of a loop nest and select program transformations to improve its memory performance. In generating the CMEs, Ghosh *et al.* use reuse vectors obtained from traditional compiler reuse analysis. A reuse vector connects any two iteration points at which a reference accesses the same memory block. Ghosh *et al.* require that loops be perfectly-nested, whereas my framework handles arbitrarily-nested loops. The method of Ghosh *et al.* does not capture reuse between loop nests, and instead assumes a cold-start cache for each loop nest in a program, in which the loop nest begins execution with the cache containing none of the data it accesses. Such an assumption leads to an inaccurate and pessimistic cache miss count because it overestimates the number of compulsory misses. My framework uses cache state to link the loop nests in a program, thereby taking into consideration the actual contents of the cache as a loop nest begins execution and accurately determining the number of cache misses incurred by the loop nest. Solving the Diophantine equations is difficult, and in general, Ghosh *et al.* do not use them to find the number of cache misses explicitly. Instead, they use the CME model indirectly to optimize the number of cache misses on a case-by-case basis, and as a result the generality of their framework is compromised. Due to the translation of Presburger formulas into DFAs, my framework has a way of finding the number of cache misses explicitly, which I have found to be quite useful in practice. Also, reuse vectors do not exist in the presence of nonlinear array layouts; thus, the CME framework is limited to arrays with row- and column-major layouts. Of course, this is not a serious limitation since row- and column-major are the standard ways of mapping arrays to memory. However, the ability of my framework to evaluate the potential benefit of nonlinear array layouts is useful.

Vera *et al.* [121, 122, 123, 124] introduce an extension of traditional reuse vectors to quantify reuse between array references contained in multiple loop nests, thereby enhancing the CME framework and matching the capability of my framework to handle multiple loop nests. In addition, Vera *et al.* offer techniques for handling if-statements, call statements, and arbitrarily-nested loops. For if-statements, the conditional guards must be expressions consisting of LCVs and constants. My framework can also handle such if-statements using the statement guards presented in Section 2.2. Vera *et al.* handle call statements by abstractly inlining information about the memory accesses of a subroutine at the sites of its call statements. As is, my framework does not handle such call statements, but it likely could using a similar approach that inserted the access trace of a subroutine at the call sites in the program's access trace. Even Vera *et al.* avoid solving the generated Diophantine equations by sampling the iteration space of a program fragment and approximating its total number of cache misses based on the miss count of the sample. Other works [54, 55, 56, 65] have also used sampling to make the calculation of cache behavior more tractable.

The sampling method for counting cache misses and our Presburger-DFA method for counting cache misses have complementary strengths. The sampling method provides a good estimation of the cache miss count when the number of cache misses is large and the misses are distributed uniformly throughout the iteration space of a program fragment. Our Presburger-DFA method gives an exact cache miss count even when the misses are sparsely distributed in the iteration space and works best when the number of cache misses is small. In the task of optimizing code, where the objective is to move from a high-miss point to a low-miss point, both methods are of use.

9.2 Work Based on Stack Distances

Computing stack or reuse distance [67], the amount of unique data requested between two accesses to the same memory location, is a technique used by both Caşcaval [30, 31] and Beyls and D'Hollander [14, 15, 16, 17, 18].

The work of Beyls and D'Hollander [14, 15, 16, 17, 18] generates cache hints that both estimate the latency of accessing data items and influence the replacement of data items in cache, with the purpose of diminishing the effect of misses in a multilevel cache hierarchy. Beyls and D'Hollander generate such cache hints based on a reuse distance locality metric. Comparison of the Beyls and D'Hollander framework with mine is useful, despite our differing goals. Reuse distance, which represents the amount of unique data addressed between two accesses to the same memory location, is very similar to the witness notion of my analysis framework. However, there is one key difference—the technique of Beyls and D'Hollander considers a trace for the entire program in the computation of reuse distance, while my framework takes a *set-centric* approach (see Chapter 4). As a consequence, Beyls and D'Hollander use their metric to indicate the behavior of fully-associative caches in which memory blocks contain only one data element, whereas my framework handles caches of any associativity and blocksize. Another point of comparison is the method of determining a locality metric for each data access, which highlights an important tradeoff. The use of analytical methods restricts my framework to program structures that are loop nests. Because their framework uses program profiling, Beyls and D'Hollander experience no such restriction, but this flexibility comes at the cost of speed. Their method is akin to simulation (whose potential drawbacks Chapter 1 describes), and its additional running time is not worth the slight precision gained over my method in the case of structured programs. Beyls investigates using the polyhedral model to calculate reuse distances analytically [14, 125], and finds the success of an analytical approach closely related to the limited equation-solving capabilities of current polyhedral tools.

Caşcaval [30, 31] uses a histogram of stack distances to estimate the number of cache misses incurred by a loop nest. At compile time, Caşcaval computes the stack distance histogram from the data dependence distance vectors in a loop nest, and the resulting histogram is a locality metric for loop nests that is independent of cache. Like for Beyls and D'Hollander, Caşcaval's technique for computing the locality metric is not *set-centric*, distinguishing it from my technique. Therefore, his method yields a valid prediction only for fully-associative caches in which memory blocks contain one data element, and it must deduce the number of misses in set-associative caches using a probabilistic argument. By not including inter-nest data dependencies in his analysis, Caşcaval considers each loop nest in isolation of the rest of the program. As in the CME model, this leads to an inaccurate and overestimated cache miss count. The cache state concept of my framework prevents such an inaccuracy.

9.3 Work Based on Linear Constraints

Many techniques for modeling program behavior use sets of linear constraints to model the behavior of interest and then count or enumerate the solutions satisfying such constraints. In particular, Presburger arithmetic has been used to model various aspects of programming languages such as the memory locations touched by a loop nest [50], as well as in other areas such as timing verification [5, 6]. The work of this dissertation and its predecessors [34, 100] fall into this category.

Clauss [39] computes the number of distinct memory locations touched by a loop nest, information used to improve the cache miss rate of a program. His problem is similar to mine, in that the objective for both is to determine the number of integer solutions resulting from a set of linear constraints. In Clauss' strategy, the integer solutions are represented geometrically as integer points belonging to polytopes, and he computes the number of points using Ehrhart polynomials [45]. Recall that in my framework, Presburger formulas are translated into DFAs in order to determine the number of integer solutions to the formulas. There exist canonical ways of translating Presburger formulas into DFAs with the minimum number of states. Moreover, it is understood how the worst-case complexity of the translation manifests (see Section 2.5). In contrast, even though the polytope approach experiences the same worst-case complexity, its manifestation is not as well-understood. There is no canonical way of dividing a set of overlapping polytopes into a set of disjoint polytopes, which is critical for obtaining an accurate count of the solutions. Clauss' method is also subject to geometric degeneracies for many problems of interest. The Polylib library [90] contains an implementation of Clauss' method.

Verdoolaege *et al.*. [125] extend Clauss' Ehrhart method for counting integer points in a parameterized polytope. In particular, they handle the problem of degenerate domains to some extent. By extending Barvinok's algorithm for computing the number of integer points in a non-parameterized polytope [11], Verdoolaege *et al.* compute Ehrhart polynomials analytically. This analytical solution is in contrast to Clauss' method of interpolation for computing the Ehrhart polynomial, which is vulnerable to failure in many cases. Despite improvements in Ehrhart methods for counting solutions, the lack of smoothness in formulas describing cache behavior (*i.e.*, solutions counts change dramatically as a function of input parameters) results in polynomials of high degree with impractically large numbers of coefficients. Ehrhart methods have the advantage of producing a symbolic expression of the solution count, but are often impractical for non-smooth formulas, such as those describing cache misses. The Presburger-DFA method for counting solutions gives solutions counts that are not symbolic, but performs well for non-smooth formulas. Furthermore, the usefulness of symbolic capabilities demonstrated by Clauss and Verdoolaege *et al.* is not limitless, since Presburger arithmetic does not allow the multiplication of two or more symbolic values. For instance, the index expression for accessing an array with two or more dimensions of symbolic size is not describable in Presburger arithmetic.

Several other works are concerned with counting solutions to Presburger formulas. Pugh's method [106] is based on computing sums and expresses the number of solutions symbolically. For a given summation problem, the choice of which techniques to apply and the order in which to apply them seems to require human intelligence, and no software implementation of Pugh's method exists. Barvinok's algorithm [11] (with subsequent improvements by Dyer and Kannan [44]) counts the number of integer points inside a convex polyhedron of fixed dimension in polynomial time. The LattE (Lattice point Enumeration) tool [42], the first known implementation of Barvinok's algorithm, is software for the enumeration of all lattice points inside a rational convex polyhedron. To count the number of solutions to a Presburger formula, Barvinok's algorithm requires a geometric preprocessing step to express the formula as a disjoint union of polyhedra, like Clauss' method. Boigelot and Latour [19, 20] represent a Presburger formula as a Number Decision Diagram and then count its accepting paths to produce the number of solutions. At first glance, this method looks quite similar to the one described in Section 5.2 and in [100]; however, there are several critical differences. In the Boigelot and Latour method, accepting paths represent satisfying variable values by interleaving the binary expansions of the values, while in my method, the binary expansions are stacked (see Section 5.1.1) resulting in DFAs with fewer numbers of states. Also, the Boigelot and Latour method encodes the binary expansions of satisfying variable values most-significant-bit first. The least-significant-bit-first ordering of my method has enabled the derivation of convergence properties (see Section 5.2.2) that determine when to terminating counting to ensure an accurate count. The LASH toolset [21] contains an implementation of the Boigelot and Latour method.

Bastoul and Feautrier [12] present a method, known as *chunking*, for improving data locality in a program. Essentially, their approach is to partition the program's operations into subsets small enough so that the data accessed by each *chunk* fits into cache. They evaluate the memory traffic of each chunk, and their optimization algorithm uses this information to find the reordering of program operations that minimizes memory traffic. In order to determine the number of memory blocks copied to cache during by each chunk, Bastoul and Feautrier must simplify the cache model significantly. They assume a fully-associative cache, and assume that the cache is flushed between execution of chunks. Clearly, such assumptions lead to an estimation of the number of cache misses a program incurs, which is sufficient for their optimization method. The goal of my cache analysis framework is different, requiring an accurate evaluation of misses for caches of any associativity.

9.4 Work Based on Reference Traces

Early approaches to analyzing memory behavior [1, 120] extracted parameters from the trace of memory references made by a program. The novel approach of Weikle [127, 128] views caches as filters. To gain insight on a program executing in a multilevel cache system, Weikle analyzes a synthetic trace of its memory accesses. Given an input sequence of memory accesses, the cache "filters" out the accesses that hit in the cache leaving the accesses that miss to pass through the cache filter. The work of Weikle introduces new locality metrics computed at every point in the trace, emphasizing recent behavior. Since these metrics are unique to Weikle's cache filter approach, my framework does not have any quantities that are comparable. Weikle's framework executes a program to generate its memory access trace, and thus, it is not limited to considering a particular program structure. Beyls and D'Hollander [14, 15, 16, 17, 18] also work from a memory access trace to compute reuse distances. Like Beyls and D'Hollander, Weikle trades speed for the flexibility of handling any program feature. Brehob and Enbody [24] offer models for measuring the locality of a reference trace and the expected cache behavior of the trace, using stack distance.

9.5 Work Based on Cache State

Ferdinand and Wilhelm [4, 48] apply abstract interpretation to the problem of classifying the cache behavior of memory references and use this information in predicting the WCET (Worst Case Execution Time) of a program or task. Their approach derives from the *abstract cache state* notion of Mueller *et al.* [97, 98], which applies to instruction caches. The work of Ferdinand and Wilhelm is similar to my framework in two ways. First, they also use a set-centric approach to cache behavior classification, considering an $\mathcal{A}$-way set-associative cache as $\mathcal{S}$ fully-associative caches. Second, Ferdinand and Wilhelm also have the notion of a cache state, although it is somewhat different from mine. They use cache state in classifying all memory accesses as cache misses or hits, whereas I use cache state only for classifying actual boundary misses. Also, the cache state technique of their framework is approximate, identifying memory blocks that are definitely in the cache (always a cache hit), memory blocks that definitely are not in the cache (always a cache miss), and memory blocks that may be in the cache (not classified). Ferdinand and Wilhelm must assume that the non-classified memory blocks incur cache misses. Therefore, they can predict an upper bound on the number of cache misses. Cache state in my framework is exact and leads to an exact prediction of cache misses.

9.6 Summary

The following revisits the six weaknesses given in Chapter 1 that are common to some or all of the existing frameworks for analyzing memory behavior discussed in this chapter.

- **Approximating cache behavior.** The CME framework of Ghosh *et al.* [59, 60, 61, 62] is based on reuse vectors, and as a result, only captures reuse among uniformly-generated references (*i.e.*, array references whose index expressions differ at most by a constant value). By sampling the iteration space of a program fragment, Vera *et al.* [121, 122, 123, 124] approximate the total number of cache misses incurred by the fragment based on the miss count of the sample. Beyls and D'Hollander [14, 15, 16, 17, 18] and Caşcaval [30, 31] use probabilistic arguments to deduce the approximate number of misses in set-associative caches from their cache behavior predictions valid in fully-associative caches with unit blocksize. Ferdinand and Wilhelm [4, 48] estimate cache behavior by giving an upper bound on the number of cache misses incurred by a program.

- **Modeling only fully-associative caches.** Beyls and D'Hollander, Caşcaval, and Bastoul and Feautrier [12] all present cache behavior models that consider only fully-associative caches in which memory blocks contain one data element.

- **Modeling data caches only, ignoring misses in the translation lookaside buffer (TLB).** None of the existing frameworks for analyzing memory behavior discussed here have been used to determine the number of TLB misses incurred by a program fragment.

- **Considering only perfectly-nested loops.** In modeling the cache behavior of a loop nest, Ghosh *et al.* require that the loop nest be perfectly-nested.
- **Considering loop nests in isolation of each other.** In modeling the cache behavior of multiple loop nests, both Ghosh *et al.* and Caşcaval assume a cold-start cache for each loop nest.
- **Handling only canonical array layout functions (*i.e.* row- and column-major).** None of the existing frameworks for analyzing memory behavior discussed here have been used to determine number of misses incurred by a program fragment that uses nonlinear array layouts.

As Chapter 1 claims, and the rest of this document demonstrates, the analysis framework presented in this dissertation does not suffer from any of the weaknesses listed above. Table 9.1 summarizes the comparison of several significant frameworks for doing static analysis of memory behavior with the analysis framework presented in this dissertation. The last row of Table 9.1 indicates whether the amount of time required by each framework to analyze the cache behavior of a loop nest (or sequence of loop nests) is small enough to make such analysis feasible within an optimizing compiler, and the entries of this row reflect my own opinion.

	Ghosh *et al.* [59,60,61,62]	Vera *et al.* [119,120,121,122]	Beyls and D'Hollander [14,15,16,17,18]	Caşcaval [30,31]	This Dissertation
Handles multiple loop nests	no	yes	yes	no	yes
Handles imperfectly-nested loops	no	yes	yes	yes	yes
Exactly models cache behavior	no	no	exact for fully-associative caches with unit blocksize	exact for fully-associative caches with unit blocksize	yes
Produces behavior for data caches	yes	yes	yes	yes	yes
Produces behavior for TLBs	no	no	no	no	yes
Handles nonlinear array layouts	no	no	no	no	yes
Handles symbolic problem sizes	no	no	in some cases	in some cases	no
Practical to run at compile time	no	yes	yes	yes	yes

Table 9.1: Comparison of several significant frameworks for doing static analysis of memory behavior with the analysis framework presented in this dissertation. (Refer to discussion in Section 9.6 for details.)

Chapter 10

Conclusions

Data access time continues to dominate the execution times of many programs. An important part of applying code and data transformations to improve program performance is understanding the behavior of programs executing in a memory hierarchy. This dissertation presents an analytical framework that determines the exact cache behavior of a program, given virtually any configuration of cache memory. In handling set-associative caches, data cache and TLB misses, imperfect loop nests, and nonlinear array layouts in an exact manner, the analytical framework given here goes beyond existing analytical frameworks for modeling cache behavior. This dissertation concludes with a list of the major thesis contributions (Section 10.1), commentary on how development of the analysis framework began and on the limitations of the framework (Section 10.2), and several directions for future research related to the analysis framework of this dissertation (Section 10.3).

10.1 Thesis Contributions

This dissertation describes and demonstrates an analytical framework for understanding the behavior of loop nests executing in a memory hierarchy. The framework has three components: 1) an alternative classification of cache misses that makes it possible to obtain the exact cache behavior of a sequence of program fragments by combining the cache behavior of the individual fragments; 2) the use of Presburger arithmetic to model data access patterns and describe events such as cache misses; and 3) algorithms exploiting the connection among Presburger arithmetic, automata theory, and graph theory to allow an exact model of cache behavior. The following are the major contributions of this dissertation revisited.

- *A new alternative cache miss classification has advantages over traditional cache miss classification schemes.* Chapter 3 introduces the interior-boundary cache miss classification scheme as an alternative to compulsory-replacement cache miss classification schemes. The interior-boundary miss classification has the vital property of composability, meaning that there is an exact way of relating the cache misses of individual program fragments to the cache misses of the entire program. As Section 3.1 shows, compulsory-replacement miss classification schemes lack composability, primarily because compulsory misses incurred by individual program fragments may not even be misses in a composition of such fragments.

The new miss classification has two types of cache misses: misses independent of cache state when a program fragment begins execution (called interior misses) and misses dependent on that cache state (called potential boundary misses). Determination of cache state before and after program fragment execution is used to resolve the potential boundary misses that actually do miss in cache (called boundary misses), and is the key to precisely combining the cache behavior of individual program fragments.

The total number of cache misses incurred by a program is the sum of the interior and boundary misses incurred by each fragment of the program. It is possible to approximate the cache behavior of a program by counting only interior misses, which omits the computation of cache state and potential boundary misses, yet yields an approximation of the total cache miss count with a tight error bound.

The interior-boundary cache miss classification and its composability property are demonstrated on a sequence of four loop nests from the `calc3` subroutine of `swim.f` in SPECfp95. Section 8.2 gives the results of using the cache analysis framework to compute the number of interior and boundary misses incurred by the `calc3` sequence. For each individual loop nest, the framework counts the number of interior misses incurred, identifies the potential boundary misses, and determines the cache state after execution of the loop nest. Then, the framework counts the boundary misses incurred by the loop nest by resolving the potential boundary misses with the cache state after execution of the preceding loop nest.

- *The cache analysis framework of this dissertation models the behavior of loop nests executing in set-associative caches.* The cache analysis framework models the exact behavior of loop nests in LRU caches of arbitrary associativity. For a fixed blocksize $\mathcal{B}$ and number of cache sets $\mathcal{S}$, the framework produces the cache behaviors of loop nests for all $\{(k, \mathcal{B}, k\mathcal{B}\mathcal{S}; \mathcal{S})\colon k \geqslant 1\}$ caches in one pass.

  In just one pass through the cache analysis framework, each set of results in Chapter 8 gives the cache miss counts for all associativity values $\mathcal{A} \geqslant 1$. Therefore, the framework is useful in determining the behavior of loop nests in real systems, and in imagining the behavior of loop nests in experimental systems.

- *The cache analysis framework of this dissertation models the data access patterns of arbitrarily-nested loops using Presburger arithmetic and exploits connections between Presburger arithmetic, automata theory, and graph theory to identify cache misses.* The cache analysis framework uses Presburger arithmetic to express various kinds of witnesses, which identify the situations that may cause a memory access to miss in cache depending on the cache's associativity. Section 4.3.2 gives the Presburger formulas describing witnesses for four types of associativity-dependent cache events: interior misses, replacement misses, potential boundary misses, and whether a memory block is in the cache state at the end of program fragment execution. For other associativity-dependent cache events of interest, the reader can define neighborhood and witness terms using these four examples as guides.

Exploiting a fundamental connection between Presburger arithmetic and automata theory, the analysis framework converts witness formulas to DFAs whose accepting paths encode the solutions of the formulas. Enumeration of the accepting paths in such DFAs determines the witnesses of each memory accesses. For each memory access, its witness count is compared with the value of $\mathcal{A}$ to determine the outcome of a cache event for a cache with associativity $\mathcal{A}$.

To determine the outcome of cache events not dependent on cache associativity, the cache analysis framework uses Presburger arithmetic to express the occurrence of the cache event directly. Section 5.4 gives the Presburger formulas describing interior misses in direct-mapped caches and compulsory misses. The analysis framework constructs DFAs recognizing the solutions of the miss formulas, exploiting the Presburger-DFA connection. To efficiently count (without enumerating) the accepting paths of such a DFA, the analysis framework treats the DFA as a weighted, directed graph. The number of solutions encoded by the accepting DFA paths is thus the number of cache misses described by the formula.

The automata-theoretic methods for counting and enumerating solutions to Presburger formulas work well for cache behavior formulas, and are relevant to many applications in program analysis that use linear constraints to model a behavior of interest.

- *The cache analysis framework of this dissertation is flexible.* The cache analysis framework models the occurrence of cache events that depend on associativity (such as interior misses, replacement misses, boundary misses, and cache state) and those that do not (such as interior misses in direct-mapped caches and compulsory misses). Chapter 8 gives the results of using the framework to produce the behaviors of various loop nests according to associativity-dependent cache events and non-associativity-dependent cache events.

  The analysis framework counts both data cache misses and TLB misses. Misses in the TLB are often overlooked in static analysis because their large associativity values and blocksizes are difficult to model. Chapter 8 gives the results of using the framework to produce data cache miss counts and TLB miss counts for various loop nests.

The cache analysis framework is also flexible in handling row- and column-major array layouts, as well as, nonlinear array layouts expressible in Presburger arithmetic. Section 8.3 gives the results of using the framework to produce cache miss counts for loop nests with various nonlinear array layouts.

- *The cache analysis framework of this dissertation is a tool for improving program behavior and exploring the space of memory design.* The analysis framework considers cache memories of virtually any configuration, which allows the determination of TLB miss counts, in addition to data cache miss counts. TLB misses have a significant impact on the performance of a loop nest. The penalty for a TLB miss can be hundreds of processor cycles, and for that reason, examining and reducing the TLB misses incurred by a program can result in a major improvement in its performance.

  In Chapter 8, results of using the cache analysis framework on a variety of problems give exact cache miss counts for many cache configurations, including data caches and TLBs. The framework is especially valuable for examining the effect of parameter change, such as for array starting addresses, blocksize, and number of cache sets. The framework is also useful in modeling configurations currently unrepresented by hardware.

  Section 8.4 considers two loop optimizations for reducing the cache misses incurred by a loop nest—loop tiling and loop permutation. In producing the cache miss counts for various tile sizes, the cache analysis framework is useful in selecting the best tile size, and in producing the cache miss counts for all permutations of a loop nest, the cache analysis framework is useful in selecting the best permutation.

10.2 Commentary

The work of this dissertation initially began in response to the Cache Miss Equations (CMEs) framework of Ghosh *et al.* [59, 60, 61, 62]. This well-known work based on reuse vectors generates linear Diophantine equations characterizing the cache misses of a single loop nest (see Section 9.1 for more description). My first observation of the CME work was the lack of support for multiple loop nests, both because traditional reuse vectors do not exist from loop nest to loop nest and because the compulsory-replacement misses of each loop nest do not combine to give those of a loop nest sequence. Furthermore, using reuse vectors to capture data access patterns is an abstraction that is not valid for all loop nests. I was struck that the CME work was not exploiting information available in loop nests to characterize the exact behavior of an entire sequence of loop nests. This led me to create a new interior-boundary classification of cache misses that addresses specifically the precise cache behavior of multiple loop nests.

My second observation of the CME work was the apparent difficulty of counting solutions to the cache miss equations. In fact, Ghosh *et al.* use the CMEs in ways that avoid counting their solutions directly, and the generality of the CME framework suffers as a result. I was encouraged to develop a method for counting formula solutions directly that works well for the types of formulas describing cache behavior. Polyhedral methods are one approach to counting solutions directly. However, while powerful mathematical results (such as the existence of Ehrhart polynomials) are known for polytopes, the corresponding algorithms are complex and subject to geometric degeneracies. Another approach is to express cache behavior as formulas of Presburger arithmetic, although the large worst-case complexity of Presburger arithmetic decision procedures is deterring at first. In order to count Presburger formula solutions, I choose to exploit the powerful connection between Presburger arithmetic and deterministic finite automata (DFAs). Taking advantage of this connection does not circumvent the difficulty of counting solutions directly, but its manifestation is better understood and manageable in the context of DFAs. Unlike parametric methods for counting solutions, the Presburger-DFA method produces solution counts that are not symbolic. Parametric methods are not always useful, in particular when arrays have two or more dimensions of symbolic size, since their index expressions are not expressible in Presburger arithmetic. Of course, the real success of the Presburger-DFA method for counting solutions is demonstrated in Chapter 8.

The overall objective of my cache analysis framework is to produce *exact* cache miss counts, in contrast to the CME work. Certainly, in achieving such a goal it is necessary to forego the shortcuts of making approximations in the cache behavior. I really want to push the limits of generating exact cache miss counts. I often find that as the complexity and problem sizes of loop nests increase, so does the likelihood of failure in one of the tools employed by the framework. This is particularly true for formula simplification (performed by the Omega Library [75]) and DFA-construction (accomplished with the construction algorithms of Bartzis and Bultan [9, 10] and MONA's DFA Library [81]). Failures are due either to limits in the underlying theory of the tools or the lack of robustness in their implementations. I believe that both reasons share the responsibility, as some instances of cache behavior attack weaknesses of the implementation, while others approach the worst-case complexity associated with Presburger arithmetic. Improvements in any of the tools utilized by my cache analysis framework will lead to more success in and improve the speed of producing cache miss counts.

Even with its limitations, the cache analysis framework presented in this dissertation is an effective tool for measuring the cache performance of a loop nest sequence. The interior-boundary classification of cache misses has the vital property of composability and provides a new way of looking at cache behavior. It can be used with its counterpart the compulsory-replacement miss classification to study the cache performance of programs. Orthogonal to the new miss classification is a new method for expressing and counting the cache misses incurred by a loop nest executing in set-associative caches. The Presburger-DFA method's flexibility in handling imperfectly-nested loops and a variety of data layouts while considering caches of any associativity permits the study of virtually any loop nest and any cache configuration.

10.3 Future Research Directions

The cache analysis framework presented in this dissertation offers several opportunities for future research. Improving the success rate and the time required to produce the cache behavior of a loop nest would have the most significant impact on the effectiveness of the framework (Section 10.3.1). Extending the framework to model more cache features and events would enhance the generality of the framework (Section 10.3.2). Finally, applying the framework to a broader class of programs would increase the framework's value (Section 10.3.3).

10.3.1 Improving Framework Robustness

To model the cache behavior of loop nests, the analysis framework presented here depends on existing tools to perform formula simplification and automata construction. The framework utilizes the Omega Library [75] to simplify cache behavior formulas before constructing the DFAs representing the formulas. Formula simplification is a critical phase of the framework since unsimplified formulas are often too complicated for efficient translation to DFAs. As the complexity and problem sizes of loop nests increase, so does the likelihood that the Omega Library will fail during cache formula simplification. The Omega Library is designed for use in data dependence analysis, where formulas typically contain small coefficients and constants. With relatively large coefficients and constants, cache behavior formulas can push the Omega Library to its limit.

The analysis framework uses the automata-construction algorithms of Bartzis and Bultan [9, 10] and the MONA tool [81] to construct DFAs recognizing the solutions of cache behavior formulas. The likelihood of failures during DFA construction also increases with the complexity and problem sizes of loop nests. Like the Omega Library, the DFA-construction tools are not designed to handle formulas as large and complicated as the cache behavior formulas. Efforts to make either the Omega Library or the automata-construction tools more robust for large formulas will certainly improve the robustness of the analysis framework. Furthermore, both the Omega Library and the automata-construction tools are generalized to handle any Presburger formula. It is possible that formula simplification and/or DFA construction specialized for cache behavior formulas could lead to improvements in the running time of the cache analysis framework. There are similarities among the formulas and DFAs generated to describe the cache behaviors of all loop nests, and exploitation of such commonalities to improve the speed of analysis is one direction for future work.

10.3.2 Modeling More Cache Features and Events

The analysis framework of this dissertation models a basic cache. It may be possible to use the analysis framework to model auxiliary cache features, if the behavior of such cache features is expressible in Presburger arithmetic. Some candidates for inclusion in the framework are common in systems today, such as software prefetching [94, 95, 96], hardware prefetching [70, 101], and victim caches [70]; and some are novel features, such as cache bypassing [69] and cache decay [72, 73]. The key to modeling prefetching lies in the description of iteration points for memory references, to capture the point at which a memory block is prefetched. It is likely that the modeling techniques developed for the main cache can be adapted to handle the victim cache, a small fully-associative cache acting as a write buffer. For cache bypassing, a technique that reads and writes data directly to main memory in cases of no reuse, it may be possible both to identify instances without reuse (*i.e.*, cache bypass candidates) and to describe the actual bypassing of the cache. Cache decay, a mechanism for leakage-reduction that turns off "dead" cache lines, would benefit from an analytical determination of the last access to a memory block before its eviction from cache.

Given my experience working with Presburger arithmetic, I believe that all of the cache features above, as well as others, can be described using Presburger formulas. The analysis framework presented in this dissertation is fundamentally flexible. The cache modeling techniques at its core may be adapted and extended to model any feature of a program or cache memory describable in Presburger arithmetic. Furthermore, the framework may be used to determine the outcome of other cache events expressed in Presburger arithmetic, using the cache events used as examples throughout this dissertation as guidelines. Applying the analysis framework presented here to model more cache features and events is one direction for future work.

10.3.3 Applying the Analysis Framework

The analysis framework of this dissertation for understanding the behavior of loop nests executing in a memory hierarchy is useful for exploring new memory system designs and guiding code and data transformations for improved program performance. Another use of the analysis framework would be to leverage the strengths of both static and dynamic analysis to profile a broader class of programs. One reason for favoring static cache analysis over explicit cache simulation is speed. Generally speaking, 90% of a program's running time is spent executing only 10% of the code. If the frequently-executed code happens to be analyzable loop nests, then static analysis of this code is often faster than explicit simulation. On the other hand, some program features are difficult or impossible to analyze at compile time, but can be captured effectively by explicit simulation. Static analysis must either neglect or attempt to approximate the cache behavior of such features.

It seems ideal to combine the strengths of both static analysis and dynamic simulation in a mixed-mode cache simulator. The static portion of the simulator would handle all analyzable loop nests, and the dynamic portion would handle everything else. The key to making the transition from dynamic simulation to static analysis is cache state, which is precisely modeled by the analysis framework presented here. Development of a mixed-mode cache simulator that switches between explicit simulation and static analysis of a program, thereby utilizing the cache state feature of this dissertation, is one direction for future work.

10.4 Summary

This dissertation presents an analytical framework for understanding the behavior of loop-oriented programs executing in a memory hierarchy. The framework has three components: the interior-boundary cache miss classification, the use of Presburger arithmetic to describe cache events, and the exploitation of connections among Presburger arithmetic, automata theory, and graph theory to model cache behavior exactly. These components allow the framework to handle set-associative caches, data cache and translation lookaside buffer (TLB) misses, imperfect loop nests, and nonlinear array layouts in an exact manner. With such capabilities, the framework of this dissertation goes beyond existing analytical frameworks for modeling cache behavior. The framework is useful in guiding code and data transformations for improved program performance and in exploring new memory system designs.

Appendix A

Example Witness Formulas and DFAs

Figures A.1 to A.3 give the Presburger formulas constructed to describe various types of witnesses for the running example loop nest $\mathbb{L}_{\mathrm{mm}}$ in Figure 2.5 on page 17. The formulas are for for cache set 0 and consider all ($\mathcal{A}$, 32, $4096 \cdot \mathcal{A}$; 128) caches with any associativity value $\mathcal{A}$. The general Presburger formulas for describing r-witnesses, b-witnesses, and s-witnesses are given in Section 4.3.2. Figures A.4 to A.6 show the DFAs that recognize the solutions of the Presburger formulas in Figures A.1 to A.3.

Figure A.7 gives the Presburger formula constructed to describe interior misses for the running example loop nest $\mathbb{L}_{\mathrm{mm}}$ in Figure 2.5. The formula is for cache set 0 and considers a (1, 32, 4096; 128) cache. The general Presburger formulas for describing interior misses in direct-mapped caches is given in Section 5.4.1. Figure A.8 shows the DFA that recognizes the solutions of the Presburger formula in Figure A.7.

Figure A.9 gives the Presburger formula constructed to describe compulsory misses for the running example loop nest $\mathbb{L}_{\mathrm{mm}}$ in Figure 2.5. The formula is for cache set 0 and considers all ($\mathcal{A}$, 32, $4096 \cdot \mathcal{A}$; 128) caches with any associativity value $\mathcal{A}$. The general Presburger formulas for describing compulsory misses is given in Section 5.4.2. Figure A.10 shows the DFA that recognizes the solutions of the Presburger formula in Figure A.9.

Figure A.11 gives the Presburger formula constructed to describe i-witnesses for the running example loop nest $\mathbb{L}_{\mathrm{mm}}$ in Figure 2.5. In this instance of the running example, $t = u = v = 16$ and all arrays are linearized according to a nonlinear array layout specified by the $(4, 4)$-interleaving $\sigma = 01010011$. The formula is for cache set 0 and considers all ($\mathcal{A}$, 32, $4096 \cdot \mathcal{A}$; 128) caches with any associativity value $\mathcal{A}$. Figure A.12 shows the DFA that recognizes the solutions of the Presburger formula in Figure A.11.

$$
\begin{aligned}
&0 \leqslant \iota_0, \iota_1, \iota_2 < 20 \,\wedge \\
&\Big(\exists d : ((\iota_2 = 0 \wedge u = 0 \wedge 32 * (128 * d) \leqslant 6400 + (\iota_0 + 20 * \iota_1)8 < 32 * (128 * d + 1)) \vee \\
&\quad (u = 1 \wedge 32 * (128 * d) \leqslant (\iota_0 + 20 * \iota_2) * 8 < 32 * (128 * d + 1)) \vee \\
&\quad (u = 2 \wedge 32 * (128 * d) \leqslant 3200 + (\iota_2 + 20 * \iota_1) * 8 < 32 * (128 * d + 1)) \vee \\
&\quad (\iota_2 = 19 \wedge u = 3 \wedge 32 * (128 * d) \leqslant 6400 + (\iota_0 + 20 * \iota_1) * 8 < 32 * (128 * d + 1))) \wedge \\
&\quad \big(\exists \kappa_0, \kappa_1, \kappa_2, v : 0 \leqslant \kappa_0, \kappa_1, \kappa_2 < 20 \wedge (\kappa_0 < \iota_0 \vee (\kappa_0 = \iota_0 \wedge \kappa_1 < \iota_1) \vee \\
&\qquad (\kappa_0 = \iota_0 \wedge \kappa_1 = \iota_1 \wedge \kappa_2 < \iota_2) \vee (\kappa_0 = \iota_0 \wedge \kappa_1 = \iota_1 \wedge \kappa_2 = \iota_2 \wedge v < u)) \wedge \\
&\qquad ((\kappa_2 = 0 \wedge v = 0 \wedge 32 * (128 * d) \leqslant 6400 + (\kappa_0 + 20 * \kappa_1) * 8 < 32 * (128 * d + 1)) \vee \\
&\qquad (v = 1 \wedge 32 * (128 * d) \leqslant (\kappa_0 + 20 * \kappa_2) * 8 < 32 * (128 * d + 1)) \vee \\
&\qquad (v = 2 \wedge 32 * (128 * d) \leqslant 3200 + (\kappa_2 + 20 * \kappa_1) * 8 < 32 * (128 * d + 1)) \vee \\
&\qquad (\kappa_2 = 19 \wedge v = 3 \wedge 32 * (128 * d) \leqslant 6400 + (\kappa_0 + 20 * \kappa_1) * 8 < 32 * (128 * d + 1))) \wedge \\
&\qquad (\neg(\exists \nu_0, \nu_1, \nu_2, t : 0 \leqslant \nu_0, \nu_1, \nu_2 < 20 \wedge (\nu_0 < \iota_0 \vee (\nu_0 = \iota_0 \wedge \nu_1 < \iota_1) \vee \\
&\qquad\quad (\nu_0 = \iota_0 \wedge \nu_1 = \iota_1 \wedge \nu_2 < \iota_2) \vee (\nu_0 = \iota_0 \wedge \nu_1 = \iota_1 \wedge \nu_2 = \iota_2 \wedge t < u)) \wedge \\
&\qquad\quad (\kappa_0 < \nu_0 \vee (\kappa_0 = \nu_0 \wedge \kappa_1 < \nu_1) \vee (\kappa_0 = \nu_0 \wedge \kappa_1 = \nu_1 \wedge \kappa_2 < \nu_2) \vee \\
&\qquad\quad (\kappa_0 = \nu_0 \wedge \kappa_1 = \nu_1 \wedge \kappa_2 = \nu_2 \wedge v < t)) \wedge \\
&\qquad\quad ((\nu_2 = 0 \wedge t = 0 \wedge 32 * (128 * d) \leqslant 6400 + (\nu_0 + 20 * \nu_1) * 8 < 32 * (128 * d + 1)) \vee \\
&\qquad\quad (t = 1 \wedge 32 * (128 * d) \leqslant (\nu_0 + 20 * \nu_2) * 8 < 32 * (128 * d + 1)) \vee \\
&\qquad\quad (t = 2 \wedge 32 * (128 * d) \leqslant 3200 + (\nu_2 + 20 * \nu_1) * 8 < 32 * (128 * d + 1)) \vee \\
&\qquad\quad (\nu_2 = 19 \wedge t = 3 \wedge 32 * (128 * d) \leqslant 6400 + (\nu_0 + 20 * \nu_1) * 8 < 32 * (128 * d + 1))))) \wedge \\
&\qquad (\exists \rho_0, \rho_1, \rho_2, w : 0 \leqslant \rho_0, \rho_1, \rho_2 < 20 \wedge (\rho_0 < \iota_0 \vee (\rho_0 = \iota_0 \wedge \rho_1 < \iota_1) \vee \\
&\qquad\quad (\rho_0 = \iota_0 \wedge \rho_1 = \iota_1 \wedge \rho_2 < \iota_2) \vee (\rho_0 = \iota_0 \wedge \rho_1 = \iota_1 \wedge \rho_2 = \iota_2 \wedge w < u)) \wedge \\
&\qquad\quad (\kappa_0 < \rho_0 \vee (\kappa_0 = \rho_0 \wedge \kappa_1 < \rho_1) \vee (\kappa_0 = \rho_0 \wedge \kappa_1 = \rho_1 \wedge \kappa_2 < \rho_2) \vee \\
&\qquad\quad (\kappa_0 = \rho_0 \wedge \kappa_1 = \rho_1 \wedge \kappa_2 = \rho_2 \wedge v < w)) \wedge \\
&\qquad\quad ((\rho_2 = 0 \wedge w = 0 \wedge 32 * (128 * e) \leqslant 6400 + (\rho_0 + 20 * \rho_1) * 8 < 32 * (128 * e + 1)) \vee \\
&\qquad\quad (w = 1 \wedge 32 * (128 * e) \leqslant (\rho_0 + 20 * \rho_2) * 8 < 32 * (128 * e + 1)) \vee \\
&\qquad\quad (w = 2 \wedge 32 * (128 * e) \leqslant 3200 + (\rho_2 + 20 * \rho_1) * 8 < 32 * (128 * e + 1)) \vee \\
&\qquad\quad (\rho_2 = 19 \wedge w = 3 \wedge 32 * (128 * e) \leqslant 6400 + (\rho_0 + 20 * \rho_1) * 8 < 32 * (128 * e + 1))))\big) \\
&\quad \wedge \neg(d = e)\Big)
\end{aligned}
$$

Figure A.1: Presburger formula describing the r-witnesses of loop nest $\mathbb{L}_{\text{mm}}$ for cache set 0.

$$
\begin{aligned}
&0 \leqslant \iota_0, \iota_1, \iota_2 < 20\ \wedge \\
&\Big(\exists d : ((\iota_2 = 0 \wedge u = 0 \wedge 32 * (128 * d) \leqslant 6400 + (\iota_0 + 20 * \iota_1) * 8 < 32 * (128 * d + 1)) \vee \\
&\quad (u = 1 \wedge 32 * (128 * d) \leqslant (\iota_0 + 20 * \iota_2) * 8 < 32 * (128 * d + 1)) \vee \\
&\quad (u = 2 \wedge 32 * (128 * d) \leqslant 3200 + (\iota_2 + 20 * \iota_1) * 8 < 32 * (128 * d + 1)) \vee \\
&\quad (\iota_2 = 19 \wedge u = 3 \wedge 32 * (128 * d) \leqslant 6400 + (\iota_0 + 20 * \iota_1) * 8 < 32 * (128 * d + 1))) \wedge \\
&\quad \big(\exists \kappa_0, \kappa_1, \kappa_2, v : 0 \leqslant \kappa_0, \kappa_1, \kappa_2 < 20\ \wedge \\
&\qquad (\kappa_0 < \iota_0 \vee (\kappa_0 = \iota_0 \wedge \kappa_1 < \iota_1) \vee \\
&\qquad (\kappa_0 = \iota_0 \wedge \kappa_1 = \iota_1 \wedge \kappa_2 < \iota_2) \vee \\
&\qquad (\kappa_0 = \iota_0 \wedge \kappa_1 = \iota_1 \wedge \kappa_2 = \iota_2 \wedge v < u)) \wedge \\
&\qquad ((\kappa_2 = 0 \wedge v = 0 \wedge 32 * (128 * e) \leqslant 6400 + (\kappa_0 + 20 * \kappa_1) * 8 < 32 * (128 * e + 1)) \vee \\
&\qquad (v = 1 \wedge 32 * (128 * e) \leqslant (\kappa_0 + 20 * \kappa_2) * 8 < 32 * (128 * e + 1)) \vee \\
&\qquad (v = 2 \wedge 32 * (128 * e) \leqslant 3200 + (\kappa_2 + 20 * \kappa_1) * 8 < 32 * (128 * e + 1)) \vee \\
&\qquad (\kappa_2 = 19 \wedge v = 3 \wedge 32 * (128 * e) \leqslant 6400 + (\kappa_0 + 20 * \kappa_1) * 8 < 32 * (128 * e + 1)))\big) \wedge \\
&\quad \big(\neg(\exists \rho_0, \rho_1, \rho_2, w : 0 \leqslant \rho_0, \rho_1, \rho_2 < 20\ \wedge \\
&\qquad (\rho_0 < \iota_0 \vee (\rho_0 = \iota_0 \wedge \rho_1 < \iota_1) \vee \\
&\qquad (\rho_0 = \iota_0 \wedge \rho_1 = \iota_1 \wedge \rho_2 < \iota_2) \vee \\
&\qquad (\rho_0 = \iota_0 \wedge \rho_1 = \iota_1 \wedge \rho_2 = \iota_2 \wedge w < u)) \wedge \\
&\qquad ((\rho_2 = 0 \wedge w = 0 \wedge 32 * (128 * d) \leqslant 6400 + (\rho_0 + 20 * \rho_1) * 8 < 32 * (128 * d + 1)) \vee \\
&\qquad (w = 1 \wedge 32 * (128 * d) \leqslant (\rho_0 + 20 * \rho_2) * 8 < 32 * (128 * d + 1)) \vee \\
&\qquad (w = 2 \wedge 32 * (128 * d) \leqslant 3200 + (\rho_2 + 20 * \rho_1) * 8 < 32 * (128 * d + 1)) \vee \\
&\qquad (\rho_2 = 19 \wedge w = 3 \wedge 32 * (128 * d) \leqslant 6400 + (\rho_0 + 20 * \rho_1) * 8 < 32 * (128 * d + 1))))\big) \\
&\quad \wedge \neg(d = e)\Big)
\end{aligned}
$$

Figure A.2: Presburger formula describing the b-witnesses of loop nest $\mathbb{L}_{\text{mm}}$ for cache set 0.

$$
\begin{aligned}
&0 \leqslant \iota_0, \iota_1, \iota_2 < 20 \wedge \\
&\Big(\exists d : ((\iota_2 = 0 \wedge u = 0 \wedge 32 * (128 * d) \leqslant 6400 + (\iota_0 + 20 * \iota_1) * 8 < 32 * (128 * d + 1)) \vee \\
&\quad (u = 1 \wedge 32 * (128 * d) \leqslant (\iota_0 + 20 * \iota_2) * 8 < 32 * (128 * d + 1)) \vee \\
&\quad (u = 2 \wedge 32 * (128 * d) \leqslant 3200 + (\iota_2 + 20 * \iota_1) * 8 < 32 * (128 * d + 1)) \vee \\
&\quad (\iota_2 = 19 \wedge u = 3 \wedge 32 * (128 * d) \leqslant 6400 + (\iota_0 + 20 * \iota_1) * 8 < 32 * (128 * d + 1))) \wedge \\
&\quad \big(\exists \kappa_0, \kappa_1, \kappa_2, v : 0 \leqslant \kappa_0, \kappa_1, \kappa_2 < 20 \wedge \\
&\qquad (\iota_0 < \kappa_0 \vee (\iota_0 = \kappa_0 \wedge \iota_1 < \kappa_1) \vee \\
&\qquad (\iota_0 = \kappa_0 \wedge \iota_1 = \kappa_1 \wedge \iota_2 < \kappa_2) \vee \\
&\qquad (\iota_0 = \kappa_0 \wedge \iota_1 = \kappa_1 \wedge \iota_2 = \kappa_2 \wedge u < v)) \wedge \\
&\qquad ((\kappa_2 = 0 \wedge v = 0 \wedge 32 * (128 * e) \leqslant 6400 + (\kappa_0 + 20 * \kappa_1) * 8 < 32(128 * e + 1)) \vee \\
&\qquad (v = 1 \wedge 32 * (128 * e) \leqslant (\kappa_0 + 20 * \kappa_2) * 8 < 32 * (128 * e + 1)) \vee \\
&\qquad (v = 2 \wedge 32 * (128 * e) \leqslant 3200 + (\kappa_2 + 20 * \kappa_1) * 8 < 32 * (128 * e + 1)) \vee \\
&\qquad (\kappa_2 = 19 \wedge v = 3 \wedge 32 * (128 * e) \leqslant 6400 + (\kappa_0 + 20 * \kappa_1) * 8 < 32 * (128 * e + 1)))\big) \wedge \\
&\quad \big(\neg(\exists \rho_0, \rho_1, \rho_2, w : 0 \leqslant \rho_0, \rho_1, \rho_2 < 20 \wedge \\
&\qquad (\iota_0 < \rho_0 \vee (\iota_0 = \rho_0 \wedge \iota_1 < \rho_1) \vee \\
&\qquad (\iota_0 = \rho_0 \wedge \iota_1 = \rho_1 \wedge \iota_2 < \rho_2) \vee \\
&\qquad (\iota_0 = \rho_0 \wedge \iota_1 = \rho_1 \wedge \iota_2 = \rho_2 \wedge u < w)) \wedge \\
&\qquad ((\rho_2 = 0 \wedge w = 0 \wedge 32 * (128 * d) \leqslant 6400 + (\rho_0 + 20 * \rho_1) * 8 < 32 * (128 * d + 1)) \vee \\
&\qquad (w = 1 \wedge 32 * (128 * d) \leqslant (\rho_0 + 20\rho_2) * 8 < 32 * (128 * d + 1)) \vee \\
&\qquad (w = 2 \wedge 32 * (128 * d) \leqslant 3200 + (\rho_2 + 20 * \rho_1) * 8 < 32 * (128 * d + 1)) \vee \\
&\qquad (\rho_2 = 19 \wedge w = 3 \wedge 32 * (128 * d) \leqslant 6400 + (\rho_0 + 20 * \rho_1) * 8 < 32 * (128 * d + 1))))\big) \\
&\quad \wedge \neg(d = e)\Big)
\end{aligned}
$$

Figure A.3: Presburger formula describing the s-witnesses of loop nest $\mathbb{L}_{\mathrm{mm}}$ for cache set 0.

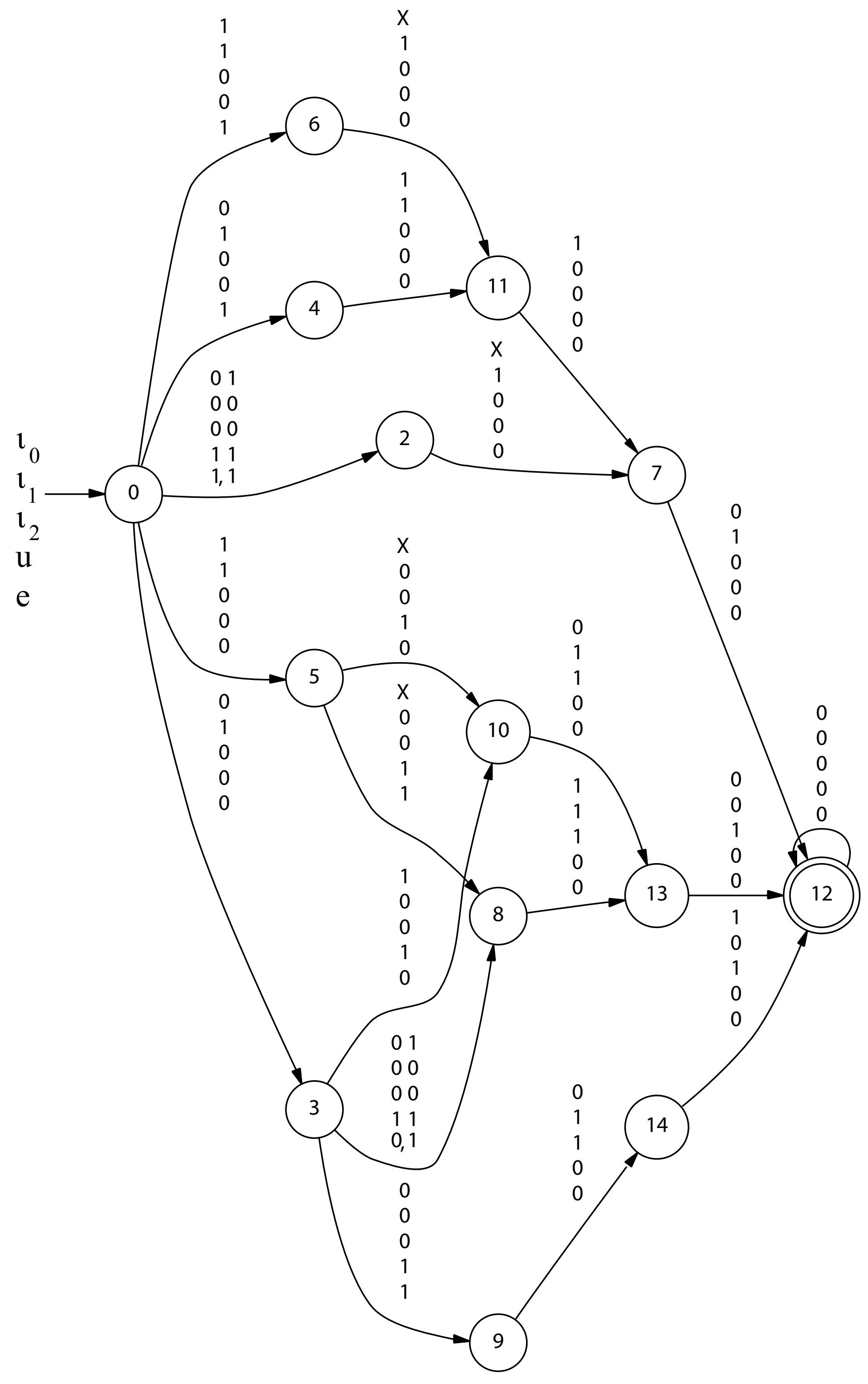


Figure A.4: DFA recognizing the solutions of the example Presburger formula in Figure A.1.

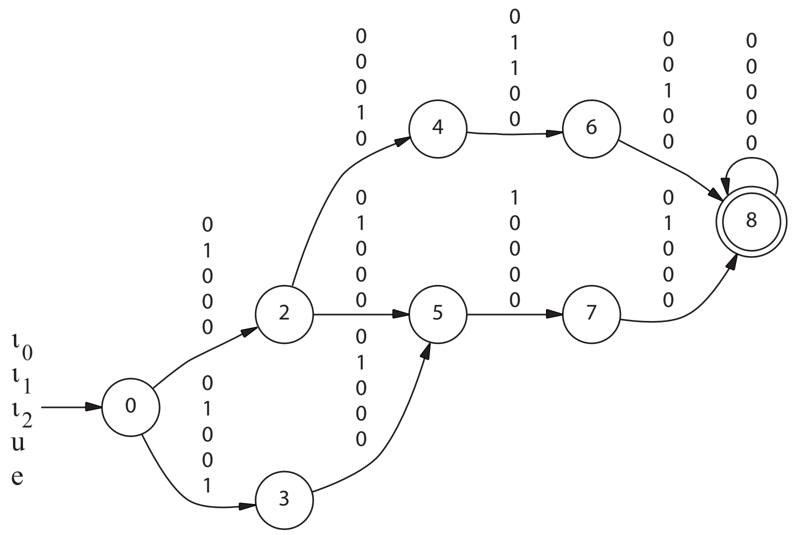


Figure A.5: DFA recognizing the solutions of the example Presburger formula in Figure A.2.

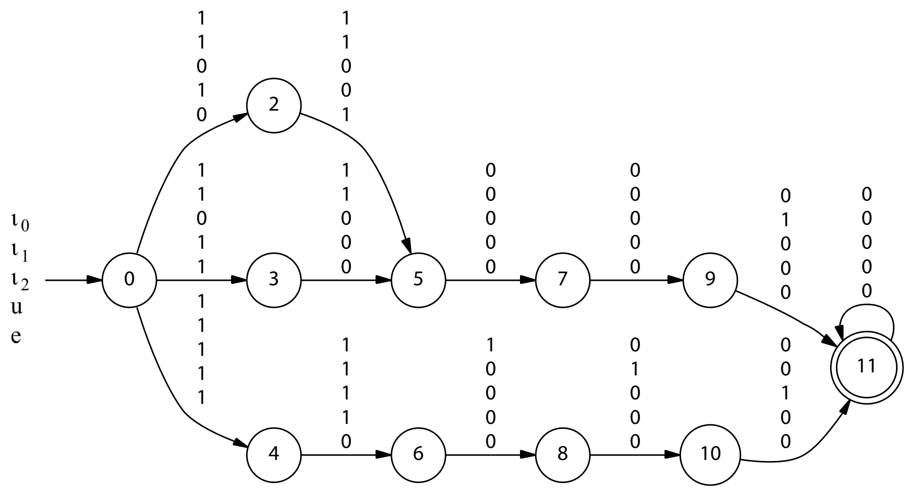


Figure A.6: DFA recognizing the solutions of the example Presburger formula in Figure A.3.

$$
\begin{aligned}
&0 \leqslant \iota_0, \iota_1, \iota_2 < 20 \wedge \\
&\Big(\exists d : ((\iota_2 = 0 \wedge u = 0 \wedge 32 * (128 * d) \leqslant 6400 + (\iota_0 + 20 * \iota_1) * 8 < 32 * (128 * d + 1)) \vee \\
&\quad (u = 1 \wedge 32 * (128 * d) \leqslant (\iota_0 + 20 * \iota_2) * 8 < 32 * (128 * d + 1)) \vee \\
&\quad (u = 2 \wedge 32 * (128 * d) \leqslant 3200 + (\iota_2 + 20 * \iota_1) * 8 < 32 * (128 * d + 1)) \vee \\
&\quad (\iota_2 = 19 \wedge u = 3 \wedge 32 * (128 * d) \leqslant 6400 + (\iota_0 + 20 * \iota_1) * 8 < 32 * (128 * d + 1))) \wedge \\
&\quad \big(\exists e, \kappa_0, \kappa_1, \kappa_2, v : 0 \leqslant \kappa_0, \kappa_1, \kappa_2 < 20 \wedge \\
&\quad\quad (\kappa_0 < \iota_0 \vee (\kappa_0 = \iota_0 \wedge \kappa_1 < \iota_1) \vee \\
&\quad\quad (\kappa_0 = \iota_0 \wedge \kappa_1 = \iota_1 \wedge \kappa_2 < \iota_2) \vee \\
&\quad\quad (\kappa_0 = \iota_0 \wedge \kappa_1 = \iota_1 \wedge \kappa_2 = \iota_2 \wedge v < u)) \wedge \\
&\quad\quad ((\kappa_2 = 0 \wedge v = 0 \wedge 32 * (128 * e) \leqslant 6400 + (\kappa_0 + 20 * \kappa_1) * 8 < 32 * (128 * e + 1)) \vee \\
&\quad\quad (v = 1 \wedge 32 * (128 * e) \leqslant (\kappa_0 + 20 * \kappa_2) * 8 < 32 * (128 * e + 1)) \vee \\
&\quad\quad (v = 2 \wedge 32 * (128 * e) \leqslant 3200 + (\kappa_2 + 20 * \kappa_1) * 8 < 32 * (128 * e + 1)) \vee \\
&\quad\quad (\kappa_2 = 19 \wedge v = 3 \wedge 32 * (128 * e) \leqslant 6400 + (\kappa_0 + 20 * \kappa_1) * 8 < 32 * (128 * e + 1))) \wedge \\
&\quad\quad (\neg(\exists \rho_0, \rho_1, \rho_2, w : 0 \leqslant \rho_0, \rho_1, \rho_2 < 20 \wedge \\
&\quad\quad\quad (\rho_0 < \iota_0 \vee (\rho_0 = \iota_0 \wedge \rho_1 < \iota_1) \vee \\
&\quad\quad\quad (\rho_0 = \iota_0 \wedge \rho_1 = \iota_1 \wedge \rho_2 < \iota_2) \vee \\
&\quad\quad\quad (\rho_0 = \iota_0 \wedge \rho_1 = \iota_1 \wedge \rho_2 = \iota_2 \wedge w < u)) \wedge \\
&\quad\quad\quad (\kappa_0 < \rho_0 \vee (\kappa_0 = \rho_0 \wedge \kappa_1 < \rho_1) \vee \\
&\quad\quad\quad (\kappa_0 = \rho_0 \wedge \kappa_1 = \rho_1 \wedge \kappa_2 < \rho_2) \vee \\
&\quad\quad\quad (\kappa_0 = \rho_0 \wedge \kappa_1 = \rho_1 \wedge \kappa_2 = \rho_2 \wedge v < w)) \wedge \\
&\quad\quad\quad ((\rho_2 = 0 \wedge w = 0 \wedge 32 * (128 * d) \leqslant 6400 + (\rho_0 + 20 * \rho_1) * 8 < 32 * (128 * d + 1)) \vee \\
&\quad\quad\quad (w = 1 \wedge 32 * (128 * d) \leqslant (\rho_0 + 20 * \rho_2) * 8 < 32 * (128 * d + 1)) \vee \\
&\quad\quad\quad (w = 2 \wedge 32 * (128 * d) \leqslant 3200 + (\rho_2 + 20 * \rho_1) * 8 < 32 * (128 * d + 1)) \vee \\
&\quad\quad\quad (\rho_2 = 19 \wedge w = 3 \wedge 32 * (128 * d) \leqslant 6400 + (\rho_0 + 20 * \rho_1) * 8 < 32 * (128 * d + 1)))))\big) \\
&\quad \wedge \neg(d = e)\Big)
\end{aligned}
$$

Figure A.7: Presburger formula describing interior misses of loop nest $\mathbb{L}_{\mathrm{mm}}$ in a direct-mapped cache.

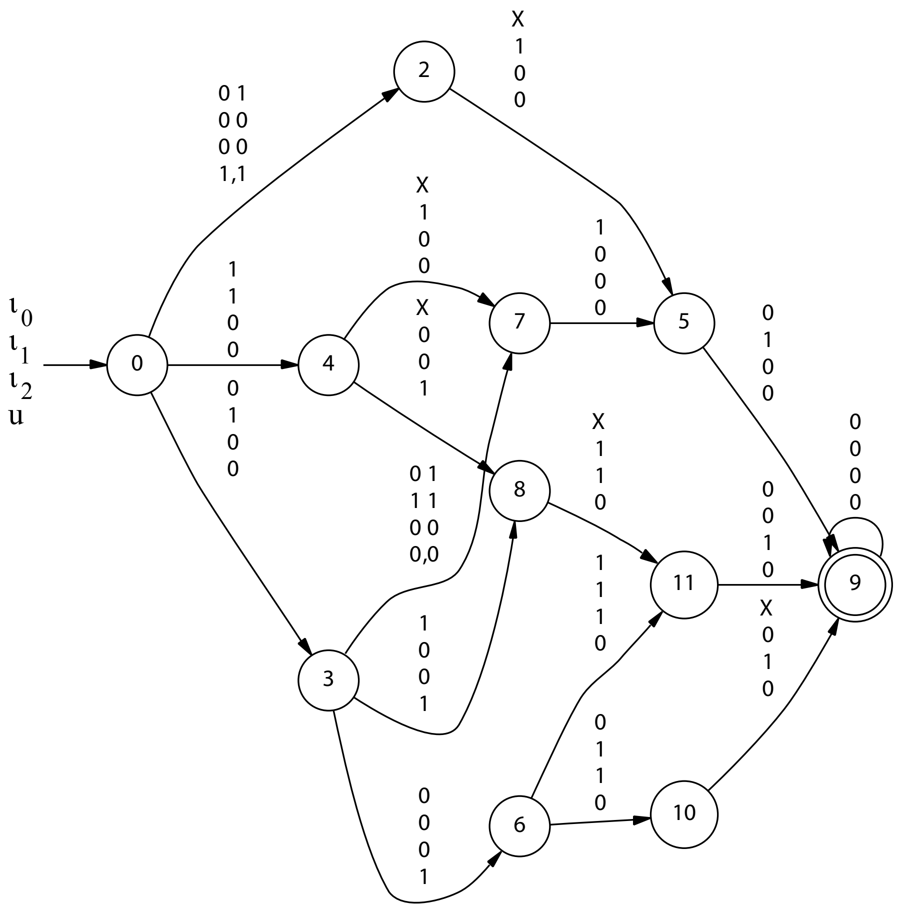


Figure A.8: DFA recognizing the solutions of the example Presburger formula in Figure A.7.

$$
\begin{aligned}
&0 \leqslant \iota_0, \iota_1, \iota_2 < 20 \wedge \\
&\Big(\exists d : ((\iota_2 = 0 \wedge u = 0 \wedge 32 * (128 * d) \leqslant 6400 + (\iota_0 + 20 * \iota_1) * 8 < 32 * (128 * d + 1))\vee \\
&\quad (u = 1 \wedge 32 * (128 * d) \leqslant (\iota_0 + 20 * \iota_2) * 8 < 32 * (128 * d + 1)) \vee \\
&\quad (u = 2 \wedge 32 * (128 * d) \leqslant 3200 + (\iota_2 + 20 * \iota_1) * 8 < 32 * (128 * d + 1)) \vee \\
&\quad (\iota_2 = 19 \wedge u = 3 \wedge 32 * (128 * d) \leqslant 6400 + (\iota_0 + 20 * \iota_1) * 8 < 32 * (128 * d + 1))) \wedge \\
&\quad \big(\neg(\exists \kappa_0, \kappa_1, \kappa_2, v : 0 \leqslant \kappa_0, \kappa_1, \kappa_2 < 20 \wedge \\
&\qquad (\kappa_0 < \iota_0 \vee (\kappa_0 = \iota_0 \wedge \kappa_1 < \iota_1) \vee (\kappa_0 = \iota_0 \wedge \kappa_1 = \iota_1 \wedge \kappa_2 < \iota_2) \vee \\
&\qquad (\kappa_0 = \iota_0 \wedge \kappa_1 = \iota_1 \wedge \kappa_2 = \iota_2 \wedge v < u)) \wedge \\
&\qquad ((\kappa_2 = 0 \wedge v = 0 \wedge 32 * (128 * d) \leqslant 6400 + (\kappa_0 + 20 * \kappa_1) * 8 < 32 * (128 * d + 1)) \vee \\
&\qquad (v = 1 \wedge 32 * (128 * d) \leqslant (\kappa_0 + 20 * \kappa_2) * 8 < 32 * (128 * d + 1)) \vee \\
&\qquad (v = 2 \wedge 32 * (128 * d) \leqslant 3200 + (\kappa_2 + 20 * \kappa_1) * 8 < 32 * (128 * d + 1)) \vee \\
&\qquad (\kappa_2 = 19 \wedge v = 3 \wedge 32 * (128 * d) \leqslant 6400 + (\kappa_0 + 20 * \kappa_1) * 8 < 32 * (128 * d + 1)))))\Big)
\end{aligned}
$$

Figure A.9: Presburger formula describing compulsory misses in loop nest $\mathbb{L}_{\mathrm{mm}}$ for cache set 0.

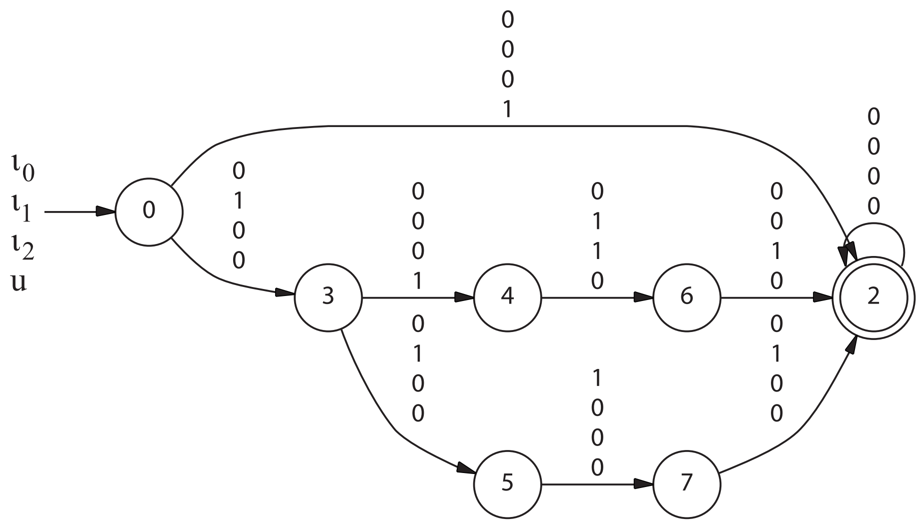


Figure A.10: DFA recognizing the solutions of the example Presburger formula in Figure A.9.

$$
\begin{aligned}
&0 \leqslant \iota_0, \iota_1, \iota_2 < 16 \wedge \\
&\Big(\exists d : (\exists o, o_7, o_6, o_5, o_4, o_3, o_2, o_1, o_0 : 0 \leqslant o_7, o_6, o_5, o_4, o_3, o_2, o_1, o_0 \leqslant 1 \wedge \\
&\quad o \geqslant 0 \wedge o = o_0 + 2 * o_1 + 4 * o_2 + 8 * o_3 + 16 * o_4 + 32 * o_5 + 64 * o_6 + 128 * o_7 \wedge \\
&\quad ((\iota_2 = 0 \wedge u = 0 \wedge \iota_0 = 8 * o_7 + 4 * o_5 + 2 * o_3 + o_2 \wedge \iota_1 = 8 * o_6 + 4 * o_4 + 2 * o_1 + o_0 \wedge \\
&\quad 32 * (128 * d) \leqslant 6400 + o * 8 < 32 * (128 * d + 1)) \vee \\
&\quad (u = 1 \wedge \iota_0 = 8 * o_7 + 4 * o_5 + 2 * o_3 + o_2 \wedge \iota_2 = 8 * o_6 + 4 * o_4 + 2 * o_1 + o_0 \wedge \\
&\quad 32 * (128 * d) \leqslant o * 8 < 32 * (128 * d + 1)) \vee \\
&\quad (u = 2 \wedge \iota_2 = 8 * o_7 + 4 * o_5 + 2 * o_3 + o_2 \wedge \iota_1 = 8 * o_6 + 4 * o_4 + 2 * o_1 + o_0 \wedge \\
&\quad 32 * (128 * d) \leqslant 3200 + o * 8 < 32 * (128 * d + 1)) \vee \\
&\quad (\iota_2 = 19 \wedge u = 3 \wedge \iota_0 = 8 * o_7 + 4 * o_5 + 2 * o_3 + o_2 \wedge \iota_1 = 8 * o_6 + 4 * o_4 + 2 * o_1 + o_0 \wedge \\
&\quad 32 * (128 * d) \leqslant 6400 + o * 8 < 32 * (128 * d + 1)))) \wedge \\
&\quad \big(\exists \kappa_0, \kappa_1, \kappa_2, v : 0 \leqslant \kappa_0, \kappa_1, \kappa_2 < 20 \wedge (\kappa_0 < \iota_0 \vee (\kappa_0 = \iota_0 \wedge \kappa_1 < \iota_1) \vee \\
&\qquad (\kappa_0 = \iota_0 \wedge \kappa_1 = \iota_1 \wedge \kappa_2 < \iota_2) \vee (\kappa_0 = \iota_0 \wedge \kappa_1 = \iota_1 \wedge \kappa_2 = \iota_2 \wedge v < u)) \wedge \\
&\qquad (\exists o, o_7, o_6, o_5, o_4, o_3, o_2, o_1, o_0 : 0 \leqslant o_7, o_6, o_5, o_4, o_3, o_2, o_1, o_0 \leqslant 1 \wedge \\
&\qquad o \geqslant 0 \wedge o = o_0 + 2 * o_1 + 4 * o_2 + 8 * o_3 + 16 * o_4 + 32 * o_5 + 64 * o_6 + 128 * o_7 \wedge \\
&\qquad ((\kappa_2 = 0 \wedge v = 0 \wedge \kappa_0 = 8 * o_7 + 4 * o_5 + 2 * o_3 + o_2 \wedge \kappa_1 = 8 * o_6 + 4 * o_4 + 2 * o_1 + o_0 \wedge \\
&\qquad 32 * (128 * e) \leqslant 6400 + o * 8 < 32 * (128 * e + 1)) \vee \\
&\qquad (v = 1 \wedge \kappa_0 = 8 * o_7 + 4 * o_5 + 2 * o_3 + o_2 \wedge \kappa_2 = 8 * o_6 + 4 * o_4 + 2 * o_1 + o_0 \wedge \\
&\qquad 32 * (128 * e) \leqslant o * 8 < 32 * (128 * e + 1)) \vee \\
&\qquad (v = 2 \wedge \kappa_2 = 8 * o_7 + 4 * o_5 + 2 * o_3 + o_2 \wedge \kappa_1 = 8 * o_6 + 4 * o_4 + 2 * o_1 + o_0 \wedge \\
&\qquad 32 * (128 * e) \leqslant 3200 + o * 8 < 32 * (128 * e + 1)) \vee \\
&\qquad (\kappa_2 = 19 \wedge v = 3 \wedge \kappa_0 = 8 * o_7 + 4 * o_5 + 2 * o_3 + o_2 \wedge \kappa_1 = 8 * o_6 + 4 * o_4 + 2 * o_1 + o_0 \wedge \\
&\qquad 32 * (128 * e) \leqslant 6400 + o * 8 < 32 * (128 * e + 1)))) \wedge \\
&\qquad (\neg(\exists \rho_0, \rho_1, \rho_2, w : 0 \leqslant \rho_0, \rho_1, \rho_2 < 20 \wedge (\rho_0 < \iota_0 \vee (\rho_0 = \iota_0 \wedge \rho_1 < \iota_1) \vee \\
&\qquad\quad (\rho_0 = \iota_0 \wedge \rho_1 = \iota_1 \wedge \rho_2 < \iota_2) \vee (\rho_0 = \iota_0 \wedge \rho_1 = \iota_1 \wedge \rho_2 = \iota_2 \wedge w < u)) \wedge \\
&\qquad\quad (\kappa_0 < \rho_0 \vee (\kappa_0 = \rho_0 \wedge \kappa_1 < \rho_1) \vee (\kappa_0 = \rho_0 \wedge \kappa_1 = \rho_1 \wedge \kappa_2 < \rho_2) \vee \\
&\qquad\quad (\kappa_0 = \rho_0 \wedge \kappa_1 = \rho_1 \wedge \kappa_2 = \rho_2 \wedge v < w)) \wedge \\
&\qquad\quad (\exists o, o_7, o_6, o_5, o_4, o_3, o_2, o_1, o_0 : 0 \leqslant o_7, o_6, o_5, o_4, o_3, o_2, o_1, o_0 \leqslant 1 \wedge \\
&\qquad\quad o \geqslant 0 \wedge o = o_0 + 2 * o_1 + 4 * o_2 + 8 * o_3 + 16 * o_4 + 32 * o_5 + 64 * o_6 + 128 * o_7 \wedge \\
&\qquad\quad ((\rho_2 = 0 \wedge w = 0 \wedge \rho_0 = 8 * o_7 + 4 * o_5 + 2 * o_3 + o_2 \wedge \rho_1 = 8 * o_6 + 4 * o_4 + 2 * o_1 + o_0 \wedge \\
&\qquad\quad 32 * (128 * d) \leqslant 6400 + o * 8 < 32 * (128 * d + 1)) \vee \\
&\qquad\quad (w = 1 \wedge \rho_0 = 8 * o_7 + 4 * o_5 + 2 * o_3 + o_2 \wedge \rho_2 = 8 * o_6 + 4 * o_4 + 2 * o_1 + o_0 \wedge \\
&\qquad\quad 32 * (128 * d) \leqslant o * 8 < 32 * (128 * d + 1)) \vee \\
&\qquad\quad (w = 2 \wedge \rho_2 = 8 * o_7 + 4 * o_5 + 2 * o_3 + o_2 \wedge \rho_1 = 8 * o_6 + 4 * o_4 + 2 * o_1 + o_0 \wedge \\
&\qquad\quad 32 * (128 * d) \leqslant 3200 + o * 8 < 32 * (128 * d + 1)) \vee \\
&\qquad\quad (\rho_2 = 19 \wedge w = 3 \wedge \rho_0 = 8 * o_7 + 4 * o_5 + 2 * o_3 + o_2 \wedge \rho_1 = 8 * o_6 + 4 * o_4 + 2 * o_1 + o_0 \wedge \\
&\qquad\quad 32 * (128 * d) \leqslant 6400 + o * 8 < 32 * (128 * d + 1))))))\big) \wedge \\
&\quad \neg(d = e)\Big)
\end{aligned}
$$

Figure A.11: Presburger formula describing the i-witnesses of loop nest $\mathbb{L}_{\text{mm}}$ for cache set 0, where all arrays are laid out according to the $(4, 4)$-interleaving $\sigma = 01010011$.

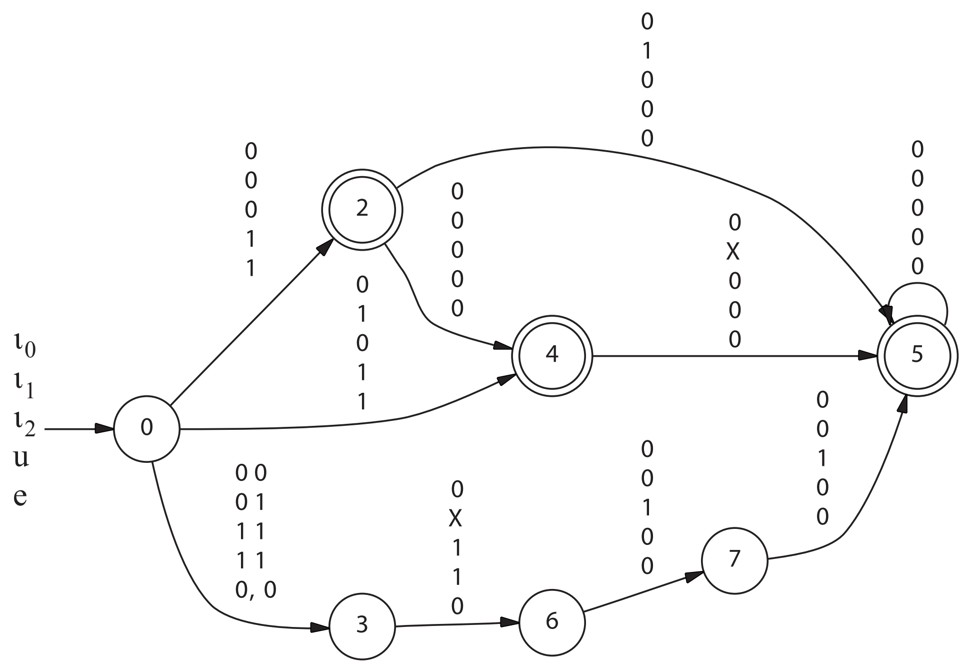


Figure A.12: DFA recognizing the solutions of the example Presburger formula in Figure A.11.

BIBLIOGRAPHY

[1] Anant Agarwal, Mark Horowitz, and John Hennessy. An analytical cache model. *IEEE Transactions on Computers*, 7(2):184–215, May 1989. 131

[2] Nawaaz Ahmed. *Locality Enhancement of Imperfectly-nested Loop Nests*. PhD thesis, Department of Computer Science, Cornell University, August 2000. 22

[3] Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. Tiling imperfectly-nested loop nests. Technical Report TR2000-1782, Department of Computer Science, Cornell University, 2000. 22

[4] Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Cache behavior prediction by abstract interpretation. In Radhia Cousot and David A. Schmidt, editors, *Proceedings of the Third International Static Analysis Symposium*, volume 1145 of *Lecture Notes in Computer Science*, pages 51–66, Aachen, Germany, September 1996. Springer. 132

[5] Tod Amon, Gaetano Borriello, Taokuan Hu, and Jiwen Liu. Symbolic timing verification of timing diagrams using Presburger formulas. In *Proceedings of the 34th Conference on Design Automation*, pages 226–231, Anaheim, CA, June 1997. ACM Press. 129

[6] Tod Amon, Gaetano Borriello, and Jiwen Liu. Making complex timing relationships readable: Presburger formula simplification using don't cares. In *Proceedings of the 35th Conference on Design Automation*, pages 586–590, San Francisco, CA, June 1998. ACM Press. 129

[7] David F. Bacon, Jyh-Herng Chow, Dz ching R. Ju, Kalyan Muthukumar, and Vivek Sarkar. A compiler framework for restructuring data declarations to enhance cache and TLB effectiveness. In *Proceedings of the 1994 Conference of the IBM Centre for Advanced Studies on Collaborative Research*, Toronto, Canada, October-November 1994. IBM Press. 4, 5, 6

[8] Jean-Loup Baer and Wen-Hann Wang. On the inclusion properties for multi-level cache hierarchies. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 73–80, Honolulu, HI, May-June 1988. IEEE Computer Society Press. 2

[9] Constantinos Bartzis and Tevfik Bultan. Automata-based representations for arithmetic constraints in automated verification. In Jean-Marc Champarnaud and Denis Maurel,

editors, *Proceedings of the 7th International Conference on Implementation and Application of Automata*, volume 2608 of *Lecture Notes in Computer Science*, pages 282–288, Tours, France, July 2002. Springer. 26, 54, 58, 64, 81, 125, 139, 140

[10] Constantinos Bartzis and Tevfik Bultan. Efficient symbolic representations for arithmetic constraints in verification. *International Journal of Foundations of Computer Science*, 14(4):605–624, August 2003. 26, 54, 64, 81, 125, 139, 140

[11] Alexander I. Barvinok. A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. *Mathematics of Operations Research*, 19(4):769–779, November 1994. 130

[12] Cédric Bastoul and Paul Feautrier. Improving data locality by chunking. In Görel Hedin, editor, *Proceedings of CC 2003 International Conference on Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 320–335, Warsaw, Poland, April 2003. Springer. 131, 132

[13] Laszlo Belady. A study of replacement algorithms for a virtual storage computer. *IBM Systems Journal*, 5(2):78–101, 1966. 18, 30

[14] Kristof Beyls. *Software Methods to Improve Data Locality and Cache Behavior*. PhD thesis, Department of Electronics and Information Systems, Ghent University, 2004. 128, 129, 131, 132

[15] Kristof Beyls and Eric D'Hollander. Reuse distance-based cache hint selection. In Burkhard Monien and Rainer Feldmann, editors, *Proceedings of the Eighth International European Conference on Parallel Processing*, volume 2400 of *Lecture Notes in Computer Science*, pages 265–274, Paderborn, Germany, August 2002. Springer. 128, 131, 132

[16] Kristof Beyls and Erik D'Hollander. Reuse distance as a metric for cache behavior. In T. Gonzalez, editor, *Proceedings of the 2001 International Association of Science and Technology for Development (IASTED) International Conference on Parallel and Distributed Computing and Systems*, pages 617–622, Anaheim, CA, August 2001. IASTED. 128, 131, 132

[17] Kristof Beyls and Erik D'Hollander. Compile-time cache hint generation for EPIC architectures. In *Proceedings of the 2nd workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Techniques*, Istanbul, Turkey, November 2002. 128, 131, 132

[18] Kristof Beyls and Erik D'Hollander. Platform-independent cache optimization by pinpointing low-locality reuse. In Marian Bubak, Geert Dick van Albada, Peter M. A. Sloot, and Jack Dongarra, editors, *Proceedings of the 4th International Conference on Computational Science*, volume 3038, pages 448–455, Kraków, Poland, June 2004. Springer. 128, 131, 132

[19] Bernard Boigelot and Louis Latour. Counting the solutions of Presburger equations without enumerating them. In Bruce W. Watson and Derick Wood, editors, *Proceedings of the 6th International Conference on Implementation and Application of Automata*, volume 2494 of *Lecture Notes in Computer Science*, pages 40–51, Pretoria, South Africa, July 2001. Springer. 130

[20] Bernard Boigelot and Louis Latour. Counting the solutions of Presburger equations without enumerating them. *Theoretical Computer Science*, 313(1):17–29, February 2004. 130

[21] Bernard Boigelot, Louis Latour, and Axel Legay. The Liège Automata-based Symbolic Handler (LASH). `http://www.montefiore.ulg.ac.be/~boigelot/research/lash/`. 131

[22] Bernard Boigelot and Pierre Wolper. Representing arithmetic constraints with finite automata: An overview. In Peter J. Stuckey, editor, *Proceedings of the 18th International Conference on Logic Programming*, volume 2401 of *Lecture Notes in Computer Science*, pages 1–19, Copenhagen, Denmark, July-August 2002. Springer. 81

[23] Alexandre Boudet and Hubert Comon. Diophantine equations, Presburger arithmetic and finite automata. In Hélène Kirchner, editor, *Proceedings of the 21st International Colloquium on Trees in Algebra and Programming*, volume 1059 of *Lecture Notes in Computer Science*, pages 30–43, Linköping, Sweden, April 1996. Springer. 26, 27, 53, 81

[24] Mark Brehob and Richard Enbody. A mathematical model of locality and caching. Technical Report TR-MSU-CPS-96-TBD, Michigan State University, November 1996. 131

[25] Shirley Browne, Jack Dongarra, N. Garner, Kevin S. London, and Philip Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, Dallas, TX, November 2000. IEEE Computer Society Press. 6

[26] Julius Richard Büchi. Weak second-order arithmetic and finite automata. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 6:66–92, 1960. 26

[27] Julius Richard Büchi. On a decision method in restricted second-order arithmetic. In E. Nagel, P. Suppes, and A. Tarski, editors, *Proceedings of the International Congress on Logic, Methodology and Philosophy of Science 1960*, pages 1–11, Stanford, CA, 1962. Stanford University Press. 26

[28] Arthur W. Burks, Hermann H. Goldstine, and John von Neumann. Preliminary discussion of the logical design of an electronic computing instrument. Report to the U.S.

Army Ordnance Department, 1946. Also in A.H. Taub, ed., 1963: Collected Works of John von Neumann. Macmillan, 5: 34-79. 2

[29] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. Compiler optimizations for improving data locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 28(11) of *SIGOPS Operating System Review*, pages 252–262, San Jose, CA, October 1994. ACM Press. 115

[30] Călin Caşcaval. *Compile-Time Performance Prediction of Scientific Programs*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2000. 128, 129, 132

[31] Călin Caşcaval. Estimating cache misses and locality using stack distances. In *Proceedings of the 2003 ACM International Conference on Supercomputing*, pages 150–159, San Francisco, CA, June 2003. ACM Press. 6, 128, 129, 132

[32] Siddhartha Chatterjee, Vibhor V. Jain, Alvin R. Lebeck, Shyam Mundhra, and Mithuna Thottethodi. Nonlinear array layouts for hierarchical memory systems. In *Proceedings of the 1999 ACM International Conference on Supercomputing*, pages 444–453, Rhodes, Greece, June 1999. ACM Press. 4, 83

[33] Siddhartha Chatterjee, Alvin R. Lebeck, Praveen K. Patnala, and Mithuna Thottethodi. Recursive array layouts and fast parallel matrix multiplication. In *Proceedings of the Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 222–231, Saint-Malo, France, June 1999. ACM Press. 4, 83

[34] Siddhartha Chatterjee, Erin Parker, Phil J. Hanlon, and Alvin R. Lebeck. Exact analysis of the cache behavior of nested loops. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 286–297, Snowbird, UT, June 2001. ACM Press. 6, 129

[35] Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-conscious structure definition. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–24, Altanta, GA, May 1999. ACM Press. 4

[36] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, Altanta, GA, May 1999. ACM Press. 4

[37] Michal Cierniak and Wei Li. Unifying data and control transformations for distributed shared memory machines. In *Proceedings of the 1995 ACM SIGPLAN Conference on*

Programming Language Design and Implementation, pages 205–217, La Jolla, CA, June 1995. ACM Press. 23

[38] Philippe Clauss. Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: Applications to analyze and transform scientific programs. In *Proceedings of the 1996 ACM International Conference on Supercomputing*, pages 278–285, Philadelphia, PA, May 1996. ACM Press. 53

[39] Philippe Clauss. Handling memory cache policy with integer points countings. In Christian Lengauer, Martin Griebl, and Sergei Gorlatch, editors, *Proceedings of the Third International European Conference on Parallel Processing*, volume 1300 of *Lecture Notes in Computer Science*, pages 285–293, Passau, Germany, August 1997. Springer. 129

[40] Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of the 1995 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 279–290, La Jolla, CA, June 1995. ACM Press. 4, 115

[41] D. C. Cooper. Theorem proving in arithmetic without multiplication. In Bernard Meltzer and Donald Michie, editors, *Proceedings of the Seventh Annual Machine Intelligence Workshop*, volume 7 of *Machine Intelligence*, pages 91–99, Edinburgh, Scotland, 1972. Edinburgh University Press. 25

[42] Jesús A. De Loera, Raymond Hemmecke, Jeremiah Tauzer, and Ruriko Yoshida. Effective lattice point counting in rational convex polytopes. `http://www.math.ucdavis.edu/~latte/pdf/lattE.pdf`, March 2003. 130

[43] Luiz De Rose, Kattamuri Ekanadham, Jeffrey K. Hollingsworth, and Simone Sbaraglia. SIGMA: A simulator infrastructure to guide memory analysis. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–13, Baltimore, MD, November 2002. ACM Press. 6

[44] Martin Dyer and Ravi Kannan. On Barvinok's algorithm for counting lattic points in fixed dimension. *Mathematics of Operations Research*, 22(3):545–549, August 1997. 130

[45] Eugène Ehrhart. *Polynômes arithmétiques et Méthode des Polyédres en Combinatoire*, volume 35 of *International Series of Numerical Mathematics*. Birkhäuser Verlag, Basel/Stuttgart, 1977. 130

[46] Jan Elder and Mark D. Hill. Dinero IV: Trace-driven uniprocessor cache simulator. `http://www.cs.wisc.edu/~markhill/DineroIV`, 1998. 6

[47] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–54, 1991. 10, 21, 23

[48] Christian Ferdinand and Reinhard Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems: The International Journal of Time-Critical Computing Systems*, 17(2/3):131–181, 1999. 132

[49] Jeanne Ferrante and Charles Rackoff. The computational complexity of logical theories. *Lecture Notes in Mathematics*, 718, 1979. 25

[50] Jeanne Ferrante, Vivek Sarkar, and Wendy Thrash. On estimating and enhancing cache effectiveness. In Uptal Banerjee et al., editors, *Proceedings of the Fourth Annual Workshop on Programming Languages and Compilers for Parallel Computing*, volume 589 of *Lecture Notes in Computer Science*, pages 328–343, Santa Clara, CA, August 1991. Springer. 129

[51] Tiago C. Ferreto, Luiz De Rose, and César A. F. De Rose. A hardware counters based tool for system monitoring. In Harald Kosch, László Böszörményi, and Hermann Hellwagner, editors, *Proceedings of the Ninth International European Conference on Parallel Processing*, volume 2790 of *Lecture Notes in Computer Science*, pages 7–16, Klagenfurt, Austria, August 2003. Springer. 6

[52] Michael J. Fischer and Michael O.Rabin. Super-exponential complexity of Presburger arithmetic. In R. M. Karp, editor, *Complexity of Computation*, volume 7 of *SIAM-AMS Symposium in Applied Mathematics Proceedings*, pages 27–41, 1974. 25

[53] International Technology Roadmap for Semiconductors. Executive summary, 2003 edition. `http://public.itrs.net/Files/2003ITRS/Home2003.htm`. 2

[54] Basilio B. Fraguela, Ramon Doallo, Juan Touriño, and Emilio L. Zapata. A compiler tool to predict memory hierarchy performance of scientific codes. *Parallel Computing*, 30(2):225–248, February 2004. 6, 128

[55] Basilio B. Fraguela, Ramon Doallo, and Emilio L. Zapata. Automatic analytic modeling for the estimation of cache misses. In *Proceedings of the 8th International Conference on Parallel Architectures and Compilation Techniques*, pages 221–231, Newport Beach, CA, October 1999. IEEE Computer Society Press. 128

[56] 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. 128

[57] Jeremy D. Frens and David S. Wise. Auto-blocking matrix-multiplication or tracking BLAS3 performance with source code. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, volume 32(7) of *SIGPLAN Notices*, pages 206–216, Las Vegas, NV, June 1997. ACM Press. 4, 83

[58] Christine Fricker, Olivier Temam, and William Jalby. Influence of cross-interference on blocked loops: A case study with matrix-vector multiply. *ACM Transactions on Programming Languages and Systems*, 17(4):561–575, July 1995. 5

[59] Somnath Ghosh. *Cache Miss Equations: Compiler analysis framework for tuning memory behavior.* PhD thesis, Department of Electrical Engineering, Princeton University, November 1999. 127, 132, 138

[60] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: An analytical representation of cache misses. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, pages 317–324, Vienna, Austria, July 1997. ACM Press. 127, 132, 138

[61] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 32(5) of *SIGOPS Operating System Review*, pages 228–239, San Jose, CA, October 1998. ACM Press. 127, 132, 138

[62] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: A compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, July 1999. 6, 22, 127, 132, 138

[63] Antonio González, Mateo Valero, Nigel P. Topham, and Joan-Manuel Parcerisa. Eliminating cache conflict misses through XOR-based placement functions. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, pages 76–83, Vienna, Austria, July 1997. ACM Press. 15

[64] Jim Handy. *The Cache Memory Book.* Academic Press, San Diego, CA, 2nd edition, 1998. 13, 18

[65] John. S. Harper, Darren J. Kerbyson, and Graham R. Nudd. Analytical modeling of set-associative cache behavior. *IEEE Transactions on Computers*, 48(10):1009–1024, October 1999. 128

[66] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quanitative Approach.* Morgan Kaufmann, San Francisco, CA, 3rd edition, 2002. 1, 2, 3, 4, 13, 14, 20

[67] Mark D. Hill and Alan J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, December 1989. 13, 19, 30, 35, 39, 76, 128

[68] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, MA, 1979. 26, 58

[69] 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, November 1999. 141

[70] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, Seattle, WA, June 1990. IEEE Computer Society Press. 5, 141

[71] Norman P. Jouppi and Steven J. E. Wilton. Tradeoffs in two-level on-chip caching. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 34–45, Chicago, IL, April 1994. IEEE Computer Society Press. 2

[72] 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, Göteborg, Sweden, June-July 2001. IEEE Computer Society Press. 141

[73] Stefanos Kaxiras, Zhigang Hu, Girija J. Narlikar, and Rae McLellan. Cache-line decay: A mechanism to reduce cache leakage power. In Babak Falsafi and T. N. Vijaykumar, editors, *Proceedgins of the First International Workshop on Power-Aware Computer Systems*, number 2008 in Lecture Notes in Computer Science, pages 82–96, Cambridge, MA, November 2000. Springer. 141

[74] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. *The Omega Calculator and Library, Version 1.1.0*, November 1996. `http://www.cs.umd.edu/projects/omega`. 81

[75] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. *The Omega Library Version 1.1.0 Interface Guide*, November 1996. `http://www.cs.umd.edu/projects/omega`. 81, 125, 139, 140

[76] Wayne Kelly and William Pugh. A framework for unifying reordering transformations. Technical Report CS-TR-3193, Department of Computer Science, University of Maryland, April 1993. 4, 22

[77] Wayne Kelly and William Pugh. Finding legal reordering transformations using mappings. Technical Report CS-TR-3297, Department of Computer Science, University of Maryland, June 1994. 22

[78] Richard E. Kessler and Mark D. Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems*, 10(4):338–359, November 1992. 13

[79] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. One level storage system. *IRE Transactions on Electronic Computers*, 12(2):223–235, April 1962. 2

[80] Felix Klaedtke. On the automata size for Presburger arithmetic. In *Nineteenth Annual IEEE Symposium on Logic in Computer Science*, page TBD, Turku, Finland, July 2004. 26, 27, 81

[81] Nils Klarlund and Anders Møller. *MONA Version 1.4 User Manual*, January 2001. `http://www.brics.dk/mona`. 64, 81, 125, 139, 140

[82] Georg Kreisel and Jean Louis Krivine. *Elements of Mathematical Logic*. North-Holland, Paris, 1967. 25

[83] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 81–88, Minneapolis, MN, May 1981. IEEE Computer Society Press. 5

[84] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 25 of *SIGOPS Operating System Review*, pages 63–74, Santa Clara, CA, April 1991. ACM Press. Special Issue. 4, 5, 115

[85] Alvin R. Lebeck and David A. Wood. Cache profiling and the SPEC benchmarks: A case study. *IEEE Computer*, 27(10):15–26, October 1994. 6

[86] Amy W. Lim and Monica S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 201–214, Paris, France, January 1997. ACM Press. 22

[87] J. S. Liptay. Structural aspects of the System/360 Model 85, part ii: The cache. *IBM Systems Journal*, 7(1):15–21, 1968. 2

[88] Yanhong A. Liu and Scott D. Stoller. Loop optimization for aggregate array computations. In *Proceedings of the 1998 IEEE International Conference on Computer Languages*, pages 262–271, Chicago, IL, May 1998. IEEE Computer Society Press. 120

[89] Yanhong A. Liu, Scott D. Stoller, Ning Li, and Tom Rothamel. Optimizing aggregate array computations in loops. *ACM Transactions on Programming Languages and Systems*. To appear. 120

[90] Vincent Loechner. *PolyLib: A Library for Manipulating Parameterized Polyhedra*, March 1999. 130

[91] Margaret Martonosi, Anoop Gupta, and Thomas E. Anderson. MemSpy: Analyzing memory system bottlenecks in programs. In *Proceedings of the 1992 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, volume 20 of *Performance Evaluation Review*, pages 1–12, Newport, RI, June 1992. ACM Press. 6

[92] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970. 71

[93] Nicholas Mitchell, Larry Carter, Jeanne Ferrante, and Karin Högstedt. Quantifying the multi-level nature of tiling interactions. In Zhiyuan Li, Pen-Chung Yew, Siddhartha Chatterjee, Chua-Huang Huang, P. Sadayappan, and David C. Sehr, editors, *Proceedings of the Tenth Annual Workshop on Programming Languages and Compilers for Parallel Computing*, volume 1366 of *Lecture Notes in Computer Science*, pages 1–15, Minneapolis, MN, August 1998. Springer. 4

[94] Todd C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Computer Systems Laboratory, Stanford University, March 1994. 5, 141

[95] Todd C. Mowry. Tolerating latency in multiprocessors through compiler-inserted prefetching. *ACM Transactions on Computer Systems*, 16(1):55–92, February 1998. 5, 141

[96] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 26 of *SIGOPS Operating System Review*, pages 62–73, Boston, MA, October 1992. ACM Press. Special Issue. 5, 141

[97] Frank Mueller. Timing analysis for instruction caches. *Real-Time Systems: The International Journal of Time-Critical Computing Systems*, 18(2/3):209–239, May 2000. 132

[98] Frank Mueller, David B. Whalley, and Marion Harmon. Predicting instruction cache behavior. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, Orlando, FL, June 1994. ACM Press. 132

[99] Derek C. Oppen. A $2^{2^{2^{pn}}}$ upper bound on the complexity of Presburger arithmetic. *Journal of Computer and System Sciences*, 16(3):323–332, July 1978. 25

[100] Erin Parker and Siddhartha Chatterjee. An automata-theoretic algorithm for counting solutions to Presburger formulas. In Evelyn Duesterwald, editor, *Proceedings of CC 2004 International Conference on Compiler Construction*, volume 2985 of *Lecture Notes in Computer Science*, pages 104–119, Barcelona, Spain, March-April 2004. Springer. 129, 131

[101] Allan Kennedy Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Department of Computer Science, Rice University, Houston, TX, May 1989. Available as technical report CRPC-TR89009. 4, 5, 141

[102] Mojżesz Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Sprawozdanie z I Kongresu matematyków krajów slowiańskich, Warszawa 1929 (Comptes-rendus du I Congrès des Mathématiciens des Pays Slaves)*, pages 92–101, Warsaw, Poland, 1930. 7, 25, 165

[103] Mojżesz Presburger. On the completeness of a certain system of arithmetic of whole numbers in which addition occurs as the only operation. *History and Philosophy of Logic*, 12(2):225–233, 1991. English translation of the article [102] by Dale Jacquette. 7, 25

[104] Betty Prince. *High Performance Memories: New Architecture DRAMs and SRAMs—Evolution and Function*. John Wiley, Chichester, NY, 1996. 13

[105] Steven A. Przybylski. *Cache and Memory Hierarchy Design: A Performance-Directed Approach*. Morgan Kaufmann, San Mateo, CA, 1990. 13

[106] William Pugh. Counting solutions to Presburger formulas: How and why. In *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 121–134, Orlando, FL, June 1994. ACM Press. 53, 130

[107] C. R. Reddy and Donald W. Loveland. Presburger arithmetic with bounded quantifier alternation. In *Conference Record of the Tenth Annual ACM Symposium on Theory of Computing*, pages 320–325, San Diego, CA, May 1978. ACM Press. 25

[108] Gabriel Rivera and Chau-Wen Tseng. Data transformations for eliminating conflict misses. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 38–49, Montreal, Canada, June 1998. ACM Press. 4, 5, 6

[109] Gabriel Rivera and Chau-Wen Tseng. Eliminating conflict misses for high performance architectures. In *Proceedings of the 1998 ACM International Conference on Supercomputing*, pages 353–360, Melbourne, Australia, July 1998. ACM Press. 4

[110] Anne Rogers and Kai Li. Software support for speculative loads. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 26 of *SIGOPS Operating System Review*, pages 38–50, Boston, MA, October 1992. ACM Press. Special Issue. 5

[111] Uwe Schöning. Complexity of Presburger arithmetic with fixed quantifier dimension. *Theory of Computing Systems*, 30(4):423428, July 1997. 25

[112] André Seznec. A case for two-way skewed-associative caches. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 169–178, San Diego, CA, May 1993. IEEE Computer Society Press. 15

[113] André Seznec and François Bodin. Skewed-associative caches. In Arndt Bode, Mike Reeve, and Gottfried Wolf, editors, *Proceedings of the 5th International PARLE Conference, Parallel Architectures and Languages Europe*, volume 694 of *Lecture Notes in Computer Science*, pages 304–316, Munich, Germany, June 1993. Springer. 15

[114] Alan J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982. 4, 5, 13

[115] Standard Performance Evaluation Corporation. SPECint95 Benchmark/SPECfp95 Benchmark, August 1995. 101

[116] The Stanford Compiler Group. *SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers.* `http://suif.stanford.edu`. 80

[117] Ryan Stansifer. Presburger's article on integer arithmetic: Remarks and translation. Technical Report TR84-639, Department of Computer Science, Cornell University, 1984. 7, 25

[118] 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 the Measurement and Modeling of Computer Systems*, volume 21 of *Performance Evaluation Review*, pages 24–35, Santa Clara, CA, May 1993. ACM Press. 30, 35

[119] Olivier Temam, Elana D. Granston, and William Jalby. To copy or not to copy: a compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 410–419, Portland, OR, November 1993. ACM Press. 4

[120] Dominique Thiebaut and Harold S. Stone. Footprints in the cache. *IEEE Transactions on Computers*, 5(4):305–329, November 1987. 131

[121] Xavier Vera, Jaume Abella, Antonio González, and Josep Llosa. Optimizing program locality through CMEs and GAs. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, pages 68–78, New Orleans, LA, October 2003. IEEE Computer Society Press. 128, 132

[122] Xavier Vera, Nerina Bermudo, Josep Llosa, and Antonio González. A fast and accurate framework to analyze and optimize cache memory behavior. *ACM Transactions on Programming Languages and Systems*, 26(2):263–300, March 2004. 6, 128, 132

[123] Xavier Vera, Josep Llosa, Antonio González, and Nerina Bermudo. A fast and accurate approach to analyze cache memory behavior. In Arndt Bode, Thomas Ludwig II, Wolfgang Karl, and Roland Wismüller, editors, *Proceedings of the Sixth International European Conference on Parallel Processing*, volume 1900 of *Lecture Notes in Computer Science*, pages 194–198, Munich, Germany, August 2000. Springer. 128, 132

[124] Xavier Vera and Jingling Xue. Let's study whole-program cache behaviour analytically. In *Proceedings of the Eighth International Symposium on High Performance Computer Architecture*, pages 175–186, Boston, MA, February 2002. IEEE Computer Society Press. 128, 132

[125] Sven Verdoolaege, Kristof Beyls, Maurice Bruynooghe, Rachid Seghir, and Vincent Loechner. Analytical computation of Ehrhart polynomials and its applications for embedded systems. In *Proceedings of the 2nd Workshop on Optimizations for DSP and Embedded Systems*, Palo Alto, CA, March 2004. 53, 129, 130

[126] Wen-Hann Wang, Jean-Loup Baer, and Henry M. Levy. Organization and performance of a two-level virtual-real cache hierarchy. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 140–148, Jerusalem, Israel, June 1989. IEEE Computer Society Press. 2

[127] Dee A. B. Weikle. *Caches As Filters: A Framework for the Analysis of Caching Systems.* PhD thesis, Department of Computer Science, University of Virginia, May 2001. 131

[128] Dee A. B. Weikle, Sally A. McKee, and William A. Wulf. Caches as filters: A new approach to cache analysis. In *Proceedings of MASCOTS'98, Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 2–12, Montreal, Canada, July 1998. IEEE Computer Society Press. 131

[129] Volker Weispfenning. Complexity and uniformity of elimination in Presburger arithmetic. In *Proceedings of the 1997 ACM SIGSAM International Symposium on Symbolic and Algebraic Computation*, pages 48–53, Maui, HI, July 1997. ACM Press. 25

[130] Maurice V. Wilkes. Slave memories and dynamic storage allocation. *IEEE Transactions on Electronic Computers*, pages 270–271, April 1965. 2

[131] David S. Wise, Jeremy D. Frens, Yuhong Gu, and Gregory A. Alexander. Language support for Morton-order matrices. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, volume 36(7) of *SIGPLAN Notices*, pages 24–33, Snowbird, UT, June 2001. ACM Press. 4, 83

[132] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of the 1991 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, Canada, June 1991. ACM Press. 4

[133] Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champagian, 1982. Available as technical report 82-1009. 4

[134] Michael J. Wolfe. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 655–664, Reno, NV, November 1989. ACM Press. 4, 115

[135] Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers.* MIT Press, Cambridge, MA, 1989. 4, 17, 21, 99

[136] Pierre Wolper and Bernard Boigelot. An automata-theoretic approach to Presburger arithmetic constraints (extended abstract). In Alan Mycroft, editor, *Proceedings of the Second International Static Analysis Symposium*, volume 983 of *Lecture Notes in Computer Science*, pages 21–32, Glasgow, UK, September 1995. Springer. 81

[137] Pierre Wolper and Bernard Boigelot. On the construction of automata from linear arithmetic constraints. In Susanne Graf and Michael I. Schwartzbach, editors, *Proceedings of the Sixth International Conference on Tools and Algorithms for Construction and Analysis of Systems*, volume 1785 of *Lecture Notes in Computer Science*, pages 1–19, Berlin, Germany, April 2000. Springer. 26, 81

[138] William A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH Computer Architecture News*, 23(1):20–24, March 1995. 1