Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory of Information Processing Science
Espoo 2004 TKO-A41/04

Using Static Program Analysis to Compile Fast Cache Simulators

Vesa Hirvisalo

Dissertation for the degree of Doctor of Science in Technology to be presented with due permission of the Department of Computer Science and Engineering for public examination and debate in Auditorium T2 at Helsinki University of Technology (Espoo, Finland) on the 26th of March, 2004, at 12 noon.

Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory of Information Processing Science

Distribution:
Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory of Information Processing Science
P.O. Box 5400
FIN-02015 HUT
Espoo
Finland

ISBN: 951-22-6994-5
ISSN: 1239-6885

Otamedia Oy
Espoo 2004

Abstract

This thesis presents a generic approach towards compiling fast execution-driven simulators, and applies this to cache simulation of programs. The resulting cache simulation method reduces the time needed for cache performance evaluations without losing the accuracy of the results.

Fast cache simulators are needed in the performance analysis of software systems. To properly understand the cache behavior caused by a program, simulations must be performed with a sufficient number of inputs. Traditional simulation of memory operations of a program can be orders of magnitude slower than the execution of the program. This leads to simulation times that are often infeasible in software development.

The approach of this thesis is based on using static cache analysis to guide partial evaluation and slicing of simulators. Because of redundancy in memory access patterns of typical programs, an execution-driven cache simulator program can be partially evaluated during its compilation. Program slicing can be used to remove the computations that have no effect on the simulation result.

The static cache analysis presented in this thesis is generic. The analysis is designed especially for programs that use dynamic addressing. The thesis assumes an address analysis that gives the cache analysis static information about cache aliases and cache conflicts between accessed memory lines.

To determine the memory references that always cause cache hits or cache misses, the thesis describes both *must* and *may* analyses of cache states. The cache state analysis is built by using abstract interpretation. Based on the use of abstract interpretation, the soundness of the analysis is proved.

The potential performance of the method was experimentally evaluated. The thesis describes both a tool set implementing the cache analysis method and experiments done with the tool set. The experiments indicate that a simple implementation is capable of significantly speeding up the simulations.

Keywords: static analysis, program analysis, performance analysis, cache simulation, program slicing.

Preface

This research is a result of my personal interest on performance issues and compilers. All kinds of gadgets with performance as their main attribute have always fascinated me. A doctoral dissertation is an important milestone, but my path has not been the shortest nor the fastest.

Because of my interest in performance, I began my university studies in physics. While being a freshman, I was introduced into computers by another freshman, Esko Nuutila, who has been my colleague and a close personal friend ever since. Thanks to him, my work with computers started as a hobby.

Long before my university studies, I was aware that there existed computers, that is, machines able to do *some* computations. However, I was not aware that there existed *universal* computers able to do *any* computations. I was amazed to learn about the machines that can modify their own programming by running compilers. So, research on performance and compilers was a choice for me.

The research that lead to this thesis was inspired by David Parnas and Alexander Ran, who taught my way into embedded software. Since then, many others have guided me in my research. These include Connie U. Smith and Murray Woodside from the software performance community and the many members of the compiler and embedded systems groups from Chalmers University, Lund University, and Saarland University.

In addition to research work, I have done a wide range of practical work with embedded systems. Especially, I want to thank Hamish Kellock and Jari Ihattula for the opportunity to learn so much. I also want to thank Mikko Reinikainen and Juha Tukkinen for their support and Niklas Holsti for his encouragement during the last steps of this work.

This work has been funded by Helsinki University of Technology, the Academy of Finland, and the National Technology Agency of Finland. It has been supervised by Professor Eljas Soisalon-Soininen, whom I thank for believing in a man with a different topic.

I thank Johan Lilius and Reinhold Heckmann for their valuable comments as reviewers of this thesis and Ruth Vilmi for helping me with the English language. Last but not least, I thank my colleagues at Helsinki University of Technology for their friendship and my family for their love during my years of study.

Otaniemi, February 2004
Vesa Hirvisalo

Contents

Notation

$\times$	set multiplication
$\cdot$	multiplication
$\vee$	logical or
$\wedge$	logical and
$2^{\{\cdots\}}$	power set
$\rightarrow$	function
$\stackrel{S}{=}$	similar to
$\Rightarrow$	implication or derivation
$\hat{x}$	abstract counterpart of x
$x^{\cap}$	x for *must* analysis
$x^{\cup}$	x for *may* analysis
x^{+}	augmentation for x
x^{A}	x with augmentation
::=	generative grammar rule
|	optional rule or a condition
$\texttt{[}x\texttt{]}^{l}$	program element x labeled l
$[\![P]\!]$	semantics of program P
$[\![P]\!]\langle I\rangle$	output from input
$\cup$	union
$\cup\{\ldots\}$	union of elements of the set $\{\ldots\}$
$\sqcup$	least upper bound
$\sqcup\{\ldots\}$	least upper bound of elements of the set $\{\ldots\}$
$\subseteq$	subset
$\sqsubseteq$	partial order
$\bot$	least element
$\top$	greatest element
$\diamond$	`nil` pointer value or empty cache line
$\emptyset$	an empty set
ϵ	a program element
π	a path (sequence of program elements)
A	cache associativity
abs	abstraction function
addr	address of location
as	abstract semantics
`aug`	augmentation (a piece of instrumentation)
$\texttt{aug}^{x}$	x-specialized augmentation

B	line length
c	a constant or a set state
C	set of set states
CFG	control flow graph
$conc$	concretization function
cs	collecting semantics
D_{as}	domain of abstract semantics
D_{cs}	domain of collecting semantics
e	expression
E	set of edges
f	a cache set
F	cache sets
F_s	static resolution
`fin`	instrumentation finalization
g	analysis function
g_C	partially evaluated analysis function
G	graph
h	age of updated cache line
I	input
I^+	instrumentation input
`init`	instrumentation initialization
J	join function
k_C	cache conflicts
k_S	cache aliases
l	memory location
L	set of memory locations
L_B	basic block length
L_L	loop length
$lin_m(x)$	concrete lines for x in state m
$line(x)$	memory line of x
$lineaddr(t)$	number of memory line
$locs(x)$	concrete locations for x
m	execution state
m^A	augmented execution state
m_f	a final execution state
m_o	an initial execution state
m^+	instrumentation state
M	set of execution states
M^A	set of augmented execution states
N	number of cache sets
N_B	number of basic blocks

N_o	total number of references
N_r	number of resolved references
N_R	number of reference instructions
N_X	number of natural loops
N_Y	number of subroutine calls
op	operation
O	output
O^+	instrumentation output
p	program point
P	a program
P^A	instrumented program
P_x	specialization x of a program
Q_f	the update queue of cache set f
r	a cache line
R	set of cache lines
R_d	speed-up coefficient
r_{ji}	a cache line (set j, age i)
s	the state of a cache set
S	the set of all cache set states
t	a memory line
T	set of memory lines
T_o	original execution time
T_r	reduced execution time
U_C	cache state update function
U_S	set state update function
var	a variable (register)
V	set of vertices or storage cells
V_V	vector of variables
V_C	vector of constants
Y	slicing criterion
Z	the set of integers

Chapter 1

Introduction

This thesis discusses how static program analysis can be used to compile fast cache simulators from programs whose cache behavior is to be analyzed. It discusses both static and dynamic program analysis and presents a framework for combining the two while applying the framework to the cache performance evaluation of programs. We call the framework *combined program analysis*.

Dynamic program analysis is a straightforward approach to program analysis. It is performed by simply executing a subject program with instrumentations that collect the analysis data. Dynamic program analysis is known by several names because of its wide applicability, e.g., execution-based testers and execution-driven simulators are often dynamic program analyzers.

Static program analysis is a sophisticated approach to program analysis. In static analysis, we try to understand the run-time behavior of a program without executing it with a specific input. Static analysis is usually motivated by its ability to simultaneously give results for a set of inputs, often for all inputs of a program. Static analysis is also known by other names; for example, flow analyzers included in compilers are static program analyzers.

For some analysis tasks, static analysis methods are fast, but not sufficiently accurate. On the other hand, dynamic analysis methods are accurate, but slow for some tasks. Our approach combines the benefits of both kinds of analysis, while our framework builds combined simulators by using program specialization. The specialization is based on abstract interpretation, partial evaluation, and program slicing.

As an application of the framework, we present a combined analysis method for the cache performance evaluation of programs. The method is especially suitable for programs using dynamic memory addressing. Such cache performance evaluation is a demanding program analysis task, and thus a good test

of the applicability of our approach.

1.1 Problem

This thesis addresses two problems: we study the problem of using static program analysis to compile fast dynamic program analyzers, which we study in general terms, and the problem of cache performance analysis of programs that use dynamic memory addressing, which we study in more specific terms.

Both dynamic analysis and static analysis have their benefits and their problems. From the point of view of this work, the main problem of static analysis is its approximative nature, which results from considering several inputs at the same time. Instead of giving an accurate answer, static analysis typically gives an upper bound or a lower bound. Different inputs can yield significantly different analysis results. Thus, the bounds can be loose and the analysis too inaccurate for practical purposes.

In dynamic analysis, we use a specific input and the analysis result is accurate. On the other hand, the analysis result is valid only for the single input. To understand the behavior of a program thoroughly, the analysis must be performed for several inputs. The time needed for such a study can be infeasible for practical purposes.

Instead of studying the problem purely on an abstract level, we concentrate on a practical application: cache performance analysis of programs. The performance of computers owes much to the use of cache memories. The speed of main memories has not developed as rapidly as the speed of processors. This performance imbalance is eased by cache memories, which hold frequently needed data so they are rapidly accessible.

Cache memory has become the performance bottleneck for many applications. Therefore, the number of lines (or instructions) executed and the related complexity measures do not have the importance that they used to have. It is important to know the number of cache misses occurring and the reason for those misses.

The crucial role of cache memories makes it hard to understand the performance of programs. The steps executed and the related memory references can be seen from the program code. However, cache misses and the related execution stalls cannot be seen. Computer aided engineering tools are needed to detect and locate them. Such methods and tools that help in designing data structures and the related memory layout are needed.

The traditional cache analysis method is simulation. Simulation is a flexible and accurate method, but sometimes it is slow. Simulation of memory operations of a program can be orders of magnitude slower than execution of the

program [156]. Furthermore, to properly understand the memory behavior of the program, simulation must be performed with a sufficient number of inputs. This leads to simulation times that are often infeasible in software development.

Recently, static cache analysis methods have been developed (see, for example, [120, 166] for a list of methods). For the programs that often use dynamic memory addressing, cache performance is related to the input for the software. Therefore, the number of the memory references potentially causing both cache hits and cache misses can be large compared to the number of memory references that can be guaranteed to cause only misses. This makes performance bottlenecks hard to locate.

1.2 Goals

In the study of combining dynamic analysis and static analysis, our goal has been to formulate a framework that gives a basis and guidelines for designing combined analyzers. We have not aimed at developing an automated method that produces combined analyzers.

Instead, we have tried to make the framework generic and flexible. The framework should have wide applicability to various analysis tasks. On the other hand, this goal of generality forces us to leave many details unspecified.

In the study of cache performance evaluation of programs, we have tried to give a detailed method for solving one problem. Our goal has not been to develop an ultimate solution to the problem. We have aimed at a solution that shows the potential of our approach.

We have concentrated on programs using dynamic addressing. For such programs, static analysis alone is often not accurate enough. On the other hand, simulation is often slow.

1.3 Methodology

This thesis is an engineering thesis that presents basic research into program analysis. It uses the tools of computer science and engineering to solve a problem within the discipline. It addresses a practical engineering problem (compiling fast cache simulators), which it approaches as a special case of a generic problem (combined program analysis).

The research presented is based on several research methods. The thesis proposes an abstract framework as a solution to the generic problem that is built on top of a set of methods previously presented in the literature of the discipline. Correctness of the abstract framework is analytically proved.

Starting from the abstract framework, the thesis builds an analysis method

that is a solution to the practical engineering problem, i.e., compiling fast cache simulators. The correctness of the proposed method is analytically proved. As a specialized research method, abstract interpretation [118] is used in the analytical work. Our use of this is explained in detail in Section 3.4.

The research included constructive work. A prototype tool for compiling cache simulators was implemented on the basis of the theory presented in this thesis. The validity of the implementation was experimentally tested. Using the implemented tool, the performance of the proposed analysis method was experimentally analyzed. The experimentation followed a well-established methodology of experimental performance analysis [81]. Our experimental method is explained in detail in Section 7.1.

1.4 Contributions

Although most of this thesis focuses on cache performance analysis of programs, it makes contributions having applicability in general. The main contributions of this thesis are:

1. It presents and analyzes a program analysis framework that uses static analysis to compile fast dynamic analyzers. The framework is applicable for analyses that are based on augmenting a subject program with instrumentation code. The framework applies partial evaluation and program slicing to speed up dynamic analyzers.

2. It presents a static cache analysis method that makes it possible to analyze programs that use dynamic memory addressing. The method is based on abstract interpretation.

3. It presents an account of how partial evaluation can be used to speed up dynamic cache performance analysis. The partial evaluation presented is based on a modified instrumentation scheme and uses data from static cache analysis.

4. It presents an account of how program slicing can be used to speed up dynamic cache performance analysis. The presented method is based on dependence graphs describing relationships inside instrumentation code.

The first contribution describes the generic framework. Together with contributions 2–4, it forms a concrete method for cache performance analysis of programs [74, 76]. The new method significantly reduces the time needed in cache performance simulations without loss of accuracy from the results.

This thesis also makes some secondary contributions:

5. It describes a tool prototype that uses static analysis to speed up cache simulations. The tool prototype demonstrates one way of implementing our method of cache performance analysis of programs.

6. It describes experiments that evaluate performance of the method for cache performance analysis of programs. The experiments use the related tool prototype.

1.5 Outline of the thesis

The structure of this thesis is as follows: The next chapter (Chapter 2) views the background of the thesis, it discusses problems in cache analysis and summarizes the difficulty of analyzing execution of software on specific hardware.

Chapter 3 describes the framework for combined program analysis. The framework described has generic applicability in program analysis, while not being limited to cache performance analysis. The framework is therefore described in an abstract way.

The next three chapters (Chapters 4–6) describe how the abstract framework can be realized to analyze cache performance. Each of the three chapters discusses a part of the cache analysis method: Chapter 4 discusses dynamic cache analysis, Chapter 5 static cache analysis, and Chapter 6 program specialization.

Chapter 7 discusses our experimentation with the method. It describes a tool that implements the cache analysis method and presents the results of experiments that were performed with that tool.

Cache analysis is one of the main issues in program performance analysis. As it has been studied intensively, there are now a number of methods developed for cache performance analysis of programs. In Chapter 8, we review the work that is related to this thesis.

Chapter 9 concludes the thesis with a discussion of the contributions of this thesis and an attempt to relate these contributions to a wider perspective of program analysis and software performance engineering.

Chapter 2

Background

The main motivation of this chapter is to remind the reader about the complexity of the relation between high-level source code and program execution. Understanding program behavior is a demanding task without proper tools – especially when hardware-related issues are considered. Designing such tools is also complicated.

There are several ways of analyzing programs, and many topics are related to program analysis. These include programming languages, compiler construction, operating systems, memory systems, and performance analysis. All these topics have been intensively studied and presented in the literature. To make the rest of this thesis more understandable, we will now discuss some of the related topics.

Our overview begins with a description of how human-readable programs are transformed into machine-executable images, and how such program images are executed in the hardware. Our hardware description concentrates on the memory subsystem. After that, we discuss how the memory operations affect the performance of a program. Finally, we describe how the run-time behavior can be analyzed and connected to the original program code.

2.1 Programs and references

A program can be a *sequential program* or a *concurrent program* [10]. A sequential program is a program whose outcome for a given input is determined by executing the instructions of the program in their explicit order, i.e., by following the explicit sequentiality and the explicit jumps in the program code. For a *concurrent program* and an input, there is no such single order to determine

the outcome.

The execution of a program can be *a serial execution* or *a parallel execution*. Instructions are executed one after the other in a serial execution. In a parallel execution, there can be multiple instructions executing at one time. There is no need to execute sequential programs strictly according to the execution order implied by the source code. In fact, pipelined and superscalar processors use parallel execution for sequential programs [9, 42, 86, 138].

A static read or write in a program is a *reference*. An execution of the read or write at runtime is an *access*. Because of loops and other structures present in programs, a single reference can access several different memory locations at runtime. We say that a program has *static addressing* if each reference of the program accesses always the same memory location whose address is known before execution of the program. Otherwise, we say that the program has *dynamic addressing*.

A *reference trace* is a list of references in the program code. An *access trace* is the list of the addresses of data accessed by an execution of the program. Reference traces are static. Access traces of program executions are usually dependent on the input given for the program.

There are two kinds of accesses: *instruction fetches* and *data accesses*. For each instruction executed, the instruction itself must be fetched from the memory. During the execution of an instruction, the data for the operation is accessed according to the addressing modes of the arguments. Typically, most of the arguments of an instruction do not refer to memory, but to a small set of internal registers of the processor.

A trace consisting solely of instruction fetches is the *control trace* of a program execution. A trace consisting solely of memory data accesses is the *data trace* of a program execution. Merged together in the proper order, the control trace and the data trace of a program execution are equal to its access trace. The operation of a typical cache memory can be simulated by using a trace; execution of the program itself is often not needed.

In this thesis, we study memory behavior related to the data trace of programs. We consider only sequential programs that are serially executed.

2.2 Program compilation

A program must be compiled, linked, and loaded before it can be executed [140]. Programs are typically written in a high level language. In a *compilation* of a program [5, 11, 162], the program is translated into binary machine code. Compilation is typically done in several phases, which are grouped into two major phases: an analysis phase and a synthesis phase.

The analysis phase is done by the *front end* of a compiler, which typically translates a source program into an *intermediate language*, which is usually machine independent and has simple syntax and clear semantics.

The synthesis phase is done by the *back end* of a compiler, which translates the intermediate representation into machine code. Because of the simplicity of the intermediate language, this is easier than a direct translation from the source language to the machine code.

A compiler does many transformations that alter the memory usage of a compiled program, e.g., register allocation [20] and common subexpression elimination [34]. Thus, subject program operations that seem to access memory actually do not access memory, and there are memory accesses that are not visible in the original program. After a compilation, we have the memory reference operations that will be executed during run time.

Linking a program means joining its compilation units together to form an executable program [109]. Linking is usually separate from compilation. Thus, analyzing inter-unit dependencies during compilation is difficult. All program units typically use the same cache memory. Therefore, the accuracy of a cache analysis can be improved if inter-unit analysis is done.

Loading a program means creating a readily executable image of the program into the memory of a computer [98]. Often, loading is not merely copying; a program can be transformed during loading. This limits the possibilities of analysis during compilation and linking.

Many systems resolve at loading-time the addresses where program text and data will reside. For cache analysis this is important. Cache analysis cannot be performed without any address data. However, as we will see in Chapter 5, it can be performed by using partial address data. There exist compilers that are aware of run-time cache alignments, and perform optimizations that are based on cache structure (see, for example, [25, 35, 93, 115, 127, 132, 161]). Further, real-time systems are often designed to be easier to analyze than common-purpose systems.

2.3 Allocation

Compiled programs usually refer to memory in three ways: static references, stack references, and heap references [5, 154]. These references typically access separate memory areas, memory allocation in those memory areas being done by using different mechanisms. From the view point of cache analysis, allocation mechanisms are important, because they decide addresses that will be used as the basis of cache analyses.

A static reference refers to the same address during the whole execution.

Typically all references to program text and global variables are static. Access addresses of static references are based on symbolic (relocatable) addresses given by the compiler. The run-time addresses are typically assigned by the loader. If the loading mechanism is known, run-time addresses can be resolved from a linked program, i.e., they can be resolved statically.

A stack reference refers to memory by using the *stack pointer* or the *frame pointer* as the base address. The stack pointer points to the top of the control stack of the program execution. The frame pointer points to the frame of the current program component (e.g., subroutine activation record containing local variables). If recursion and dynamically addressed structures in the stack are not allowed, then the possible access addresses of stack references are limited, and can be resolved statically.

A heap reference refers to memory by using an address given by the heap allocator [164] or garbage collector [163] as the base address. Heap references are truly dynamic, i.e., they cannot be statically resolved. However, a heap allocator (and a garbage collector) can have rules that they obey. For example, rules can align allocated heap objects to cache lines of the underlying hardware [24, 61, 121, 132]. Such rules give partial address information that can be used in cache analysis. The analysis that we will present in Chapter 5 assumes such support.

2.4 Memory hardware

Fast memories are significantly smaller than slow ones [69]. Because of this, the memory of computers has become layered. A typical memory hierarchy with four layers is illustrated in Figure 2.1. The topmost layer is the set of registers of the processor. Programs refer to data in registers by using the register address space. Data in other layers are referred to by using the virtual address space of the processor[1].

Virtual address [41] space makes multitasking simple [147]. Before any datum in memory is accessed, its virtual address is translated to a physical address. The address translation to physical addresses is process specific.

Virtual address spaces are usually paged, i.e., they consist of pages of equal size. There is a translation that maps virtual pages onto physical pages. The mapping is usually stored on the page table of an operating system. A page table is large and accessing it is slow. Therefore, the most frequently used entries of a page table are stored in a small intra-processor cache, which is called the Translation Lookaside Buffer (TLB) [30, 152].

[1]There are processors without a proper virtual address space, e.g., the Atmel AVR family [13], and processors, whose registers are in the virtual address space, e.g., Digital PDP-10 [18].

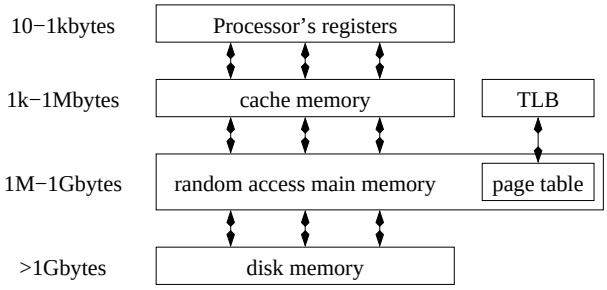


Figure 2.1: Typical layers of memory and their typical sizes.

A physical address needs not uniquely identify the actual location of the addressed datum. In addition to a main memory, there is often cache memory. Cache memory can have several layers. In this thesis, we assume that there is only one layer of cache memory.

Cache memories [128] consist of blocks called *cache lines*, which store frequently used blocks of memory. Cache lines are organized into *cache sets* of equal size. A *memory line* is an aligned block in memory that is of the size of a cache line. Each memory line (and thus also each memory address) is uniquely mapped to a set that can hold its contents. The data transfer between main memory and cache memory consists of reading and writing blocks, which can be parts of cache lines.[2]

Associativity describes a mapping between two layers. It indicates the number of cache lines that are stored in a single set. A cache is called *fully associative*, if it has only one set. Thus, any of the cache lines can hold any of the memory lines. A cache is called *direct mapped* if the size of its sets is one line, i.e., each memory line has only one cache line to store it.

If a cache is neither fully associative nor direct mapped, it is called *set-associative*. An example of a set-associative cache is given in Figure 2.2. Its set size is two, i.e., the cache consists of sets each having two lines. Thus, each of the memory lines has only two cache lines where it can be stored. Typically, set size is 1–16 lines[3].

The *fetch policy* is the algorithm for deciding when to fetch from a lower level of memory and what is to be fetched. Often, caches work *on demand*, i.e., they do not pre-fetch memory lines.

Let memory lines 1, 2, and 3 in Figure 2.2 be fetched into the cache in succession. First, memory line 1 is stored in either of the cache lines. Then,

[2]There are exceptions to this basic organization (e.g., [84]).
[3]Exceptions exists, for example, ARM9 processors can have 64-way associative caches [53].

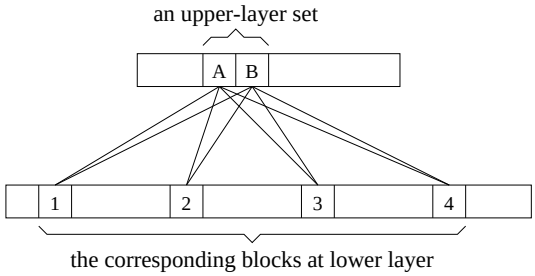


Figure 2.2: An example of associativity when set size is two.

memory line 2 is stored to the one that remains free. As memory line 3 is referred to, either memory line 1 or 2 must be replaced.

The *write policy* is the algorithm that decides on storing data down to the memory. The typical choices are write-through and write-back. In *write-through*, modified lines are immediately written to the memory. In *write-back*, modified lines are written to the memory when they need to be replaced.

The *replacement policy* is the algorithm for deciding which line will be replaced in the cache when a new line is fetched from the memory. The LRU method is a common choice [128].

The LRU method (Least Recently Used) replaces the least recently used line in the cache [17, 107]. The LRU method can be implemented by maintaining a data structure, which represents memory lines in their temporal order of last reference. If the LRU method is used in the example, then line 1 is substituted for line 3 at the last step.

2.5 Program execution

Application programs are rarely executed on an empty machine[4]. Usually, there is an operating system that provides the application programs with an interface to the hardware resources [148, 109]. The operating system also manages the use of hardware resources. It allocates hardware resources to application programs. Since there may be conflicting requests for hardware resources, the operating system must do the allocation in a fair and efficient way.

Operating systems usually allow multitasking, which means that several programs can be at some intermediate point of their execution simultaneously. Each such program execution forms a process that has its own flow of control, program text (executable code) and data that reside in the memory. The processor

[4]Small real-time systems are exceptions to this. They run without supporting software.

is shared by the processes. The operating system controls this processor sharing by allowing each process to execute a slice of time before switching to another process.

When a process switch happens, the execution context is also switched. The state of the execution is saved so that it can be restored and the execution continued. Thus, application programs can be written without any knowledge of these suspensions.

The state is saved only to the extent that enables functionally correct continuation of the execution. This includes, for example, program counter and register contents. However, cache contents are typically not saved and restored. Thus, other processes that interrupt the execution can change the contents of the cache. Further, the code of the operating system, run-time system of the compiler, dynamic libraries etc. can affect the cache.

Programs typically execute in a local manner, i.e., there is a limited set of memory locations that are frequently referred to. The set is called the *working set* of the program [40]. A *cold cache* does not contain this set. If a program is executed for a while, the cache *warms up* to contain the working set. The time slices that an operating system gives to processes are usually adjusted so that the cache has enough time to warm up.

In cache behavior study, two areas of research are separated [16]: *intrinsic* cache behavior study studies cache behavior of individual processes, while *extrinsic* cache behavior study studies inter-process effects on caches. The study of this thesis is intrinsic.

2.6 Cache misses

Cache memories [128] improve memory access times. They reduce the number of cycles a processor is waiting for data; in the best case, the processor can continue its operation without any stall. Current first level caches can give access to data over an order of magnitude faster than main memory [125].

Before accessing the main memory, the computer hardware checks whether the addressed datum is stored in a cache line. The requested address is compared with all the addresses of the memory lines in the cache set of the requested address. If the datum is in cache, then a *hit* occurs. Otherwise, a *miss* occurs and a memory line in the cache is replaced by a new one.

Cache misses affect program performance, because accessing main memory is much slower than accessing cache. As a result of this development, the traditional metrics used in designing software performance do not apply. Cache memory has become the bottleneck for many applications. Therefore, the number of lines (or instructions) executed and the related complexity measures do

not have the importance that they used to have. It is important to know the number of cache misses occurring, and the reason for those cache misses.

The crucial role of cache memories makes it hard to understand the performance of programs. The steps executed and the related memory references can be seen from the program code. However, cache misses and the related execution stalls cannot be seen. Computer aided engineering tools are needed to detect and locate references slowing down execution.

The misses can be categorized into three [69]:

- *Compulsory misses*, the first access to a line always causes a miss.
- *Capacity misses* occur when the cache is too small to hold all of the lines needed during an execution of a program.
- *Conflict misses* occur when the cache has sufficient space, but the organization of the cache does not allow the data to be kept in the cache[5].

For a programmer, it is important to know the pieces of the source code that cause the cache misses. Programs can be redesigned to have a smaller working set to avoid capacity misses. Conflict misses can be avoided by modifying the layout of data. There exists a number of studies on cache performance design that are based on understanding the cache behavior of a program (see, for example, [12, 19, 28, 27, 29, 31, 52, 55, 94, 112, 124, 126, 133, 134, 157]).

2.7 Real-time systems

A real-time system is a computer-based system in which the timing of a computed result is important [102]. Typical real-time systems are various control mechanisms, multimedia systems, and communication systems. The control actions, data transfer etc. must happen in time, otherwise such systems will not operate correctly.

Real-time systems are usually divided into *soft* and *hard* real-time systems. In hard real-time systems, a failure to meet timing requirements can be fatal, i.e., severe damage can result. In soft real-time systems, an occasional failure to meet timing requirements does not have permanent negative effects. Typically, the quality of the service provided is reduced, but the service is useful. This thesis is more focused on soft real-time systems than hard real-time systems.

Real-time systems are often *embedded systems*, and vice versa. An embedded system is a dedicated computer system that is a component of another system. Embedded systems – and thus, also real-time systems – are very common. A typical mobile phone has two embedded processors [26]. A Mercedes-Benz S

[5]These misses are also called collision or interference misses

automobile has 48 processors [60]. It has been estimated that 98%–99% of the total number of processors produced are in embedded systems [64].

Contrary to the impression that seems common, typical real-time systems are not fast. In real-time systems, predictability of timing behavior is more important than speed. However, speed requirements for real-time systems are increasing (for example, fast speed is required by multimedia applications). In the development of real-time systems, software and hardware are often co-designed. Design decisions are made so that performance analysis (including timing analysis) of the systems are easy. This means that the software is carefully written for the specific hardware.

In hard real-time systems, no software or hardware features that hamper the predictability of timing behavior are included. In the past, this has meant that processors with pipelines and cache memories have not been used in hard real-time systems. The rapid development of program analysis has changed this scene. Currently, there are complex hard real-time systems that use processors with modern architecture, e.g., the Airbus A380 airplane has such embedded systems [46]. Also, complex software structures are used in modern real-time systems [72].

2.8 Software performance engineering

Software performance engineering is a methodology for developing software systems to meet performance objectives [144]. It provides software development processes with various methods and tools that give analyses and guidelines. Such methods and tools cover the software development from early development throughout the whole lifetime of the software.

As such, software alone has no performance that can be stated. An understanding of the underlying hardware and the workload of the software is always needed. Consequently, the analyses needed in software performance engineering are difficult.

Because of the difficulty of analyzing performance of software systems, approximate models are often used. There exists a number of modeling techniques and related software performance engineering methods (see [62, 81, 111, 146] for examples).

Software performance work is often based on designing program code at various levels (from the statement level to the architectural level). When memory is the hardware bottleneck, it is important to design data structures and their layout in the memory. In this task, program analysis tools play a central role.

At least, the memory behavior of critical program parts should be designed and tested. In our experience [72], the performance critical structures are often

only a small part of a software. Thus, they can be closely studied. In addition to locating the origins of cache misses, we must understand their cause and type (compulsory, conflict, or capacity) to make changes that improve performance.

2.9 Dynamic memory analysis

Because of the complexity of the memory hardware, interactions of memory references are complex. We must understand the accesses that the references make to understand their interactions. Typical hardware does not support analysis of memory operations [78]. Therefore, the memory operations of such programs are often simulated.

The most common memory simulations are trace-driven simulations [156]. A trace-driven simulation has two main phases. In the first phase, an access trace is collected. Because hardware support for cache tracing is rare [78], the collection is typically completed by augmenting the subject program with trace emitting code. In the second phase, a memory simulation is executed using the trace of the first phase as the input. Between these two main phases, there can be a trace reduction phase that makes the trace easier to handle.

Trace-driven simulations are especially suitable for hardware performance studies. In such studies, the trace, once collected, can be used several times; it serves as a benchmark for hardware. In a software analysis, the input is varied and a different trace is usually generated for each simulation. Therefore, it is practical to use on-the-fly (execution-driven) simulation in software performance studies. In such simulations, the trace is consumed as it is produced [56].

The major problem in trace-driven simulation studies is the size of traces [129, 156]. The traces resulting from program executions can be huge. To understand thoroughly the memory behavior of a program, the program must be executed with several inputs and several traces must be generated. Thus, very long simulation times can be needed because of the huge amount of data to be analyzed.

2.10 Static program analysis

In static program analysis [118], run-time behavior of a program is analyzed without executing it with a specific input. Static program analysis is usually motivated by its ability to simultaneously give results for a set of inputs (often for all inputs of a program). Because the behavior of a program can be different for different inputs, static analysis is usually approximative. Instead of giving precise results, static analysis usually gives upper and lower bounds for possible values.

Static program analyses often do not result in a single value. Typically, they attach analysis information to the program's structure. For example, first-order program analyses attach state information to program structure, e.g., values bounding possible program states are attached before and after each statement in a program.

There are several ways to approach the problem of statically analyzing programs. Typical approaches [118] are the *equational approach*, the *constraint-based approach*, and *abstract interpretation*. Selecting the approach means selecting the way that we use to present our analysis problem and its solution, i.e., the resulting analysis.

In the equational approach, we use two classes of equations to describe an analysis. One class of equations relates the information before an operation to the information after an operation. The other class of equations relates the information of an operation to the information of another operation. In the constraint-based approach, inclusions (often inequations) are used instead of equational relations.

Abstract interpretation is a semantic-based approach to program analysis. To some extent it is independent of the specification style, i.e., it works on a level different from the equational approach or constraint-based approaches. It can be considered as a general methodology that can be used to derive analyses. This thesis will use parts of the theory of abstract interpretation. We will review them in Section 3.4.

Implementation of a program analysis is a separate issue. There are several ways to implement an analysis. Selecting the implementation means selecting the way that we are going to compute the solution for a specific analysis task. Implementation techniques include worklist algorithms [89], round-robin algorithms [85], structural analysis [139], graph-based algorithms [79], and path algebras [22].

2.11 Execution time analysis

Execution time analysis is important in real-time systems. There are several ways to analyze the execution time of a system. These include static program analysis, performance models, and evolutionary testing. The selection of the analysis method should be based on the system properties and the analysis information needed. Hard real-time systems require execution time guarantees, i.e., analysis of worst-case and best-case execution times (WCET and BCET analysis). In soft real-time systems, average-case analysis can be more useful.

Static program analysis is useful in WCET and BCET analysis, because it can give guarantees of performance that are based on the software itself. Three

different ways of computing timing bounds are usually applied: tree-based techniques, path-based techniques, and implicit path enumeration techniques. In tree-based techniques [122], the timing bound is calculated using bottom-up traversal of the program. Path based techniques [149] search for the execution path that gives the bounding execution time. Implicit path enumeration techniques [99] use algebraic or logical constraints to describe timing behavior. Static cache analysis (including the one presented in this thesis) can be used as a component of such methods.

Performance modeling is useful for both hard and soft real-time systems. Compared to static analysis, using performance modeling is demanding; usually a lot of expertise and effort is needed to reliably model systems. In performance modeling, description of system performance is constructed. Typical performance models are based on abstract descriptions of the system structure. To describe timing performance, timing information is attached to the abstract structure. Examples of software performance models are sequence charts [145] and task graphs [14]. In addition to the software, the input and the underlying hardware must be modeled. The combined models can be solved using analytical techniques or simulation.

Statically analyzing or modeling timing behavior of some systems can be infeasible. Evolutionary testing [116] uses search-based methods to analyze the timing behavior of programs. Evolutionary testing is based on simulating or executing the subject program with a number of inputs. The approach is sensitive to the size of the search space, which is exponential in the number of input variables to a program. Search heuristics are often based on the observation that all input variables are not important to performance. Various methods have been used to improve the search of test cases, e.g., program slicing [65]. The simulation methods (including the one in this thesis) can be used as a basis for evolutionary testing.

2.12 Conclusions

The relationship between the source code of a program and its behavior during execution is complex. This complexity is not only caused by the semantics of the source language. From the performance point of view, the features of the underlying hardware significantly increase the complexity.

Analyzing this complex relationship is made difficult by all the things that happen before the hardware actually executes the software. These include – but are not limited to – the actions that are done by the compiler, the linker, and the loader. Further, during the execution several factors affect the performance, e.g., memory management and process management of the operating system.

In theory, measuring can give accurate information about the execution behavior of a program, but in practice, it is increasingly difficult to do so, because of the increasing complexity of hardware and software. Software emulation tools often do not contain all the peripheral hardware components that affect performance. On the other hard, hardware tools often do not contain all the probes that would be needed for a proper understanding of the behavior of a program.

When predictability of performance is a must (e.g., in hard real-time systems), analysis is usually made easier. This is typically achieved by using such software and hardware that the current analysis methods are capable of analyzing. Thus, developing analysis methods will increase the number of software and hardware features that can be used in such systems.

Chapter 3

Combined program analysis

In program analysis, we find out the execution behavior of programs. For example, we might be interested in whether a program uses uninitialized variables in its computation. Such use can lead to errors, because of the unknown (and possibly erratic) values contained in the variables. Other uses include – but are not limited to – performance evaluation, optimizations, debugging, maintenance, and testing.

Program analysis can be performed dynamically, i.e., by executing the program. It can also be performed statically, i.e., without an execution of the program with a specific input. A specific input is used in dynamic analysis, but static analysis considers a set of inputs. Often all inputs of a program are considered at the same time. Results of a dynamic analysis are accurate, but are valid only for the single input. Results of a static analysis are valid for all the inputs considered, but usually are approximative.

In the following, we consider both dynamic and static analysis. We define a simple (but generic) programming language and present a dynamic analysis approach and a static analysis approach for the language in a semantics-based manner. Then, we present the combined analysis, which uses dynamic analysis to improve the results of static analysis and static analysis to speed up dynamic analysis.

3.1 A programming language

We use a simple programming language in describing our framework. We do not define the formal semantics of the language. The main motivation for the language is to illustrate the concepts presented. The language is designed to make

instrumentation easy. Despite this, our approach is generic in its nature. Programs written in complex languages can be instrumented – including programs in machine languages [95].

A program in our language is a statement P that can contain a finite number of substatements. The syntax (and informal semantics) of the language is the following:

$P ::= P_1 P_2$	statement concatenation
$\mid$ `(`op`)`^{l}	operation computing value for flag l
$\mid$ `[`op`]`^{l}	operation altering memory state
$\mid$ `[skip]`^{l}	do nothing
$\mid$ `while`^{l} P_1 `do` P_2 `end`	loop while flag l is not zero
$\mid$ `if`^{l} P_1 `then` P_2 `else` P_3 `end`	if flag l is zero P_3 is executed else P_2

The syntax given is an abstract one. We use labels to identify program elements, e.g., `[skip]`^{5} is the program element labeled with 5. Condition testing in `if` and `while` statements is based on labeled flags. The statement setting the flag must have the same label as the `if` statement or the `while` statement, e.g.:

```
if² [output a]¹ (a < b)² [output b]³ then
  [m = a]⁴
else
  [m = b]⁵
end
```

The statements between `if` and `then` are executed before the flag is tested. The flag of the `if`^{2} statement is set by `(`a `<` b`)`^{2}, because it has the same label as the `if`^{2} statement. Thus, the code fragment outputs both a and b, and then, assigns the smaller of them to m.

In the language, operations (statements with *op*) are the only statements that can change the state of the memory or perform I/O. Other statements affect the flow of control, i.e., the program counter.

3.2 Execution semantics

We define execution semantics using a notion of the execution state of a program on an abstract level. A program begins in some starting state, and each execution step transforms the current state into a new state. Informally, a state consists of data structures, remaining input data, and emitted output data. We consider all of them as values that are held in memory cells.

Definition 3.1 (execution state). We define the *execution state* of a program as a function $m : V \to Z$, where V is the set of memory cells and Z is the set of integers. We denote the set of all execution states by M.

Execution of a program element ϵ_p at point p of a program P can cause a change of state by altering the value of some memory cell $v_i \in V$, i.e., ϵ_p computes a new value for $m(v_i)$. We call the pair $(v_i, m(v_i))$ the *computed value* (at ϵ_p).

The state of an execution at a specific point depends on the initial state and the program elements executed so far. If the program terminates, it reaches the final state m_f corresponding to its initial state m_o. Alternatively, we can consider program P to have computed some output O from its input I. We denote that computation by $[\![P]\!]\langle I\rangle = O$, i.e., $[\![P]\!]$ is the function computed by program P (possibly a partial function).

Our framework is based on program transformations. These transformations are program instrumentation, partial evaluation, and program slicing, which are described in the following sections. The transformations must preserve some semantics of the transformed program. Therefore, we define a concept of program similarity.

Definition 3.2 (program similarity). Let programs A and B share a set of program elements. We say that programs A and B are *similar* if for any input (for which A and B terminate) the sequence of computed values at the shared program elements in the execution of A is the same as the sequence of computed values at the shared program elements in the execution of B.

The above concept of program similarity is a simplified form of the one used in [15]. In the following, we base the correctness of our program slicing on their work.

3.3 Dynamic analysis

In dynamic program analysis, we consider the state of a program execution or values that can be computed from the state. To do the analysis, we augment a subject program with *instrumentation code* that implements our analysis. In this thesis, we will use the word *augmentation* to mean a single addition of code and the word *instrumentation* to mean the measuring system that results from all the augmentations.

We do not want to affect the original computation of a subject program. Therefore, we use an instrumentation state that is separated from the original execution state of the subject program. Informally, an instrumentation state

contains values of memory cells that are not shared by the original subject program, remaining input for the instrumentation code, and output emitted by the instrumentation code.

Definition 3.3 (augmented execution state). We define the execution state of an instrumentation as a function $m^+ : V^+ \to Z$, where V^+ is a set of memory cells so that V^+ and V are disjoint. We denote the *augmented execution state* of the whole program by $m^A = m \cup m^+$ and the set of all augmented execution states by M^A.

In addition to the original program elements, an instrumented program contains program elements implementing the instrumentation. We denote such an instrumented program by P^A and the function computed by the instrumented program by:

$$[\![P^A]\!]\langle I, I^+\rangle = \langle O, O^+\rangle$$

where I^+ is the input for the instrumentation (analysis input) and O^+ the output from the instrumentation (analysis output).

Definition 3.4 (probes). Augmentations ϵ_{op} and ϵ_{of} are called *probes* if

- an original program element ϵ_o is replaced by the sequence of program elements $\epsilon_{op}\epsilon_o\epsilon_{of}$,
- any execution of ϵ_{op} and ϵ_{of} always terminate, and
- for any $v \in V$, $m(v)$ is not changed by execution of ϵ_{op} or ϵ_{of}.

Thus, we want probes not to affect the state of the original subject program or its flow of control. (In the definition, ϵ_{op} is the augmentation *preceding* the element ϵ_o and ϵ_{of} is the augmentation *following* it.)

Theorem 3.5 (correctness of probes). Let P^A be given the initial state $m_o^A = m_o \cup m_o^+$ whenever P is given the initial state m_o (i.e., m_o^+ contains the initial values for the memory cells for the instrumentation). If a program P is instrumented with probes, then the instrumented program P^A is similar to the original program.

Proof. The shared program elements are the program elements of P, because P^A is constructed by inserting code to P.

The initial state m_o corresponds to an input I. Let m_i be an execution state resulting from an execution of a program element ϵ in the program execution computing $[\![P]\!]\langle I\rangle$ and $\{\epsilon_1, \ldots, \epsilon_n\}$ the program elements whose execution can

immediately follow the execution of ϵ.

Assume that in execution of $[\![P^A]\!]\langle I, I^+\rangle$ there is execution of ϵ that results state m_j^A, and assume that $m_j^A \cap m_i = m_i$. If any original program element ϵ_f will follow in execution of P^A, then it must be one of $\{\epsilon_1, \ldots, \epsilon_n\}$. Let m_k^A be the state before the execution of ϵ_f. Then, $m_k^A \cap m_i = m_i$ must hold, because augmentation code does not change the original memory cells of P.

The original program elements use only cells $v \in V$. Initially, $m_o^A \cap m_o = m_o$. Thus (by induction), the sequences of computed values at original program elements in the executions are the same. □

We illustrate this by a simple example that finds out uses of uninitialized variables. In Chapters 4–6, we describe a demanding application of our framework (i.e., the cache analysis).

Example 3.6. Consider the program on left in Figure 3.1. The program uses variables a and b to compute an arithmetic function. We build a dynamic analyzer that checks that uninitialized variables have not been used. The instrumented code is on the right. Operator `|` is the bitwise-or.

The instrumentation state consists of the instrumentation variables u_a, u_b, and e. Variables u_a and u_b indicate whether variables a and b are uninitialized, respectively (0 for an initialized variable, 1 for an uninitialized variable). The variable e indicates whether a use of an uninitialized variable has occurred (0 for no such error, 1 for such an error).

The instrumentation is simple. Each statement ϵ_i assigning variable x is followed by augmentation $[u_x \texttt{ = } 0]^{if}$. The augmentation indicates that x has

```
[input a]^1                          [input u_a, u_b, e]^Pp
if^2 (a > 0)^2 then                  [input a]^1 [u_a = 0]^1f
  [input b]^3                        if^2 [e = e | u_a]^2p (a > 0)^2 then
else[skip]^4end                        [input b]^3 [u_b = 0]^3f
while^5 (a > 0)^5 do                 else[skip]^4end
  [a = a - 10]^6                     while^5 [e = e | u_a]^5p (a > 0)^5 do
end                                    [e = e | u_a]^6p [a = a - 10]^6 [u_a= 0]^6f
[b = a + b]^7                        end
[output b]^8                         [e = e | u_a | u_b]^7p [b = a + b]^7 [u_b = 0]^7f
                                     [e = e | u_b]^8p [output b]^8
                                     [output e]^Pf
```

Figure 3.1: A subject program (left) and the same program with an instrumentation (right).

been initialized. Each statement ϵ_i using variable x is preceded by augmentation $[e = e \mid u_x]^{ip}$. The augmentation checks that x has been initialized before the use.

Statement labeled Pp initializes the instrumentation state, i.e., has there been errors prior to the execution, and have the variables been initialized prior to the execution. Statement labeled Pf outputs the analysis result.

3.4 Static analysis

In static analysis, we estimate the execution state of a program without actually executing the program. Our static analysis follows the concept of abstract interpretation presented by [36].

In abstract interpretation, the analysis task is described using collecting semantics, which maps each program point to the set of *concrete states* that are possible at the point. The analysis task is solved using an *abstract state*, which is a safe approximation of possible concrete states. Thus, abstract interpretation is executing the program using imprecise (but computable) abstract states instead of the precise (and often statically uncomputable) collecting semantics.

Let M_o be the set of all possible initial states of a program P. The collecting semantics $cs : P \mapsto D_{cs}$ maps the program points p to sets of states:

$$cs(p) = \bigcup_{i \in M_o} \cup\{[\pi]_{conc}(i) \mid \pi \text{ is a path to } p\}$$

where $[\pi]_{conc}$ describes the state change corresponding to the execution of the path π.

In an analysis, sets of states are described by elements of a new (simpler) abstract domain D_{as}. The meaning of each element of the abstract domain is defined by a concretization function $conc : D_{as} \to D_{cs}$, and the reverse relation by abstraction function $abs : D_{cs} \to D_{as}$.

The abstract semantics $as : P \mapsto D_{as}$ maps the program points to the abstract domain:

$$as(p) = \bigsqcup\{[\pi]_{as}(abs(M_o)) \mid \pi \text{ is a path to } p\}$$

where $\bigsqcup$ is the least upper bound operator on D_{as} and $[\pi]_{as}$ describes the abstract state change corresponding to the execution of the path π.

Figure 3.2 illustrates this process. The meaning of a program function f (concrete domain) is described by an abstract function $\hat{f}$ (abstract domain). The set of possible concrete states can be statically approximated by $f(x) \subseteq$

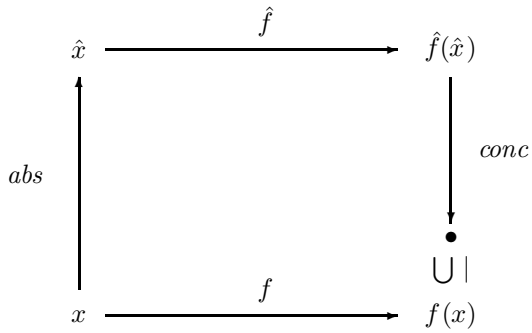


Figure 3.2: Abstract interpretation.

$conc(\hat{f}(abs(x)))$, where x is the set of initial concrete states and *conc* and *abs* are the concretization function and the abstraction function between the domains.

With abstract interpretation, we can show soundness of our static analysis. Abstract interpretation is based on lattice theory and Galois connections [118]. In the following, we define some basic concepts, and then state our notion of soundness.[1]

Definition 3.7 (lattice). A *join semi-lattice* $D = (D, \sqcup)$ is a non-empty set, D, with a *join* operation, $\sqcup$, which is idempotent, commutative and associative.

Idempotency means that $x \sqcup x = x$, commutativity means that $x \sqcup y = y \sqcup x$, and associativity means that $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$. The join operation implies a partial ordering, which we denote by $\sqsubseteq$. Thus, a subset of a lattice has an upper bound d_0 if $d \sqsubseteq d_0$ for all members d of the subset.

Definition 3.8 (complete lattice). A *complete lattice* is a lattice, where all subsets of D have a least upper bound d_0, i.e., $d_0 \sqsubseteq d$ whenever d is another upper bound of the subset.

A complete lattice has a least element $\bot$, which is the unit of the join operation. For implementing an analysis, it is useful that all ascending chains in the property space eventually stabilize. Chains are sequences $(l_n)_{n \in N}$ that are totally ordered subsets of D. They eventually stabilize if and only if $\exists n_0 \in N : \forall n \in N : n \geq n_0 \Rightarrow l_n = l_{n_0}$.

Usually, lattices are the key concept in any static program analysis technique. Because of the properties of a lattice structure, we can use a bound that describes

[1] We will base our definitions on a *join* operation, ascending chains, and Galois injection properties. Other ways exist, but the approach is useful for our development.

the possible program states instead of listing them.

Definition 3.9 (monotony). A function $f : D_1 \to D_2$ is *monotone* if $\forall l, l' \in D_1 : l \sqsubseteq_1 l' \Rightarrow f(l) \sqsubseteq_2 f(l')$, where $\sqsubseteq_1$ is the ordering in D_1 and $\sqsubseteq_2$ is the ordering in D_2.

Definition 3.10 (adjointness). Abstraction *abs* and concretization *conc* are strongly adjoint if $\forall x \in D_{cs} : x \subseteq conc(abs(x))$ and $\forall x \in D_{as} : x = abs(conc(x))$.

If abstraction and concretization are monotone and strongly adjoint, then they form a *Galois injection* between the lattices of abstract and concrete domains. This means that we can safely use a bound in the *abstract* domain to describe possible program states in the *concrete* domain.

Definition 3.11 (consistency). Functions $f : D_{cs} \to D_{cs}$ and $\hat{f} : D_{as} \to D_{as}$ are locally consistent if $\forall x \in D_{cs} : f(x) \subseteq conc(\hat{f}(abs(x)))$.

Local consistency means that our abstract interpretation of a single step in a program is sound: the concrete states given by collecting semantics of the step is a subset of the concretized abstract state given by abstract semantics of the step. Using a Galois injection, we can consider a whole program by means of abstract interpretation.

Theorem 3.12 (soundness of abstract interpretation). If an abstract interpretation satisfies the following conditions, then the set of concrete states given by collecting semantics is a subset of the concretized abstract state given by abstract semantics.

- $(D_{cs}, \subseteq, \cup, \bot_{cs})$ and $(D_{as}, \sqsubseteq, \sqcup, \bot_{as})$ are complete join semi-lattices;
- *abs* and *conc* are monotone and strongly adjoint; and
- the abstract operation $\hat{f}$ is locally consistent with the concrete operation f.

Proof can be found in, e.g., [36].

The meaning of concrete program functions $f(x)$ can be described by abstract functions $\hat{f}(\hat{x})$. The result is a safe approximation; we do not miss any concrete semantics of a program, but can include some false ones.

Using static analysis, we can find out that some of the values needed or yielded by some program element ϵ are constants. The program specializations that we use are based on this.

Example 3.13. We consider again Example 3.6, but we use static analysis for the same analysis task and assume analysis input $u_a = 1$, $u_b = 1$, and $e = 0$. Using static analysis, we can solve some values needed in our analysis (values on the left in Figure 3.3). For such a simple program, the uses of uninitialized variables can be seen by looking at the program code, but computational methods exist for such tasks, e.g., a derivation of algorithms for computing reaching definitions [5].

[input a]^1	$u_a = 0$	$u_b = 1$	$e = 0$
if^2($a > 0$)^{2}then	$u_a = 0$	$u_b = 1$	$e = 0$
[input b]^3	$u_a = 0$	$u_b = 0$	$e = 0$
else[skip]^{4}end	$u_a = 0$	$u_b = 0 \vee 1$	$e = 0$
while($a > 0$)^{5}do	$u_a = 0$	$u_b = 0 \vee 1$	$e = 0$
[a = a - 10]^6	$u_a = 0$	$u_b = 0 \vee 1$	$e = 0$
end	$u_a = 0$	$u_b = 0 \vee 1$	$e = 0$
[b = a + b]^7	$u_a = 0$	$u_b = 0$	$e = 0 \vee 1$
[output b]^8	$u_a = 0$	$u_b = 0$	$e = 0 \vee 1$

Figure 3.3: A static analysis of a program.

Statement 1 defines a value for a, therefore $u_a = 0$ after that. Statement 3 defines a value for b, therefore $u_b = 0$ after that. Because statement 3 is inside an `if` statement, u_b can also be 1, until statement 7 unambiguously defines a value for variable b.

A use of an uninitialized variable can happen in statement 7, because u_b can be 1. Therefore e can be 0 or 1 after that point. Our static solution is approximate (e can be 0 or 1); obviously the analysis result is dependent on the input for the program.

3.5 Combined analysis

Our combined analysis has three phases: a compilation phase, an execution phase, and a summary phase. In the compilation phase, a subject program is statically analyzed and a dynamic analyzer is built. In the execution phase, the dynamic analyzer is executed (typically with several inputs). In the summary phase, the analysis information of the compilation phase and the analysis information of the execution phase are combined.

The execution phase follows the typical procedures of simulation studies, which can be found in a suitable text book (e.g., [96]). The summary phase simply merges the static and dynamic analysis results. As the compilation

phase is special, we will describe it in detail.

The compilation phase consists of three program transformation steps: program instrumentation, partial evaluation, and program slicing. The first step creates a straightforward dynamic analyzer and the last two are program specializations, which make it faster and more compact. The two specialization steps need static analysis information.

Step 1: Instrumentation

The first phase is an instrumentation that creates dynamic program analyzers. Let P be the original subject program and O its output corresponding to input I, i.e., the program computes:

$$[\![P]\!]\langle I\rangle = O$$

The corresponding instrumented program computes:

$$[\![P^A]\!]\langle I, I^+\rangle = \langle O, O^+\rangle$$

where I^+ is the input for the instrumentation and O^+ is the analysis result measured by the probes.

Corollary 3.14 (correctness of step 1). Directly from Theorem 3.5 follows that if P is transformed into P^A by adding probes, then P^A is similar to P.

This shows only the correctness of our framework. The correctness of the instrumentation itself must be separately shown for each application.

Step 2: Partial Evaluation

Partial evaluation is a program transformation that is given a subject program with part of its input data. It constructs a new program that, when given the remaining input, will yield the same result that the original subject program would have produced given full input.

Consider the program P^A computing $[\![P^A]\!]\langle I, I^+\rangle = \langle O, O^+\rangle$. Let *peval* denote the partial evaluator, then

$$([\![peval]\!]\langle P^A, I^+\rangle = P^A_{I^+}) \Rightarrow ([\![P^A]\!]\langle I, I^+\rangle = [\![P^A_{I^+}]\!]\langle I\rangle)$$

for all I. Thus, we fix the analysis initialization (input) and evaluate statically part of the analysis.

Our partial evaluation is based on static analysis of the original program P. The static analysis is performed for the same task as the preceding instrumentation for the dynamic analysis (which produced P^A from P). Static analysis can give us static values of both the original program and its instrumentation.

Let an augmentation compute a function $g(v_0, v_1, \ldots, v_n) = g(V)$, where the argument vector $V = v_0, v_1, \ldots, v_n$ is a subset of some program state. Based on static analysis, we can know that some of the arguments in an argument vector are constants. Therefore, we can substitute $g(V)$ for $g(V_V \mid\mid V_C)$, where V_C contains the constant arguments and V_V the variable arguments. The operator $\mid\mid$ denotes merging two argument vectors into one argument vector (in the correct order).

Because of the constant values, the function g can be partially evaluated resulting in a simpler function g_C. In the best case, V_V is empty: no computations are needed at run time.

Now we can formulate a (rather trivial) condition for correctness of our partial evaluation. At this point, we do not explain how such a program transformation could be performed. We will discuss partial evaluation in more detail in Chapter 6.

Theorem 3.15 (correctness of step 2). An original analyzer is similar to the partially evaluated analyzer, if for all original instrumentations g and the corresponding partially evaluated instrumentations g_C:

$$\forall\, V_V \in M_g^A : g_C(V_V) = g(V_V \mid\mid V_C) \;\wedge\; V_C \text{ is constant in } M_g^A$$

where V_V is a variable argument vector, V_C is a constant argument vector, and M_g^A are the possible augmented program states at the point preceding g.

Proof. Let m_i^A be a state before execution of g, and m_{i+1}^A the state after its execution. Let m_i^C be the state before execution of g_C, and m_{i+1}^C the state after its execution. If $m_i^C = m_i^A$ then $m_{i+1}^C = m_{i+1}^A$, because g and g_C compute the same value. □

Essentially, Theorem 3.15 demands that the argument vector V_C describes the same substate in all states $m \in M_g^A$, i.e., V_C truly is a constant. In such a situation, g and g_C must compute the same result.

If we assume that the instrumentation input I^+ is constant, then we can apply this also to the input statements of the instrumentation. In this way, we will get an analyzer that is specialized for a fixed instrumentation input.

Example 3.16. Consider our previous examples. If we use the static informa-

tion of Example 3.13, then we can partially evaluate the dynamic analyzer of Example 3.6 into the one in Figure 3.4.

```
[u_a = 1]^Pp1 [u_b = 1]^Pp2 [e = 0]^Pp3
[input a]^1 [u_a = 0]^1f
if^2 [e = 0]^2p (a > 0)^2 then
  [input b]^3 [u_b = 0]^3f
else[skip]^4end
while^5 [e = 0]^5p (a > 0)^5 do
  [e = 0]^6p [a = a - 10]^6 [u_a = 0]^6f
end
[e = u_b]^7p [b = a + b]^7 [u_b = 0]^7f
[output b]^8
[output e]^Pf
```

Figure 3.4: A partially evaluated analyzer.

The analyzer does not read any analysis input. Instead, it has initializing assignments indicating that all variables are uninitialized and no uses of uninitialized variables have happened (statements $Pp1$, $Pp2$, and $Pp3$).

Several computations of new values have been replaced by assignments of constants, because their values are statically known. These are statements $2p$, $5p$, and $6p$. For them, we know that $e = 0$ and $u_a = 0$, thus $(e \mid u_a) = 0$.

Statement $7p$ is simplified, because we know that $e = 0$ and $u_a = 0$, thus $(e \mid u_a \mid u_b) = u_b$. Nothing remains from statement $8p$, because we know that $u_b = 0$, thus $(e \mid u_b) = e$. The resulting dynamic analyzer is faster than the analyzer of Example 3.6, because we have used static information to speed it up.

Step 3: Slicing

As described in the preceding, a partially evaluated analyzer computes both the original output and the analysis output:

$$[\![P^A_{I^+}]\!]\langle I\rangle = \langle O, O^+\rangle$$

We do not need the original output. Further, we do not need the program elements that do not affect our analysis result.

We use program slicing to implement a program transformation that yields programs $P^A_{I^+Y}$ computing only the analysis output. Let *slicer* be the transformation, then for all I:

$$([\![slicer]\!]\langle P^A_{I^+}, Y\rangle = P^A_{I^+Y}) \Rightarrow ([\![P^A_{I^+Y}]\!]\langle I\rangle \stackrel{O^+}{=} [\![P^A_{I^+}]\!]\langle I\rangle \ \wedge \ [\![P^A_{I^+Y}]\!]\langle I\rangle \stackrel{O}{=} \emptyset)$$

where Y is called the *slicing criterion*, $\stackrel{O^+}{=}$ denotes equality of analysis output and $\stackrel{O}{=}$ denotes equality of original output.

Now we can formulate a condition for correctness of our program slicing. In our framework of dynamic analysis, the issue of program slicing is much more complicated than partial evaluation. As for partial evaluation, we do not explain here how such a program transformation could be performed in practice. We return to the matter in Chapter 6, where we discuss it in more detail.

Theorem 3.17 (correctness of step 3). If all input statements and the analysis data output statements are used as the slicing criterion, then a sliced dynamic analyzer consumes the same input and yields the same analysis output as the original dynamic analyzer.

Proof. The principal property of slicing is producing similar programs at the slicing criterion. A proof for this can be found, e.g., in [15].

Example 3.18. Consider our previous example. If we use all the original input statements (i.e., statements 1 and 3) and the analysis output statements (i.e., statement Pf) as the slicing criterion, then we can slice the dynamic analysis program into the following:

$[u_b$ `=` $1]^{Pp2}$
$[$`input` $a]^1$
`if`^{2} $(a > 0)^2$ `then`
  $[$`input` $b]^3$ $[u_b$ `=` $0]^{3f}$
`else`$[$`skip`$]^4$`end`
$[e$ `=` $u_b]^{7p}$
$[$`output` $e]^{Pf}$

Statement $7p$ assigns e its final value. Other assignments to e are useless. Therefore, statements $6p$, $5p$, $2p$, and $Pp3$ have been removed. Statement $7p$ uses only the values of u_b that reach it. Therefore, statements $7f$, $6f$, $1f$, and $Pp1$ have been removed.

The analyzer program does not need to produce output other than analysis output. Therefore, statement 8 producing the output of the original program has been removed. Statements 7, 6, and 5 compute only the output value. Thus, they too have been removed. Note that the analyzer program has to read the same input as the program of Example 3.6, therefore statement 1 has not been

removed.

A significant part of the original program has been sliced away; especially the loop that can take a long time to execute. The program could be further improved by using u_b instead of e in statement *Pf*, thus, also $7p$ could be removed[2].

3.6 Conclusions

In this chapter, we described a framework for program analysis. The framework combines dynamic program analysis and static program analysis: it uses dynamic analysis to improve results of static analysis and static analysis to speed up dynamic analysis.

We use two kinds of program specialization in speeding up dynamic analysis: partial evaluation and program slicing. They supplement each other. Considering the flow of control in a program, partial evaluation and program slicing work in opposite directions. In our example of partial evaluation, we stepped forward while partially evaluating statements, and looked backward for values of our variables. In our example of program slicing, we stepped backward while slicing statements, and looked forward for possible uses of computed values.

Our framework is an abstract one. This leaves several options available in implementing a specific analysis by using combined analysis. The augmentations needed in dynamic analysis depend on the analysis to be performed. The same is true for the static analysis suggested. Abstract interpretation gives a theoretical framework for static analysis. Several implementations are possible within the framework, e.g., the work-list algorithm [118].

Alternatives exist also for implementing partial evaluation and program slicing. For example, simple methods like constant folding or complex methods like polyvariant specialization can be used in partial evaluation [82]. In program slicing, there are methods that are based on data-flow equations, information-flow relations, and dependence graphs [153].

In the next three chapters, we apply our framework to a demanding real-world problem: cache performance evaluation of programs. In the application, we will give the reader a more concrete picture, how steps of our abstract framework can be realized to solve an analysis problem.

As the example in this chapter indicated, program specialization can be applied both to the subject program and to its augmentations. This can lead to significant performance improvements in dynamic analysis. However, the performance of the method is dependent on the subject program. On one extreme, nothing can be evaluated or sliced away during the program specializa-

[2]Compilers do such transformations, see [5].

tion. Thus, no data result from the compilation phase, and all is left to the simulation phase.

On the other extreme, all simulation steps can be evaluated statically and the whole input program sliced away during the compilation phase (except the part related to reading the original input and outputting the result). Thus, full data result from the compilation phase, and an empty simulation phase follows.

For the extreme cases, our method is of no value. According to our experience, such cases are very rare in practice, i.e., our method is useful in most practical cases. We will discuss this further in Chapter 7 in which our experimentation with our cache performance analysis method is presented.

Chapter 4

Dynamic cache analysis

In the previous chapter, we described a combined program analysis framework. In the following, we describe how combined program analysis can be applied to cache performance analysis. This chapter describes a dynamic cache analysis method (it could also be called an execution-driven cache simulation method). The two subsequent chapters describe how the dynamic analysis can be improved.

We start our description by defining a less general version of the programming language of the previous chapter. The version defines a simple load/store-mechanism for accessing memory. After defining the language, we define cache semantics in a generic way. The cache semantics describe what happens in a cache memory when a program is executed. The semantics are defined in more detail than is needed in this chapter because the next chapter is based on the semantics.

Our method of program analysis is based on augmenting programs with instrumentation code. The rest of this chapter describes the augmentations that are needed in our cache analysis. We define the augmentations at an abstract level to allow freedom for implementation.

4.1 A programming language

In the previous chapter, our description of the programming language did not consider the memory in any detail. Now, we assume that the memory is divided into two parts: registers (including flags) and a heap containing heap cells. Further, we assume that the data in the registers (excluding flags) and cells of the heap can contain pointers to heap cells in addition to plain data.

In our modified language, a program is a statement P, which can contain a finite number of substatements. We use variable names (var) to specify registers, literals to denote constants (num), and expressions. The syntax (and informal semantics) of the language is the following.

e ::=	$var \mid num \mid op(e_1, \ldots, e_n)$	expression
P::=	$P_1 P_2$	statement concatenation
|	$(e)^l$	expression setting flag l
|	$[var = e]^l$	assignment
|	$[var_1 = *var_2]^l$	load
|	$[*var_1 = var_2]^l$	store
|	$[\texttt{skip}]^l$	do nothing
|	$\texttt{while}^l\ P_1\ \texttt{do}\ P_2\ \texttt{end}$	loop while flag l is not zero
|	$\texttt{if}^l\ P_1\ \texttt{then}\ P_2\ \texttt{else}\ P_3\ \texttt{end}$	if flag l is zero P_3 is executed else P_2

Note that the heap can be accessed only by the load and store statements. Other statements use only registers, which cannot be cached. Thus, only load and store statements affect the state of the cache memory. In the following, we do not separate reading and writing, and assume LRU replacement policy.

4.2 Cache semantics

We define concrete semantics using a notion of the execution state of a program. We denote a program execution state by m and the set of all states by M. In this chapter, we give a detailed formulation only for states describing the contents of a data cache. We follow [47] in our formulation of concrete cache semantics.

The cache contains recently used memory cells. To identify the cells, we assign them locations (i.e., memory addresses).

Definition 4.1 (location). A location uniquely identifies a memory cell. Locations describe the pointer values that point to memory cells. We denote a location by l and the set of all locations by L.

We consider a memory T consisting of memory lines $t_i \in T$ of the size B. The memory line that a location $l \in L$ belongs to depends on its address $addr(l)$, and is determined by function $line : L \to T$:

$$line(l) = t_{addr(l)\,\texttt{div}\,B}$$

Let $F = \langle f_1, \ldots, f_N \rangle$ be the sets of an A-way set associative cache, and $R_i = \langle r_{i1}, \ldots, r_{iA} \rangle$ the lines of the set f_i. The cache set that a memory line

$t_i \in T$ belongs to depends on its line address $lineaddr(t_i) = i$, and is determined by function $set : T \rightarrow F$:

$$set(t) = f_{lineaddr(t) \bmod N+1}$$

Figure 4.1 illustrates these address calculations. The total size of the cache is $N \cdot A \cdot B$. Memory can be seen to consist of blocks of the size $N \cdot B$ containing a line per each cache set. Locations are contained in the lines. The cache associativity A does not affect this address calculation.

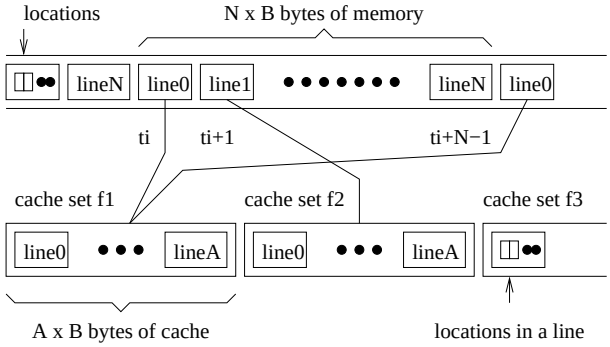


Figure 4.1: Mapping of cache sets and memory lines.

The capacity of a cache set is smaller than the number of memory lines belonging to it: only a fraction of its memory lines can be stored. Further, lines of a cache set can be empty, which we denote by $\diamond$. No duplicates are stored in a cache set. Thus, we define the state of a cache set by the following function.

Definition 4.2 (set state). A *set state* is a function $s : R \rightarrow T \cup \{\diamond\}$, where

$$\forall r_a, r_b \in R : s(r_a) = s(r_b) \Rightarrow s(r_a) = s(r_b) = \diamond \vee r_a = r_b$$

S denotes the set of all set states.

Note that each memory line belongs to a set (not to a specific cache line). We will use LRU replacement of cache lines, and order the cache lines of a set accordingly, i.e.:

$\forall r_a, r_b \in R : a > b \Rightarrow b$ is more recently referred to (younger) than a

For any memory line $t \in T$ in cache, the index i of its cache line $r_i \in R$ is

its *age*. Age is $\top$ for any memory line that is not in cache. Further, $\top > A$ and $\top + 1 = 1 + \top = \top$. The age of any location is the age of the memory line that holds it.

The state of a cache consists of the states of its sets. Memory lines belong to a unique set; they are not shared among sets. Thus, we define:

Definition 4.3 (cache state). A *cache state* is a function: $c : F \rightarrow S$, where

$$\forall f_y \in F : \forall r_x \in R : c(f_y)(r_x) \neq \diamond \Rightarrow set((c(f_y)(r_x)) = f_y$$

We denote the set of all cache states by C. Memory operations of a program change the state of the cache. By describing the state changes we can define the cache semantics.

Definition 4.4 (set update). The set update function $U_S : S \times T \rightarrow S$ describes the change of the state of a set. If the accessed memory line t is already in cache at line r_h, i.e., its age is h, then

$$U_S(s,t) = [r_1 \rightarrow t, r_i \rightarrow s(r_{i-1}), r_j \rightarrow s(r_j)]$$

where $i \in [2, \ldots, h]$ and $j \in [h+1, \ldots, A]$. Otherwise,

$$U_S(s,t) = [r_1 \rightarrow t, r_i \rightarrow s(r_{i-1})]$$

where $i \in [2, \ldots, A]$.

The memory line that is referred to becomes the youngest. In the first case, the memory lines newer than h are shifted one step older. In the second case, all lines in the set are shifted and the least recently used line will disappear.

Example 4.5. Let the state of a cache set be [4, 3, 2, 1], i.e., 4 is the most recently referred to memory line, and 1 is the least recently referred to memory line. If memory line 2 is referred to (i.e., h = 3) then the new set state is [2, 4, 3, 1]. If a memory line that is not already in cache is referred to, say memory line 5, then the new set state is [5, 4, 3, 2]. Thus, memory line 1 has been replaced by memory line 5 in the cache.

Definition 4.6 (cache update). The cache update function $U_C : C \times T \rightarrow C$ describes the change of the state of the cache:

$$U_C(c,t) = c[set(t) \rightarrow U_S(c(set(t)), t)]$$

The semantics define the LRU replacement policy that works on demand.

Example 4.7 Consider the following program:

```
[y = *b] 1
[x = *a] 2
if 3 (gt(x, y)) 3 then
  [x = *c] 4 else [skip] 5 end
if 6 (gt(x, y)) 6 then
  [y = *a] 7
else
  [y = *d] 8
end
```

Let associativity of a cache be two. The program uses four memory locations pointed by registers a, b, c, and d. Assume that they point to different memory lines $T = \{t_a, t_b, t_c, t_d\}$ belonging to a single cache set $R = \{r_1, r_2\}$. At the program point before statement 6, the possible concrete states are $\{(r_1, t_a), (r_2, t_b)\}$ and $\{(r_1, t_c), (r_2, t_a)\}$. (In the first state, t_a is the most recently referred to memory line and t_b is the other memory line cached.) Thus, statement 7 will always hit, and statement 8 will always miss. For a simple program, as the one above, such an analysis is computable. For many practical programs, such an analysis is not computable.

4.3 Analysis augmentation

The cache semantics of the previous section gives an abstract specification for analysis augmentations. However, it leaves the augmentations undefined. Instead of explicitly writing augmentations with our programming language, we define an abstract algorithm for cache analysis and define the points of a subject program that are instrumented. Thus, any actual implementation that conforms with the abstract algorithm is valid.

The abstract augmentation uses a mapping `incache` and a set of replacement queues Q_f, where $f \in [1, \ldots, N]$. For each cached line t, `incache`(t) is 1. Otherwise `incache`(t) is 0. For each cache set f, Q_f is the replacement queue of the set. The replacement queues represent the total order needed by LRU management; they are last-in-first-out queues. For each cache set, the head of its replacement queue is the least recently used cache line.

Definition 4.8. Analysis augmentation `aug`(l) for memory reference l is:

```
(1)   t = line(l, B)
(2)   f = set(t, N)
(3)   if not incache(t) then
(4)     set_not_incache(remove_head(Q_f))
(5)     set_incache(t)
(6)   else
(7)     remove(Q_f, t)
(8)   end
(9)   insert_tail(Q_f, t)
```

In the augmentation, B is the line size, N is the number of cache sets, `line` returns the memory line referred to, `set` returns the cache set of the line, `remove_head` removes the head of a queue and returns it, `insert_tail` inserts a cache line to the tail of a queue, and `remove` removes a cache line from the inside of a queue.

Definition 4.9. Cache analysis instrumentation is the following:

1. A program P is replaced by a program `init` P `fin`, where `init` is a list of input statements that initialize data structures of `aug` (including those describing the parameters of the cache) and `fin` is a list of output statements that print measurement data.
2. Each statement `[`$var_1 = *var_2$`]` is replaced by `aug`(var_2)`[`$var_1 = *var_2$`]`.
3. Each statement `[`$*var_1 = var_2$`]` is replaced by `aug`(var_1)`[`$*var_1 = var_2$`]`.

Theorem 4.10 (exclusion of side-effects). The augmentation (Definitions 4.8 and 4.9) does not affect the original computation of the program.

Proof. The augmentations are probes. First, the augmentations are of the form $P_o \Rightarrow P_{op}P_oP_{of}$, where P_{op} = `init` and P_{of} is `fin` for augmentation 1, and P_{op} is empty and P_{of} = `aug` for augmentations 2 and 3. Second, the cells of the data used by `init`, `fin`, and `aug` are separate from cells of the original program. □

Theorem 4.11 (correctness). The augmentation (Definitions 4.8 and 4.9) computes the cache update U_C (Definition 4.6.) if `incache` is initially consistent with cache contents.

Proof. The augmentation handles each cache set separately. Thus, it computes the cache update U_C correctly if it computes the set update U_S correctly.

1. Assume that the line is already in cache. Mapping `incache` will be consistent with the queues also after the execution of `aug`, because lines 1–3, 7, and 9 will be executed, and they do not change `incache`. At line 9, the line is moved to the tail of the queue. Thus,

   $$U_S(s,t) = [r_1 \to t, r_i \to s(r_{i-1}), r_j \to s(r_j)]$$

   will hold after the execution of `aug`.
2. Otherwise, the line t is not in cache. Lines 1–5 and 9 are executed, because `incache`(t) is 0. After the execution, `incache`(t) is 1 (line 5) and t is at the tail of the queue (line 9). The head of the queue is removed and `incache` consistently updated (line 4). Thus,

   $$U_S(s,t) = [r_1 \to t, r_i \to s(r_{i-1})]$$

   will hold after the execution of `aug`.

The augmentation computes the cache update, because conditions of Definition 4.3 hold. □

Example 4.12 Consider again the program of Example 4.7.

```
init
aug(b)[y = *b] 1
aug(a)[x = *a] 2
if 3 (gt(x, y)) 3 then
  aug(c)[x = *c] 4 else [skip] 5 end
if 6 (gt(x, y)) 6 then
  aug(a)[y = *a] 7
else
  aug(d)[y = *d] 8
end
fin
```

The statements accessing memory have been augmented with preceding code `aug`. The whole program has been augmented with preceding code `init` and following code `fin`. This instrumentation simulates cache hit and misses. Computation of the original program remains the same.

4.4 Conclusions

In this chapter, we described how a dynamic cache analyzer can be built. We did not give an explicit implementation. Instead, we described the analyzer on an abstract level. In defining cache states, we followed [47] in our formulation.

The initialization `init` will input at least the cache parameters N, A, and B. For many applications, the cache can be assumed to be cold when the simulation starts. For such a purpose, an implementation of our augmentation must be able to handle queues that contain elements describing $\diamond$ (i.e., an empty cache line).

Sometimes it can be useful to begin a simulation with a warm cache. For such a purpose, the initialization `init` must input the contents of the cache and set the replacement queues correspondingly.

The finalization `fin` will typically output some metrics describing cache hits and cache misses. An implementation of our abstract augmentation must collect such information to data structures of the instrumentation. To point out reasons for hits and misses, line number information etc. is typically needed.

Thus, in addition to input and output of the original subject program, an instrumented program has input and output that is specific for the instrumentation. The transformation described in this chapter will modify a program computing $[\![P]\!]\langle I\rangle = O$ into a program computing $[\![P^A]\!]\langle I, I^+\rangle = \langle O, O^+\rangle$ as described in step 1 of our abstract framework.

Typically, the individual augmentations themselves do not output anything. Such output would make a cache simulator very slow. Even with a fast implementation of our augmentation, an execution-driven simulator is several times slower than the original program. In Chapter 6, we discuss how a fast analyzer can be compiled.

Chapter 5

Static cache analysis

Our compilation phase consists of three steps: instrumentation, partial evaluation, and slicing. The last two steps are static program transformations that need static information of program behavior.

In the previous chapter, a program instrumentation for dynamic cache analysis was described. This chapter describes, how a similar analysis can be performed statically. We divide our static cache analysis into two parts: address analysis and cache state analysis. We describe the state analysis in detail. For the address analysis, we give only an interface description.

We use abstract interpretation to derive abstract cache states from the concrete cache states of Section 4.2. Each abstract cache state describes a set of concrete cache states. To determine references that always hit and references that always miss, we describe both *must* and *may* analysis of abstract cache states.

5.1 Address analysis interface

The use of dynamic addressing makes cache analysis difficult. Because of the dynamic addressing, we do not know the absolute memory addresses of data cells statically. At a reference, we cannot tell the memory line (absolute address) that is accessed or its cache set.

However, knowing the concrete locations (absolute addresses) and cache sets is not necessary in our analysis. It is sufficient to know about the cache conflicts and the cache aliases statically. We use abstract locations to describe cache conflicts and cache aliases.

Definition 5.1 (abstract location). An *abstract location* $\hat{l}$ is a symbolic data item that is pointed by a non-empty subset of variables of a program. We denote the set of abstract locations by $\hat{L}$.

Each abstract location $\hat{l}$ can describe a set of concrete locations, which we denote by $locs(\hat{l})$. Locations are stored on memory lines. In program state $m \in M$, we denote the concrete memory line of $\hat{l}$ by $lin_m(\hat{l})$.

We use two kinds of rules: conflict and alias rules. The alias rule states the abstract locations that belong to the same cache line. The conflict rule states the abstract locations that can replace each other in the cache.

Statically, we do not know how the abstract locations are placed in the concrete memory. We can get lower and upper bounds for abstract locations that are cache conflicts or cache aliases. We define *must* and *may* rules (and sets) respectively.

Definition 5.2 (abstract cache alias). Two abstract locations $\hat{l}_1, \hat{l}_2$ are *must* cache aliases, if and only if

$$\forall l_1 \in locs(\hat{l}_1), l_2 \in locs(\hat{l}_2) : line(l_1) = line(l_2)$$

where $line(l)$ means memory line address of l (see Section 4.2). They are *may* cache aliases, if and only if

$$\exists l_1 \in locs(\hat{l}_1), l_2 \in locs(\hat{l}_2) : line(l_1) = line(l_2)$$

Locations that are cache aliases cannot replace each other in a cache. Other locations sharing a common cache set compete over the cache space available in the set.

Definition 5.3 (abstract cache conflict). Two abstract locations $\hat{l}_1, \hat{l}_2$ are in a *must* cache conflict, if and only if they are not *may* cache aliases and

$$\forall l_1 \in locs(\hat{l}_1), l_2 \in locs(\hat{l}_2) : set(line(l_1)) = set(line(l_2))$$

They are in a *may* cache conflict, if and only if they are not *must* cache aliases and

$$\exists l_1 \in locs(\hat{l}_1), l_2 \in locs(\hat{l}_2) : set(line(l_1)) = set(line(l_2))$$

The set of abstract locations in *must* and *may* cache alias with $\hat{l}$ are denoted $k_S^{\cap}(\hat{l})$ and $k_S^{\cup}(\hat{l})$, respectively. The set of abstract locations in *must* and *may*

conflicts with $\hat{l}$ are denoted $k_C^{\cap}(\hat{l})$ and $k_C^{\cup}(\hat{l})$, respectively.

5.2 Cache state analysis

In our static cache analysis, we use abstract cache states instead of the concrete cache states defined in Chapter 4. Each abstract cache state describes a set of concrete cache states. Our abstract cache states are mappings of abstract locations to ages (in the sense of a LRU ordering).

Definition 5.4 (abstract cache state). An abstract cache state $\hat{c}$ maps abstract locations to cache set positions $\hat{c} : \hat{L} \rightarrow \{1, \ldots, A, \top\}$. $\hat{C}$ denotes the set of all abstract cache states.

Static analysis is approximative. Without running a program with a specific input we cannot tell the cache states. Instead, we can compute bounds for possible cache states at a given program point. In *must* analysis, we determine a set of abstract locations that definitely are cached. In *may* analysis, we determine a set of abstract locations that can be cached.

We will use the same notation for both analyses. In the cases where the analyses need to be handled differently, we mark our symbols for *must* analysis with $^{\cap}$ and *may* analysis with $^{\cup}$. Otherwise we use the generic symbols.

In both analyses, we have a unique least element $\bot_{\hat{C}}$. In *must* analysis, it consists of minimum ages ($\{\hat{l} \rightarrow 1 \mid \hat{l} \in \hat{L}\}$). In *may* analysis, it consists of maximum ages ($\{\hat{l} \rightarrow \top \mid \hat{l} \in \hat{L}\}$).

The meaning of abstract cache states is given by a concretization function $conc_{\hat{C}} : \hat{C} \rightarrow 2^C$.

Definition 5.5 (cache concretization). For *must* analysis $conc_{\hat{C}}$ is:

$$conc_{\hat{C}}^{\cap}(\hat{c}) = \{c \mid c \in C \wedge \forall m \in M : (\forall \hat{l} \in \hat{L}, 1 \leq j \leq N, 1 \leq i \leq A : c(f_j)(r_{ji}) = lin_m(\hat{l}) \Rightarrow \hat{c}(\hat{l}) \geq i)\}$$

For *may* analysis $conc_{\hat{C}}$ is:

$$conc_{\hat{C}}^{\cup}(\hat{c}) = \{c \mid c \in C \wedge \forall m \in M : (\forall \hat{l} \in \hat{L}, 1 \leq j \leq N, 1 \leq i \leq A : c(f_j)(r_{ji}) = lin_m(\hat{l}) \Rightarrow \hat{c}(\hat{l}) \leq i)\}$$

The definition means that in *must* analysis the abstract age $\hat{c}(\hat{l})$ is a maximum limit for concrete ages i. In *may* analysis it is a minimum limit for concrete ages i. Note that $\top > A$ and $\hat{c}(\hat{l})$ can be $\top$.

The function $abs_C : 2^C \to \hat{C}$ gives the abstract state that describes a set of concrete states.

Definition 5.6 (cache abstraction). For *must* analysis abs_C is:

$$\begin{aligned} abs_C^{\bigcap}(X) = \{\hat{l} \to n \mid \forall m \in M, \hat{l} \in \hat{L}, c \in X : (\exists i, j : c(f_j)(r_{ji}) = lin_m(\hat{l})) \wedge \\ n = max\{i \mid \forall i, j : c(f_j)(r_{ji}) = lin_m(\hat{l})\}\} \bigcup \\ \{\hat{l} \to \top \mid \exists m \in M, \hat{l} \in \hat{L}, c \in X : (\nexists i, j : c(f_j)(r_{ji}) = lin_m(\hat{l}))\} \end{aligned}$$

where $1 \leq i \leq A, 1 \leq j \leq N$. For *may* analysis abs_C is:

$$\begin{aligned} abs_C^{\bigcup}(X) = \{\hat{l} \to n \mid \exists m \in M, \hat{l} \in \hat{L}, c \in X : (\exists i, j : c(f_j)(r_{ji}) = lin_m(\hat{l})) \wedge \\ n = min\{i \mid \forall i, j : c(f_j)(r_{ji}) = lin_m(\hat{l})\}\} \bigcup \\ \{\hat{l} \to \top \mid \forall m \in M, \hat{l} \in \hat{L}, c \in X : (\nexists i, j : c(f_j)(r_{ji}) = lin_m(\hat{l}))\} \end{aligned}$$

The definition means that for each concrete location we have some abstract location that is mapped to a bound. In *must* analysis, the bound is the maximum age ($\top$ if not cached in some concrete state). In *may* analysis, the bound is the minimum age ($\top$ if not cached in all concrete states).

Although we use the same representation for *must* and *may* analysis, their semantics are different because of the different concretization and abstraction functions. For example, in *may* analysis, $\top$ means "a cache miss will happen", but, in *must* analysis, it means "we do not know anything".

Abstract cache states are different at different points of a source program. By applying the update function, we get the abstract cache state at the point after a statement, if we know the abstract cache state at the point before the statement.

Definition 5.7 (abstract update). The abstract cache state update is given by function $U_{\hat{C}} : \hat{C} \times \hat{L} \to \hat{C}$. For *must* analysis it is:

$$U_{\hat{C}}^{\bigcap}(\hat{c}, \hat{l}) = \begin{cases} \{\hat{l}_i \to 1, & \text{if } \hat{l}_i \in k_S^{\bigcap}(\hat{l}) \\ \hat{l}_i \to \hat{c}(\hat{l}_i) + 1, & \text{else if } \hat{l}_i \in k_C^{\bigcup}(\hat{l}) \wedge \hat{c}(\hat{l}_i) < \hat{c}(\hat{l}) \wedge \hat{c}(\hat{l}_i) < A \\ \hat{l}_i \to \top, & \text{else if } \hat{l}_i \in k_C^{\bigcup}(\hat{l}) \wedge \hat{c}(\hat{l}_i) = A \wedge \hat{c}(\hat{l}) > A \\ \hat{l}_i \to \hat{c}(\hat{l}_i)\} & \text{otherwise} \end{cases}$$

For *may* analysis it is:

$$U_{\hat{C}}^{\bigcup}(\hat{c}, \hat{l}) = \begin{cases} \{\hat{l}_i \to 1, & \text{if } \hat{l}_i \in k_S^{\bigcup}(\hat{l}) \\ \hat{l}_i \to \hat{c}(\hat{l}_i) + 1, & \text{else if } \hat{l}_i \in k_C^{\bigcap}(\hat{l}) \wedge \hat{c}(\hat{l}_i) < \hat{c}(\hat{l}) \wedge \hat{c}(\hat{l}_i) < A \\ \hat{l}_i \to \top, & \text{else if } \hat{l}_i \in k_C^{\bigcap}(\hat{l}) \wedge \hat{c}(\hat{l}_i) = A \wedge \hat{c}(\hat{l}) > A \\ \hat{l}_i \to \hat{c}(\hat{l}_i)\} & \text{otherwise} \end{cases}$$

To get the maximum ages in *must* analysis, we mark as most recent (i.e., age 1) only the locations that definitely share the line referred to, and shift them to older only if they may be affected. To get the minimum ages in *may* analysis, we mark as most recent all locations that may share the line referred to, and shift them to older if they definitely are affected. We compare the ages, because accesses can increase the ages of only the locations that are younger in the cache than the location that is referred to.

In concrete cache semantics, having a cache update function was sufficient. In the abstract cache semantics, we must be able to handle the joining of abstract cache states, because branches of control can join in a program.

Definition 5.8 (abstract join). The abstract cache states are joined by function $J_{\hat{C}} : \hat{C} \times \hat{C} \to \hat{C}$. For *must* analysis it is:

$$J^{\cap}_{\hat{C}}(\hat{c}_1, \hat{c}_2) = \begin{cases} \{\hat{l}_i \to \top, & \text{if } \hat{c}_1(\hat{l}_i) = \top \vee \hat{c}_2(\hat{l}_i) = \top \\ \hat{l}_i \to max(\hat{c}_1(\hat{l}_i), \hat{c}_2(\hat{l}_i))\} & \text{otherwise} \end{cases}$$

For *may* analysis it is:

$$J^{\cup}_{\hat{C}}(\hat{c}_1, \hat{c}_2) = \begin{cases} \{\hat{l}_i \to \top, & \text{if } \hat{c}_1(\hat{l}_i) = \top \wedge \hat{c}_2(\hat{l}_i) = \top \\ \hat{l}_i \to \hat{c}_1(\hat{l}_i), & \text{if } \hat{c}_1(\hat{l}_i) \neq \top \wedge \hat{c}_2(\hat{l}_i) = \top \\ \hat{l}_i \to \hat{c}_2(\hat{l}_i), & \text{if } \hat{c}_1(\hat{l}_i) = \top \wedge \hat{c}_2(\hat{l}_i) \neq \top \\ \hat{l}_i \to min(\hat{c}_1(\hat{l}_i), \hat{c}_2(\hat{l}_i))\} & \text{otherwise} \end{cases}$$

The join operation induces a partial ordering $\sqsubseteq_{\hat{C}}$ for our abstract domain[1]:

$$\forall \hat{c}_1, \hat{c}_2 \in \hat{C} : \hat{c}_1 \sqsubseteq_{\hat{C}} \hat{c}_2 \Leftrightarrow J_{\hat{C}}(\hat{c}_1, \hat{c}_2) = \hat{c}_2$$

To prove the soundness of our analysis, we must prove the properties listed in Chapter 3. First, we prove properties of the domains. Then, we show the Galois injection. Finally, we prove the update functions. The proofs for the update functions in particular describe the thinking behind our analysis.

Proposition 5.9 (lattice properties). $(2^C, \subseteq, \cup, \emptyset)$ and $(\hat{C}, \sqsubseteq_{\hat{C}}, J_{\hat{C}}, \bot_{\hat{C}})$ are complete join semi-lattices, and the ascending chain condition holds.

Proof. For $(2^C, \subseteq, \cup, \emptyset)$ the proposition holds, because it is based on a power set. For $(\hat{C}, \sqsubseteq_{\hat{C}}, J_{\hat{C}}, \bot_{\hat{C}})$ the issue is more complicated.

[1]Note that because $J_{\hat{C}}$ is different for *must* and *may* analyses, the least element and the ordering are also different. However, for simplicity, we use the same notation for both cases, whenever possible.

1. The join operation is idempotent. This holds for both *must* and *may* analysis, because *max* and *min* are idempotent.

2. The join operation is commutative. For *must* analysis, this results from the commutativity of *max* operation and disjunction. For *may* analysis, this results from the commutativity of *min* operation and conjunction, and the symmetry of lines 2 and 3 in the definition of $J^{\cup}_{\hat{C}}$.

3. The join operation is associative. For *must* analysis, this results from the associativity of *max* operation and disjunction. For *may* analysis, this results from the associativity of *min* operation and conjunction, and from the observation that lines 2 and 3 in the definition of $J^{\cup}_{\hat{C}}$ cannot result $\top$.

4. There is a unit, because $\forall \hat{c} \in \hat{C} : J_{\hat{C}}(\bot_{\hat{C}}, \hat{c}) = \hat{c}$ for both *must* and *may* analysis.

5. All ascending chains eventually stabilize, because cache associativity is finite and the number of abstract locations is finite. □

Proposition 5.10 (monotony). The abstraction is monotone, i.e.:

$\forall \hat{c}_1 \sqsubseteq_{\hat{C}} \hat{c}_2 \in \hat{C} : conc_{\hat{C}}(\hat{c}_1) \subseteq conc_{\hat{C}}(\hat{c}_2)$ and
$\forall C_1, C_2 \subseteq 2^C : C_1 \subseteq C_2 \Rightarrow abs_C(C_1) \sqsubseteq_{\hat{C}} abs_C(C_2)$

Proof. Consider concretization functions. In *must* analysis, a greater or equal abstract state means that all age limits must be greater or equal (see Definition 5.8 that implies the ordering). In Definition 5.5, the greater the age limit i is, the more concrete states are included. Similarly in *may* analysis, a greater or equal abstract state means that all age limits must be less or equal, and the less the age limit is, the more concrete states are included. Therefore concretization functions are monotone.

Consider abstraction functions. The set C_2 includes all concrete states that the set C_1 includes. Therefore in *must* analysis, the age limit for C_2 cannot be less than for C_1, because maximum is selected. In *may* analysis, the age limit for C_2 cannot be greater than for C_1. Based on the *join* functions (Definition 5.8), $abs_C(C_1) \sqsubseteq_{\hat{C}} abs_C(C_2)$ for both cases. Therefore, abstraction functions are monotone. □

Proposition 5.11 (adjointness). The abstraction is strongly adjoint, i.e.:

$\forall C' \subseteq 2^C : C' \subseteq conc_{\hat{C}}(abs_C(C'))$ and
$\forall \hat{c} \in \hat{C} : \hat{c} = abs_C(conc_{\hat{C}}(\hat{c}))$

Proof. Consider the first part of the proposition and *must* analysis. Let $\hat{l}$ be any abstract location. $\hat{c}(\hat{l})$ will be the maximum of ages of its concrete locations in C'. If $C' \not\subseteq conc_{\hat{C}}(abs_C(C'))$, then C' would hold a state, where age of a concrete location of $\hat{l}$ is greater than the maximum. This is a contradiction. The same contradiction arises for *may* analysis with minimums. Thus, the first part of the proposition holds.

Consider the second part of the proposition and *must* analysis. $conc_{\hat{C}}(\hat{c})$ will hold all those concrete cache states, for which ages are less than $\hat{c}(\hat{l})$ for any abstract location $\hat{l}$. On the other hand, $conc_{\hat{C}}(\hat{c})$ is the maximum for these ages. The same is true with *may* analysis and minimums. Thus, the second part of the proposition holds. □

Proposition 5.12 (local consistency of must analysis). The concrete and abstract update function are locally consistent, i.e.:

$$\forall m \in M, c \in conc^{\cap}_{\hat{C}}(\hat{c}), \hat{l} \in \hat{L} : U_C(c, lin_m(\hat{l})) \in conc^{\cap}_{\hat{C}}(U^{\cap}_{\hat{C}}(\hat{c}, \hat{l}))$$

Proof. Let $\hat{l}$ be an abstract location. Consider the possible cache states before a reference. Definition 5.6 implies that $\hat{c}(\hat{l})$ will be greater than or equal to the age of any concrete location, for which $c \in conc^{\cap}_{\hat{C}}(\hat{c})$ holds. On the other hand, consider the possible cache states after a reference, and the memory line holding the location. Definition 5.5 tells us that all cache states where age of the memory line is less than or equal to $\hat{c}(\hat{l})$ are included.

Thus, to prove the proposition, we must show that the new age of any memory line is less or equal than the new age limits for all abstract locations that may have a concrete location on it.

Consider a memory line t of a location l. $U_C(c, t)$ will modify only the cache set that t belongs to (Definition 4.5). Three things can happen:

1. The memory line t holds several locations. Line 1 of $U^{\cap}_{\hat{C}}$ sets age to 1 only for those abstract locations that must share the line (i.e., some abstract locations may remain too old, but this is safe).

2. In the same set as t, there can be other memory lines that are younger than A. Ages of the locations on those lines are increased by one. Line 2 of $U^{\cap}_{\hat{C}}$ increases ages of all abstract locations that may share the set (i.e., some abstract locations may become too old, but this is safe).

3. The line t is not cached. Thus, the least recently used line is removed. Line of 3 $U^{\cap}_{\hat{C}}$ removes (sets their age $\top$) all the abstract locations that are marked least recently used (age is A) and may share the set. (i.e., some abstract locations still in cache may be marked $\top$, but this is safe).

The ages of other locations will remain the same. Because $U^{\cap}_{\hat{C}}$ does not change the limits for other cases (line 4 of $U^{\cap}_{\hat{C}}$), these age limits are also correct. Thus, if $c \in conc^{\cap}_{\hat{C}}(\hat{c})$ then $U_C(c,t)$ will be in the set of updated states. □

Considering *may* analysis is similar to *must* analysis.

Proposition 5.13 (local consistency of may analysis). The concrete and abstract update function are locally consistent, i.e.:

$$\forall m \in M, c \in conc^{\cup}_{\hat{C}}(\hat{c}), \hat{l} \in \hat{L} : U_C(c, lin_m(\hat{l})) \in conc^{\cup}_{\hat{C}}(U^{\cup}_{\hat{C}}(\hat{c}, \hat{l}))$$

Proof. To prove the proposition, we must show that the new age of any memory line is greater than or equal to the new age limits for all abstract locations that may have a concrete location on it.

Consider a memory line t of a location l. $U_C(c,t)$ will modify only the cache set that t belongs to (Definition 4.5). Three things can happen:

1. The memory line t holds several locations. Line 1 of $U^{\cup}_{\hat{C}}$ sets age to 1 for all those abstract locations that may share the line.

2. In the same set as t, there can be other memory lines that are younger than A. Ages of the locations on those lines are increased by one. Line 2 of $U^{\cup}_{\hat{C}}$ increases ages of only those abstract locations that must share the set.

3. The line t is not cached. Thus, the least recently used line is removed. Line 3 $U^{\cup}_{\hat{C}}$ removes (sets their age $\top$) only those the abstract locations that are marked least recently used (age is A) and must share the set.

The ages of other locations will remain the same. Because $U^{\cup}_{\hat{C}}$ does not change the limits for other cases (line 4), these age limits are also correct. Thus, if $c \in conc^{\cup}_{\hat{C}}(\hat{c})$ then $U_C(c,t)$ will be in the set of updated states. □

Now finally, it is simple to prove the soundness.

Theorem 5.14 (soundness). The *must* and *may* analyses (Definitions 5.4–5.8) are sound.

Proof. Because Propositions 5.9–5.13 hold, the analyses fulfill the conditions given by Theorem 3.12, and thus are sound. □

Example 5.15. Consider again Examples 4.7 and 4.12.

```
[y = *b] 1
[x = *a] 2
if3 (gt(x, y)) 3 then
  [x = *c] 4 else [skip] 5 end
if6 (gt(x, y)) 6 then
  [y = *a] 7
else
  [y = *d] 8
end
```

Consider the state of the cache before statement 6. In our *must* analysis (with $\hat{L} = \{\hat{a}, \hat{b}, \hat{c}, \hat{d}\}$), the abstract state is $\{(\hat{a}, 2), (\hat{b}, \top), (\hat{c}, \top), (\hat{d}, \top)\}$, i.e., the location pointed by a must be in the cache: its maximum age is 2. In our *may* analysis, the abstract state is $\{(\hat{a}, 1), (\hat{b}, 2), (\hat{c}, 1), (\hat{d}, \top)\}$, i.e., three locations may be cached. Thus, we know that the memory reference a at statement 7 is always a hit and the memory reference d at statement 8 is always a miss.

5.3 Conclusions

In this chapter, we described a static cache performance analysis. The analysis is based on abstract interpretation. Instead of concrete values, abstract interpretation uses abstract values (i.e., descriptions of values), which are safe approximations of possible concrete values.

Crucial for our analysis is that we do not try to understand the whole execution state of the subject program. We concentrate only on the abstract cache state. We do not solve the concrete locations that the cache contains.

We described two analyses: a *must* analysis and a *may* analysis. The *must* analysis describes the locations that definitely are in the cache. Using the analysis, we can detect references that must always hit. The *may* analysis describes the locations that can be in the cache. Using the analysis, we can detect references that must always miss. If a reference is to an abstract location that is not in the *must* set (*must* age is $\top$) and is in the *may* set (*may* age is not $\top$), then we cannot classify it.

Accuracy of the static analysis described is dependent on the address data that we get. For the unclassified references, we need dynamic analysis. In the next chapter, we describe how static cache behavior information can be used to build cache simulators that solve the remaining references.

Chapter 6

Program specialization

This is the last chapter describing our three-step method for cache analysis. In the previous chapters, we described how a subject program can be instrumented to form a dynamic cache analyzer, and how the cache behavior of a subject program can be statically analyzed. In this chapter, we describe how a dynamic cache analyzer can be made fast and compact by using program specializations. We will use the word *augmentation* to mean a single addition of code and the word *instrumentation* to mean the measuring system that results from all the augmentations.

We use two program specializations. First, we partially evaluate the dynamic analyzer. Second, we slice the resulting dynamic analyzer. Both steps are based on the static analysis of the previous chapter. In the first step, we propagate static information forward in the dynamic analyzer. In the second step, we propagate the static information backward in the dynamic analyzer.

Partial evaluation and program slicing can be used to solve a number of problems. They are generic methods. In a broad sense, partial evaluation can be applied to a program if we know enough of its task in advance. Program slicing can be applied to a program, if we know that some values in a program are not needed.

There are two applications for program specialization in speeding up cache analysis. We can apply the program specializations to the original subject program code and its instrumentation. In the following, we give an overall description of how the code of a subject program can be specialized. We will handle the specialization of the instrumentation in detail.

The structure of this chapter is as follows: We start by describing partial evaluation in general. After the generic description, we apply a simple partial evaluation to the augmentations described in Section 4.3 to yield faster

code. Then, we discuss program structure representations and program slicing in general. After that, we apply a specialized form of program slicing to the augmentations.

6.1 Partial evaluation

Partial evaluation [82] is a program transformation that is given a subject program P with part of its input data, I_1. It constructs a new program P_{I_1} that, when given the remaining input I_2, will yield the same result that P would have produced given both inputs. Let *peval* denote the partial evaluator, then for all I_1 and I_2:

$$(\llbracket peval \rrbracket \langle P, I_1 \rangle = P_{I_1}) \Rightarrow (\llbracket P \rrbracket \langle I_1, I_2 \rangle = \llbracket P_{I_1} \rrbracket \langle I_2 \rangle)$$

Partial evaluation is a generic method. Other forms of partial evaluation can be defined in addition to the one above.

The theoretical basis for partial evaluation was formulated by Kleene [91], but obviously not with the intention of improving programs. Lombardi [103] was probably the first to use the name. The theory was later refined by Futamura [54] and Ershov [45]. There has been advanced methods for various versions of partial evaluation, e.g., polyvariant specialization [23] and arity raising [113].

Our application of partial evaluation is simple compared to advanced partial evaluation methods. It is performed during compilation of a source program. Such partial evaluation is usually based on *binding-time analysis* [83], which divides program elements of P into two categories: static and dynamic. Static program elements are evaluated during the partial evaluation and the dynamic elements remain to be executed later.

Optimizing compilers typically apply simple forms of partial evaluation. Typical partial evaluations performed by compilers are constant folding and jump optimizations. Constant folding evaluates program elements yielding constant results during compilation. Jump optimizations compute jump addresses at compile-time.

6.2 Partially evaluated instrumentation

The static cache analysis of the previous chapter can be used as binding-time analysis for our analysis instrumentation. Using the static cache analysis, we can classify memory references into three classes:

- those that always hit,
- those that always miss, and
- those that we cannot decide about.

Consider our analysis augmentation from Section 4.3. Our analysis input contains the values N and B. If we specialize our program for an input, then we can replace the variables with the values that they are given. Therefore, we use a faster operation $\texttt{line}_B(l)$ instead of $\texttt{line}(l,\ B)$, and a faster operation $\texttt{set}_N(l)$ instead of $\texttt{set}(l,\ N)$. We denote the improved augmentation by $\texttt{aug}^{any}$.

```
(1)   t = line_B(l)
(2)   f = set_N(t)
(3)   if not incache(t) then
(4)     set_not_incache(remove_head(Q_f))
(5)     set_incache(t)
(6)   else
(7)     remove(Q_f, t)
(8)   end
(9)   insert_tail(Q_f, t)
```

Because of static cache analysis, we may statically know the value of the condition on line 3. Thus, we use the following augmentation $\texttt{aug}^{miss}$ at miss references.

```
(1)   t = line_B(l)
(2)   f = set_N(t)
(4)   set_not_incache(remove_head(Q_f))
(5)   set_incache(t)
(9)   insert_tail(Q_f, t)
```

At hit references, we use the following augmentation $\texttt{aug}^{hit}$:

```
(1)   t = line_B(l)
(2)   f = set_N(t)
(7)   remove(Q_f, t)
(9)   insert_tail(Q_f, t)
```

Definition 6.1 (partial evaluation). A cache analyzer (Definition 4.8) is partially evaluated by replacing each augmentation aug for:

- $\texttt{aug}^{hit}$ if the reference always hits.
- $\texttt{aug}^{miss}$ if the reference always misses.
- $\texttt{aug}^{any}$ otherwise.

Augmentation init is replaced by augmentation $\texttt{init}_{NAB}$ that initializes the cache state, i.e., it assigns initial values to *incache* and the replacement queues.

Theorem 6.2 (correctness). The partially evaluated analyzer yields the same analysis result as the original, if the analysis input for the original and the initialization of the partially evaluated analyzer are equivalent.

Proof. Based on Theorem 3.15, the whole program must compute the same result for the remaining input, if each partially evaluated augmentation computes the same function as the original. $\texttt{aug}^{miss}$ and $\texttt{aug}^{hit}$ compute the same function as the original augmentation, if the classification of references is correct. Theorem 5.14 tells us that references classified as hits always hit and references classified as misses always miss. As $\texttt{aug}^{any}$ works for both hits and misses, our classification is correct. □

6.3 Program structure

In static program analysis, various graphs are often used to describe program structure. We use two structure descriptions: control flow graphs and dependence graphs. Control flow graphs describe how program parts relate to each other, when their execution order is considered. Dependence graphs describe how program parts relate to each other, when their computation is considered.

A *directed graph* $G = (V, E)$ consists of a set of *vertices* V and a set of edges E, where $E \subseteq V \times V$. We say that v is the *source* of the edge $(v, w) \in E$ and w is *target*, and mark that $v \rightarrow w$. Further, we call w an immediate successor of v and v an immediate predecessor of w.

A *path* is a non-empty sequence of vertices $v_1, ..., v_n$ so that

$$\forall\, 1 \leq i < n : v_i, v_{i+1} \in V \wedge (v_i, v_{i+1}) \in E.$$

Vertex w is a successor of v and v predecessor of w, if there is a path starting from v and ending to w.

Definition 6.3 (control flow graph). A control flow graph is a quadruple $CFG = (V, E, S, T)$, where V is a set of vertices, E is a set of edges, $S \in V$ is an entry vertex with no predecessors, and $T \in V$ is an exit vertex with no successors.

In our analysis, we restrict our attention to control flow graphs, in which

- any vertex has at most two immediate successors,
- any vertex is a successor of the entry vertex and
- the exit vertex is a predecessor from any vertex.

If a vertex has two immediate successors, we call one of the edges the `true`-*branch* (denoted by $\overset{\texttt{true}}{\rightarrow}$) and we call the other the `false`-*branch* (denoted by $\overset{\texttt{false}}{\rightarrow}$).

A control flow graph is formed in the following way. For each operation a in a subject program, there is a vertex v_a in its control flow graph. If operation a can be immediately followed by operation b in an execution, then there is an edge from v_a to v_b. Thus, a control flow graph describes all possible flows of control in a program.

Definition 6.4 (postdominance) Vertex w of a control flow graph *postdominates* an other vertex v of the control flow graph, if and only if all paths from v to the end of the control flow graph (i.e., T) go through w.

Let L be `true` or `false`. Vertex w *postdominates* the L-branch of vertex u, if and only if

$$(u \overset{L}{\rightarrow} w) \in E \bigvee (\exists v \in V : \exists (u \overset{L}{\rightarrow} v) \in E \wedge w \textit{ postdominates } v)$$

Definition 6.5 (control dependence). Let $v, w \in V$. Vertex w is *directly L-control dependent* on v (written $v \rightarrow_c^L w$), if and only if w postdominates the L-branch of v and w does not postdominate v.

Definition 6.6 (data dependence). Vertex w is *directly data dependent* on v (written $v \rightarrow_f w$), if and only if vertex v assigns to value x, vertex w uses x, and there is a path from v to w that does not include any assignments to x.

The program dependence graph consists of vertices of the control flow graph, control dependence edges, and data dependence edges.

6.4 Program slicing

Program slicing is an operation that identifies semantically meaningful decompositions of programs. Slicing is performed by using a subset Y of program elements as a base. The subset Y is called the *slicing criterion*.

Program slicing is a generic method. It was originally proposed by [159]. We use the slicing approach that is based on dependence graphs [80]. Usually, two kinds of slices are identified:

- A *backward slice* of a program P with set of program elements Y consists of all program elements that might affect the values computed by Y.
- A *forward slice* of program P with set of program elements Y consists of all program elements that might be affected by the values computed by members of Y.

In our analysis, we compute a backward slice. The slicing criterion consists of input statements of the original program and output statements of the analysis instrumentation.

Let P be a program computing two values:

$$[\![P]\!]\langle I\rangle = \langle O_1, O_2\rangle$$

We use slicing to implement a program transformation that yields a program computing only O_1 from I. Let *slicer* be the transformation:

$$([\![slicer]\!]\langle P, Y\rangle = P_Y) \Rightarrow ([\![P_Y]\!]\langle I\rangle \stackrel{O_1}{=} [\![P]\!]\langle I\rangle \ \wedge \ [\![P_Y]\!]\langle I\rangle \stackrel{O_2}{=} \emptyset)$$

where $\stackrel{O_1}{=}$ denotes equality of output O_1 and $\stackrel{O_2}{=}$ denotes equality of output O_2. The slicing criterion Y lists the program elements that read I and produce O_1.

Program slicing is performed by analyzing relations between program elements. Program elements do computations by using values to define new ones or to control program flow. If a value or flow of control is not used, then the elements defining the value or controlling the flow can be deleted.

Optimizing compilers typically apply simple forms of program slicing, e.g., dead code elimination. Dead code elimination removes program elements that compute unused values. Analyzing control dependencies caused by jumps and especially subroutine calls is more complex. Compilers rarely do such an analysis.

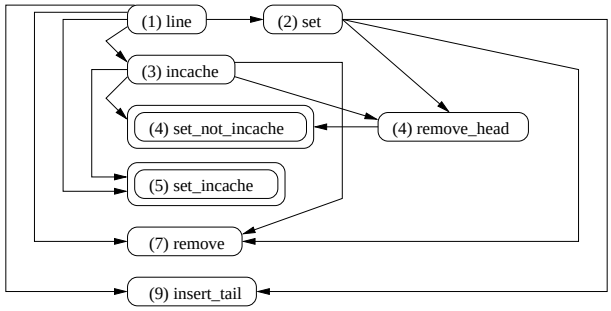
6.5 Sliced instrumentation

In our analysis, the slicing criterion consists of input statements of the original program and output statements of the analysis instrumentation. In this section, we consider only the instrumentation code, and therefore, only analysis output.

The augmentations in Section 6.2 do not contain any explicit output operations. However, for our analysis purposes, the analysis output is dependent on the cache contents, which are described by the mapping *incache* in our augmentation. Therefore, the operations modifying the contents of *incache* are the slicing criterion for our instrumentation. These operations are *set_not_incache*

(on line 4) and *set_incache* (line 5).

Other operations in an augmentation are needed only if our slicing criterion is directly or indirectly dependent on *set_not_incache* or *set_incache*. The dependence arcs inside an augmentation are illustrated below. Augmentation line numbers are in parentheses, and the members of the slicing criterion are encircled by double lines.



The cache set computation (line 2) has always a path of data dependence arcs to *set_not_incache* (line 4). The branching (line 3) has always control dependence arcs to the same operation. The cache set computation (line 2) has data dependence arc from cache line computation (line 1). Thus, those operations cannot be sliced if *set_not_incache* (line 4) is present.

However, operations *remove* (line 7) and *insert_tail* (line 9) have no such dependencies in a single augmentation. Only the state of the replacement queue Q_f is dependent on *remove* and *insert_tail*. The state of the queue is observed by *remove_head* (line 4), which has data dependence arc to *set_not_incache*. Thus, *remove* and *insert_tail* can have path of data dependence arcs to some following *set_not_incache* operations.

Consider a partially evaluated cache simulator. Each hit reference has the augmentation aug^{hit}. The instrumentation removes a cache line from the replacement queue. If we know that a reference is always a hit, then it must be preceded by a reference that places the cache line in the replacement queue Q_f. Thus, we have a pair of the form:

insert_tail(Q_f, t)
...
remove(Q_f, t)

Actually, because of the following hit, there is no need to insert the cache line in the replacement queue. We know that the inserted line will never reach

the head of the queue. Thus in such a situation, no *remove_head* operation has data dependence arc from the insertion and removal. Thus, the insertion and the related removal can be sliced away.

Compared to ordinary slicing, this is more complicated. Instead of slicing single operations (i.e., nodes in the flow graph), we are slicing several operations as a group. We cannot slice only insertions, because removals would have nothing to remove. On the other hand, we cannot slice only removals, because replacement queues would contain duplicates of the same line at erroneous positions.

Therefore, we must define their dependencies, and act accordingly. For clarity in the following text, we use the word *observer* to denote the operation *remove_head*, the word *insertion* to denote the operation *insert_tail*, and the word *removal* to denote the operation *remove*.

Definition 6.7 (queue head dependence) An insertion of line t to queue $Q_{set(t)}$ has dependence arc to an observer that may observe queue $Q_{set(t)}$, if

- there is a path from the insertion to the observer that may have no insertions of line t to queue $Q_{set(t)}$, and
- cache line t can be at the head of $Q_{set(t)}$ at the point preceding the observer.

The subject program code does not address cache lines. Instead, it uses abstract locations $\hat{l}$, from which lines are computed at run time (by operation *line* in our augmentation). Cache line t can be at the head of $Q_{set(t)}$, if the cache age of the corresponding abstract location $\hat{l}$ can be A, i.e., $\hat{c}^{\cap}(\hat{l}) \leq A \leq \hat{c}^{\cup}(\hat{l})$.

Definition 6.8 (queue tail dependence) An insertion of line t has dependence arcs from and to a removal that must remove line t, if there is a path from the insertion to the removal and the path has no insertions or removals of t. Such dependent insertions and removals form a group.

Our static cache analysis is not based on exact address data. Instead, we know only the conflicts and cache aliases between the addresses. This may yield aliasing, i.e., two or more operations that statically seem to use separate *incache* flags (and replacement queues) are actually using the same flag (and the same replacement queue).

Slicing a group of insertions and removals causes a pending insertion. The line to be inserted is marked *incache*, but is not yet inserted in the corresponding replacement queue. Because of aliasing, we can have several insertions pending. Consider the following example:

$\mathtt{aug}^{any}(a)$
...
$\mathtt{aug}^{any}(b)$
...
$\mathtt{aug}^{hit}(a)$
...
$\mathtt{aug}^{hit}(b)$

Assume that the insertion at $\mathtt{aug}^{any}(a)$ is pended until $\mathtt{aug}^{hit}(a)$ and the insertion at $\mathtt{aug}^{any}(b)$ is pended until $\mathtt{aug}^{hit}(b)$. If a and b are cache aliases, then the memory line of a and b is inserted twice in the replacement queue, because $\mathtt{aug}^{hit}$ does not check the cache state.

Because of such aliasing, we modify our augmentation to use a counter of pending insertions instead of a simple flag.

Definition 6.9 (insertion counter) In our augmentations, *incache* is a table of counters that is used in the following way:

- *set_incache*(t) increments *incache*(t) by two if it is zero and by one otherwise.
- *set_not_incache*(t) sets *incache*(t) to zero.
- *insert_tail*$(Q_{set(t)}, t)$ decrements *incache*(t) by one, and inserts t only if *incache*(t) is initially two.
- *remove*$(Q_{set(t)}, t)$ increments *incache*(t) by one, and removes t only if *incache*(t) is initially one.

Initially *incache*(t) is zero for all t not in cache, and one for all t in cache.

Without slicing, our augmentations work as previously (i.e., any *incache* counter is 0 or 1 between the augmentations). If the instrumentation of a program is sliced, then *incache* can contain counter values greater than 1 between augmentations.

Theorem 6.10 (correctness of insertion counter) The augmentation of Definition 6.9 computes the same result as the augmentation of Definition 6.1.

Proof. We assume an invariant that between augmentations, *incache*$(t) = 0$ if t is not in cache, and *incache*$(t) = 1$ if t is in cache. We must consider all our three augmentations.

$\mathtt{aug}^{hit}$ According to our invariant, *incache*$(t) = 1$ before the removal (line 7). Thus, the removal will happen, and *incache*$(t) = 2$ before the insertion. Because of that, the insertion will happen, and *incache*$(t) = 1$ after the

augmentation.

aug^{miss} We must consider two lines: t and the line that will be removed. According to our invariant, $incache(t) = 0$ before *set_incache*. Thus, it will be set to 2 and the insertion will happen. According to our invariant, *incache* of the line to be removed is one. After *set_not_incache* it will be zero. Therefore, the invariant will hold for both lines after the augmentation.

aug^{any} If $incache(t) = 0$, the augmentation works like aug^{miss}, otherwise it works like aug^{hit}. □

Now, we are finally ready to define our slicing, and prove that it yields simulators that correctly simulate caches. Let P be a simulator, and P_Y the corresponding sliced simulator.

Definition 6.10 (sliced instrumentation). A group of insertions and removals is deleted from the simulator if there are no dependencies to the members of the group from the outside of the group.

Theorem 6.11 (correctness of slicing). If and only if $incache(t) = 0$ in a simulation by P, $incache(t) = 0$ at the corresponding step in the simulation by P_Y.

Proof. The contents of *incache* depend on lines returned by *remove_head* (at line 4). To show the theorem, we must show that *remove_head* results in the same line in execution of P_Y as in execution of P. We assume the following invariants:

1. $pending(t) = incache(t) - 1$ if $incache(t) > 0$ and $pending(t) = 0$ otherwise,
2. a memory line t is in its queue $Q_{set(t)}$, if and only if $incache(t) = 1$, and
3. the ordering of any $Q_{set(t)}$ in the execution of P_Y is a subset of its ordering at the corresponding step in execution of the P,

where $pending(t)$ is the number of pending insertions of line t. Consider each augmentation:

aug^{miss} If *remove_head* (line 4) in execution of P results t, then in execution of P_Y the line t must be in $Q_{set(t)}$, because there must be an insertion (Definition 6.7 and Theorem 5.14) and the insertion cannot be pending (Definition 6.8 and Theorem 5.14). Because of invariant 3, t must be the head of $Q_{set(t)}$ in execution of P_Y. Therefore, *remove_head* in execution of P_Y results t.

If *insert_tail* (line 9) is deleted, then there is one more insertion pending after execution of the augmentation. The line t is not inserted in its queue.

If *insert_tail* (line 9) is not deleted, then the insertion of t will be done, if there are no insertions pending. Because insertion is done at the tail in both P and P_Y, invariant 3 will hold. If there are insertions pending, then no insertion will be done.

Thus, all invariants hold also after execution of the augmentation.

$\texttt{aug}^{hit}$ If *remove* (line 7) is deleted, then there is one less insertion pending after execution of the augmentation. Because of the deletion, incrementation of $incache(t)$ is also deleted.

If $remove(t)$ (line 7) is not deleted, then the removal of t will be done, if there are no insertions pending. If there are insertions pending, then no removal will be done.

Deletion of the insertion causes the same effects as in $\texttt{aug}^{miss}$. Thus, all invariants hold also after execution of the augmentation.

$\texttt{aug}^{any}$ If $incache(t) = 0$, the augmentation works like $\texttt{aug}^{miss}$, otherwise it works like $\texttt{aug}^{hit}$. □

Example 6.12 Consider again the program of Example 4.12.

```
init_NAB
aug^miss(b)[y = *b] 1
aug^miss(a)[x = *a] 2
if 3 (gt(x, y)) 3 then
  aug^miss(c)[x = *c] 4 else [skip] 5 end
if 6 (gt(x, y)) 6 then
  aug^hit(a)[y = *a] 7
else
  aug^miss(d)[y = *d] 8
end
fin
```

The augmentations have been replaced by faster ones. Note that the example is artificial. In a typical program, most of the references of the program are *hits*.

Example 6.13. Consider a miss followed by two hits. The instrumentation code resulting after partial evaluation is shown below on the left. Because of the first hit, the *insert_tail* operation of the miss and the *remove* operation of the first hit can be removed. Further, because of the second hit, the *insert_tail* operation of the first hit and the *remove* operation of the second hit can be removed. The resulting code is on the right.

(1)	t = line$_B$(l)	t = line$_B$(l)
(2)	f = set$_N$(t)	f = set$_N$(t)
(4)	set_not_incache(remove_head(Q$_f$))	set_not_incache(remove_head(Q$_f$))
(5)	set_incache(t)	set_incache(t)
(9)	insert_tail(Q$_f$, t)	
	...	...
(7)	remove(Q$_f$, t)	
(9)	insert_tail(Q$_f$, t)	
	...	
(7)	remove(Q$_f$, t)	
(9)	insert_tail(Q$_f$, t)	insert_tail(Q$_f$, t)

Nothing remains from the augmentation of the first hit. The cache line is marked to be in the cache from the beginning of the sequence, but inserted in its replacement queue at the end.

6.6 Conclusions

The program specializations described in this chapter affect a cache simulator in three ways: they reduce the number of steps to be executed, simplify the code by removing branching, and reduce the size of the code. Partial evaluation does not significantly reduce the number of steps, but it does reduce the size of the code and its complexity.

The effect of slicing may seem like a minor one. It affects a special case: unconditional hits. However, hits and sequences of hits are common in programs [40]. Programmers tend to write code that uses memory in a local manner. The structure of cache memories takes advantage of this. After an access to a specific cache set, there may be a number of other accesses. But, when the cache set is accessed again, the access is typically a hit – and it is followed by hits. Our slicing handles such chains of hits very efficiently: except for the first and the last in the chain, all augmentations are removed. Furthermore, augmentations for the first and the last are simplified.

In addition to the instrumentation, slicing can be applied to the source program. In our cache analysis, the original computation of the source program is

needed only to produce address data for the analysis. The rest of the computation can be removed. Further computations may become unnecessary because address data of many hit references are not needed.

Chapter 7

Experiments

Based on the method described in Chapters 3–6, we implemented a set of tools that were designed as a prototype for practical engineering work. We experimented with the tools to find out, whether partial evaluation and slicing that are guided by static cache analysis form a potential method to compile fast cache simulators. We discovered from our experiments that the method is sufficient to yield significant speed-up of simulations. In the following, we describe our tool and our experiments that were solely based on *must* analysis of abstract cache states.

We begin our description with two introductory sections. First, we describe the performance analysis method that we have used in our experiments. Second, we describe the tool set and give an example how the tool set has been applied in practice.

After the introductory section, we describe the experimental analysis that we did. We did two kinds of performance experiments, which we call static and dynamic experiments. We discuss the experimental setting and results of the experiments in separate sections.

7.1 Experimental method

In computer science and engineering, performance analysis has wide applicability. Most performance problems are unique. The metrics, workload, and evaluation techniques used for one problem generally cannot be used for the next problem. However, there are steps typically common to all performance analyses, and a variety of techniques that can be used during those steps [81]. We used the following steps in our analysis:

1. *Defining the system* under study and *stating the goals* of the study. What constitutes a system is usually defined by delineating system boundaries. A clear goal is needed not only to guide the study, but also in presenting the results: we must know the question in order to understand and judge the answer.

2. Each system provides a set of services; it has input and output. Usually the performance of a system depends on its input, and it gets feedback by giving output. To analyze the performance of a system, we must *identify the services* and the related inputs and outputs.

3. To state, compare, and discuss performance we need to select *metrics*. Typically, metrics are numeric indicators of performance. However, in general, they can be any indicators related to the speed, the size, the accuracy, the availability etc. of the system.

4. Many things affect the performance of a system. In performance analysis, they are called *parameters* of the system. Typically, parameters are divided into system parameters and workload parameters. The parameters to be varied are called *factors*.

5. An *evaluation technique* is needed for finding out the performance of a system. Typically, three broad categories of techniques are identified: mathematical analysis, simulation, and measuring.

6. For an evaluation, a *workload* must be selected. Workload is the input given to a system. There are many types of workloads, e.g., real workloads, synthetic workloads, and artificial workloads. Some workloads are based on analyzing and modeling a real workload, e.g., a random workload generator that is based on a probabilistic workload model.

7. *Evaluation* of the performance must be carried out. If a simulation or a measuring technique is selected, *experiments* must be designed and run.

8. Finally, the gathered data must be interpreted, conclusions drawn, and results presented.

Performance analyses are often performed in an iterative manner. The steps above are repeated until the analysis of a system provides the information needed. Such an iterative process typically starts with an initial analysis having generic goals and ends with an final analysis having specific goals.

7.2 Tools

Based on the method described in Chapters 3–6, a set of tools were designed and implemented as a prototype for practical engineering work. The tool set is called MSE (Memory Simulation Environment) [73].[1] It is designed for studying memory behavior of programs.

Using MSE, a subject program is compiled and instrumented to form a simulator with the run-time components of MSE. The size of the tool set is approximately 24600 lines of source code. MSE was designed and implemented by the author as a part of this research.

MSE is designed for general studies of memory behavior. Instead of considering some specific hardware, MSE uses an abstract machine, which we call SM (Simple Machine). SM is a register machine with a simple instruction set. The main feature of SM is that its memory system can be parameterized to conform with various memory configurations. In the context of cache analysis, only instructions addressing the memory are significant. SM is representative for a set of machines using the load/store architecture.

We used six tools of MSE in our experiments: SMC (Simple Machine Compiler), ISC (Instruction Simulator Compiler), MTC (Memory Trace Coder), MSC (Memory Simulation Coder), MSS (Memory Stack Simulator), and MEA (Memory Event Analyzer).

MSE is designed to be used with the GNU C/C++ programming environment. The components of the GNU environment that are needed to use MSE are: CCCP (GNU C-Compatible Compiler Preprocessor), GCC (GNU C/C++ Compiler), and ld (GNU linker).

Programs whose memory behavior is to be simulated are written in ANSI C [88]. SMC does not implement full ANSI C; some complicated language constructs[2] are omitted because we have not needed such constructs in our studies of program memory behavior.

In our experiments, we used MSE in two configurations: in the first, MSE implements a traditional memory simulator, while in the second, MSE implements a memory simulator based on the method described in Chapters 3–6. The first configuration is given in Figure 7.1 and the second in Figure 7.2.

The first two phases are common to both configurations. The user source code is processed with CCCP to handle macros, especially file inclusion. This results in a single file in which all textual macro substitutions are done. After this, the preprocessed file is given to SMC, which translates the file into SM code.

In the traditional approach, the SM code is given to MTC, which inserts

[1] An earlier version of MSE was called DBE [70].

[2] The omitted constructs consist mostly of some complex type specification features.

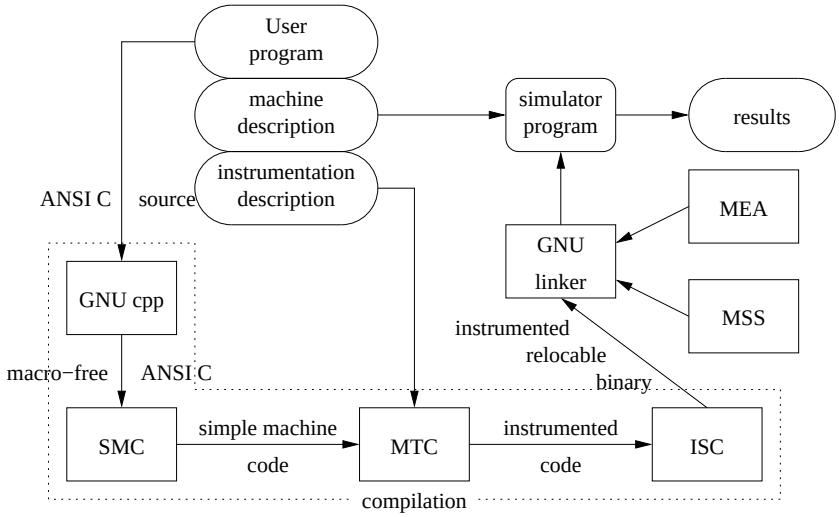


Figure 7.1: Trace-based simulation using MSE.

trace generating statements into the code, and compiles it into an execution-driven trace generator. ISC compiles this instrumented code to the relocatable binary code of the host machine. The compiled code is linked with MSS and MEA to form an executable simulation program that uses the instrumentation of Chapter 4.

During execution of the simulation program the instrumentation embedded in the user code generates an implicit memory access trace. The trace is fed to MSS, which simulates memory operations according to memory parameters that it is given. The simulation results in a sequence of memory events, which describe data transfers between different layers of memory, i.e., cache misses.

The memory event sequence is fed to MEA, which produces measurement results according to measurement requests that it is given. Measurement results are typically numeric values, e.g., they indicate how many cache misses occurred during a simulation.

In the combined analysis approach, the SM code is given to MSC, which constructs a memory simulator. The operation of MSC is guided by a machine description, which describes the memory configuration of SM. The resulting code is given to ISC, which compiles the code to relocatable binary code. Using ld, the code is linked with MEA to form an executable simulation program.

Based on the method described in Chapters 3–6, MSC omits simulation at references, the effects of which are statically known, and uses simplified simula-

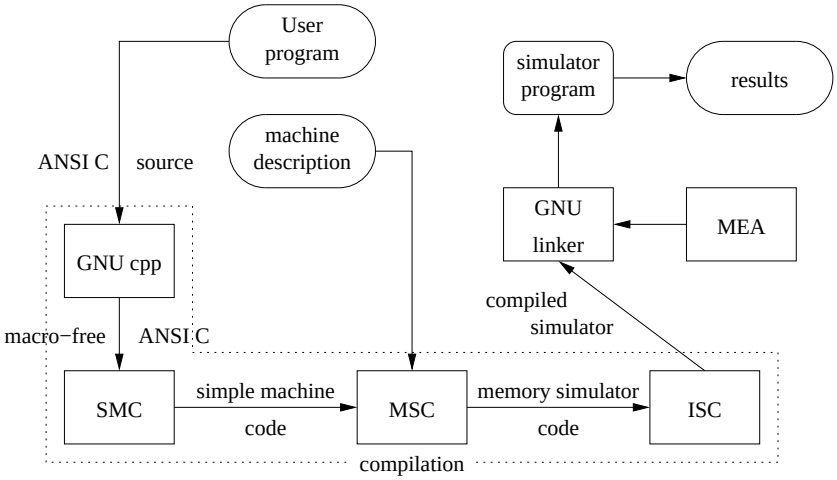


Figure 7.2: Combined simulation using MSE.

tion at references, the effects of which are partially known.

MSC does not implement the *may* analysis described in Chapter 5, because the *must* analysis alone is sufficient to yield significant speed up of simulation. Without the *may* analysis, cache misses cannot be statically found. Thus, the tool uses only the $\mathtt{aug}^{any}$ and $\mathtt{aug}^{hit}$ augmentations. Further, MSC sliced only the instrumentation code in our experiments. The subject program code was not sliced.

MSE uses the address analysis described in [72] to support the cache analysis. The analysis is a *must* cache alias analysis that is based on the value equivalence algorithm of [6]. To handle global analysis, MSC uses fix point iteration [118].

We give an example that demonstrates the use of MSE in a design process. The example is based on an actual case, where an earlier version of MSE was used by the author, and whose results have been published in [76].

Example 7.1. We describe the design of a memory layout scheme for a program that computes the full transitive closure of a binary relation in an environment that has a two-level memory. Transitive closure computation is a basic computational task. It is required, for instance, in the reachability analysis of transition networks representing distributed and parallel systems and in evaluating recursive database queries.

Consider a directed graph $G = (V, A)$, where V is the set of vertices and $A \subseteq V \times V$ is the set of arcs. The *transitive closure* of G is a graph $G^+ = (V, A^+)$

such that for all $v, w \in V$ there is an arc $(v, w) \in A^+$ if and only if there is a non-null path from v to w in G. The *successor set* of a vertex v is the set $Succ(v) = \{w \mid (v, w) \in A^+\}$. A *strong component* of G is a maximal subset $C \subseteq V$ such that for each $v, w \in C$ there is a path from v to w (and vice versa). A *topological order* of the vertex set V of a graph $G = (V, A)$ is any total order $\leq$ of V such that $v \leq w$ if arc $(v, w) \in A$.

The program is based on the algorithm COMP_TC [119], which was designed for operation in a homogeneous memory. Processing large amounts of data is characteristic to transitive closure algorithms. If the memory consists of layers of different speed, then the data transfer between the memory layers is typically the performance bottleneck of the computation. The amount of data transferred depends on the memory layout. Thus, a layout that is suitable for COMP_TC is needed.

The algorithm COMP_TC is given in Figure 7.3. COMP_TC uses Tarjan's algorithm [151] to detect the strong components of a graph. The algorithm uses two stacks: *nstack* to store vertices and *cstack* to store components. To construct the successor set of a strong component C, COMP_TC uses the components adjacent from C. These components are collected during a depth-first traversal of the input graph (lines 6–11 in Figure 7.3). COMP_TC scans the components, and for a component X checks whether X already is in $Succ(C)$. If it is not, then X and $Succ(X)$ are added into $Succ(C)$ (lines 16–19 in Figure 7.3). The component itself is constructed at lines 20–24 in Figure 7.3.

Based on the algorithm, a program designed for operation in a two-level memory was written [76]. The program is over 1000 lines of C code. An abstract description of the program is given in Figure 7.4. Because of the complexity and the number of memory references, it is practically impossible to understand the memory behavior of the actual program without automated tools. MSE is a useful tool for such tasks.

Despite the complexity, some overall observations and design decisions can be explained using the abstract program in Figure 7.4. Simulations with MSE reveal that traversing the input graph by recursion (line 6 in Figure 7.3) causes a significant proportion of the memory transfers. Therefore, instead of using recursion and local variables, the program uses a separate control stack to store depth-first-search paths and the adjacent vertices. When a vertex v is entered, all arcs leaving v are stored on top of the control stack (line 4 in Figure 7.4). If a suitable stack structure and alignment is used, MSE can statically guarantee that most of the related memory references are hits.

When this memory bottleneck is solved, simulations reveal an other memory bottleneck that is caused by accessing vertices and components in two different stacks (e.g., lines 4 & 10 and 17 & 21 in Figure 7.3). Storing the vertices and components in a single work stack (lines 2 and 11 in Figure 7.4) localizes

```
(1)   procedure COMP_TC(v);
(2)   begin
(3)     root(v) := v; C(v) := Nil;
(4)     PUSH(v, nstack);
(5)     hsaved(v) := HEIGHT(cstack);
(6)     for each vertex w such that (v, w) ∈ E do begin
(7)       if w is not already visited then COMP_TC(w);
(8)       if C(w) = Nil then root(v) := MIN(root(v), root(w))
(9)       else if (v, w) is not a forward edge then
(10)        PUSH(C(w), cstack);
(11)    end;
(12)    if root(v) = v then begin
(13)      create a new component C;
(14)      if TOP(nstack) = v then Succ(C) := ∅
(15)      else Succ(C) := {C};
(16)      while HEIGHT(cstack) ≠ hsaved(v) do begin
(17)        X := POP(cstack);
(18)        if X ∉ Succ(C) then Succ(C) := Succ(C) ∪ {X} ∪ Succ(X);
(19)      end;
(20)      repeat
(21)        w := POP(nstack);
(22)        C(w) := C;
(23)        insert w into component C;
(24)      until w = v
(25)    end
(26)  end;
(27)  begin /* Main program */
(28)    nstack := ∅; cstack := ∅;
(29)    for each vertex v ∈ V do
(30)      if v is not already visited then COMP_TC(v)
(31)  end.
```

Figure 7.3: Algorithm COMP_TC (adapted from [119]).

```
(1)   procedure VISIT(vertex v)
(2)     push v onto the work stack;
(3)     Root(v) := v;
(4)     fetch all arcs leaving v to buffer (push them onto the control stack);
(5)     for each arc (v, w) on the control stack do begin
(6)       if w is not visited then VISIT(w);
(7)       if w is not already a member of a component then
(8)         if Root(w) < Root(v) in the depth-first order then
(9)           Root(v) := Root(w)
(10)      else if arc (v, w) is not a forward arc then
(11)        push the component containing w onto the work stack;
(12)    end;
(13)    if Root(v) = v then begin
(14)      create a new component C;
(15)      for all items on the work stack between the top and vertex v do
(16)        if item is a vertex then move it into component C;
(17)      Succ(C) := if component C is cyclic then {C} else {};
(18)      sort the components that were above vertex v in the work stack
(19)        into a topological order and remove duplicates;
(20)      for each remaining component X in topological order do begin
(21)        pop X from the work stack;
(22)        if X ∉ Succ(C) then Succ(C) := Succ(C) ∪ Succ(X) ∪ {X}
(23)      end
(24)    end
(25)  end;
(26)  for each vertex v of the graph do /* main program */
(27)    if v is not visited then VISIT(v)
```

Figure 7.4: A variant of COMP_TC for hierarchic memories (adapted from [76]).

the memory references. A further improvement can be achieved by storing the components in a topological order. This process of analyses and improvements is continued towards the details of the program.

The result is a program that efficiently uses a two-level memory to support its computation. During each step of the design process, static analysis could be used both to guide and to speed up the simulations needed in understanding the memory behavior.

7.3 Experimental setting

In our performance analysis of the cache analysis method, we used the performance analysis method described in the first section. The following enumerated items correspond to the enumerated items of the first section.

1. It is clear that the performance of our method depends on the nature of the memory references in the subject program to be analyzed. The more statically resolvable the memory behavior of the subject program is, the better opportunity we have to partially evaluate and slice the corresponding simulator.

   The goal of this experimental study is to find out the simulation speed-up yielded by our implementation, when the proportion of statically classifiable[3] memory references is assumed known. In addition to this goal, we are interested in finding out the parameters that affect the performance of our method.

   Our primary goal is not to find out statistically confirmed bounds for performance. Rather, we are interested to know, whether partial evaluation and slicing that are guided by static analysis form a potential method to compile fast cache simulators.

   There are some studies addressing static resolvability of data cache behavior (e.g., [90, 105, 160]), but to our knowledge, there are no previous experimental studies that address a slicing problem similar to this study.

   In our experiments, we compared the performance of MTC-based simulation to the performance of MSC-based simulation. Thus, we had two systems under study. The first consists of MTC and MSS. The second is MSC.

   The systems were studied as a part of MSE version 1.2/IN running on a Linux PC with double processor 450MHz Pentium II and 256Mb of DRAM memory. We used GCC version 3.2, ld version 2.13.90, and Linux kernel version 2.4.18.

2. The systems under study have two phases: static compilation and dynamic memory simulation. The static service of MTC is instrumentation of SM code. Its static input is abstract machine code, and its static output is the same code augmented with instrumentation code. MSS works only during the dynamic phase. The dynamic service given by MTC and MSS

[3] We say that we can statically classify a memory reference, if we know statically that it always hits or always misses.

is memory simulation. Its input is the input of the subject program, and its output is the sequence of memory events resulting from the simulation.

The static service of MSC is the generation of a memory simulator. Its static input is abstract machine code and its static output is a memory simulator. The dynamic service, input, and output of MSC is the same as for MTC and MSS.

3. To analyze speed-up, we used the speed-up coefficient R_d as the metric. The speed-up coefficient is $R_d = T_o/T_r$, where T_o is the elapsed time for an execution of an MTC-compiled simulator and T_r is the elapsed time for the corresponding execution of the MSC-compiled simulator. The time was read from a clock of the PC by using system call `gettimeofday`, which yielded microsecond precision. The time was measured from the point immediately after MSE runtime system initialization actions to the point immediately before MSE runtime system termination actions.

   To analyze the static performance, we used static resolution as the metric. Static resolution $F_s = N_r/N_o$, where N_o is the number of memory references analyzed and N_r is the number of memory references statically classified (i.e., the behavior is known). All memory references of the subject programs were analyzed and measured, except the ones related to MSE runtime system set-up and shut-down (MSE subject programs usually include code controlling the simulations).

4. There are many parameters affecting the performance of combined cache analysis. Static workloads, i.e., subroutines of the subject programs, were characterized by their average *basic block length* $L_B = N_R/N_B$ and average *loop length* $L_L = N_R/(N_X + N_Y + 1)$, where N_R is the number of reference instructions, N_B is the number of basic blocks, N_X is the number of natural loops, and N_Y is the number of subroutine calls.

   The most important cache parameter affecting the performance seems to be the cache associativity A. Therefore, we used it as a factor.

5. We decided to use measuring as the evaluation technique. We measured the instrumentation of the programs (static experiments), and execution of the instrumented programs (dynamic experiments). Static experiments were performed by instrumenting each subject program with MTC and MSC. Dynamic experiments were performed by randomly selecting input points, and comparing execution times of simulators generated with MTC to simulators generated with MSC.

6. In the static experiments, we generated the workload for MTC and MSC

by using SMC as an input generator. SMC was given selected programs to be compiled.

In the dynamic experiments, we generated the workload by using random number generators of MSE.

The workload selection and generation is a complex issue; we discuss it in more detail in separate sections.

7.4 Static workload

Selecting a workload for experiments that study combined program analysis presents a difficult problem. The reason for this is that combined program analysis uses programs as its input: It is not easy to automatically generate or manually select a representative set of programs.

The workload selection is further complicated by the fact that programs need input. We cannot measure the efficiency of program transformations, such as our instrumentation, without running the transformed programs. To get comprehensive results, we need a number of inputs per workload program. Thus, we can only consider programs that can be provided with a sufficient set of inputs.

A model of programs would be needed for automatic generation of workload programs, i.e., making an artificial workload. To our knowledge, no sufficient model exists for the generation of input programs for such experiments. It is hard to design such a model, because programs are typically very complex, but not random.

There are some real workloads that are designed as comprehensive representatives of typical programs. Many benchmarks are designed to measure CPU performance instead of memory performance, e.g., the Whetstone benchmark [37], the Linpack benchmark [43], the Dhrystone benchmark [158], and Lawrence Livermore Loops [110]. Some benchmarks are more generic. For example, the SPEC benchmarks [150] are designed for measuring the computational capacity of computer systems, while the TPC benchmarks [155] for measuring transaction systems.

However, such comprehensive benchmarks are not needed in this study. Our primary goal is not to measure properties of typical programs (e.g., the proportion of statically classifiable memory references). The goal of this study is to find out the simulation speed-up yielded by our implementation, when the proportion of statically classifiable memory references is assumed known. The workload must be representative for typical programs only in this sense.

In our experimental setting, simulation speed-ups result solely from partial evaluation and slicing of instrumentation code. Considering this, the essential

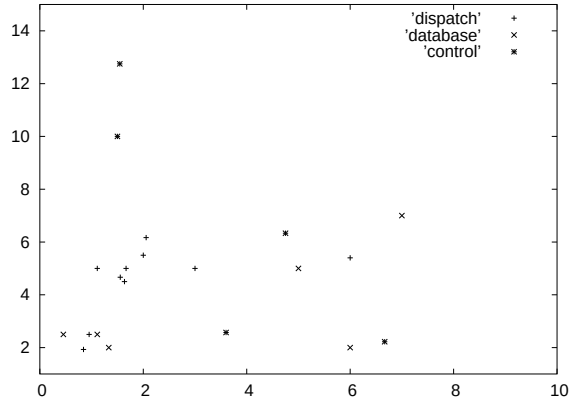


Figure 7.5: Properties of analyzed subroutines: horizontal axis is block length L_B and vertical axis is loop length L_L.

property is the distribution of the instrumentation code among the subject program code that drives the simulation. The memory references of the subject program are instrumented. We get biased results, if we use a workload that has an unnatural distribution of memory references. Thus, we think that we can accept a workload consisting of real programs that have a natural distribution memory references.

We ended up selecting three different applications as our static workload. The applications are small, but not artificial, i.e., they are programs that compute real results.

- `Dispatch`: A message dispatcher, which receives messages, decodes them, and routes them further. The decoding and routing is implemented hard coded. Addresses of most memory references are dynamically computed.
- `Database`: A relational database application, whose index is implemented as an unbalanced binary tree. Addresses of most memory references are dynamically computed.
- `Control`: A control application that operates like a device driver. Addresses of most memory references are static.

The programs are coded in C and consist of subroutines (C functions) and related data structures. The characteristics of subroutines are plotted in Figure 7.5. Each subroutine is a point in the L_B, L_L-space (block length, loop length). According to our input characterization, the workload includes subroutines having different characteristics.

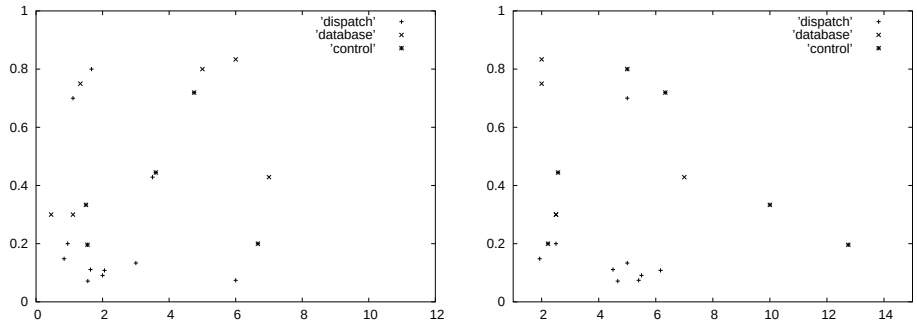


Figure 7.6: Static resolution F_s (vertical axis) of each subroutine compared with block length L_B (horizontal axis, left) and loop length L_L (horizontal axis, right).

We cannot state that our workload comprehensively represents all other programs in every respect. However, we think that the workload is suitable for the goal of this study.

7.5 Static experiments

In our static experiments, we gave our workload programs to SMC, which translated them to SM code. For each subroutine we measured static resolution F_s with cache associativity $A = 16$. The results are in Figure 7.6. Each subroutine is a point in a two dimensional space.

The left side of Figure 7.6 shows how F_s (vertical axis) depends on block length L_B (horizontal axis). The right side of Figure 7.6 shows how static resolution F_s (vertical axis) depends on loop length L_L (horizontal axis). The figures show significant variance in static resolution F_s. This supports our choice of workload.

We repeated the same measurement altering cache associativity in the range from 1 to 16. We summed the counts N_o and N_r for each program, and plotted the resulting $F_s = N_r/N_o$ as a function of the associativity.

Figure 7.7 shows how static resolution F_s (vertical axis) depends on the associativity A (only in the range 1–8). The performance depends both on the subject program and on the cache associativity.

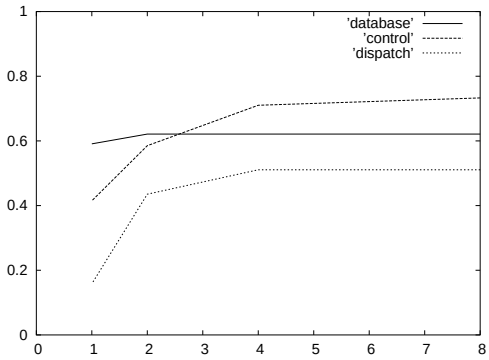


Figure 7.7: Static resolution F_s (vertical axis) as a function of associativity A (horizontal axis).

7.6 Dynamic workload

Dynamic workload was generated separately for each of the programs in our static workload:

- `Database`: x insertions and x queries were made to the database. Keys for both the insertions and the queries were drawn from a random uniform distribution.
- `Dispatch`: x messages were sent to the dispatcher. The type of each message was drawn from a random uniform distribution.
- `Control`: x control commands were send to the controller. Type and parameters for the commands were drawn from a random uniform distribution.

A pseudo-random number generator of MSE was used get random numbers. The generator is a simple linear congruential generator [96].

7.7 Dynamic experiments

The static workload programs were compiled by MTC and MSC, and simulators corresponding to each static workload program were built. The experiments were repeated for cache associativities ranging from 1 to 16.

Dynamic experiments were performed by randomly selecting 30 input points. At each input point, we executed a two-phased analysis: validation and measurement.

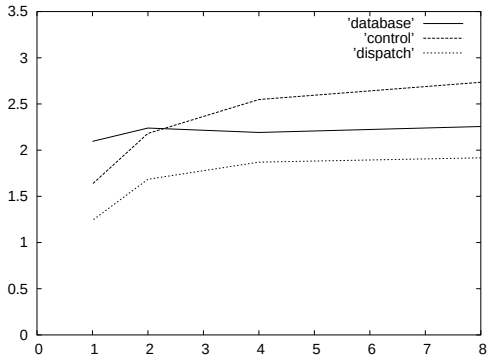


Figure 7.8: Speed-up coefficient R_d (vertical axis) as function of cache associativity A (horizontal axis).

First, for each subject program, an interleaved simulation was run with both systems (MTC-generated and MSC-generated). The validity was checked by comparing similarity of the *incache* tables step-by-step.

Second, we run separate simulations with MTC-generated and MSC-generated simulators, and compared the simulation times. We repeated the measurements until 95% statistical confidence was reached [108].

The results are in Figure 7.8. The figure shows how the speed-up coefficient R_d depends on cache associativity (only in the range 1–8). For all three programs, the effect is similar to the static experiments. This indicates that the performance is mostly dependent on the static phase of our method.

7.8 Conclusions

We performed two kinds of experiments, which we call static and dynamic experiments. In both experiments, we used two tools: one that built traditional simulators and one that built specialized simulators. In static experiments, we measured and compared the simulators. In dynamic experiments, we measured and compared their execution.

The performance of the method depends mostly on the target program and associativity of the cache. Our static cache analysis could classify 15% to 70% of the memory references. The performance of the static analysis depended on the dynamism of the addressing and on the interleaving of the memory references. Accesses of `Database` are more local than accesses of `Dispatcher`. They both use dynamic addressing, but `Control` uses static addressing. The actual (i.e.,

simulated) cache hit ratio for all the applications was typically around 90%.

The goal of this study was to find out the simulation speed-up yielded by our implementation, when the proportion of statically classifiable memory references is assumed known. While the proportion of classifiable memory references varied from 15% to 70% in the static experiments, the specialized simulators were 25% to 180% faster than the original simulators in the dynamic experiments.

The speed-up is mostly caused by slicing. The direct effect of partial evaluation is minor, but it makes the slicing possible by removing branching in the instrumentations.

In addition to our measurement results, it is important to note the things that were not measured. These included:

- We did not measure the time consumed in the static analysis and program transformations. This time is typically negligible, because several simulations can use the same simulator.
- We did not measure the speed-up caused by using an integrated simulator instead of a traditional separate simulator. The speed-up can be significant (see [97] for an evaluation).
- We did not measure the speed-up caused by slicing the input program. Based on our experience, such speed-up depends heavily on the workload, but can be significant.
- Our experiments were solely based on *must* analysis of abstract cache state. Further, we did no interprocedural analysis. Including both *may* and interprocedural analysis of cache states could improve the performance.

The program analysis and program specializations described in Chapters 5–6 may seem complicated. The code implementing the corresponding program analysis and program specializations in MSC is only 460 lines of C++ code. This is a small fraction of a practical memory analysis system – the size of the whole MSE environment is approximately 24600 lines of source code. The price of statically speeding up memory simulations is not high.

Chapter 8

Related work

There has been a variety of approaches for fast cache performance analysis of programs. Also, the goals that have motivated these approaches have varied. There has been work trying to guarantee performance bounds (especially worst-case execution time), work trying to support hardware designs, work trying to estimate average-case performance etc. Some approaches are very specific, e.g., concentrating only on direct-mapped data caches or programs without loops. Some approaches have a much broader scope, e.g., they analyze realistic combinations of data caches, instruction caches, and instruction pipelines.

The traditional cache performance analysis method is trace-based simulation [156], which involves simulating every memory access for all executions (i.e., inputs) considered representative. The methods proposed for fast cache performance analysis can roughly be placed in three major categories: analytic methods, dynamic methods, and static methods.

Analytic methods are based on modeling the hardware and the software in a way resulting in a model that can be solved by mathematical analysis. Typically, such methods are very different from ours (e.g., [3]). Therefore, we present here (Section 8.1) only one line of research, because it resembles our work in some respects. It is based on specific loop models, which makes analytical solving possible.

Dynamic methods are simulation methods that speed up analysis by reducing the number of accesses simulated. They are often called *trace compaction* or *trace filtering* methods, because they are implemented as compactors or filters between the trace generating program and the memory simulator. There are dynamic methods that are based on sampling [4], redundancy [63], spatial analysis [4], and temporal analysis [142]. Three methods that have affected our research are stack deletion, trace stripping, and spatial blocking. We will review

these in Sections 8.2–8.4.

Some simulation methods have concentrated on speeding up the trace consumption instead of manipulating the trace. The methods are based on using several processors to share the load of simulation. In the control of the parallel execution, both cache analysis specific methods and generic parallel simulation methods have been applied. We will review such methods in Section 8.5.

Most of the modern cache performance analysis methods are static. Static methods speed up analysis by simultaneously simulating memory behavior for all inputs considered significant. Many of the static analysis methods are WCET methods (worst-case execution time analysis methods). Thus, their motivation is typically different from ours.

WCET analysis is often done for fairly simple microcontrollers and programs that are not data intensive [44]. Therefore, static WCET analysis is often concentrated on pipeline analysis or instruction cache analysis. Pipeline analysis differs significantly from cache analysis. The state space is smaller, but rather difficult anomalies can occur (e.g., out-of-order execution). Pipeline analysis can be handled by more local analysis than cache analysis.

Instruction cache analysis is similar to data cache analysis. Some methods have been applied both to data cache analysis and instruction cache analysis. Instruction cache analysis is less difficult than data cache analysis, because the address data is usually static and addresses are often accessed in a sequence. Many data cache analysis methods also assume static address data.

There are several approaches to static cache analysis. Some approaches embed cache analysis into WCET calculation, but most perform separate cache behavior analysis. The cache behavior analysis is performed by studying relationships of memory references or by using an explicit cache state concept. We review the methods that have affected our work in Sections 8.7–8.11. In Section 8.6, we shortly describe our previous research on the topic.

8.1 Cache miss equations

Using cache miss equations is an analytic method designed to understand cache behavior of programs and identify their memory bottlenecks. The method is based on loop models, which make the cache behavior analytically solvable. The methods are applicable for deep loop nests, which are typical for scientific code. A typical loop model for cache miss equation assumes that the loops must be perfectly nested, have linear bounds, and memory references must have linear indices with constant steps. Further, only a limited form of branching is allowed inside the loops.

The restrictions make memory references very regular. A nested loop exe-

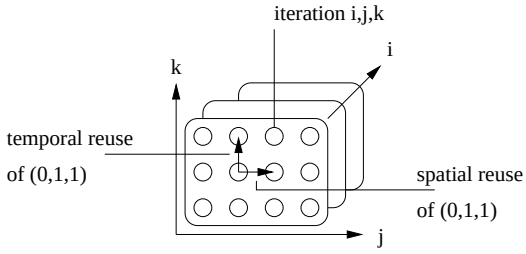


Figure 8.1: Iteration space formed by the loop nest `for i=1,N;k=1,N;j=1,N do Z(i,j)+= X(i,k)*Y(k,j)`. Temporal and spatial reuse vector are shown as arrows. Adapted from [57].

cution spans a multidimensional iteration space, where each point describes a single iteration point (see Figure 8.1). Memory reference patterns can be described by using reuse vectors [167]. Reuse vectors describe repeated accesses to the same memory (or cache) line.

Ghosh, Martonosi, and Malik [57] assume a uniprocessor with one-level direct-mapped cache. They classify reuse as *temporal* if the same location is accessed and *spatial* if the same line is accessed. Based on the reuse vectors they write two kinds of cache miss equations that describe cache misses. *Cold miss equations* identify cache misses caused by first uses of lines and *replacement miss equations* identify cache misses caused by reuses of lines.

Cache miss equations describe program cache behavior qualitatively. Solving the cache miss equations yields quantitative results. Ehrhart polynomials are suggested as a solution method by [58]. The method is highly accurate, but applicable only to a restricted class of loops that cause regular patterns of memory accesses typical for some scientific applications. In more recent work [59], they describe how cache miss equations can be used for compiler optimizations.

8.2 Stack deletion

Stack deletion [141] is a compaction method of traces for the simulation of stack-based memory management algorithms, such as the LRU (see Section 2.5). The idea is to remove those accesses that refer to memory locations that are near the top of the stack of the management algorithm. Thus, probable hits are removed from the trace and probable misses are left in the trace. If the cache miss ratio is low, then the degree of compaction achieved is high.

Ignoring an access of a memory location near the top of the stack does not directly cause any error in the simulation. However, the order of memory lines

original trace	
T	a, b, c, c, c, c, b, b, b, d, d, a, b, c, a
time-independent compaction	
T_2	a, b, c, b, d, a, b, c, d, c, b, c, a
T_3	a, b, c, d, a, b, c, d, b, a
T_4	a, b, c, d, a, c, d, a
time-dependent compaction	
T_2	(a 1), (b 1), (c 1), (b 4), (d 3), (a 2), (b 1), (c 1), (d 1), (c 1), (b 1), (c 1), (a 1)
T_3	(a 1), (b 1), (c 1), (d 7), (a 2), (b 1), (c 1), (d 1), (b 2), (a 2)

Figure 8.2: Examples of stack deletion (adapted from [141]).

in the stack of the management algorithm can be different from that with the original trace. Later on, this can cause an error in the simulation, but typically the probability of the error is low [142].

Smith presented two stack deletion methods: one for time-dependent management algorithms and one for time-independent management algorithms. In an access trace, both methods delete the accesses that are hits to levels $1, \ldots, k-1$, where k is the deletion parameter (the minimum size of the simulated cache).

In the first method, the compacted trace consists of the remaining accesses. In the second method, the compacted trace consists of a two-tuple for each access that remains after deletion. A two-tuple is composed of the address accessed and the increment of the time counter needed in time-dependent memory-management algorithms [33, 40].

Figure 8.2 gives examples of stack deletion. T_k denotes the reduced trace of deletion parameter k. The number of misses observed in time-independent processing of the original trace T for memory sizes of 1, 2, 3, and 4 is (12, 10, 8, 4), and similarly for T_2, T_3, and T_4 we obtain (13, 10, 8, 4), (10, 10, 9, 4), and (8, 8, 5, 4).

Smith states that for LRU the simulation error is bound by $F(C-k+2) \leq F_k(C) \leq F(C+k-2)$, where C is the cache size, k the deletion parameter, F the number of misses for the original trace with cache, and F_k is the number of misses for a compacted trace. This theoretical bound is very loose. In his measurements, he reported significantly smaller errors (typically 1%–5%).

The stack deletion method has weaknesses. In many situations, it is too slow. During the trace compaction, the stack must be maintained. This makes the compaction as slow as many memory simulators. Thus, stack deletion is worth while doing only if the compacted trace is to be used several times.

8.3 Trace stripping

Puzak proposed a method called *trace stripping* [129], which performs temporal filtering of traces. The method uses a direct-mapped cache, which is called a *cache filter*. The cache filter is given an input trace. The accesses, which cause misses in the cache filter, are written to an output trace. Thus, the accesses not causing misses are filtered.

Let the cache filter have 2^f direct-mapped sets and G be a cache, which has 2^g, $g \geq f$ sets and the same line size as the cache filter. If G is given the output trace of the cache filter, then the same number of cache misses is generated as if G was given the original input trace of the cache filter. The method is not efficient for full-associative caches, because the filter would consist of a single line.

Trace stripping is illustrated in Figure 8.3, in which we use the same graphical representation as in Section 2.4. On the left, there is a secondary-layer memory having 8 lines and an associative cache having 2 sets of the size 2. On the right, the same 8-line secondary memory has a 1-way (direct-mapped) cache. It can be used as a cache filter for the left-hand-side cache, because both have the same number of sets. In both caches, set a is for lines 1, 3, 5, 7 and set b for 2, 4, 6, 8.

Let an original trace be (3, 6, 3, 1, 6, 3). If the trace is given to both of the caches, the following actions happen:

2-way cache	1-way filter cache
3: miss, read to a1	3: miss, read to a1
6: miss, read to b1	6: miss, read to b1
3: hit	3: hit
1: miss, read to a2	1: miss, read to a1
6: hit	6: hit
3: hit	3: miss, read to a1

Thus, the cache filter yields four misses: 3, 6, 1, and 3. The bigger cache yields three misses: 3, 6, and 1. If the bigger cache is given the filtered trace,

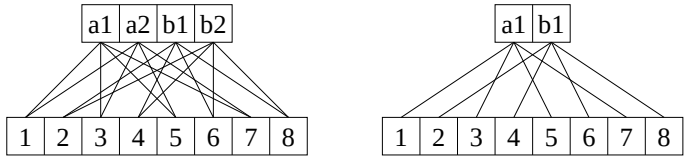


Figure 8.3: A memory with a 2-way cache (left) and a cache filter for it (right).

i.e., cache miss sequence of the filter (3, 6, 1, 3), then exactly the same misses happen.

3: miss, read to a1
6: miss, read to a1
1: miss, read to a2
3: hit

The compaction achieved by trace stripping is significant. Typically, a program trace is compacted over 90%. Puzak suggested that the method could also be used to filter traces with line sizes different from the one to be used in simulations. Such a method can increase the compaction, but also cause errors in simulations [38].

8.4 Spatial blocking

The previous compression methods are based solely on temporal analysis of a trace. Agarwal and Huffman have presented a method that also applies spatial analysis. They call their spatial trace compaction *spatial blocking* [4]. Spatial blocking is designed to be applied to traces that are already temporally filtered. Their temporal filtering is based on trace stripping.

The spatial blocking method is based on statistics. It assumes that all the memory references in a spatial neighborhood have similar properties. It uses a representative reference from each spatial neighborhood to predict the performance of the entire neighborhood. In mathematics, this method of sampling is called stratified sampling.

Consider a large population of data over which some mean parameter is desired. Assume that the population can be divided into strata within which the value of the parameter of interest is more nearly constant than it is in the population as a whole. Stratified sampling draws small unrelated samples from each of the strata separately to estimate the value of the given parameter. Such sampling is statistically superior to taking a single sample of the same size from the entire population.

Consider a population P in which the mean value m of some parameter M is desired. Let the population have s strata. Within each stratum the value of M can vary. Let the size of ith stratum be N_i, the sum being the total population size N (i.e., the trace length).

An estimate of m, called $m*$, is obtained by taking the weighted average of the selected representative items, called m_i's, from each stratum. Then

$$m* = (\frac{N_1}{N}m_1 + \frac{N_2}{N}m_2 + \ldots + \frac{N_s}{N}m_s)$$

If the variance of M in stratum i is σ_i^2, then the variance of $m*$ is approximately given by

$$Var(m*) = (\frac{N_1^2}{N^2}\sigma_1^2 + \frac{N_2^2}{N^2}\sigma_2^2 + \ldots + \frac{N_s^2}{N^2}\sigma_s^2)$$

assuming that the strata are uncorrelated and the variance of M in each stratum is small compared to the variance in the whole population.

If the variances within the strata are the same and the strata are of the same size, then $Var(m*) = \sigma^2/s$, which is typically negligible.

For program traces, the strata size varies and the strata are correlated. However, typical programs do not refer memory in a random manner. The memory references are usually local and regular. Agarwal and Huffman made some experiments with their method. They report miss-rate-errors about 10% for traces compacted over a magnitude.

8.5 Parallel simulation

Parallel cache simulation methods use several processors to speed up cache simulations. A parallel simulation method needs a mechanism that shares the simulation load among the processors. Such methods can be cache simulation specific or generic.

Set partitioning and time partitioning [68] are based on the organization of typical caches. In set partitioning, a trace is seen as a composition of independent subtraces. Each subtrace contains references that map onto one single set of the simulated cache. For each subtrace, there exists a processor that runs a separate simulator processing the subtrace. The method works best for set-associative caches. For directed mapped caches the subtasks are too small, while for full associative caches there is no parallelism.

In time partitioning, a trace is seen as a sequence of subtraces. The processing of a trace T is iterative. First, the trace T is cut into n parts, which we denote T_i $(1 \leq i \leq n)$. Each resulting subtrace is processed by a separate simulator. In the first iteration, only the first subtrace T_1 can be simulated correctly, because the initial cache state is known only for it. At subsequent steps, each T_i $(2 \leq i \leq n)$ is resimulated using the final state of T_{i-1} as the initial. Because of the stability of typical caches, the iteration terminates quickly. According to [68], time partitioning is statistically faster for typical caches than set partitioning. However, the method is not suitable for execution-driven simulations.

Parallel discrete-event simulation methods are generic methods that can be applied to cache simulation. In such an approach, each cache line is seen as an

object and the cache is seen as communication and synchronization mechanism for the objects. Memory accesses are the events that cause message passing in the system. Lin et al. have applied the Chandy-Misra simulation method to solve the problem [77]. Some parallel simulation methods have applied program slicing to improve performance [39, 131].

8.6 Static filtering

Static filtering is a method for speeding up trace-based memory simulations. Dynamic filtering compacts the trace after it has been generated, but static filtering modifies the trace generation to yield more compact traces. Such a static method speeds up both the generation and consumption of a trace, as dynamic methods speed up only the latter. The method yields a smaller degree of compaction than dynamic methods, but it can be combined with them.

Static filtering is based on compile-time data flow analysis, which analyzes redundancies in the memory reference patterns of programs. If the memory behavior caused by a pattern can be caused by a smaller pattern, then a program is instrumented to emit the reduced pattern to its memory traces instead of the complete one.

In my licentiate's thesis [72], I proposed the approach and presented a method and a naive implementation, by which 30%–70% reduction of trace size was achieved. The filtering method consists of a variant of traditional flow analysis of programs and a related instrumentation scheme. This method finds static and induced chains of computed values, which are used as addresses for memory accesses.

The method does not perform an explicit cache state analysis. It analyzes the cache behavior of the references in a program by using a temporal and spatial locality window. As many other filtering methods, the method is approximative. The approximation can cause an error in a memory simulation that is based on the compacted trace.

8.7 Cache constraints

Cache constraints are equations or in-equations that describe the cache behavior of a program. They state how cache hits (or misses) depend on execution counts of program elements. Cache constraints form a generic framework for program cache performance analysis. For automated applications, a method for constructing cache constraints and a method for solving the constraints is needed.

Li, Malik, and Wolfe [99] use cache conflict graphs to construct cache constraints. They assume a direct-mapped cache and study instruction caches. A cache conflict graph is constructed for every cache line. The conflict graph describes how fragments of basic blocks can replace each other in the cache during the execution of a program.

Each conflict graph has a start node (s), an end node (e), and a node $B_{i,j}$ for every fragment j of every basic block B_i that is mapped to the line that the conflict graph models. A directed edge $B_{k,l} \rightarrow B_{m,n}$ is drawn if there exists a path from $B_{k,l}$ to $B_{m,n}$ without passing through any other fragment on the same cache line.

The start node represents the start of a program and the end node represents the end of a program. Entering a fragment node in the graph implies a cache miss. At each node $B_{i,j}$, the sum of control flow going into the node must be equal to the sum of control flow leaving the node, and it must also be equal to the execution count of the block itself. This results in a set of constraints:

$$x_i = \sum p(u.v, i.j) = \sum p(i.j, u.v)$$

where p stands for count, $i.j$ the fragment itself, $u.v$ is any node (including s and e), and x_i is execution count of the block. Counts of the form $p(i.j, i.j)$ represent cache hits, for which the graphs yield constraints like:

$$p(i.j, i.j) \leq x_{i.j}^{hit} \leq p(s, i.j) + p(i.j, i.j)$$

Together with some simpler constraints derived from the control flow graph, these form a constraint system that implicitly describes the cache behavior.

Li, Malik, and Wolfe [99] use integer linear programming (ILP) to solve constraint systems. To get a WCET bound, they use an ILP solver with the goal of maximizing a cost function consisting of execution times of program fragments. The method has been further developed to incorporate more complex cases [100, 104, 120].

8.8 Structural analysis

Structural analysis methods are based on the syntax structure of a source program. In such methods, analysis information is passed bottom-up or top-down in the syntax tree to yield the desired analysis result.

Lim et al. [101] use a bottom-up structural method for cache performance analysis. The bottom-up analysis uses partial state descriptions. They assume a direct-mapped cache and static address data. For each syntax structure c_i

and each cache line l_j, they compute the first and last known cache content.

For simple statements accessing memory, the partial state description contains the locations accessed. For compound statements, the partial state description is constructed from the state descriptions of their component statement. For example, for the statement

$s \rightarrow \texttt{if}\ (e)\ \texttt{then}\ s_1\ \texttt{else}\ s_2$

the construction is

$c_s = \{c_e \oplus c_1\} \cup \{c_e \oplus c_2\}$

where $\oplus$ is a concatenation operator that simulates operation of the cache and c_x is state description of structure x. Based on the partial state information, some cache misses and hits can be determined. In their method, the cache analysis is embedded into a bottom-up WCET calculation. The method adds cache analysis to the timing schema approach of Park and Shaw [123].

Kim, Min, and Ha [90] extend the analysis by using a version of the pigeonhole principle to get bounds on data cache performance. The pigeon-hole principle states that if there are fewer containers than items to be contained, then there must be more than one item in some container. In loops, they determine the maximum number of accesses by single memory references and the maximum number of distinct locations accessed. The method is limited in its ability to detect both temporal and spatial locality.

8.9 Graph coloring

Graph coloring is a technique that ensures that adjacent nodes in a graph have different colors. Based on register interference graphs, the method has been used in compilers to perform register allocation [32].

Rawat [130] uses a graph coloring technique to analyze data cache performance. Based on the variables contained, cache lines are defined live ranges, which are represented by cache nodes. Conflicting accesses to the same line cause splitting of the nodes. Node splitting represents cache misses.

The applicability is limited: the method considers only local scalar variables within a single function. Further, the method is applicable only to direct mapped cache memories.

8.10 Static cache simulation

Static cache simulation is a group of methods for cache performance analysis. The methods are based on computing static abstract cache states at program points and using additional program flow information to yield classification of memory references. Abstract cache states are constructed by straightforward iterative flow analysis methods. At each point, the cache state gives a loose bound for possible cache contents. This bound is tightened by using various flow analysis methods.

Mueller and Whalley [117] describe a data flow analysis method for the prediction of instruction cache behavior. They base their method on control-flow partitioning. They use reaching-state and control-flow information to classify references more accurately.

The original method has been further developed in many ways. The variants include methods for WCET analysis for processor with pipelines and data caches. A typical memory reference classification is:

- *Always miss.* The referred to location is never in cache.
- *Always hit.* The referred to location is always in cache.
- *First miss.* The referred to location is not cache for the first access, but is for rest of them.
- *First hit.* The referred to location is in cache for the first access, but not for rest of them.

[66] presents a variant for data cache analysis that handles dynamic addressing and yields *may* and *must* analysis of cache behavior. The analysis uses data-flow techniques to determine the bounded range of addresses for each data reference.

8.11 Abstract interpretation

Abstract cache state interpretation is the cache analysis method used in this thesis. It is based on the generic theory of abstract interpretation. In addition to the data cache analysis for dynamic references (presented in this thesis), the method has been used for a number of performance analysis problems.

In [7, 47, 50] Alt, Ferdinand, Martin, and Wilhelm present the general approach for using abstract interpretation for cache state analysis. Variants of this basic method include better data cache analysis, pipeline analysis, and more precise modeling of program structure, especially loops and interprocedural analysis [49, 67]. To implement their analyses, they use a generic static analysis tool called PAG [8].

They have considered complex real-life processors such as PowerPC 750/755 and SuperSPARC I [137]. In [48], Ferdinand et al. describe analysis of the ColdFire MCF 5307 processor that has a pipeline and a unified instruction and data cache. In their work, they have also considered scheduling issues in the presence of cache memory [87, 136].

There are two major differences between their work and ours. First, they consider WCET analysis while we consider bottleneck analysis. Thus, their focus is on giving tight WCET bounds for all possible inputs. Our focus is on understanding program behavior for representative set of inputs. Second, they consider programs with static memory addressing while we consider programs with dynamic memory addressing.

8.12 Conclusions

Cache memories are one of the main components affecting the performance of computers. Therefore, methods for cache performance evaluation have been intensively studied. A wide spectrum of methods have been developed by varying communities. Some methods are hardware-oriented while others are software-oriented. Thus, the methods differ in their applicability.

All static cache analysis methods are based on the observation that runtime behavior of a program can be more dependent on the static program text than its input. For such programs, cache behavior can be resolved to an extent that makes static analysis methods practical. The methods are not useful for programs having dynamic behavior. Thus, all static analysis methods have explicit or implicit restriction on programs that can be analyzed. However, compared to an approach like cache miss equations, the static methods are far more generic.

Some static cache analysis methods are built ad hoc, but there exists methods that have a unified and firm theoretical basis. Such a basis makes modeling of complex systems reasonable. Abstract interpretation especially seems like a competitive approach. The most complex static cache analysis tasks completed so far have been performed by applying abstract interpretation.

Simulation methods that do trace compaction or speed up trace generation are based on address trace redundancy that is typical for programs. Figure 8.4 illustrates this. In the figure, traces are plotted in two-dimensional space. Each point is a memory reference.

On the left side of the figure is a trace where accesses are almost randomly and uniformly scattered in the address space. Such a trace has very little redundancy and cannot be compacted. Also, data caches are not useful for programs having such a trace.

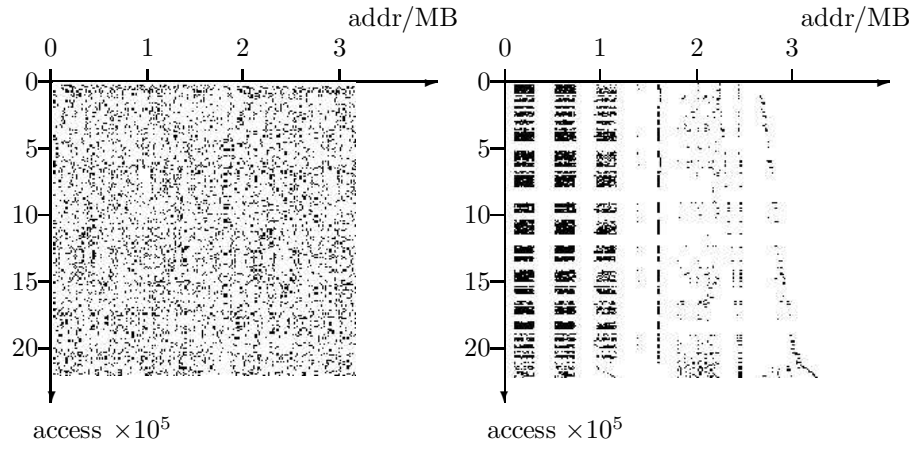


Figure 8.4: Example traces plotted in two-dimensional space: A random-like trace (left) and a typical program trace (right). Horizontal dimension is the address referred to and vertical dimension is the number of the reference.

On the right side of Figure 8.4 is a typical program data trace. A number of regularities appear in the trace. Some addresses are accessed more often than others. Accesses are both temporally and spatially local, which can be seen as various formations in the figure, e.g., lines. All the methods try to benefit from such patterns.

Most of the compaction methods presented in the literature are designed for trace-driven simulation. Typically, they are not so well suited for execution-driven simulation. Parallel simulation method that speed up trace consumption differ in this respect. They are orthogonal to the approach presented in this thesis.

Chapter 9

Discussion

Because of the complexity of memory hardware, interactions between memory references are complex. We must understand the accesses that the references make to understand their interactions. Typical hardware does not support analysis of memory operations [78]. Thus, simulation has often been the choice for performance studies and the basis for related tools giving detailed information (e.g., [106]).

Trace-driven simulations [156] are common. A trace-driven simulation has two main phases. In the first phase, an access trace is collected. Because hardware support for cache tracing is rare [78], the collection is typically performed by instrumenting the subject program with trace-emitting code. In the second phase, a memory simulation is executed using the trace of the first phase as the input.

The major problem in trace-driven simulation studies is the size of traces [129, 156]. The traces resulting from program executions can be huge. To understand thoroughly the memory behavior of a program, the program must be executed with several inputs and several traces must be generated. Thus, very long simulation times can be needed because of the huge quantity of data to be analyzed.

To make the size and processing time of traces feasible, a trace compaction phase can be added between the two main phases [156]. Compaction is especially suitable for hardware performance studies. In such studies, the trace once collected can be used several times; it serves as a benchmark for hardware [92].

In a software performance study, the input is varied and a different trace is usually generated for each simulation. Rapid prototyping with various designs improving the performance of the software are needed [144]. Therefore, execution-driven (on-the-fly) simulation is practical in software performance

studies; the trace is consumed as it is produced.

Cache analysis can be performed statically, i.e., without executing the program or simulating its execution with an input. Recently developed static cache analysis methods have their roots in compiler technology, which has used static analysis for optimization for decades [118].

Methods using static cache analysis have given a new tool for revealing cache-related performance-bottlenecks of programs. Static analysis is performed without executing the program, thus the results are applicable for all inputs. Static analysis can yield upper and lower bounds of cache misses and also execution times. Such absolute guarantees are very useful in the design of hard real-time system [49]. Further, static analysis methods can pin-point reasons for bad cache behavior.

Static analysis is a fast method, but in some cases, its approximative nature is a problem. Typical static analysis divides memory references of a program into three classes: those who always miss, those who always hit, and those about which we cannot decide. (With auxiliary information, we can get a more detailed classification, e.g., first-hit and first-miss classifications by conceptual unrolling of loops.) For applications using dynamic memory addressing, the bounds that we get can be too loose (the example in Figure 9.1 illustrates this).

The use of dynamic memory addressing makes programming easier. In the past, dynamic memory addressing has not been used in hard real-time systems with cache memory, because there have not been sufficient analysis methods to guarantee their performance. In soft real-time systems, such absolute guarantees are not needed.

Dynamic memory addressing is difficult for static analysis, because typical

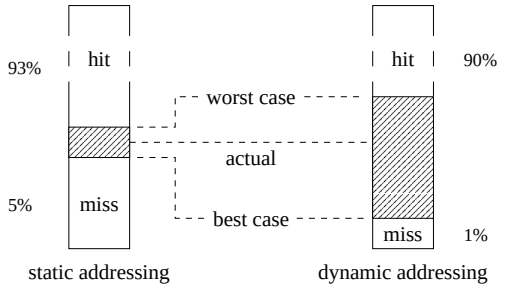


Figure 9.1: Using dynamic addressing instead of static addressing makes bounds loose. If 1% of accesses are known to be misses that cost 40 cycles each and 90% are known to be hits that cost 3 cycles each, the upper bound of performance is almost double compared to the lower bound.

cache memories have parallel architecture. Instead of a single memory unit, cache memories are composed of a number of small memories, which execute in parallel. This set-associative organization causes conflict misses, – but, on the other hand, – it makes cache memory very fast, because the signal paths in the hardware are small.

With typical dynamic memory addressing, we cannot statically compute the cache set (or sets) that a reference accesses during execution. Therefore, our understanding of the micro-level parallelism in hardware is typically incomplete.

Our approach solves the problem by using a static method that can analyze programs using dynamic addressing, and supplementing the input-independent static analysis with input-dependent dynamic analysis. Static analysis can be used to give absolute guarantees of memory behavior, but it typically cannot give full memory behavior description for any input. Dynamic analysis gives full memory behavior description for a single input, but it cannot give absolute guarantees. We believe that the combination makes program performance analysis and design easier.

In the following, we discuss our research in more detail. First, we summarize the abstract framework, the related cache analysis method, and their evaluation. Then, we discuss the applicability of our work and its position in performance analysis. Finally, we discuss possible future work.

9.1 Method

In this thesis, we presented an abstract framework for combining static analysis and dynamic analysis, and applied the framework to cache performance analysis of programs. Our approach is based on three simple observations that we have made:

- Memory simulators typically spend most of their time in processing hits, i.e., practically running idle. Those hits could be statically handled, and thus not simulated at all.
- From the cache simulation point of view, the actual task of the subject program is irrelevant. Simulation can be made faster by removing the parts of the computation that do not affect the simulation result.
- In software performance analysis, a trace is not as useful as in hardware performance analysis. By joining trace collection and consumption, we can apply the improving transformations to the whole simulator.

Our combined analysis has three phases: a compilation phase, an execution phase, and a summary phase. In the compilation phase, a subject program is statically analyzed and a dynamic analyzer is built. In the execution phase,

the dynamic analyzer is executed. The first phase gives the results of the static analysis, i.e., common for all inputs. During the second phase, each simulation gives results specific to its input. In the summary phase, the analysis information of the compilation phase and the analysis information of the execution phase are combined.

This thesis is focused on the compilation phase. The compilation phase consists of three steps:

1. Based on a memory model, each reference in the subject program is augmented with a piece of code that simulates its memory behavior. The result is an integrated memory simulator. It produces both the output of the original subject program and simulation output describing the memory behavior of the original subject program.
2. The integrated simulator is statically analyzed. This includes address analysis, i.e., analyzing cache aliasing and cache conflicts of memory references. Based on the address analysis, cache behavior of memory references is solved.
3. The integrated simulator is partially evaluated and sliced. All those memory simulation operations that can be found independent of the input to the simulator are evaluated. All instructions that are not needed in computing the simulation output are removed. The result is a simulator, which uses inputs of the original subject program, but produces only the simulation output describing the memory behavior of the original subject program.

Our static cache analysis approximates concrete cache states. The concrete state of a cache consists of the states of its sets. The state of the whole cache (consisting of several cache sets) is approximated by an abstract cache state.

In *must* analysis of cache states, abstract locations are mapped to their maximum cache age. In *may* analysis of cache states, abstract locations are mapped to their minimum cache age. Memory references that always hit are found by *must* analysis and memory references that always miss are found by *may* analysis.

The partial evaluation is based on using specialized augmentation variants. The program slicing is based on deleting groups of operations that have no effect on the simulation result.

9.2 Evaluation

We proved the correctness of both the abstract framework and its application to cache performance evaluation. We used analytical methods in proving the

correctness. We selected abstract interpretation as the basis of our static analysis, because it supports semantics-based proofs of various properties of static analysis methods.

Our experience with abstract interpretation is positive. Working with the mechanisms of abstract interpretation helps to avoid pitfalls of static program analysis. Other approaches to static program analysis exist, but we are satisfied with abstract interpretation.

In the early phases of this research, we used a window concept [71] instead of abstract cache states and equational analysis instead of abstract interpretation. While using such methods, the development of static program analysis was error prone.

We selected partial evaluation and program slicing as our speed-up mechanisms, because they are generic methods. Other speed-up mechanisms exist. Compiler literature includes a long list of various optimizations (actually, they are code improving transformations). They can be applied to simulators. Many of them were applied in our experimentation, because we used an optimizing compiler as a part of our system.[1]

Our method is a safe one. Simulation is fast, but no information is lost. In the early phases of this research, we experimented with information-loosing methods because they seemed to give much better performance than safe ones. Our experiments with information-loosing methods did not pay off. Performance gain was unsatisfying compared to the number of problems caused. With information-loosing methods, correctness proofs are difficult. Furthermore, validating an implementation is even harder.

We implemented a tool to show that our approach is practical. The tool uses fixed-point iteration to implement static analysis and stack simulation [107] to implement dynamic analysis. Other implementation techniques exist, but our selection was sufficient to demonstrate a practical implementation. The correctness of our implementation was experimentally validated.

The tool described in this thesis is a research tool. It is not suitable for analyzing large production software-systems or for other industrial use. It lacks a proper user interface and cannot be integrated into a software-development environment with its present facilities. However, it is sufficient for research purposes.

To show the potential of our method, we experimentally evaluated its performance by applying our tool. The experimental work is not sufficient to properly state the performance of the method. For such a purpose, the presented performance analysis is too narrow. Further, our implementation was based only on *must* analysis of cache states. Including *may* analysis should improve perfor-

[1] To analyze the effect of our method, we used the same optimizations for both the traditional and the specialized simulators in our experiments.

mance. However, our experimentation shows that the method has the potential to significantly improve the performance of cache simulators.

9.3 Applicability

The growth of performance mismatch of memory speed and processor speed, and the development of layered memory architectures have caused many profound changes in program design [143]. For data intensive applications, the number of processing steps is no longer the most important measure of performance. Instead, limiting the number of data accesses and especially scheduling and localizing them has become more important [2, 61, 167].

Understanding memory performance is difficult. The steps executed and the related memory references can be seen from the code of a program. However, cache misses and the related execution stalls cannot be seen. Therefore, there is a need for automated tools aiding software and algorithm designers.

Our analysis is safe. It introduces no error in the simulation. However, this does not guarantee that a practical analysis has no errors. Errors are characteristic to simulation studies. In a simulation study, a *model* is evaluated [96]. Typically, a significant error is caused by using a model, which usually is an abstract simplification and approximative. Thus, the problem in simulation studies is often the verification and validation [135] of the simulation model and its implementation, not the accuracy of the solution technique.

If accuracy is wanted, a direct evaluation technique should be used, i.e., the system should be measured or emulated. Emulation is a choice if the system has not yet been built, otherwise measuring will be very hard or impossible. When a memory emulation is performed, real and accurate traces should be used.

Because of the complexity of real systems, understanding their behavior by using direct methods is typically much more laborious than by using simulations [81]. Furthermore, a simulation model can be generic and the results can be applicable for a range of real systems, as direct methods are specific only to the particular system studied.

This thesis presented a method for speeding up LRU-based cache simulation and a related tool implementation, which was built for analyzing cache performance of an abstract machine. The analysis does not separate data reading and data writing. The effect of data writing with various write policies is an especially important concern in writing programs for high-speed applications [128]. Our method and implementation has been sufficient for our algorithm research (e.g., [76]).

Analysis of real-life hardware is needed for practical software development. Already, there exist static analysis tools [1] that are based on static address

data. Such tools can give WCET bounds many times tighter than tools without static microarchitecture modeling.

9.4 Contribution

The contributions of this thesis show that, if we use simulation for program cache behavior analysis, it is possible and reasonable to apply program specializations that speed up the simulation.

The contributions of this thesis are related to software performance engineering. Consider a typical iterative software performance development process that is illustrated in Figure 9.2. Analyzing performance is only a part of the whole process [75, 144]. Applying an evaluation technique such as the cache analysis method described in this thesis is only one part of a performance analysis (as already seen in Chapter 7).

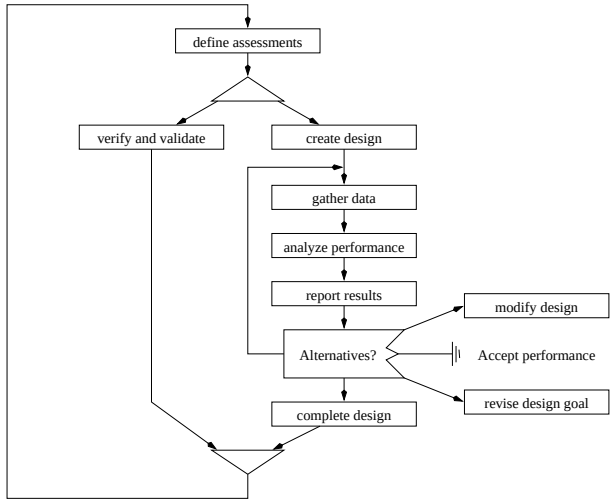


Figure 9.2: A typical software performance engineering process (according to [144]).

On the other hand, static analysis and simulation are generic evaluation methods. The abstract framework presented in this thesis can be used to design other program analyses, including as those not related to performance analysis. The static cache analysis can be used for other purposes than speeding up simulations. An example of such are intelligent memory systems [114, 165].

The use of a specialized cache performance analysis method may seem problematic. However in our experience [72], the performance critical structures are often only a small fraction of a software. Thus, they can be closely studied. In addition to locating the origins of cache misses, we must understand what causes them in order to make changes that improve performance.

Software performance work is often based on designing program code at various levels (from the statement level to the architectural level). When memory is the hardware bottleneck, it is important to design data structures and their layout in the memory. Bad design can cause performance to drop an order of magnitude [21].

9.5 Future work

This thesis is a part of ongoing research. Several directions for future research exist. These include address analysis, analysis of parallel execution, programming language design, and parallel simulation.

We have also considered applying our approach to pipeline simulations. However, according to our initial studies on the subject, there are a number of things that make such an analysis harder. Small details in pipeline implementation or changes in cache state can cause large changes in pipeline operation [51]. Therefore, pipeline analyses must be more detailed than cache analysis.

The theory presented in this thesis is based on traditional serial simulation. The approach can be used to speed up parallel simulation. A parallel simulation could be significantly faster. The current formulation can be used with simple parallel techniques (set partitioning and time partitioning). However, further research is needed to combine our approach with simulation methods that use advanced synchronization methods.

Static cache performance analysis is a demanding task. The analysis is even more demanding, when dynamic addressing in references is allowed. One of the key elements in our approach has been the division of the problem. We have divided static cache analysis into two parts: address analysis and cache state analysis.

The difficult part of static cache performance analysis is predicting cache misses. Cache misses cause the execution stalls. In such an analysis, it is important to understand conflicts and cache states in a *may-not* sense. An advanced static address analysis is needed. With present address analyses, the performance of our static cache analysis (without simulation to support it) is not sufficient for practical purposes.

Our static analysis works nicely, when we compile fast cache simulators. In speeding up cache analysis, understanding cache hits is important. If we

statically predict a miss, we can speed up the simulation only slightly, because the change of the cache state is significant. If we statically predict a hit, we can significantly speed up the simulation, because the change of the cache state is modest.

In analyzing cache hits, it is important to understand cache aliases and cache states in a *must* sense. Our experiments demonstrated that significant speed-up could be achieved with simple address analysis, if the cache is not direct mapped. For direct mapped caches, better address analysis is needed.

The task of developing address analysis could be eased by design of programming languages and the related libraries. Typical programming languages and libraries do not support data layout design. Typical allocators are given no instruction, where they should allocate memory. Explicit data layout instructions could help both the designer tuning the performance of a software and the tool analyzing the performance of the software.

Typical static cache analysis methods, including our method, assume serial execution. This severely restricts the mechanisms determining how a system can be scheduled. Allowing flexible scheduling would greatly increase the number of software systems that can be analyzed. Currently, parallel execution is a challenge – probably the biggest challenge – for cache analysis.

Bibliography

[1] AbsInt Angewandte Informatik GmbH. aiT Worst-Case Execution Time Predictor. http://www.absint.com/ (product information).

[2] A. Agarwal, J. Hennessy, and M. Horowitz. Cache Performance of Operating Systems and Multiprogramming. *ACM Transactions on Computer Systems*, 6(4):393–431, May 1988.

[3] A. Agarwal, M. Horowitz, and J. Hennessy. An Analytical Cache Model. *ACM Transactions on Computer Systems*, 7(2):184–215, May 1989.

[4] A. Agarwal and M. Huffman. Blocking: Exploiting Spatial Locality for Trace Compaction. In *Proceedings of the ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, Performance Evaluation Review, 18(1), pages 48–57, May 1990.

[5] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[6] B. Alpern, M.N. Wegman, and F.K. Zadeck. Detecting Equalities of Variables in Programs. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 1–11, January 1988.

[7] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache Behavior Prediction by Abstract Interpretation. In *Proceedings of the Static Analysis Symposium (SAS)*, pages 52–66. Springer-Verlag, Lecture Notes in Computer Science 1145, 1996.

[8] M. Alt and F. Martin. Generation of Efficient Interprocedural Analyzers with PAG. In *Proceedings of the Static Analysis Symposium (SAS)*, Springer-Verlag, Lecture Notes in Computer Science 983, Springer Verlag, pages 33–50, September 1995.

[9] D. Anderson and T. Shanley. *Pentium Processor System Architecture.* Mindshare Press, 1993.

[10] G.R. Andrews. *Concurrent Programming, Principles and Practice.* Addison-Wesley, 1991.

[11] W.A. Appel. *Modern Compiler Implementation in C.* Cambridge University Press, 1998.

[12] L. Arge, M.A. Bender, E.D. Demaine, B. Holland-Minkley, and J.I. Munro. Cache-Oblivious Priority Queue and Graph Algorithm Applications. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 268–276, 2002.

[13] Atmel Corporation. AVR Enhanced RISC Microcontroller Data Book, 1997.

[14] F. Baccelli, A. Jean-Marie, and Z. Liu. A Survey on Solution Methods for Task Graph Models. In *Proceedings of the QMIPS-Workshop on Formalism, Principles and State-of-the-art*, 1993.

[15] T. Ball and S. Horowitz. Slicing Programs with Arbitrary Control-flow. In *Proceedings of the International Workshop on Automated and Algorithmic Debugging*, pages 206–222, May 1993. Springer-Verlag, Lecture Notes in Computer Science 749.

[16] S. Basumallick and K. Nilsen. Cache Issues in Real-Time Systems. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems (LCT-RTS)*, June 1994.

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

[18] C. Bell, J. Mudge, and J. McNamara. *Computer Engineering: A DEC View of Hardware Systems Design*. Digital Press, 1978.

[19] M.A. Bender, E.D. Demaine, and M. Farach-Colton. Cache-Oblivious B-Trees. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, pages 339–409, 2000.

[20] M.E. Benitez. *Register Allocation and Phase Interactions in Retargetable Optimizing Compilers*. PhD Dissertation, University of Virginia, April 1994.

[21] J. Black, C. Martel, and H. Qi. Graph and hashing algorithms for modern architectures: design and performance. In *Proceedings of the Workshop on Algorithm Engineering (WAE)*, pages 37–48, Saarbrücken, Germany, 1998.

[22] J. Blieberger. Data-Flow Frameworks for Worst-Case Execution Time Analysis. *Real-Time Systems*, 22(3):183–227, 2002.

[23] M.A. Bulyonkov. Polyvariant Mixed Computation for Analyzer Programs. *Acta Informatica*, (21):473–484, 1984.

[24] B. Calder, K. Chandra, S. John, and T. Austin. Cache-Conscious Data Placement. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 139–149, San Jose, USA, October 1998.

[25] D. Callahan, S. Carr, and K. Kennedy. Improving Register Allocation for Subscripted Variables. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 53–65, 1990.

[26] A. Chandrakasan and R. Brodersen. *Low power digital CMOS design*. Kluwer, 1995.

[27] S. Chatterjee, V.V. Jain, A.R. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear Array Layouts for Hierarchical Memory Systems. In *Proceedings of the ACM International Conference on Supercomputing*, pages 444–453, 1999.

[28] S. Chatterjee, A.R. Lebeck, P.K. Patnala, and M. Thottethodi. Recursive Array Layouts and Fast Parallel Matrix Multiplication. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 222–231, 1999.

[29] S. Chatterjee and S. Sen. Cache-Efficient Matrix Transposition. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 195–205, Toulouse, France, January 2000.

[30] J. Chen, A. Borg, and N.P. Jouppi. A Simulation Based Study of TLB Performance. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 114–123, May 1991.

[31] T.M. Chilimbi, M.D. Hill, and J.R. Larus. Cache-Conscious Structure Layout. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, May 1999.

[32] F. Chow and J.L. Hennessy. Register Allocation by Priority-Based Coloring. *ACM SIGPLAN Notices*, 19(6):222–232, 1984.

[33] W.W. Chu and H. Opderbeck. The Page Fault Frequency Replacement Algorithm. In *Proceedings of the Fall Joint Computing Conference*, 1972.

[34] J. Cocke. Global Common Subexpression Elimination. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Optimization*, pages 20–24, July 1970.

[35] S. Coleman and K.S. McKinley. Tile Size Selection Using Cache Organization and Data Layout. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 279–290, June 1995.

[36] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 238–252, Los Angeles, USA, 1977.

[37] H.J. Curnow and B.A. Wichmann. A Synthetic Benchmark. *The Computer Journal*, 1(19):43–49, 1975.

[38] S. Das and E.E. Johnson. Accuracy of Filtered Traces. In *Proceedings of the International Conference on Computers and Communications*, pages 82–86. IEEE, April 1995.

[39] E. Deelman, R. Bagrodia, R. Sakellariou, and V. Adve. Improving Lookahead in Parallel Discrete Event Simulations of Large-Scale Applications using Compiler Analysis. In *Proceedings of the Workshop on Parallel and Distributed Simulation (PADS)*, 2001.

[40] P. J. Denning. The Working Set Model for Program Behaviour. *Communication of the ACM*, 11(5):323–333, 1968.

[41] P. J. Denning. Virtual Memory. *ACM Computing Surveys*, 2(3):158–189, September 1970.

[42] R.B.K. Dewar and M. Smosna. *Microprocessors: A Programmer's View*. McGraw-Hill, 1990.

[43] J.J. Dongorra. Performance of Various Computers Using Standard Linear Equations Software in FORTRAN Environment. *Computer Architecture News*, 5(11), 1983.

[44] J. Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. Acta Universitatis Upsaliensis, Uppsala Dissertations from the Faculty of Science and Technology 36, 2002.

[45] A.P. Ershov. Mixed Computation: Potential applications and problems for study. *Theoretical Computer Science*, (18):41–67, 1982.

[46] TNI Europe. TNI-Valiosys' DO-178B Software Design Tool Suite Selected by Airbus for the A380. Press release, April 2002.

[47] C. Ferdinand. *Cache Behavior Prediction for Real-Time Systems*. Pirrot Verlag, University of Saarland, Saarbruecken, 1997. Doctoral Dissertation.

[48] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and Precise WCET Determination for a Real-Life Processor. In *Proceedings of the ACM International Workshop on Embedded Software (EMSOFT)*, October 2001. Springer-Verlag, Lecture Notes in Computer Science 2211.

[49] C. Ferdinand, D. Kästner, M. Langenbach, F. Martin, M. Schmidt, J. Schneider, H. Theiling, S. Thesing, and R. Wilhelm. Run-Time Guarantees for Real-Time Systems - The USES Approach. In *Proceedings of Informatik '99 – Arbeitstagung Programmiersprachen*, Paderborn, Germany, 1999.

[50] C. Ferdinand, F. Martin, and R. Wilhelm. Applying Compiler Techniques to Cache Behavior Prediction. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems (LCT-RTS)*, pages 37–46, Las Vegas, USA, June 1997.

[51] M.J. Flynn. *Computer Architecture, Pipelined and Parallel Processor Design*. Jones and Bartlett Publishers, 1995.

[52] M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran. Cache-Oblivious Algorithms. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 285–297, 1999.

[53] S. Furber. *ARM System-on-Chip Architecture*. Addison-Wesley, 2000.

[54] Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[55] K.S. Gatlin and L. Carter. Memory Hierarchy Considerations for Fast Transpose and Bitreversals. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 33–44, January 1999.

[56] A. Gefflaut and P. Joubert. SPAM: a Multiprocessor Execution-Driven Simulation Kernel. *International Journal in Computer Simulation*, 6(1):69–88, 1996.

[57] S. Ghosh, M. Martonosi, and S. Malik. Cache Miss Equations: An Analytical Representation of Cache Misses. In *Proceedings of the ACM International Conference on Supercomputing*, July 1997.

[58] S. Ghosh, M. Martonosi, and S. Malik. Precise Miss Analysis for Program Transformations with Caches of Arbitrary Associativity. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 1998.

[59] S. Ghosh, M. Martonosi, and S. Malik. Automated Cache Optimizations using CME Driven Diagnosis. In *Proceedings of the ACM International Conference on Supercomputing*, Santa Fe, USA, 2000.

[60] A. Grote. Im Schatten des Komforts. *c't Magazin für Computertechnologie*, April 1997.

[61] D. Grunwald, B. Zorn, and R. Henderson. Improving the Cache Locality of Memory Allocation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 28(6) of *ACM SIGPLAN Notices*, pages 177–186, June 1993.

[62] N.J. Gunther. *The Practical Performance Analyst*. iUniverse, 2000.

[63] J. Ha and E.E. Johnson. PDATS: Lossless Address Trace Compression for Reducing File Size and Access Time. In *Proceedings of the IEEE International Conference on Computers and Communications*. IEEE, April 1994.

[64] T.R. Halfhill. Embedded Market Breaks New Ground. *Microprocessor Report*, January 2000.

[65] M. Harman, L. Hierons, A. Baresel, and H. Sthamer. Improving Evolutionary Testing by Flag Removal. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, 2002.

[66] C.A. Healy, R.D. Arnold, F. Mueller, D.B. Whalley, and M.G. Harmon. Bounding Pipeline and Instruction Cache Performance. *IEEE Transactions on Computers*, 48(1):53–70, January 1999.

[67] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The Influence of Processor Architecture on the Design and Results of WCET Tools. *Proceedings of the IEEE Symposium on Real-Time System (RTSS)*, 91(7):1038–1054, July 2003. Special Issue on Real-Time Systems.

[68] P. Heidelberg and H. Stone. Parallel Trace-Driven Simulation by Time Partitioning. In *Proceedings of the Winter Simulation Conference*, 1990.

[69] J.L. Hennessy, D.A. Patterson, and D. Golberg. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2002.

[70] V. Hirvisalo. DBE – A Tool for Trace Driven Memory Simulation. In *Tool Descriptions of International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS)*, pages 5–7, St.Malo, France, June 1997.

[71] V. Hirvisalo. *Compile-time Compaction of Traces for Memory Simulation*. Licentiate's Thesis, Helsinki University of Technology, Laboratory of Information Processing Science, 1998.

[72] V. Hirvisalo. Experience in Performance Analysis of Large Real-Time Systems. In *Proceedings of the Workshop on Software and Performance (WOSP98)*, pages 88–92, Santa Fe, USA, October 1998.

[73] V. Hirvisalo. *A Tool Prototype for Memory Performance Simulation*. Helsinki University of Technology, Laboratory of Information Processing Science, Otaniemi, Finland, 2001. Technical report Pet-41-GEN.

[74] V. Hirvisalo. Combining Static Analysis and Simulation to Speed up Cache Performance Evaluation of Programs. In *Presentations of the Tenth Nordic Workshop on Programming and Software Development Tools and Techniques*, Copenhagen, Denmark, August 2002.

[75] V. Hirvisalo and E. Nuutila. Information Needs in Performance Analysis of Telecommunication Software - a Case Study. In *Proceedings of the ARES International Workshop on Development and Evolution of Software Architectures for Product Families*, Las Navas del Marques, Avila, Spain, November 1996.

[76] V. Hirvisalo, E. Nuutila, and E. Soisalon-Soininen. Transitive Closure Algorithm MEMTC and its Performance Analysis. *Discrete Applied Mathematics*, 110(1):77–84, May 2001.

[77] M. Holliday. Techniques for Cache and Memory Simulation using Address Reference Traces. *International Journal in Computer Simulation*, 1:129–151, 1991.

[78] M. Horowitz, M. Martonosi, T.C. Mowry, and M.D. Smith. Informing Memory Operations: Providing Memory Performance Feedback in Modern Processors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 260–270, May 1996.

[79] S. Horwitz, A. Demers, and T. Teitelbaum. An Efficient General Iterative Algorithm for Dataflow Analysis. *Acta Informatica*, (24):679–694, 1987.

[80] S. Horwitz and T. Reps. The Use of Program Dependence Graphs in Software Engineering. In *Proceedings of the International Conference on Software Engineering*, pages 392–411, Melbourne, Australia, May 1992.

[81] R. Jain. *The Art of Computer Systems Performance Analysis: Technique for experimental design, measurement, simulation and modeling*. John Wiley and Sons, Inc, 1991.

[82] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[83] N.D. Jones, P. Sestoft, and H. Sondergaard. An Experiment in Partial Evaluation: The Generation of a Compiler Generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications, Lecture Notes in Computer Science 202*, pages 124–140. Springer-Verlag, 1985.

[84] N.P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 364–373, June 1990.

[85] B.J. Kam and J.D.. Ullman. Global Data Flow Analysis and Iterative Algorithms. *Journal of the ACM*, 23(1):158–171, 1976.

[86] G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice Hall, 1992.

[87] D. Kästner and S. Thesing. Cache Sensitive Pre-Runtime Scheduling. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 131–145, 1999.

[88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, 1988.

[89] G. Kildall. A Unified Approach to Global Program Optimization. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 194–206, 1973.

[90] S-K. Kim, S.L. Min, and R. Ha. Efficient Worst Case Timing Analysis for Data Caching. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS)*, pages 230–240, Bookline, USA, June 1996.

[91] S.K. Kleene. *Introduction to Metamathematics*. North-Holland, 1952.

[92] S. Laha, J.H. Patel, and K.I. Ravishankar. Accurate Low-Cost Methods for Performance Evaluation of Cache Memory Systems. *IEEE Transactions on Computers*, 37(11):1325–1336, 1988.

[93] M.S. Lam, E.E. Rothberg, and E.W. Wolf. The Cache Performance and Optimization of Blocked Algorithms. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 63–74, April 1991.

[94] A. LaMarca and R.E. Ladner. The Influence of Caches on the Performance of Sorting. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 370–379, January 1997.

[95] J.R. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 291–300, La Jolla, USA, June 1995.

[96] A.M. Law and W.D. Kelton. *Simulation, Modeling and Analysis*. McGraw-Hill, 1991.

[97] A.R. Lebeck and D.A. Wood. Active Memory: A New Abstraction for Memory-System Simulation. In *Proceedings of the ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, pages 220–231, 1995.

[98] J.R. Levine. *Linkers and Loaders*. Morgan Kaufmann, 2000.

[99] Y.-T.S. Li, S. Malik, and A. Wolfe. Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, pages 298–307, 1999.

[100] Y.-T.S. Li, S. Malik, and A. Wolfe. Performance Estimation of Embedded Software with Instruction Cache Modeling. *ACM Transactions on Design Automation of Electronic Systems*, 4(3):257–279, July 1999.

[101] S-S. Lim, S.L. Min, M. Lee, C. Park, H. Shin, and C.S. Kim. An Accurate Instruction Cache Analysis Technique for Real-Time Systems. In *Proceedings of the Workshop on Architectures for Real-Time Applications*, 1994.

[102] J.W.S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[103] L.A. Lombardi. Incremental Computation. In *Advances in Computers*, volume 8, pages 247–333. Academic Press, 1967.

[104] T. Lundqvist and P. Stenström. An Integrated Path and Timing Analysis Method Based on Cycle-Level Symbolic Execution. *Journal of Real-Time Systems*, pages 183–207, November 1999.

[105] T. Lundqvist and P. Stenström. *Empirical Bounds on Data Caching in High-Performance Real-Time Systems*. Chalmers University of Technology, Department of Computer Engineering, 1999. Technical report 99-4.

[106] M. Martonosi, A. Gupta, and T. Anderson. MemSpy: Analyzing Memory System Bottlenecks in Programs. In *Proceedings of the ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, pages 1–12, 1992.

[107] R.L. Mattson, J. Gecsei, D.R. Slutz, and I.L. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[108] C. McGeoch. Analyzing Algorithms by Simulation. *Computing Surveys*, 24(2):195–212, June 1992.

[109] M.K. McKusick, K. Bostic, M.J. Karels, and J.S. Quarterman. *The Design and Implementation of the 4.4BSD UNIX Operation System*. Addison-Wesley, 1996.

[110] F.H. McMahon. Livermore FORTRAN Kernels: A Computer Test of the Numerical Performance. Technical report, Lawrence Livermore National Laboratories, California, USA, 1986.

[111] D.A. Menasce. *Capacity Planning and Performance Modeling*. Prentice Hall, 1994.

[112] N. Mitchell, L. Carter, and J. Ferrante. Localizing Non-affine Array References. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 1999.

[113] T. Mogensen. Partially Static Structures in a Self-Applicable Partial Evaluator. In D. Bjorner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 325–347. North-Holland, 1988.

[114] C.A. Moritz, M.I. Frank, and S. Amarasinghe. FlexCache: A Framework for Flexible Compiler Generated Data Caching. In *Proceedings of the Workshop on Intelligent Memory Systems*. Springer-Verlag, November 2000.

[115] S.S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[116] F. Mueller and J. Wegener. A Comparison of Static Analysis and Evolutionary Testing for the Verification of Timing Constraints. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS)*, pages 179–188, June 1998.

[117] F. Mueller and D.B. Whalley. Efficient On-the-fly Analysis of Program Behavior and Static Cache Simulation. In *Proceedings of the Static Analysis Symposium (SAS)*, 1994.

[118] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[119] E. Nuutila. An Efficient Transitive Closure Algorithm for Cyclic Digraphs. *Information Processing Letters*, 52:207–213, 1994.

[120] G. Ottosson and M. Sjödin. Worst-Case Execution Time Analysis for Modern Hardware Architectures. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems (LCT-RTS)*, June 1997.

[121] P.R. Panda, N.D. Dutt, and A. Nicolau. Memory Data Organization for Improved Cache Performance in Embedded Processor Applications. *ACM Transactions on Design Automation of Electronic Systems*, 2(4), October 1997.

[122] C.Y. Park and A.C. Shaw. Experiments with a Program Timing Tool Based on a Source-Level Timing Schema. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, pages 72–81, December 1990.

[123] C.Y. Park and A.C. Shaw. Experiments with a Program Timing Tool Based on a Source-Level Timing Schema. *IEEE Computer*, 24(5):48–57, May 1991.

[124] N. Park, D. Kang, K. Bondalapati, and V.K. Prasanna. Dynamic Data Layouts for Cache-conscious Factorization of DFT. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 693–702, May 2000.

[125] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A Case for Intelligent RAM. *IEEE Micro*, 17(2):34–44, April 1997.

[126] M. Penner and K. Prasanna. Cache-Friendly Implementations of Transitive Closure. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Barcelona, Spain, September 2001.

[127] K. Pettis and R.C. Hansen. Profile Guided Code Positioning. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 16–27, 1990.

[128] S.A. Przybylski. *Cache and Memory Hierarchy Design*. Morgan Kaufmann Publishers, Inc., Palo Alto, 1990.

[129] T.R. Puzak. *Analysis of Cache Replacement Algorithms*. University of Massachusetts, Department of Electrical and Computer Engineering, 1985. PhD thesis.

[130] J. Rawat. Static Analysis of Cache Performance for Real-Time Programming, 1993. Master thesis TR93-19, Iowa State University of Science and Technology.

[131] S.K. Reinhardt, M.D. Hill, J.R. Larus, A.R. Lebeck, J.C. Lewis, and D.A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proceedings of the ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, 1993.

[132] G. Rivera and C.-W. Tseng. Data Transformations for Eliminating Conflict Misses. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 38–49, May 1998.

[133] P. Sanders. Accessing Multiple Sequences Through Set Associative Caches. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP), Lecture Notes in Computer Science 1644*, pages 655–664, Prague, Czech Republic, December 1999. Springer-Verlag.

[134] P. Sanders. Fast Priority Queues for Cached Memory. In *Proceedings of the Workshop on Algorithm Engineering and Experimentation, Lecture Notes in Computer Science 1619*, pages 312–327, 1999.

[135] R.G. Sargent. A Tutorial on Validation and Verification of Simulation Models. In *Proceedings of the Winter Simulation Conference*, ACM SIGPLAN Notices, pages 33–39, 1988.

[136] J. Schneider. *Combined Schedulability and WCET Analysis for Real-Time Operating Systems*. Shaker Verlag, University of Saarland, Saarbruecken, 2003. Doctoral Dissertation.

[137] J. Schneider and C. Ferdinand. Pipeline Behavior Prediction for Superscalar Processors. Technical Report A/02/99, Universität des Saarlandes, 1999.

[138] T. Shanley. *PowerPC 601 System Architecture*. Mindshare Press, 1994.

[139] M. Sharir. Structural Analysis: A New Approach to Flow Analysis in Optimizing Compilers. *Computer Languages*, (5):141–153, 1980.

[140] A. Silberschatz, J. L. Peterson, and P. Galvin. *Operating System Concepts*. Addison-Wesley, 1991.

[141] A.J. Smith. Two Methods for the Efficient Analysis of Memory Address Trace Data. *IEEE Transaction on Software Engineering*, SE-3(1), 1977.

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

[143] C.U. Smith. Applying synthesis principles to create responsive software systems. *IEEE Transaction on Software Engineering*, SE-14(10), October 1988.

[144] C.U. Smith. *Performance Engineering of Software Systems*. Addison-Wesley, 1990.

[145] C.U. Smith and L.G. Williams. Performance Engineering Models of CORBA-based Distributed-Object Systems. In *Proceedings of the Computer Measurement Group Conference*, pages 886–898, 1998.

[146] C.U. Smith and L.G. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley, 2002.

[147] W. Stallings. *Computer Organization and Architecture - Designing for Performance*. Prentice Hall, 1996.

[148] W. Stallings. *Operating Systems*. Prentice Hall, 2002.

[149] F. Stappert and P. Altenbernd. Complete Worst-Case Execution Time Analysis of Straight-Line Hard Real-Time Programs. *Journal of System Architecture*, 46(4):339–355, 2000.

[150] System Performance Evaluation Cooperative. SPEC Benchmark Suite Release 1.0. *SPEC Newsletter*, 2(2):3–4, 1990.

[151] R.E. Tarjan. Depth First Search and Linear Graph Algorithms. *SIAM Journal of Computing*, 1(2):146–160, June 1972.

[152] P.J. Teller. Translation-Lookaside Buffer Consistency. *IEEE Computer*, 23(6):26–36, 1990.

[153] F. Tip. A Survey of Program Slicing Techniques. *Journal of programming languages*, 3(3):121–189, September 1995.

[154] M. Tofte and J.-P. Talpin. Region-based Memory Management. *Information and Computation*, 132(2):109–176, February 1997.

[155] Transaction Processing Performance Council. TCP Benchmark A, Proposed Standard 5E. Technical report, ITOM International Co., California, USA, 1986.

[156] R.A. Uhlig and T.N. Mudge. Trace-driven Memory Simulations: A Survey. *ACM Computing Surveys*, 29(2):128–170, June 1997.

[157] J.S. Vitter and M.H. Nodine. Large-Scale Sorting in Uniform Memory Hierarchies. *Journal of Parallel and Distributed Computing*, 17(1–2):107–114, January and February 1993.

[158] R. Weicker. Dhrystone. Siemens, 1984.

[159] M. Weiser. *Program Slicing: Formal, Psychological and Practical Investigations of an Automatic Program Abstraction Method*. The University of Michigan, Ann Arbor, USA, 1979. PhD thesis.

[160] R.T. White, F. Mueller, C.A. Healy, D.B. Whalley, and M.G. Harmon. Timing Analysis for Data Caches and Set-Associative Caches. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS)*, pages 192–202, June 1997.

[161] D. Whitfield and M.L. Soffa. An Approach to Ordering Optimizing Transformations. In *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming*, pages 137–146, March 1990.

[162] R. Wilhelm and D. Maurer. *Compiler Design*. Addison-Wesley, 1995.

[163] P.R. Wilson. Uniprocessor Garbage Collection Techniques. In *Proceedings of the International Workshop on Memory Management*, pages 1–42. Lecture Notes in Computer Science 637, Springer-Verlag, 1992.

[164] P.R. Wilson, M.S. Johnstone, M. Neely, and D. Boles. Dynamic Storage Allocation: A Survey and Critical Review. In *Proceedings of the International Workshop on Memory Management*, pages 1–116. Lecture Notes in Computer Science 986, Springer-Verlag, 1995.

[165] E. Witchel, S. Larsen, C.S. Ananian, and K. Asanovic. Direct Addressed Caches for Reduced Power Consumption. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, December 2001.

[166] F. Wolf and R. Ernst. Data Flow Based Cache Prediction Using Local Simulation. In *Proceedings of the IEEE High Level Design Validation and Test Workshop*, pages 155–160, November 2000.

[167] M.E. Wolf and M.S. Lam. A Data Locality Optimizing Algorithm. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 26(6), pages 30–44, 1991.