THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

BOUNDING WORST-CASE DATA CACHE PERFORMANCE

By

RANDALL T. WHITE

A Dissertation submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Degree Awarded:
Spring Semester, 1997

The members of the Committee approve the dissertation of Randall T. White defended on April 1, 1997.

David B. Whalley
Professor Directing Dissertation

Steven F. Bellenot
Outside Committee Member

Theodore P. Baker
Committee Member

R. C. Lacher
Committee Member

Gregory A. Riccardi
Committee Member

Approved:

R. C. Lacher, Chair, Department of Computer Science

To

My parents, R. B. and Margie White,

My grandmother, Martha Cox,

and

In Loving Memory of

My grandfather, L. T. Cox, Sr.

Acknowledgements

I would first and foremost like to thank Dr. David Whalley, my major professor, for his guidance, patience, and support throughout this entire project. He was always available to listen to my questions and concerns, to offer insights and solutions, and to discuss my progress on the often quite difficult programming tasks involved with this dissertation. He also deserves a reward for putting up with my constant questions and grouching about VPO!

I am also grateful to the members of my committee, Dr. Ted Baker, Dr. Greg Riccardi, Dr. Steven Bellenot, and Dr. Chris Lacher for their help and support.

I would also like to thank Frank Mueller for his helping me with my initial understanding of the static simulator code. I've abused it so much now that he would hardly recognize it! I would also like to thank Chris Healy for his help with my modifications to both the timing analyzer and the pipeline simulation portion of the Ease execution simulator.

I am forever grateful to my family, especially my parents and my sister Debbie, for their generous support throughout my seemingly endless years in graduate school. I would have never made it this far without their love, prayers, and belief in me.

The research on which this dissertation is based was supported in part by the Office of Naval Research under contract number N00014-94-1-006.

Contents

List of Tables vii

List of Figures viii

Abstract ix

1 Introduction 1

1.1 Challenges 1

1.2 Related Work 3

1.3 Organization of Dissertation 5

2 Overview of Research Environment 6

3 Calculation of Relative Addresses in the Compiler 8

3.1 Front-End 8

3.2 Back-End 8

3.3 The DNF File 15

4 Calculation of Virtual Addresses 22

5 Static Simulation to Produce Data Reference Categorizations 26

6 Timing Analysis 34

7 Execution Simulation of Data Cache and Pipeline Effects 41

8 Results 46

9 Future Work 53

9.1 Merging Instruction and Data Caching Prediction and Simulation 53

9.2 Wrap-Around Fill for Data Caches 53

9.3 Write Buffer 54

9.4 Best Case 54

9.5 Compiler Optimizations 54

10 Conclusions 55

Appendices 57

A Memory References in Annulled Delay Slots 57

B Description of the DNF File 63

References 68

Bibliographical Sketch 71

List of Tables

6.1 Timing Analysis Steps for the loop in Figure 6.2 39

8.1 Test Programs 47

8.2 Dynamic Results 48

List of Figures

1.1 Data vs. Instruction Memory Mapping 2

2.1 Overview of Bounding Data Cache Performance 6

3.1 Example C Program, RTLS, and SPARC Assembly 10

3.2 Algorithmic Range of Relative Addresses for the load in Figure 3.1 11

3.3 Example in Figure 3.1 without Strength Reduction in Outer Loop 12

3.4 Algorithmic Range of Relative Addresses for the load in Figure 3.3 14

3.5 Portion of a DNF File for Function Initialize 17

3.6 C Code, RTLS, and SPARC Assembly for Function Initialize . . 19

4.1 Virtual Address Space Organization in SunOS 23

4.2 Algorithmic Range of Virtual Addresses for the Load in Figure 3.3 25

5.1 Algorithm to Calculate Data Cache States 27

5.2 Instruction Vs. Data Cache State Representation 28

5.3 Detecting Spatial Locality 30

5.4 Detecting Temporal Locality Across and Within Loops 32

6.1 Worst-Case Loop Analysis Algorithm 35

6.2 Example to Illustrate Worst-Case Loop Analysis Algorithm . . . 38

6.3 Pipeline Diagrams for the Two Paths in Figure 6.2 38

A.1 Example C Program, RTLS, and SPARC Assembly, Revised . . 59

Abstract

Tightly predicting worst-case execution times (WCETs) of programs on real-time systems with caches is difficult. Whether or not a particular reference is in cache depends on the program's previous dynamic behavior. While much progress has been accomplished recently on predicting instruction cache performance of programs, bounding worst-case data cache performance is significantly more challenging. Unlike instruction caching, many of the data addresses referenced by load and store instructions can change during the execution of a program. This dissertation describes an automatic tool-based approach for statically bounding the worst-case data cache performance of large code segments. It also presents the work done to verify the validity of the computed bounds. The given approach works on fully optimized code, performs the analysis over the entire control flow of a program, detects and exploits both spatial and temporal locality within data references, produces results typically within a few seconds, and produces, on average, 30% tighter WCET bounds than can be predicted without analyzing data cache behavior.

The given method of timing analysis involves several steps. First, data flow analysis within an optimizing compiler is used to determine the bounded range of addresses of each data reference relative to a global symbol or activation record. Second, virtual address ranges are calculated from the relative address ranges by examining the order of the assembly data declarations and the call

graph of the entire program. Third, the control flow of the program is analyzed to statically categorize the caching behavior of each data reference. Fourth, these categorizations are used when calculating the pipeline performance of sequences of instructions representing paths within the program. Finally, the pipeline path analysis is used to estimate the worst-case execution performance of each loop and function in the program.

Overall, this dissertation presents a comprehensive report on methods and results of worst-case timing analysis of data cache behavior and shows that such an analysis can lead to a significantly tighter worst-case performance prediction. The given approach is unique and provides a considerable step towards realistic worst-case execution time prediction of contemporary architectures and its use in schedulability analysis for real-time systems.

Chapter 1
Introduction

Real-time systems rely on the assumption that the worst-case execution time (WCET) of hard real-time tasks be known to ensure that deadlines of tasks can be met – otherwise the safety of the controlled system is jeopardized. Static analysis of program segments corresponding to tasks provides an analytical approach to determine the WCET for modern processors. The complexity of these processors requires a tool-based approach since *ad hoc* testing methods may not exhibit the worst-case behavior of the architecture. This dissertation presents a working system of tools that can statically analyze optimized code and produce the data cache timing analysis for real-time programs without requiring interaction from the user. Furthermore, those programs are allowed to take full advantage of many features of modern architectures, including pipelining and data caches.

1.1 Challenges

Obtaining WCETs for real-time applications on systems that use a data cache is quite challenging. Unlike instruction caching, many of the addresses of references to data can change during the execution of a program. A reference to an item within an activation record could have different addresses depending on the sequence of calls associated with the invocation of the function. Many data references, such as indexing into an array, are dynamically calculated and can vary

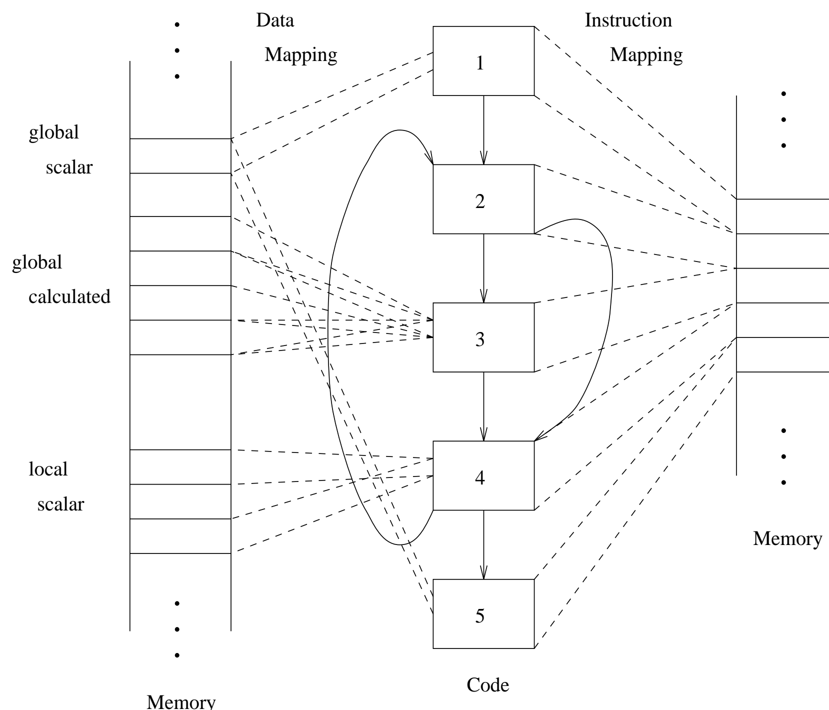


Figure 1.1: Data vs. Instruction Memory Mapping

each time the data reference occurs. Pointer variables in languages like C may be assigned addresses of different variables or an address that is dynamically calculated from the heap. Instruction caches also allow for a simple 1-1 mapping from instructions to memory. This is not possible for data caches. Figure 1.1 shows how the mapping of data addresses differs from that of instructions. A global variable may be accessed from more than one place in the code (blocks 1 and 5). A global calculated reference may appear textually in the program only once (block 3), but will access many different addresses. Likewise, a local variable may appear only once (block 4), but will map to different addresses in the stack depending on the calling sequence of the function in which it appears.

Initially, it may appear that obtaining a reasonable bound on worst-case

data cache performance is just not feasible. However, this problem is far from hopeless since the addresses for many data references can be statically calculated. Static or global scalar data references do retain the same addresses during the execution of a program. Run-time stack scalar data references can often be statically determined as a set of addresses depending upon the sequence of calls associated with an invocation of a function. The pattern of addresses associated with many calculated references, such as indexing through an array, can also often be resolved statically.

1.2 Related Work

There has been some work in the area of locality analysis to assist compilers performing optimizations [18, 27]. Generally, this work only attempts to estimate the locality of data within a set of nested loops. The effect of scalar references is not typically considered. Furthermore, this analysis usually occurs for high-level optimizations, where the exact machine instructions and virtual addresses of the data references are unknown.

In the past few years, research in the static analysis of WCET of programs has increased. Conventional methods for static analysis have been extended from unoptimized programs on simple CISC processors [7, 22, 21] to optimized programs on pipelined RISC processors [9, 17, 28] and from uncached architectures to instruction caches [3, 11, 15]. However, there has been little previous work on predicting WCET for data caching. Only three previous attempts have been reported. Rawat and Nilsen [23] used a graph coloring approach to bound data caching performance. However, only the live ranges of local scalar variables

within a single function were analyzed. Unfortunately, these types of references are fairly uncommon since most local scalar variables are allocated to registers by optimizing compilers and actually do not reference memory.

Kim *et. al.* [12] have recently published work about bounding data cache performance for calculated references, which they refer to as occurring from dynamic load and store instructions. Their approach uses a pigeonhole principle. For each loop they determine the maximum number of references from each dynamic load/store instruction. They also determine the maximum number of distinct locations in memory referenced by these instructions. The difference between these two values is the number of data cache hits for the loop given that there are no conflicting references. This technique works quite well when all of the data references fit into cache and the size of each data reference is the same size as a cache line. Unfortunately, their technique does not detect any spatial locality (i.e., when the line size is greater than the size of each data reference and the elements are accessed contiguously) and detects no temporal locality across different loop nests. Furthermore, no general analysis of induction variables and loop invariant values is performed to calculate relative ranges of data references. Instead, they rely on analyzing unoptimized code, where registers associated with a memory address have to correspond to a loop index variable. As will be shown in this dissertation, compiler optimizations can make the process of calculating ranges of relative addresses significantly more challenging.

Li *et. al.* [16] have described a framework to integrate data caching into their integer linear programming (ILP) approach to timing prediction. Their implementation performs data-flow analysis to find conflicting blocks. However,

their linear constraints describing the range of addresses of each data reference currently have to be calculated by hand. They also require a separate constraint for *every* element of a calculated reference, which causes a scalability problem for large arrays. No actual WCET results on data caches were reported. Their approach can also incur significant overhead. In fact, one program in their test suite required over 50,000 seconds (nearly 14 hours) for WCET results to be produced with instruction caches being considered. The number of constraints needed for large arrays can make overhead for data cache prediction even higher.

1.3 Organization of Dissertation

Chapter 2 gives a brief overview of the different stages involved in the bounding of worst-case data cache performance. Chapter 3 discusses changes made to the *vpo* compiler [4] to allow the calculation of relative address information. In Chapter 4 the method of computing virtual addresses is discussed. Chapter 5 describes the method of static cache simulation to categorize the data references in a program. In Chapter 6 the timing analysis algorithm is given along with an example showing its use. Chapter 7 describes the changes made to the ease cache simulator [6] in order to check the validity of the computed addresses and to simulate data caching and pipeline effects of executing the test programs. Chapter 8 shows the results of the work on various representative programs. Chapter 9 suggests topics for future research, and Chapter 10 gives the conclusions. In addition there are two appendices. Appendix A discusses the difficulties of dealing with a load or store in an annulled delay slot. Appendix B gives a grammatical description of the data information (`dnf`) file.

Chapter 2
Overview of Research Environment

There are several stages involved in the approach used for bounding data cache performance of large code segments. Figure 2.1 depicts an overview of these stages. An optimizing compiler [4] has been modified to store information about the control flow, which includes the calling structure of functions, data declarations, and relative address ranges of data references within each function as the side effect of the compilation of each source file. This information is passed to an address calculator, which converts the relative data addresses to virtual data addresses. The control flow is input to the static cache simulator, which uses it to construct a control-flow graph of the entire program. The static simulator uses the information produced by the address calculator and the specified data cache configuration when analyzing this graph to produce a categorization of

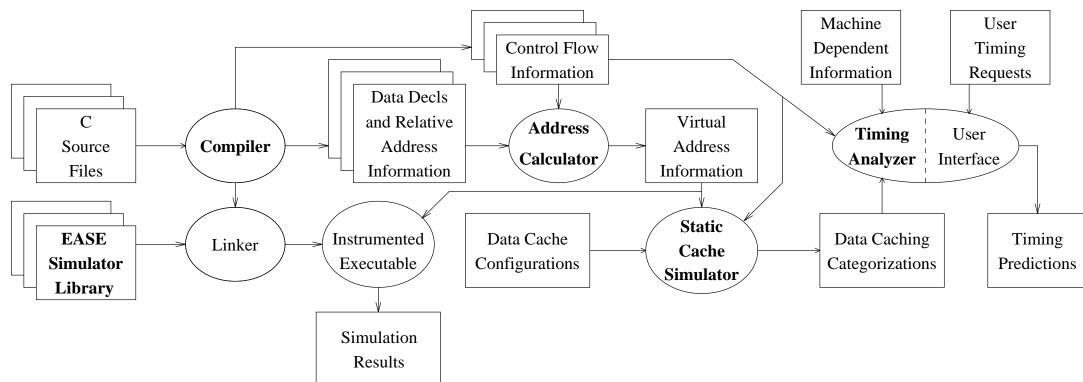


Figure 2.1: Overview of Bounding Data Cache Performance

each data reference in the program. This static stimulation is based on previous work described in [19]. The timing analyzer reads machine-dependent information to determine how each instruction proceeds through the pipeline. It uses the data categorizations to determine whether each data reference should be treated as a hit or miss during the pipeline analysis. The timing analyzer also uses the control flow information to produce a program control-flow graph, which is analyzed to produce a worst-case estimate for each loop and function within the program. The timing analysis is based on earlier work described in [2, 8]. Finally, a graphical user interface is displayed that allows a user to request timing bounds for portions of the program [14, 13]. Excerpts of this dissertation describing the approach to statically bounding WCETs for programs that use a data cache, along with extensions to the static simulator to produce categorizations for instructions in programs using a set associative cache, can be found in [26].

In addition to the components mentioned above, an execution simulator was developed to check the validity of the statically computed WCETs. An existing cache simulator (EASE, described in [6]) was modified to measure the simulated pipeline and data caching behavior of a program's execution. This simulator is also used to check the validity of the computed addresses by comparing them against the addresses that are actually accessed at run time. If any actual address is found to be outside the computed range, an error message is output and the simulation terminates.

Chapter 3

Calculation of Relative Addresses in the Compiler

3.1 Front-End

The front-end of the compiler (*cfe – C Front End*) [5] was modified to pass information about declarations of static data to the compiler back-end. Normally, this information is discarded after the static data assembly declarations are emitted. However, it is necessary for the address calculator to know some of this information, which includes the name, size, and alignment requirements for each static data declaration.

3.2 Back-End

The back-end of the compiler (*vpo – Very Portable Optimizer*) [4] required more intensive modifications than cfe. Since a goal of the research was to allow the user to take advantage of fully optimized code, there were many instances where traditional techniques for calculating ranges of relative addresses were not adequate due to interference from various compiler optimizations.

Vpo attempts to calculate relative addresses for each data reference associated with load and store instructions after other compiler optimizations have been performed. First, the compiler determines for each loop the set of its induction variables, their initial values and strides, and the loop-invariant registers.[1]

[1] A basic loop induction variable only has assignments of the form `v := v` $\pm$ `c`, where `v` is a variable or register and `c` is an integer constant. Nonbasic induction variables are also only incremented or decremented by a constant value on each loop iteration, but get their values

Expansion of addresses used in loads and stores is then performed. Expansion is accomplished by examining each preceding RTL and replacing registers used as source values in the address with the source of the RTL which set that register. Induction variables associated with a loop are not expanded. Loop invariant values are expanded by moving to the end of the preheader block of that loop. Expansion of addresses of scalar references to the run-time stack (e.g. local variables) is trivial. Expansion of references to static data (e.g. global variables) often requires expanding loop invariant registers since these addresses are constructed with instructions that may be moved out of a loop. Expansion of calculated address references (e.g. array indexing) requires knowledge of loop induction variables.

Consider the source code and corresponding RTLs, which represent SPARC [2] assembly instructions, in Figure 3.1. Note that although delay slots are actually filled by the compiler, they have not been filled when compiling code for the examples in this section in order to simplify the examples for the initial explanation of the expansion and simplification of memory address information. A more detailed discussion of the ramifications of memory references in (possibly annulled) delay slots is given in Appendix A.

The memory reference in the load (instruction 9) references the register `r[13]`, which is a basic induction variable for the inner loop (instructions 9-13) and will not be expanded. The stride for `r[13]` was calculated to be 4. The

---

either directly or indirectly from basic induction variables. A variety of forms of assignment for nonbasic induction variables are allowed. Loop invariant values do not change during the execution of a loop. A discussion of how induction variables and loop invariant values are identified can be found elsewhere [1].

[2]SPARC is a registered trademark of Sparc International, Inc.

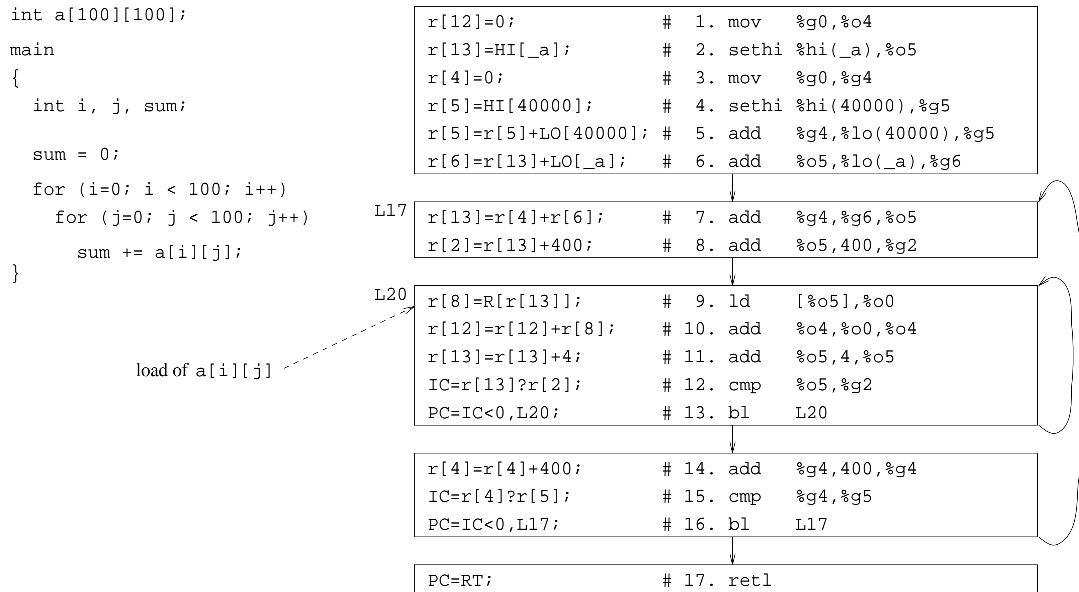


Figure 3.1: Example C Program, RTLS, and SPARC Assembly

compiler determined its initial value to be `r[4]+r[6]` by expanding `r[13]` at the end of the inner loop preheader, which precedes the header block at `L20`. The register `r[4]` is a basic induction variable for the outer loop (instructions 7-16). However, the `r[6]` register is loop invariant in the outer loop and this initial address also needs to be expanded. The expansion of the initial address accessed by the load is shown in the following steps.

1. `r[13]` # address expression in load at instruction 9
2. `r[4]+r[6]` # from instruction 7
3. `r[4]+(r[13]+LO[_a])` # from instruction 6
4. `r[4]+(HI[_a]+LO[_a])` # from instruction 2

Address simplification is performed to obtain the following expression as an initial address. The `HI` and `LO` correspond to the high and low portions of

```
for (r[4] = 0; r[4] < 40000; r[4] += 400)
   for (r[13] = r[4]+_a; r[13] < r[4]+_a+400; r[13] += 4)
      address{r[13]}
```

Figure 3.2: Algorithmic Range of Relative Addresses for the load in Figure 3.1

the address since the address calculation cannot be accomplished in a single instruction on the SPARC.

5. `r[4]+_a` # eliminate HI and LO and unnecessary parentheses

The relative range of addresses is calculated using the initial value and stride of these induction variables and the number of iterations of the loop to determine a limit. The compiler determined that the initial value for `r[4]` is 0 and its stride is 400. The number of iterations for each loop is calculated to be 100 by the compiler. The range of relative addresses for this example can be depicted algorithmically as shown in Figure 3.2.

Unfortunately, the calculation of a range of relative addresses is not always so simple. Occasionally, the compiler optimization called loop strength reduction cannot be applied due to lack of available registers. To address this problem a more sophisticated address simplification algorithm was developed. To illustrate the capabilities of this algorithm, the same example in Figure 3.1 was recompiled without loop strength reduction being applied to the outer loop. Figure 3.3 contains the instructions generated in this case.

The effect of the memory address in the load (instruction 12), which is `r[2]`, has to be expanded. The register `r[2]` cannot be immediately expanded since it is an induction variable for the inner loop (instructions 12-16). The compiler

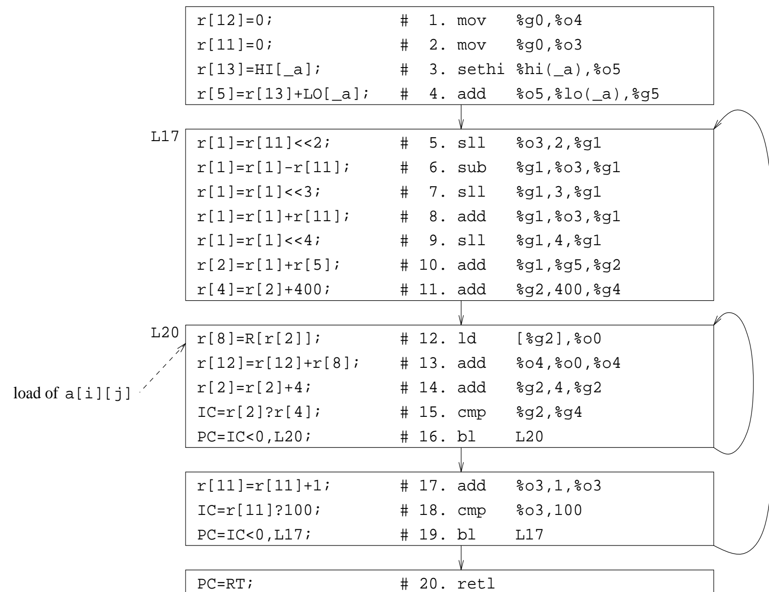


Figure 3.3: Example in Figure 3.1 without Strength Reduction in Outer Loop

next attempts to determine the initial value for this induction variable. This is accomplished by moving to the end of the preheader of the inner loop, which precedes the loop header block at `L20`. The initial value of `r[2]` is `r[1]+r[5]`. The result of expanding the initial memory address within the preheader is shown in the following steps.

1. `r[2]` # from load at instruction 12
2. `r[1]+r[5]` # from instruction 10
3. `(r[1]<<4)+r[5]` # from instruction 9
4. `((r[1]+r[11])<<4)+r[5]` # from instruction 8
5. `(((r[1]<<3)+r[11])<<4)+r[5]` # from instruction 7
6. `((((r[1]-r[11])<<3)+r[11])<<4)+r[5]` # from instruction 6
7. `(((((r[11]<<2)-r[11])<<3)+r[11])<<4)+r[5]` # from instruction 5

The register `r[11]` is an induction variable for the outer loop and will not be expanded. The register `r[5]` cannot be immediately expanded since it is loop invariant. The compiler moves to the end of the outer loop's (instructions 5-19) preheader, which precedes the header block at `L17`. The assignments to `r[5]` are used to further expand the initial address in the two following steps.

8. `(((((r[11]<<2)-r[11])<<3)+r[11])<<4)+(r[13]+LO[_a])`
   \# from instruction 4
9. `(((((r[11]<<2)-r[11])<<3)+r[11])<<4)+(HI[_a]+LO[_a])`
   \# from instruction 3

Now the expression needs to be simplified. The reason it appears so complicated is that the compiler had performed an optimization that turned a multiply by an integer constant into a sequence of shifts, adds, and subtracts. This multiplication was unnecessary when loop strength reduction had been performed on the outer loop. The HI and LO calculation is eliminated and the effect of this optimization is reversed in the following steps.

10. `(((((r[11]<<2)-r[11])<<3)+r[11])<<4)+_a` # eliminate HI and LO
11. `(((((r[11]*4)-r[11])<<3)+r[11])<<4)+_a` # change `<<2` to `*4`
12. `((((r[11]*3)<<3)+r[11])<<4)+_a` # change *n*`*4-`*n* to *n*`*3`
13. `((((r[11]*3)*8)+r[11])<<4)+_a` # change `<<3` to `*8`
14. `(((r[11]*24)+r[11])<<4)+_a` # change `*3*8` to `*24`
15. `((r[11]*25)<<4)+_a` # change *n*`*24+`*n* to *n*`*25`
16. `((r[11]*25)*16)+_a` # change `<<4` to `*16`
17. `(r[11]*400)+_a` # change `*25*16` to `*400`

```
for (r[11] = 0; r[11] < 100; r[11] += 1)
   for (r[2] = r[11]*400+_a; r[2] < r[11]*400+_a+400; r[2] += 4)
      address{r[2]}
```

Figure 3.4: Algorithmic Range of Relative Addresses for the load in Figure 3.3

At this point the initial address of the load has been fully expanded. The only remaining register in the initial address is an induction variable. The initial value and stride of these induction variables and the number of iterations of the loop to determine a limit is used by the compiler to calculate the relative range of addresses. The initial value of the inner induction variable `r[2]` was given at step 17 and its stride is 4. The initial value of the outer induction variable `r[11]` is 0 and the stride is 1. The number of iterations of each loop is 100. Thus, the range of relatives addresses for this example can be depicted algorithmically as shown in Figure 3.4.

This approach to expanding addresses allows us the ability to handle non-standard induction variables. We are not limited to simple induction variables in simple `for` loops that are updated only at the head of the loop. For example, consider the following code segment.

```
for (i = 0; i < 100; i++) {
   ... a[i] ...          – first use of i as index into a
     ⋮
   i++;                  – induction variable i incremented inside body of the loop
     ⋮
   ... a[i] ...          – second use of i as index into a
     ⋮
}
```

The first use of `i` to index into `a` within the loop body uses the value to which it was initialized or incremented in the `for` loop header statement. However, since `i` is incremented in the middle of the loop, the next use of `i` as an index into `a` will have a value that is one more than it was before. Our approach will detect this difference and correctly determine the relative address range for each memory reference.

3.3 The DNF File

After relative address information is computed, it needs to be passed to the address calculator. The data information, or `dnf`, file serves this purpose. Its format was chosen for ease of input into the address calculator and for human readability. A grammar showing the full organization of the `dnf` file is given in Appendix B. The `dnf` file contains lines giving both the static location of the data references in the file, i.e., which function and basic block in which each particular reference resides, and the dynamic control flow of the program. However, its primary purpose is to give information about each data reference in a program to the address calculator so that virtual addresses can be computed. The following describes typical information the `dnf` file contains for each data reference in the program.

variable name: This is either the name of the local or global variable or an address string consisting of a base name with possible registers and constant offsets for calculated references.

type: One of the letters `B`, `W`, `R`, `F`, or `D`, for bytes (1 byte), shorts (2 bytes), ints (4 bytes), floats (4 bytes), or doubles (8 bytes), respectively, giving

the size and alignment requirements of a particular memory reference or a sequence of memory references.

relative offset: There are lines that give offsets relative to the current activation record for locals and relative to the beginning of global data for globals. For calculated references these offsets are used to calculate a base address, and there are additional lines which give a list of induction variables from which the actual virtual range of addresses that can be accessed, as well as the order in which these addresses will be accessed, may be computed.

access type: This is the letter `r` or `w`, telling whether this reference is a read from or a write to memory. Reads (loads) and writes (stores) may be accessed differently and can cause different effects on the data cache.

instruction number: The number of the load or store instruction, relative to the current function.

data reference number: The number of the current data reference, starting at 0, relative to the current source file.

Figure 3.5 shows a portion of the `dnf` file containing the data reference information for a function `Initialize`, which takes two arrays as parameters and sets the corresponding elements in each to the same random integers. Note that the line numbers are not actually part of the file but are given in the figure for demonstration purposes. The calculated reference information lines in this file portion, along with the two lines following each of them (lines 23-28), are the

```
          ⋮
 1   FN Main
          ⋮
 2   FC Initialize
 3   PM 2 ArrayA ArrayB
          ⋮
 4   FN Initialize
 5   SP 96
 6   -- 1 --
 7   SL 2
 8   NI 5
 9   -- 2 --
10   LI 10
11   SL 3
12   P 6 1 -1
13   NI 6
14   -- 3 --
15   LI 10
16   SL 4
17   P 5 2 -1
18   FC Rand
19   NI 2
20   -- 4 --
21   SL 5
22   P 3 -1
23   CR w 13 R 9 0
24   BA r[21]+r[25]-r[24] 4 10 10 3 0
25   IV 1 r[21] 2 4 10 10 r[24]+r[18]+4 1 r[18] 1 44 10 10 44 1 ...
     ...r[26] 1 44 10 10 r[24]+88 0
26   CR w 14 R 10 0
27   BA r[21] 4 10 10 3 0
28   IV 1 r[21] 2 4 10 10 r[24]+r[18]+4 1 r[18] 1 44 10 10 44 1 ...
     ...r[26] 1 44 10 10 r[24]+88 0
29   NI 2
30   -- 5 --
31   SL 6
32   SR 3
33   P 4 -1
34   NI 4
          ⋮
```

Figure 3.5: Portion of a DNF File for Function Initialize

`dnf` file result of an interesting example of the expansion and simplification of address strings for calculated references. These lines show how the system can handle an optimization called *index reduction* in which the induction variable calculation for one memory reference is "piggy-backed" onto that of another when there are two different memory references with identical strides in the same loop. They also show how the computation of relative address information is accomplished when dealing with calculated references that have address components that are passed in as parameters.

Consider the C source code, RTLs, and SPARC assembly instructions in Figure 3.6 for the `Initialize` function. The first memory address, `r[21] + r[22]` (instruction 14), is for the store of `B[i][j]`. The second memory address, `r[21]` (instruction 15), is for the store of `A[i][j]`. Register `r[21]` is the induction register for the inner loop (instructions 12-19) and thus cannot be expanded. It has an initial value, a stride, and a maximum and minimum number of iterations associated with it that were computed and stored earlier in the compilation process.[3] These values for induction register `r[21]` can be seen in the `IV` lines (lines 25 and 28) following the `CR` lines (lines 23 and 26) for both arrays `A` and `B` in Figure 3.5.

The initial value for `r[21]` is `r[24]+r[18]+4`. `r[24]` is the same as register `i[0]`, the first input register. The SPARC uses the concept of a register window [24] to pass the first six arguments to a function via six special input registers. Since `r[24]` was seen to be a special register during the initial value

[3]This earlier computation and expansion of the initial value string of an induction register proceeds in basically the same manner as has already been discussed except that loop invariant registers are expanded as well.

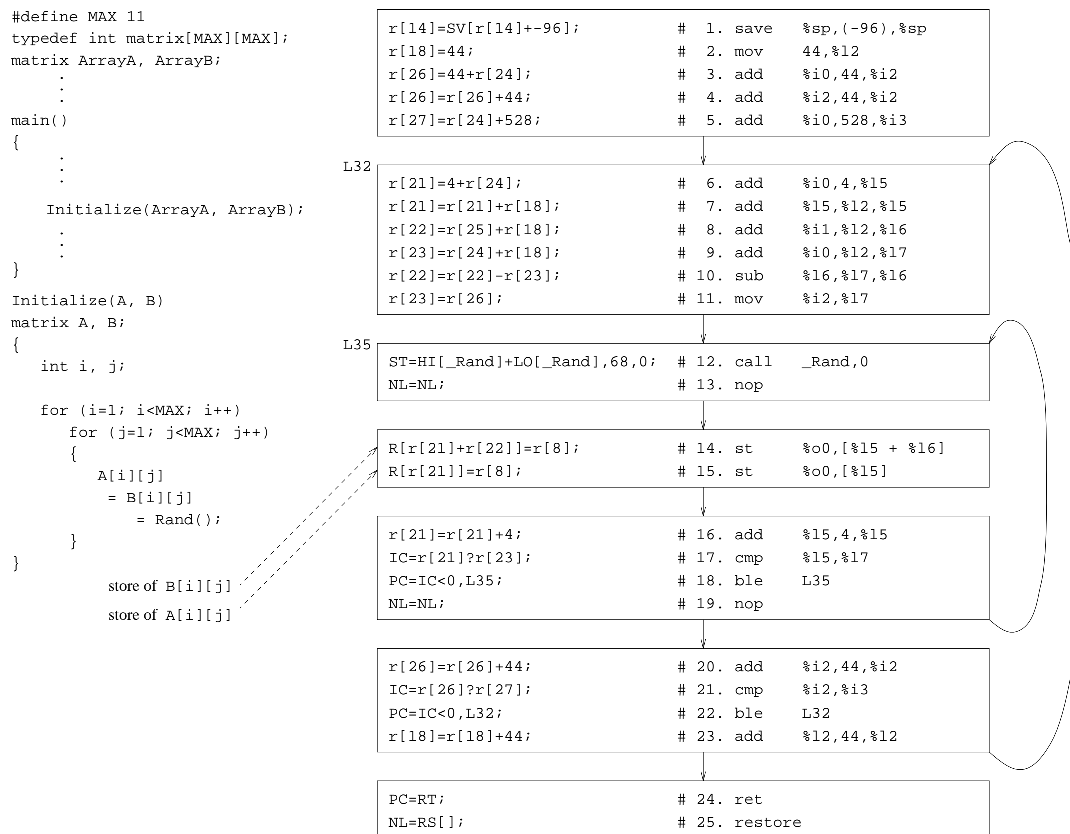


Figure 3.6: C Code, RTLS, and SPARC Assembly for Function Initialize

expansion, no attempt was made to expand it further. During virtual address calculation later, parameter information will be searched to find the appropriate global or local (to the caller) base address to use in its place. The parameter information dumped for this example can be seen in the PM line (line 3) of Figure 3.5. Such a line is dumped out after every function call in the program for which parameter information could be readily expanded. Therefore, the initial value of `r[21]` can be seen to consist of the first element accessed `r[24]+4` plus the offset that comes from computing the row location, that of the induction variable for the outer loop (instructions 6-23), `r[18]`. The stride is 4 and the

minimum and maximum number of iterations are both 10.

Once the initial value, stride, and number of iterations are available, there is enough information to compute the sequence of addresses that will be accessed by the store of `A[i][j]`. Knowing that both references had the same stride, the compiler used index reduction to avoid having to use another induction register for the address computation for `B[i][j]` since it shares the same loop control variables as that for `A`. Therefore, the memory address, `r[21] + r[22]` for `B[i][j]`, includes the address for `A` (`r[21]`) plus the difference between the two arrays (`r[22]`). This can be seen from the following sequence of expansions and simplifications. Remember that register `r[21]` cannot be immediately expanded since it is an induction register for the inner loop, so the expansion continues with register `r[22]` as follows. Also, register `r[18]` will not be expanded since it is the induction variable for the outer loop. Furthermore, registers `r[24]` and `r[25]` are input registers corresponding to parameters and will not be expanded either.

1. `r[21] + r[22]` # from load at 14
2. `r[21]+(r[22]-r[23])` # from inst 10
3. `r[21]+(r[22]-(r[24]+r[18]))` # from inst 9
4. `r[21]+((r[25]+r[18])-(r[24]+r[18]))` # from inst 8

The effect of this expansion is simplified in the following steps.

5. `r[21]+r[25]+r[18]-(r[24]+r[18])` # remove ()'s and distribute +'s
6. `r[21]+r[25]+r[18]-r[24]-r[18]` # remove ()'s and distribute -'s
7. `r[21]+r[25]-r[24]` # remove negating terms (r[18]-r[18])

Thus, we are left with the induction register `r[21]` plus the difference between the two arrays. Although, for now, we can only see that the arrays will be found in the parameter information. This simplified address expression string is then dumped to the `dnf` file.

When the address calculator attempts to resolve this string to an actual virtual address, it will first search the parameter information to find out that r[24] is `ArrayA` and r[25] is `ArrayB`. It will then use these base names and the initial value of `r[21]`, which is `r[24]+r[18]+4` or `ArrayA+r[18]+4`, and the `ArrayA`'s will cancel out as the following steps show.

1. `r[21]+r[25]-r[24]` # address string of `B[i][j]`
2. `r[21]+ArrayB-ArrayA` # substitute parameter information
3. `ArrayA+r[18]+4+ArrayB-ArrayA` # substitute initial value of `r[21]`
4. `r[18]+4+ArrayB` # result when `ArrayA`'s cancel out

Note that this result gives the initial address of the row in the `ArrayB` array.

Chapter 4

Calculation of Virtual Addresses

Calculating addresses that are relative to the beginning of a global variable or an activation record is accomplished within the compiler since much of the data flow information required for this analysis is immediately available due to its use in compiler optimizations. However, calculating virtual addresses can not be done in the compiler since the analysis of the call graph and data declarations across multiple files is required. Figure 4.1 shows the general organization of the virtual address space of a process executing under SunOS.[1] There is some startup code preceding the instructions associated with the compiled program. Following the program code segment is the static data, which is aligned on a page boundary. The run-time stack starts at high addresses and grows toward low addresses. Part of the memory between the run-time stack and the static data is the heap, which is not depicted in the figure since addresses in the heap could not be calculated statically by the environment described in this dissertation.

Static data consists of global variables, static variables, and nonscalar constants (e.g. strings). In general, the Unix linker (*ld*) places the static data in the same order that the declarations appeared within an assembly file. Also, static data within one file will precede static data in another file specified later

[1] SunOS is a registered trademark of Sun Microsystems, Inc.

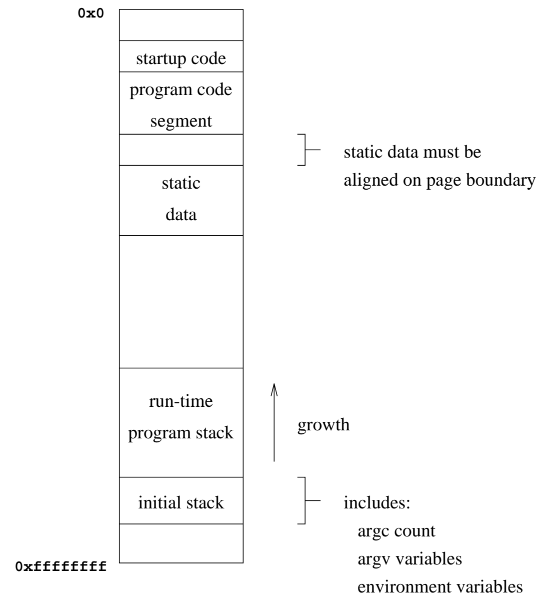


Figure 4.1: Virtual Address Space Organization in SunOS

in the list of files to be linked.[2] In addition, padding between variables sometimes occurs. For instance, variables declared as `int` and `double` on the SPARC are aligned on word and doubleword boundaries, respectively. In addition, the first static or global variable declared in each of the source files comprising the program is aligned on a doubleword boundary. Finally, the beginning of the static data area is aligned on a page boundary.

Run-time stack data includes temporaries and local variables not allocated to registers. Some examples of temporaries include parameters beyond the sixth word passed to a function and memory used to move values between integer and floating-point registers since such movement cannot be accomplished directly on a SPARC. While the code size and static data addresses would not change

[2]There were some exceptions to these rules depending upon how such data is statically initialized.

between executions of a program, the addresses for run-time stack data could. The reason is that the size of the initial stack area that appears before the `main` function is invoked can vary. This initial stack area includes the `argc` count, the strings for `argv` (command line) arguments, and the strings for the logical names associated with a Unix *setenv* command that can be accessed using the `envp` argument. Another complication for calculating run-time stack addresses is that the address of the activation record for a function can vary depending upon the actual sequence of calls associated with its activation. For instance, a local array variable accessed in a function `B` would have a different address if `B` was invoked directly from `main` or if the `main` function invoked function `A` which in turn invoked `B`. The sum of the sizes of the activation records associated with this sequence of calls along with the initial run-time stack address is used to determine the virtual address of the activation record containing the local variable. The address calculator (along with the static simulator and timing analyzer) distinguishes between different function instances and evaluates each instance separately.

Once the static data names and activation records of functions are associated with virtual addresses, the relative address ranges can be converted into virtual address ranges. For instance, consider the relative range shown in Figure 3.4. Say the address calculator determines that array `_a` starts at virtual address 20,000. Then this address would be substituted for the name in the expression as shown in Figure 4.2.

Only virtual addresses have been calculated so far. There is no guarantee that the virtual address will be the same as the actual physical address, which

```
for (r[11] = 0; r[11] < 100; r[11] += 1)
   for (r[2] = r[11]*400+20000; r[2] < r[11]*400+20400; r[2] += 4)
      address{r[2]}
```

Figure 4.2: Algorithmic Range of Virtual Addresses for the Load in Figure 3.3

is typically used to access cache memory on most machines. The assumption for this work is that the system page size is an integer multiple of the data cache size, which is often the case. For instance, the MicroSPARC I has a 4KB page size and a 2KB data cache [25]. Thus, both a virtual and corresponding physical address would have the same relative offset within a page and would map to the same line within the data cache.

Chapter 5

Static Simulation to Produce Data Reference Categorizations

The method of static cache simulation is used to statically categorize the caching behavior of each data reference in a program for a specified cache configuration. A program control-flow graph is constructed that includes the control flow within each function and a function instance graph, which uniquely identifies each function instance by the sequence of call sites required for its invocation. This program control-flow graph is analyzed to determine the possible data lines that can be in the data cache at the entry and exit of each basic block within the program. Static simulation in general and its particular use in the simulation of caches for instruction categorization is explored in detail by Mueller in [19].

The iterative algorithm used for static instruction cache simulation [3, 19] will not be sufficient for static data cache simulation. The problem is that the calculated references can access a range of possible addresses. At the point that the data access occurs, the data lines associated with these addresses may or may not be brought in cache, depending upon how many iterations of the loop has been performed at that point. To deal with this problem, an additional state was created to indicate whether or not a particular data line could *potentially* be in the data cache due to calculated references. When a block with an incoming transition that exits a loop is encountered, then the data lines associated with a calculated reference in that loop that are still in cache at that point are unioned

```
WHILE any change DO
    FOR each basic block instance B DO
        IF B == top THEN
            input_state(B) = calc_input_state(B) = all invalid lines
        ELSE
            input_state(B) = calc_input_state(B) = NULL
        FOR each immed pred P of B DO
            input_state(B) += output_state(P)
            calc_input_state(B) += output_state(P) + calc_output_state(P)
            IF P is in another loop THEN
                input_state(B) += calc_output_state(P) & data_lines(remaining in that loop)
        output_state(B) = input_state(B)
        FOR each data reference D in B DO
            IF D is scalar reference THEN
                output_state(B) += data_line(D)
                output_state(B) -= data_lines(D conflicts with)
                calc_output_state(B) += data_line(D)
                calc_output_state(B) -= data_lines(conflicts with)
            ELSE
                output_state(B) -= data_lines(D could conflict with)
                calc_output_state(B) += data_lines(D could access)
                calc_output_state(B) -= data_lines(D could conflict with)
```

Figure 5.1: Algorithm to Calculate Data Cache States

into the input cache state of that block. The iterative algorithm in Figure 5.1 was used to calculate the input and output cache states for each basic block in the program control-flow graph.

Another problem to resolve is the internal representation of the cache state as a bit vector. With instructions, because of the simple 1-1 mapping from instructions to program lines and program lines to memory (see Figure 1.1), it is easy to represent the cache state directly in a bit vector following this mapping. However static data starts in low memory, growing upward, and stack data starts at very high memory and grows downward. Representing all of the possible memory addresses to which data could map in a single contiguous

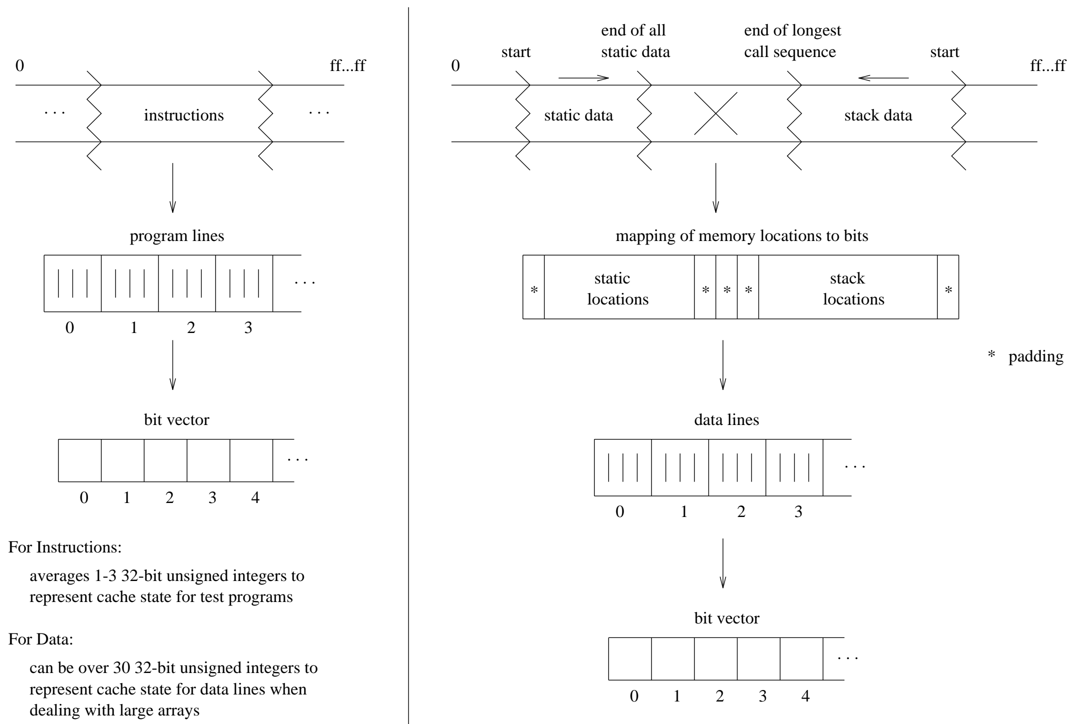


Figure 5.2: Instruction Vs. Data Cache State Representation

bit vector is not possible since the vector would be too large. Therefore, a new mapping method, shown in Figure 5.2, was devised. The upper limit of global and other static data can be computed since the starting address and size of each object is known. Likewise, the lower limit of stack data can be computed using the call graph since we can find the longest calling sequence in the program and sum the sizes of the activation record of each function involved. Knowing these two limits, we can effectively remove the unused memory space from the middle and push the two pieces together with padding as needed to preserve alignment requirements. We can now represent a very large address space in a smaller and more manageable state vector.

Once the cache state vectors have been produced, they are used to determine whether or not each of the memory references within the bounded virtual address range associated with a data reference will be in cache. The static cache simulator needs to produce a categorization of each data reference in the program. The four worst-case categories of caching behavior used in the past for static instruction cache simulation were as follows.

1. **Always Miss (m)**: The reference is not guaranteed to be in cache.

2. **Always Hit (h)**: The reference is guaranteed to always be in cache.

3. **First Miss (fm)**: The reference is not guaranteed to be in cache the first time it is accessed each time the loop is entered, but is guaranteed thereafter.

4. **First Hit (fh)**: The reference is guaranteed to be in cache the first time it is accessed each time the loop is entered, but is not guaranteed thereafter.

These categorizations are still used for scalar data references. However, for nonscalar (calculated) data references, they are not adequate by themselves.

To obtain the most accuracy, a worst-case categorization of a calculated data reference for each iteration of a loop could be determined. For example, some categorizations for a data reference in a loop with 20 iterations might be as follows:

```
m h h h m h h h m h h h m h h h m h h h
```

With such detailed information the timing analyzer could then accurately determine the worst-case path on each iteration of the loop. However, consider

```
/* row order sum */                /* column order sum */
int a[100][100];                   int a[100][100];
main()                             main()
{                                  {
   int i, j, sum;                     int i, j, sum;

   sum = 0;                           sum = 0;
   for (i = 0; i < 100; i++)          for (j = 0; j < 100; j++)
      for (j = 0; j < 100; j++)          for (i = 0; i < 100; i++)
         sum += a[i][j];                    sum += a[i][j];
}                                  }
```

```
row order:  c 25 2500   from [m h h h m h h h m h h h ...  m h h h]
col order:  m           from [m m m m m m m m m m m m ...  m m m m]
```

Figure 5.3: Detecting Spatial Locality

a loop with 100,000 iterations. Such an approach would be very inefficient in space (storing all of the categorizations) and time (analyzing each loop iteration separately). A new categorization was created called **Calculated (c)** that would also indicate the maximum number of data cache misses that could occur at each loop level in which the data reference is nested. The previous data reference categorization string can now be represented as follows:

```
c 5
```

since there are only five total misses and there is only one loop level involved.

The order of access and the cache state vectors were used to detect cache hits within calculated references due to **spatial locality**. Consider the following two code segments in Figure 5.3 that sum the elements of a two dimensional array. The left code code segment is the same as that given in Figure 3.1.

The two code segments are equivalent, except that the left code segment accesses the array in row order and the right code segment uses column order

(i.e., the `for` statements are reversed). Assume that the scalar variables (`i`, `j`, and `sum`) are allocated to registers. Also, assume the size of the direct-mapped data cache is 256 bytes with 16 cache lines containing 16 bytes each. Thus, a single row of the array `a` requiring 400 bytes cannot fit into cache. The static cache simulator is able to detect that the load of the array element in the left code segment had at most one miss for each of the elements that are part of the same data line. This is accomplished by inspecting the order in which the array is accessed and detecting that no conflicting lines are accessed in these loops. The categorizations for the load data reference in the two segments are given in the same figure. Note in this case that the array happens to be aligned on a line boundary. The specification of a single categorization for a calculated reference is accomplished in two steps for each loop level after the cache states are calculated. First, the number of references (iterations) performed in the loop is retrieved. Next, the maximum number of misses that could occur for this reference in the loop is determined. For instance, at most 25 misses will occur in the innermost loop for the left code segment. The static cache simulator determined that all of the loads for the right code segment would result in cache misses. Its data caching behavior can simply be viewed as an always miss. Thus, the range of 10,000 different addresses referenced by the load are collapsed into a single categorization of `c 25 2500` (calculated reference with 25 misses at the innermost level and 2500 misses at the outer level) for the left code segment and an `m` (always miss) for the right code segment.

Likewise, cache hits from calculated references due to **temporal locality** both across and within loops are also be detected. Consider the code segment

```
int i, j, sum, same, a[50], b[50];
...
sum = 0;
for (i = 0; i < 50; i++)
   sum += a[i];                 /* ref 1 */
same = 0;
for (i = 0; i < 50; i++)
   for (j = 0; j < 50; j++)
      if (a[i] ==               /* ref 2 */
          b[j])                 /* ref 3 */
         same++;
```

---

ref 1: c 13 from [m h m h h h m h h h m h h h ... m h h h]
ref 2: h from [h h ... h h] due to temporal locality across loops.
ref 3: c 13 13 from [m h h m h ... m h] on first execution of inner loop, and [h h h h ... h] on all successive executions of it.

Figure 5.4: Detecting Temporal Locality Across and Within Loops

in Figure 5.4. Assume a cache configuration with 32 16-byte lines (512 byte cache) so that both arrays a and b requiring 400 bytes total (200 each) fit into cache. Also assume the scalar variables are allocated to registers. The accesses to the elements of array a after the first loop were categorized as h (always hit) by the static simulator since all of the data lines associated with the array will be in the cache state once the first loop is exited. This shows the detection of temporal locality *across* loops. After the first complete execution of the inner loop, all the elements of b will be in cache, so then all references to it on the remaining executions of the inner loop are also categorized as hits. Thus, the categorization of c 13 13 is given. There are 13 misses relative to the innermost loop due to spatial locality from bringing b into cache during the first complete execution of the inner loop. But there are also only 13 misses relative to the outermost loop since b will now be completely in cache on each iteration after

the first iteration. This shows the detection of temporal locality *within* loops.

The current implementation of the static data cache simulator (and timing analyzer) imposes some restrictions. First, only direct-mapped cache configurations are allowed. Obtaining categorizations for set-associative data cache organizations can be done in a manner similar to that described in [20]. Second, recursive calls are not allowed since it would complicate the generation of unique function instances. Third, indirect calls are not allowed since an explicit call graph must be generated statically.

Chapter 6
Timing Analysis

The timing analysis of data caches is based on earlier work. Arnold in [2] details work predicting the performance of programs using instruction caches. Healy in [8] extends this work to handle the integration of pipeline and instruction cache analysis.

The pipeline path analysis calculates the performance of a sequence of instructions representing paths through loops or functions. Pipeline information about each instruction type is obtained from the machine-dependent data file. Information about the specific instructions in a path is obtained from the control-flow information files. As each instruction is added separately to the pipeline state information, the timing analyzer uses the data caching categorizations to determine whether the MEM (data memory access) stage should be treated as a cache hit or a miss.

The worst-case loop analysis algorithm was modified to appropriately handle calculated data reference categorizations. The timing analyzer will conservatively assume that each of these misses for a calculated reference has to occur before any of its hits. Furthermore, the timing analyzer cannot assume that the penalty for these misses will overlap with other long running instructions since the analyzer may not evaluate these misses in the exact iterations in which they occur. Thus, each calculated reference miss is always viewed as a hit within

```
total_cycles = curr_iter = 0.
pipeline_information = first_misses_encountered = first_hits_encountered = NULL.
WHILE curr_iter != n - 1 DO
    Find the longest continue path.
    first_misses_encountered += first misses that were misses in this path.
    first_hits_encountered += first hits that were hits in this path.
    IF a first miss or first hit was encountered in this path THEN
        curr_iter += 1.
        Subtract 1 from the remaining misses of each calculated reference in this path.
        Concatenate pipeline_information with the union of the information for all paths.
        total_cycles += additional cycles required by union.
    ELSE IF a calculated reference was encountered in this path as a miss THEN
        min_misses = the minimum of the number of remaining misses of each
                     calculated reference in this path that is nonzero.
        min_misses = min(min_misses, n - 1 - curr_iter).
        curr_iter += min_misses.
        Subtract min_misses from the remaining misses of each calc ref in this path
        Concatenate pipeline_information with the union of the information
            for all paths min_misses times.
        total_cycles += (additional cycles required by union) * min_misses.
    ELSE
       break
Concatenate pipeline_information with the union of the pipeline information
    for all paths (n - 1 - curr_iter) times.
total_cycles += (additional cycles required by union) * (n - 1 - curr_iter).
FOR each set of exit paths that have a transition to a unique exit block DO
    Find the longest exit path in the set.
    first_misses_encountered += first misses that were misses in this path.
    first_hits_encountered += first hits that were hits in this path.
    Concatenate pipeline_information with the union of the information
        for all exit paths in the set.
    total_cycles += additional cycles required by exit union.
    Store this information with the exit block for the loop.
```

Figure 6.1: Worst-Case Loop Analysis Algorithm

the pipeline path analysis and the maximum number of cycles associated with a data cache miss penalty is added to the total time of the path. This strategy permits an efficient loop analysis algorithm with some potential overestimations when a data cache miss penalty could be overlapped with other stalls. However, the results in Chapter 8 indicate that any such overestimations were small when they occurred at all.

The worst-case loop analysis algorithm is given in Figure 6.1. The additions

to the previously published algorithm [9] to handle calculated references are shown in boldface. Let n be the maximum number of iterations associated with a loop. The WHILE loop terminates when the number of processed iterations reaches n - 1 or no more first misses, first hits, or calculated references are encountered as misses, hits, and misses, respectively. This WHILE loop will iterate no more than the minimum of (n - 1) or (p + r) times, where p is the number of paths and r is the number of calculated references in the loop.

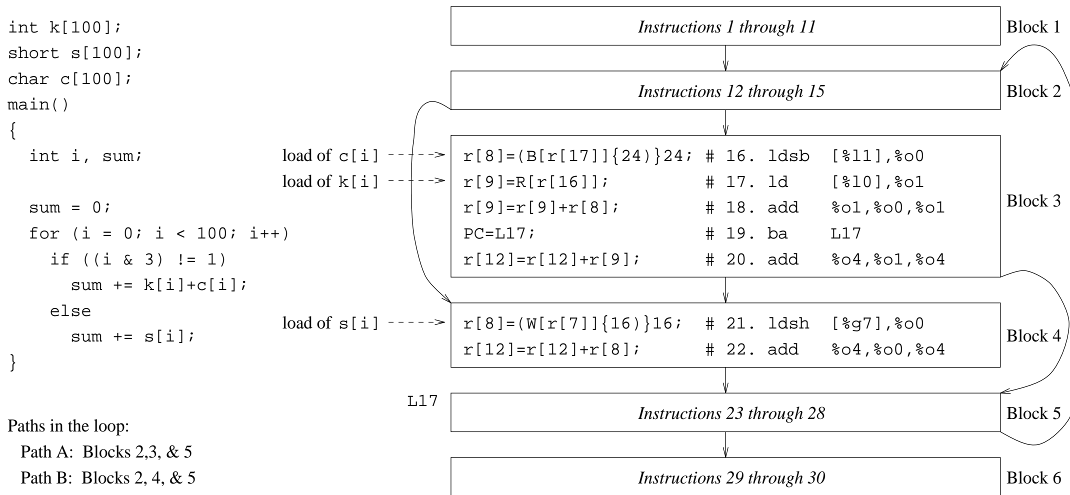
The algorithm attempts to select the longest path for each loop iteration. In order to demonstrate the correctness of the algorithm, one must show that no other path for a given iteration of the loop will produce a longer time than that calculated by the algorithm. Since the pipeline effects of each of the paths are unioned, it only remains to be shown that the caching effects are treated properly. All categorizations are treated identically on repeated references, except for first misses, first hits, and calculated references. Assuming that the data references have been categorized correctly for each loop and the pipeline analysis was correct, it remains to be shown that first misses, first hits, and calculated references are interpreted appropriately for each loop iteration. A correctness argument about the interpretation of first hits and first misses is given in [3].

The WHILE loop will subtract one from each of the calculated reference's miss count in the longest path chosen on each iteration whenever there are first misses or first hits encountered as misses or hits, respectively. Once no such first misses and first hits are encountered in the longest path, the same path will remain the longest path as long as its set of calculated references that were

encountered as misses continue to be encountered as misses since the caching behavior of all of the references will be treated the same. Thus, the pipeline effects of this longest path are efficiently replicated for the number of iterations associated with the minimum number of remaining misses of the calculated references that are nonzero within the longest path. After the WHILE loop, all of the first misses, first hits, and calculated references in the longest path will be encountered as hits, misses, and hits, respectively. The unioned pipeline effects after the WHILE loop will not change since the caching behavior of the references will be treated the same. Thus, the pipeline effects of this path are efficiently replicated for all but one of the remaining iterations. The last iteration of the loop is treated separately since the longest exit path may be shorter than the longest continue path.

A short example is given in Figure 6.2 to illustrate the algorithm. In this example delay slots have been filled. The `if` statement condition was contrived to force the worst-case paths to be taken when executed. Assume a data cache line size of 8 bytes and enough lines to hold all three arrays in cache. The figure also shows the iterations when each element of each of the three arrays will be referenced and whether or not each of these references will be a hit or a miss.

Two different paths can be taken through the loop on each iteration as shown in Figure 6.3. Note that the pipeline diagrams reflect that the loads of the array elements were found in cache. The miss penalty from calculated reference misses is simply added to the total cycles of the path and is not directly reflected in the pipeline information since these misses may not occur in the same exact iterations as assumed by the timing analyzer.



data lines: data line 0, data line 1, data line 2, data line 3, ... data line 50, data line 51, ... data line 76, data line 77, ...

array elements: k: 0 1 2 3 4 5 6 7 ... s: 0 1 2 3 4 5 6 7 ... c: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 ...

iteration accessed: 1 3 4 5 7 8 2 6 1 3 4 5 7 8 9 11 12 13 15 16

result: miss miss hit miss miss hit miss miss m h h h h h m h h h h h

k[i]: c 50 from [m h m h ... m h]

s[i]: c 25 from [m h h h m h h h ... m h h h]

c[i]: c 13 from [m h h h h h h h m h h h h h h h ... m h h h]

Figure 6.2: Example to Illustrate Worst-Case Loop Analysis Algorithm

Pipeline Diagram for Path A: Instructions 12-20 and 23-28 (blocks 2,3,5)

		cycle																			
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
stage	IF	12	13	14	15	16	17	18	19	19	20	23	24	25	26	27	28				
	ID		12	13	14	15	16	17	18	18	19	20	23	24	25	26	27	28			
	EX			12	13		15	16	17		18		20	23	24	25	26		28		
	FEX																				
	MEM				12	13		15	16	17		18		20	23	24	25	26		28	
	WB					12	13		15	16	17		18		20	23	24	25	26		28
	FWB																				

Pipeline Diagram for Path B: Instructions 12-15 and 21-28 (blocks 2,4,5)

		cycle																
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
stage	IF	12	13	14	15	21	22	23	23	24	25	26	27	28				
	ID		12	13	14	15	21	22	22	23	24	25	26	27	28			
	EX			12	13		15	21		22	23	24	25	26		28		
	FEX																	
	MEM				12	13		15	21		22	23	24	25	26		28	
	WB					12	13		15	21		22	23	24	25	26		28
	FWB																	

Figure 6.3: Pipeline Diagrams for the Two Paths in Figure 6.2

Table 6.1: Timing Analysis Steps for the loop in Figure 6.2

step	start iter	longest path cycles	min_misses	iters handled	additional cycles	total cycles
1	1	20+18=38	min(13,50)=13	13	20+((20-4)*12)+(18*13)=446	446
2	14	20+9=29	min(37)=37	37	((20-4)*37)+(9*37)=925	1371
3	51	17+9=26	min(25)=25	25	((17-4)*25)+(9*25)=550	1921
4	76	20+0=20	N/A	24	(20-4)*24=384	2305
5	100	20+0=20	N/A	1	20-4=16	2321

Table 6.1 shows the steps the timing analyzer uses from the algorithm given in Figure 6.1 to estimate the WCET for the loop in the example shown in Figures 6.2 and 6.3. The longest path detected in the first step is Path A, which contains references to `k[i]` and `c[i]`. The pipeline time required 20 cycles and the misses for the two calculated references (`k[i]` and `c[i]`) required 18 cycles. Note that each miss penalty was assumed to require 9 cycles. Since there were no first misses, the timing analyzer determines that the minimum number of remaining misses from the two calculated references is 13. Thus, the path is replicated an additional 12 times. The overlap between iterations is determined to be 4 cycles. Note that 4 is not subtracted from the first iteration since any overlap for it would be calculated when determining the worst-case execution time of the path through the `main` function. The total time for the first 13 iterations will be 446. The longest path detected in step 2 is also Path A. But this time all references to `c[i]` are hits. There are 37 remaining misses to `k[i]`. The total time for iterations 14 through 50 is 925 cycles. The longest path detected in step 3 is Path B, which has 25 remaining misses to `s[i]`. This results in 550 additional cycles for iterations 51 through 75. After step 3 the worst-case loop analysis has exited the WHILE loop in the algorithm shown in Figure 6.1.

Step 4 calculates 384 cycles for the next 24 iterations (76-99). Step 5 calculates the last iteration to require 16 cycles. The timing analyzer calculates the last iteration separately since the longest exit path may be shorter than other paths in a given loop. The total number of cycles calculated by the timing analyzer for this example was identical to the number observed via execution simulation.

A timing analysis tree is constructed to predict the worst-case performance. Each node of the tree represents either a loop or a function in the function instance graph. The nodes representing the outer level of function instances are treated as loops that will iterate only once. The worst-case time for a node is not calculated until the time for all of its immediate child nodes are known. For instance, consider the example shown in Figures 6.2 and 6.3 and Table 6.1. The timing analyzer would calculate the worst-case time for the loop and use this information to next calculate the time for the path in `main` that contains the loop (block 1, loop, block 6). The construction and processing of the timing analysis tree occurs in a similar manner as described in [3, 9].

Chapter 7

Execution Simulation of Data Cache and Pipeline Effects

The timing analyzer produces as its result the number of cycles a program takes to execute. In order to verify that this number is correct, a method was developed to simulate a program's worst-case execution time. The Ease execution simulator [6] works by instrumenting the assembly code generated by the compiler in order to invoke measurement routines during its execution. One of these routines was heavily modified in earlier work in order to calculate the number of cycles required to execute a program by simulating the program's pipeline and instruction caching behavior during its execution [8]. This routine was further modified for this dissertation in order to simulate data caching behavior as well. Also, other portions of the Ease simulator were modified so as to check the accuracy of the statically computed addresses.

In the original Ease implementation, a call to a special instruction cache trace function was inserted into the assembly code at the beginning of each basic block. This function was modified as reported in [8] to go through each instruction in the basic block, using instruction information produced by the compiler and dumped into the assembly file, and to simulate the pipeline and instruction cache effects that would result from that instruction's execution. This simulation of each instruction in the basic block happens before the actual instructions in each block are executed. As each new basic block is reached either

by falling through or branching, a new sequence of instructions is simulated and its effect on the instruction cache and pipeline states is recorded. After the program terminates, the final results of this simulation, most importantly the total number of cycles used by the program, are dumped to a file.

For data cache simulation, however, this approach is not sufficient. This is due to the fact that memory access to data, unlike that for instructions, does not occur sequentially within a basic block. Also, unlike instructions, data access addresses can change each time they are referenced. Yet another consideration that required this approach to be modified is the desire to have the simulator automatically do a verification of the accuracy of the virtual addresses computed by the address calculator.

In addition to inserting a special trace routine at the beginning of each basic block for data cache simulation, the original Ease implementation inserted a call to a special simulation function after each load or store in the assembly code. This function, called `ease_read` for loads and `ease_write` for stores, would take the data address and its size as parameters and update the running data cache state as necessary, based on whether load or store was determined to be a hit or a miss. This would continue until the instrumented program completed execution.

In order to update Ease to merge data cache simulation with that of pipeline and instruction caching as well as check the validity of the computed virtual addresses, the following changes were made.

- The special trace routine at the beginning of each basic block was modified to call the previously modified block instruction simulation routine

immediately if there were no data references in this basic block. Thus, the pipeline effects of executing this block can be simulated. In order for the start-up routine to know if there are data references in the block or not, the compiler was modified to identify the number of data references associated with each basic block and to write this number to a special file of data declarations that is linked in the instrumented assembly code.

- If there are data references in the current basic block, then the entire block is executed and the data cache results—whether the access is a hit or a miss—is recorded as the result of the execution of either `ease_read` or `ease_write` after each load or store in the block.

- The `ease_read` and `ease_write` routines were modified to recognize if they were called after the last load or store in the basic block, and if so, to call the block instruction simulation routine. They were also modified to take an additional parameter—the overall number, starting from 0, of the data reference they were simulating. This number is used to index into a global array of addresses, initialized from the `adr` file produced by the address calculator, when verifying the validity of the address for the current load or store.

In the previously modified block instruction routine, which simulated the pipeline and instruction cache behavior for each instruction, a load or store of a word required only a one cycle delay in the pipeline MEM stage since all data cache accesses were assumed to be hits. However, this routine was updated in order to correctly simulate data caching by requiring extra cycles in MEM

stage when the cache access is a miss since the needed data cache line had to be fetched from memory. A delay is also needed in all cases when the data reference is a store, since a write-through cache is assumed and, thus, there will be a memory access for every store. Also, for the present, the routine was modified to assume that all instruction cache accesses are hits.

When verifying the accuracy of the virtual addresses, the simulator actually checks each data address it receives from `ease_read` or `ease_write` against the *range* of possible virtual addresses given to it via the `adr` file written by the address calculator. If any discrepancies are found, an error is reported and the simulation terminates. Note that the virtual address range of a global scalar variable is a single virtual address. The virtual address range of a local scalar variable is the list of addresses that may result from the different locations of its activation record on the stack due to the function in which it is declared being called from different places in the code. The virtual address range of a calculated reference includes all addresses between the lowest and highest addresses accessed, inclusive. None of the executions of the test programs discussed in Chapter 8 reported any incorrect data addresses.

One of the most difficult problems encountered with modifying the Ease simulator was changing the compiler to produce correctly instrumented code. Many situations that were not a problem for instruction caching arose when trying to instrument the code for data caching. These problems mainly occurred because whenever there were data references in a basic block, function calls had to be inserted in the middle of the block after each load or store. Thus, the state of the machine had to be saved before each call and restored after it.

Instrumentation for instruction caching only had calls inserted at the beginning of each block. Less state information is typically live at basic block boundaries. Therefore, less information had to be saved and restored for instruction caching, and the inserted measurement instructions caused fewer problems.

Chapter 8

Results

Measurements were obtained on code generated for the SPARC architecture by the *vpo* optimizing compiler [4]. The machine-dependent information contains the pipeline characteristics of the MicroSPARC I processor. The programs described in Table 8.1 are used to evaluate the effectiveness of the environment for bounding worst-case data cache performance. Note that these programs were restricted to specific classes of data references, which did not include any dynamic allocation from the heap. It is doubtful that heap references can ever be analyzed to obtain known addresses. In addition, pointer variables containing the addresses of other variables were limited to those used as formal parameters. A more general pointer analysis may be used in the future to determine a set of possible address ranges associated with a dereferenced pointer variable. Both heap references and non-parameter pointer address references are currently assumed to be able to access any line within the data cache. A direct-mapped data cache containing 16 lines of 32 bytes for a total of 512 bytes was used. The MicroSPARC I uses write-through/no-allocate data caching [25]. The static simulator was able to categorize store data references. However, these categorizations were ignored by the timing analyzer since stores always accessed memory and a hit or miss associated with a store data reference had the same effect on performance. While pipeline and data caching behavior was evaluated,

Table 8.1: Test Programs

Name	Num Bytes	Description or Emphasis
Matcnta	40060	Counts and Sums Nonnegative Values in a 100x100 Integer Matrix
Matcntb	460	Counts and Sums Nonnegative Values in a 10x10 Integer Matrix
Matmula	30044	Multiplies 2 50x50 Matrices into a third 50x50 Integer Matrix
Matmulb	344	Multiplies 2 5x5 Matrices into a third 5x5 Integer Matrix
Matsuma	40044	Sums Nonnegative Values in a 100x100 Integer Matrix
Matsumb	444	Sums Nonnegative Values in a 10x10 Integer Matrix
Sorta	2044	Bubblesort of 500 Integer Array into Ascending Order
Sortb	444	Bubblesort of 100 Integer Array into Ascending Order
Statsa	16200	Calcs Sum, Mean, Var, Std Dev, Cor Coef of 2 arrays of 1000 doubles
Statsb	600	Calcs Sum, Mean, Var, Std Dev, Cor Coef of 2 arrays of 25 doubles
Des	1346	Data Encryption Standard—Encrypts and Decrypts 64 bits

instruction fetches were assumed to be all hits.

Table 8.1 describes the test programs used to assess the effectiveness of bounding worst-case data cache performance. Two versions were used for each of the first five test programs. The **a** version had the same size arrays that were used in previous studies [3, 9]. The **b** version of each program used smaller arrays that would totally fit into a 512 byte cache. The number of bytes reported in the table is the total number of bytes of variables in the program. Note that some of these bytes will be in the static data area while others will be in the run-time stack. The sixth test program is `des`, a data encryption program. The amount of data in this program is not changed since the encryption algorithm is based on using the large static arrays as they are. This program is included here because of its other interesting properties. It includes arrays of structures, arrays indexed by other arrays, and many address parameters to functions.

Table 8.2 depicts the dynamic results from executing the test programs. The

Table 8.2: Dynamic Results

Name	Hit Ratio	Observed Cycles	Estimated Cycles	Estim. Ratio	Naive Ratio
Matcnta	71.86%	1,143,014	1,143,023	**1.000**	1.148
Matcntb	70.73%	12,189	12,189	**1.000**	1.148
Matmula	62.81%	7,245,830	7,952,807	**1.098**	1.240
Matmulb	89.40%	11,396	11,396	**1.000**	1.332
Matsuma	71.86%	1,122,944	1,122,953	**1.000**	1.151
Matsumb	69.98%	11,919	11,919	**1.000**	1.152
Sorta	97.06%	4,768,228	9,826,909	**2.061**	2.883
Sortb	99.40%	188,696	371,977	**1.971**	2.915
Statsa	90.23%	1,237,698	1,447,572	**1.170**	1.290
Statsb	89.21%	32,547	37,246	**1.144**	1.290
Des	75.71%	155,340	191,564	**1.233**	1.448

hit ratios were obtained from the data cache execution simulation. Only *Sort* had very high data cache hit ratios due to many repeated references to the same array elements. The *observed cycles* were obtained using the ease cache simulator discussed in Chapter 7. The *estimated cycles* were obtained from the timing analyzer discussed in Chapter 6. The *estimated ratio* is the quotient of these two values. The *naive ratio* was calculated by assuming that all data cache references were misses and dividing those cycles by the observed cycles. It is used to show the advantage of doing the data cache analysis versus assuming all data cache references are misses.

The timing analyzer was able to tightly predict the worst-case number of cycles required for pipelining and data caching for most of the test programs. In fact, for five of them, the prediction was exact or over by less than $\frac{1}{10}$ of a percent. Also, each of the small examples shown earlier in this dissertation resulted in exact predictions when the worst-case paths were executed. For those that were overestimated, there were clear reasons as to why it happened.

Matmula had an overestimation of about 10% whereas the smaller data version *Matmulb* was exact. As shown in the following code segment, the *Matmul* program has repeated references to the same elements of three different arrays: A, B, and Res.

```
    ⋮
    for (Outer = 0; Outer < UPPERLIMIT; Outer++)
       for (Inner = 0; Inner < UPPERLIMIT; Inner++)
       {
         Res [Outer][Inner] = 0;
         for (Index = 0; Index < UPPERLIMIT; Index++)
             Res[Outer][Inner]  +=
                A[Outer][Index] * B[Index][Inner];
       }
    ⋮
```

These references would miss the first time they were encountered, but would be in cache for the smaller *Matmulb* when they were accessed again since the arrays fit entirely in cache. Also, since they fit into cache, there is no interference between them. A reference to B could not knock any lines of A or Res out of cache. However, when they do not fit into cache they can interfere with each other and possibly with other elements of themselves, if individually they are sufficiently larger than cache. In such a case, the static simulator conservatively assumes that any *possible* interference must result in a cache miss. Therefore, the categorizations are more conservative and the overestimation is larger.

The inner loop in the function within *Sort* that actually sorts the values has a varying number of iterations that depends upon a counter of an outer loop. The number of iterations performed was overrepresented on average by about two for this inner loop.

The *Stats* program had about 17% and 14% overestimations for the large and small versions of the program, respectively. The strategy of treating a calculated

reference miss as a hit in the pipeline and adding the maximum number of cycles associated with the miss penalty to the total time of the path caused overestimations with these two programs. The *Statsa* and *Statsb* programs were the only floating-point intensive programs in the test set. Often delays due to long-running floating-point operations could have been overlapped with data cache miss penalty cycles.

The *Des* program had the worst overestimation of the test set at 23%. This may seem large, but is in fact quite satisfactory when considering the nature of the data accesses within the program. There are many places in the code where an element of a locally defined static character array is used as an index into a global array of integers as the following code segment shows.

```
   ⋮
unsigned long bit[33];
   ⋮
   static char iet[49]={0,32,1,2,3,4,5,4,5,6,7,8,9, ... ,32,1};
   ⋮
   for (j=16,l=32,m=48;j>=1;j--,l--,m--) {
      ie.r = (ie.r <<=1) | (bit[iet[j]] & ir ? 1 : 0);
      ie.c = (ie.c <<=1) | (bit[iet[l]] & ir ? 1 : 0);
      ie.l = (ie.l <<=1) | (bit[iet[m]] & ir ? 1 : 0);
   }
   ⋮
```

Since the character array `iet` is statically initialized, there is no simple method to determine which value from it will be used to index into the integer array `bit`, and, thus, no way to establish a meaningful pattern of access. In a case like this, we assume that *any* element of the `bit` array may be accessed any time the data reference occurs in the program. This forces all conflicting data lines to be knocked out of the cache state during the iterative flow analysis phase of the static simulation. Thus, the resulting categorizations are quite conservative.

The *Des* program also has an overestimation due to data dependences in the program. A longer path deemed feasible by the timing analyzer could not be taken in a function due to a variable's value in an `if` statement. In [9], a 13% overestimation is reported for this program when doing pipeline and instruction caching evaluation only.

Despite the overestimations detailed above, the results given in this dissertation show that with certain restrictions it is possible to tightly predict much of the data caching behavior of many programs.

The difference between the *naive* and *estimated* ratios shows the benefits for performing data cache analysis when predicting worst-case execution times. The benefit of worst-case performance from data caching is not as significant as the benefit obtained from instruction caching [3, 9]. An instruction fetch occurs for each instruction executed. The performance benefit from a write-through/no-allocate data cache only occurs when the data reference from a load instruction is determined to be in cache. Load instructions only comprised on average 14.28% of the total executed instructions for these test programs. However, the results do show that performing data cache analysis for predicting worst-case execution time does still result in substantially tighter predictions. In fact, for the programs in the test set the prediction improvement averages over 30%.

The overhead associated with predicting WCETs for data caching using this method comes primarily from that of the static cache simulation. The time required for the static simulation increases linearly with the size of the data. However, even with large arrays as in the given test programs this time is rather small. The average time for the static simulation to produce data reference

categorizations for the 11 programs given in Tables 8.1 and 8.2 is only 2.89 seconds. The average time for the timing analyzer to produce the worst-case execution times for the test programs is 1.05 seconds.

Chapter 9

Future Work

There are several areas of further investigation that can be carried on for bounding worst-case data cache performance.

9.1 Merging Instruction and Data Caching Prediction and Simulation

An eventual goal of this research is to merge all components of the data caching worst-case analysis with those of the instruction caching worst-case analysis [9]. Timing predictions could then be obtained for the complete machine. Actual measurements from the machine using a logic analyzer could be used to gauge the effectiveness of the entire timing analysis environment.

9.2 Wrap-Around Fill for Data Caches

Other future work includes analyzing the effect of wrap-around-fill data caching. This analysis would result in tighter estimations since the MicroSPARC I does use wrap-around fill for both instruction and data caching [25]. Wrap-around-fill analysis for data caching can probably be accomplished in a manner similar to that used for wrap-around-fill instruction caching [10]. It is currently assumed that each store requires a constant penalty for accessing memory with a write-through data cache.

9.3 Write Buffer

Another area to be explored is evaluating the effect of the use of a write buffer. The MicroSPARC I actually has a write buffer that can hold the address and data for a single outstanding store. Also, at most a single load or store can access memory at one time. Both wrap-around fill and the write buffer can probably be handled by representing the access to memory and the write buffer as a pipeline stage and multiple simultaneous accesses would be prevented due to the detection of structural hazards.

9.4 Best Case

The timing analyzer may be modified in order to predict the best-case execution time of programs for data caching. All preceding elements of the data cache analysis project—the compiler, the address calculator, and the static simulator—already have some code in place to deal with best case. There would also have to be some changes made to the Ease execution simulator.

9.5 Compiler Optimizations

The research presented in this dissertation could also be applied to developing new or improving existing compiler optimizations for programs on machines with data caches. Current data cache compiler optimizations are only applied on tightly nested loops without function calls. This research would allow optimizations to be performed more accurately and in an interprocedural fashion.

Chapter 10

Conclusions

This dissertation has presented an approach for bounding the worst-case performance for programs using a data cache. This approach involves several steps. Data flow analysis is used within a compiler to determine a bounded range of relative addresses for each data reference. An address calculator converts these relative ranges to virtual address ranges by examining the order of data declarations and the call graph of the program. Categorizations of the data references are produced by a static simulator. A timing analyzer uses the categorizations when performing pipeline path analysis to predict the worst-case performance for each loop and function in the program. A method of quantitatively verifying the results using a data cache and pipeline simulator has also been presented.

The following accomplishments have been demonstrated:

- The creation of a tool-based system that *automatically* produces WCET results with no interaction from the user,
- A *static* analysis technique that requires no complete simulation or execution of the program to be timed and, thus, no need to find the appropriate input data to drive the worst-case paths,
- The ability to analyze the complete control flow of a program, including functions and all their instances, loops at all nesting levels, and conditional control flow,

- The ability to handle address information that is sent into functions via parameters,
- Detection and exploitation for tighter timing bounds of spatial locality within calculated data references and temporal locality both within and across loops, and
- A performance overhead when predicting WCETs measured in seconds.

The results of using the system on various representative programs indicate that the approach is valid and can result in significantly tighter worst-case performance predictions. For the given test programs, an improvement averaging over 30% is shown for the WCET predictions.

Appendix A
Memory References in Annulled Delay Slots

One of the main advantages of the approach to bounding WCETs for processors with data caches that is discussed in this dissertation is the ability to work with completely optimized code. However, doing so makes the analysis needed to statically compute addresses much more complicated. Certain optimizations can also affect the manner in which both the static simulation and timing analysis is performed. Perhaps the most difficult of the optimizations dealt with is that of filling delay slots, particularly those of annulled branches.

On the SPARC processor, every instruction that can transfer control from one place in the program to another (branches and calls) is followed by a delay instruction. This is an instruction that will be executed before control is transferred to the target of the branch or call, and is said to be in the *delay slot*. Without optimization, these slots can be filled with `nop` instructions to make compilation faster. However, this wastes an instruction cycle where something useful could actually be accomplished. So, a compiler optimization called *filling delay slots* is usually performed to move another useful instruction from somewhere else in the program into this slot.

While the filling of delay slots increases execution efficiency, it can radically alter the structure of a program. This makes the expansion of data address information much more difficult since instructions that set registers that contain

part of the address may be moved away from the area in which the load or store that uses that address is located. Furthermore, a load or store instruction itself may be used to fill a delay slot. This often results in a duplicate load or store instruction being created when loops are involved.

Matters may be complicated further if the branch instruction before the delay slot is an *annulled branch*. If an annulled branch is not taken, then the instruction in the delay slot will be *annulled*. This means that although it will occupy all stages in the pipeline, the results of the instruction will not be committed. If this instruction is a load or a store, it will be flushed out of the pipeline before a read from or write to memory is performed, respectively. For example, consider the following assembly code segment.

```
              ⋮
        add     %o2,%o0,%o2          # 9
        cmp     %o2,%g1              # 10
        ble,a   L15                  # 11
        ld      [%o2],%o0            # 12
        sethi   %hi(_a),%o3          # 13
              ⋮
L15:    add     %o1,%o5,%o1          # 27
              ⋮
```

If the branch (instruction 11) is taken, the instructions 9, 10, 11, 12, and 27 will be executed completely, including the `ld` instruction in the delay slot (instruction 12). However, if the branch is not taken, only instructions 9, 10, 11, and 13 will be executed fully. As mentioned above, the `ld` will begin in the pipeline but will not complete execution as a load.

To see the difficulties involved with predicting WCET for programs containing a data reference in an annulled delay slot, consider the source code and

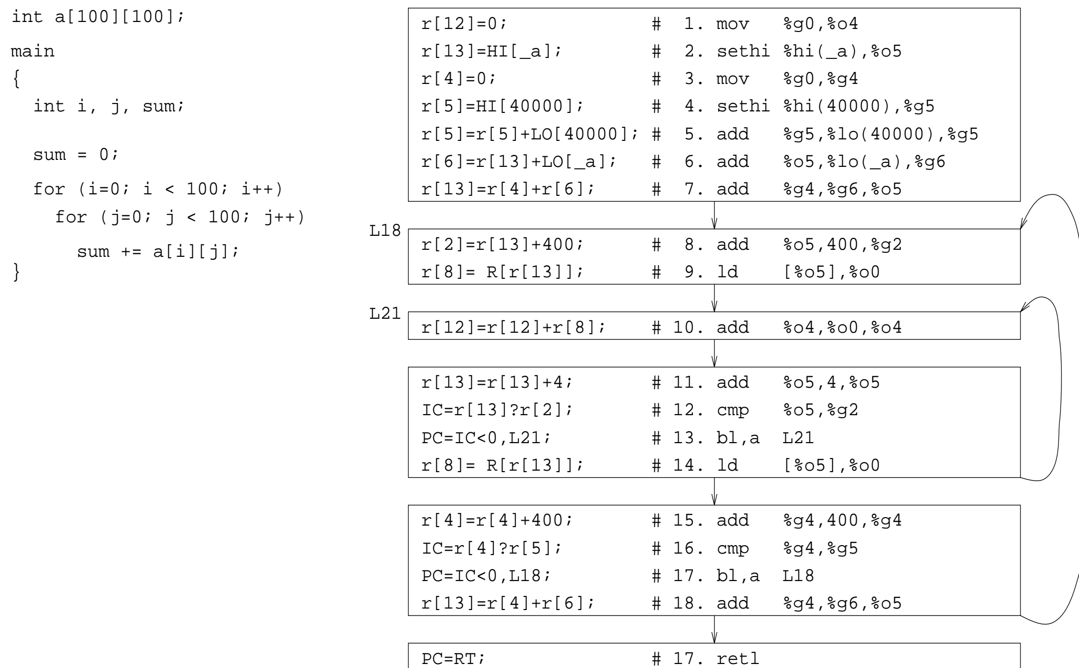


Figure A.1: Example C Program, RTLS, and SPARC Assembly, Revised

corresponding RTLs and SPARC assembly instructions, in Figure A.1. This is the same example as that in Figure 3.1 except that the filling of delays slots has been turned back on. Note that although there is only one memory reference in the source code, the reference to `a[i][j]`, it is actually turned into two loads (instructions 9 and 14) in the machine code due to the filling of delay slots. In this case, a load instruction that was originally at the beginning of the innermost loop was used to fill the delay slot at the end of the loop, so a copy of it was placed right before the innermost loop. Thus, the effective behavior of the data reference is the same – all 10,000 elements of the array will be summed. This is true because the load at instruction 9 will access the first element of every row of array `a` and the load at instruction 14, the one in the delay slot, will access

the remaining elements of every row of `a`. It may appear that the second load will actually go beyond the last element of each row to the first element of the next row. However, remember that because of the annulled branch, the load at instruction 14 will *not* be fully executed as a load when the branch is not taken. Therefore, register `r[8]` will not be set when control falls out of the inner loop (i.e., the branch at instruction 13 is not taken).

For this example, the filling of the annulled delay slot with a load did not cause problems with expansion of addresses. The load at instruction 9 was expanded and recognized as a simple calculated reference with one induction register since it is in the outer loop. The load at instruction 14 was also expanded and recognized as a simple CR with two induction registers. Enough information is available for both to compute the range of virtual addresses that will be accessed by each.

However, there is a problem to be solved in doing the static simulation to produce the appropriate categorizations for these calculated references. Remember that the iterative analysis phase of the static simulator that produces the cache states does not know and does not care about loop iteration information. It has no way of knowing that the last access to the load at instruction 14 for each complete execution of the inner loop will not actually update the cache since it is in an annulled delay slot. A method was devised so that lines brought in by this array access on the last iteration of the inner loop would not update the cache. The compiler recognizes the case when this happens and puts a flag in the DNF file for the load in the delay slot as well as the corresponding load that was moved outside the loop. When the static simulator sees these flags when

building the control flow information, it effectively changes these two loads to a single load that happens at the beginning of the innermost loop – exactly the way the code appeared before delay slots were filled. This way, when the iterative flow algorithm executes, it produces the correct cache state information. After the cache states are produced, the loads are put back in their original form.

In order to produce the appropriate calculated categorizations for these loads, the categorization phase of the static simulator also has to recognize them as special cases. When the first load is identified (usually the one that has been moved out), the corresponding one is located and a single categorization string for both is produced. This works by using the stored induction variable register information to step through both of the references at the same time. During each iteration of the algorithm, the element of the reference that would come next is considered when determining if the corresponding place in the categorization string should be a hit or a miss.

For example, if the references have a positive stride, the lowest of the two is chosen. Next, its individual categorization is determined (hit or miss). Then, it is incremented according to its stride. This process repeats until the whole categorization string is produced.

This method works correctly since, even though there are two different loads, they are both needed to completely iterate through the array. As mentioned above, the one right before the innermost loop iterates through the first element of each row, and the one in the delay slot iterations through all of the other elements in each row.

After the categorization string is produced it can be used to produce the correct calculated (`c`) categorization for each load for every loop level involved.

Modifications were also necessary to allow both the timing analyzer and the execution simulator to deal correctly with loads in an annulled delay slot. Changes to both of these took the form of adding extra code that prevents the load from completing execution as a load in such a case. This prevents extra time from being counted when the load is a miss but the branch is not taken.

Appendix B

Description of the DNF File

<DNF_File>	::=	*<ItemList>* <EOF>
<ItemList>	::=	*<DNF_Item>* <EOLN>
	::=	*<ItemList>* *<DNF_Item>* <EOLN>
<DNF_Item>	::=	*<LocalRef>* | *<GlobalRef>* | *<CalcRef>* | *<FuncName>* | *<StackSpace>* | *<FuncCall>* | *<Parameters>* | *<BlockNum>* | *<LeftSucc>* | *<RightSucc>* | *<PredList>* | *<NumInstructs>* | *<G_Rec>* | *<LocalInf>* | *<LoopIters>*
<LocalRef>	::=	*<LocalType>* <rw> <name> <finum> <offset> <type> <drefnum> <annulled>
<LocalType>	::=	**LV** | **LA**
<GlobalRef>	::=	*<GlobalType>* <rw> <name> <finum> <type> <drefnum> <annulled>
<GlobalType>	::=	**GV** | **GR**
<CalcRef>	::=	**CR** <rw> <finum> <type> <drefnum> <annulled> <EOLN> *<CalcInfoLine>*
<CalcInfoLine>	::=	**BA** <address> <stride> <min_iters> <max_iters> <loop> <caseflag> <EOLN> *<IndVarLine>*
	::=	**CM** <address> <EOLN> *<IndVarLine>*
	::=	**OE** <address>
	::=	**AW** <gl> <name> <basereg>
<IndVarLine>	::=	**IV** *<IndVarList>* **0**
	::=	**IV** *<IndVarList>* *<ReplaceList>* **0**
	::=	**IV** *<ReplaceList>* **0**
<IndVarList>	::=	**1** <indreg> <loopnum> <stride> <min_iters> <max_iters> <initval>
	::=	*<IndVarList>* **1** <indreg> <loopnum> <stride> <min_iters>

		<max_iters> <initval>
<ReplaceList>	::=	**-1** <indreg> <initval>
	::=	*<ReplaceList>* **-1** <indreg> <initval>
<FuncName>	::=	**FN** <name>
<StackSpace>	::=	**SP** <amount>
<FuncCall>	::=	**FC** <name>
<Parameters>	::=	**PM** <number> <ParamList>
<ParamList>	::=	<address>
	::=	*<ParamList>* <address>
<BlockNum>	::=	**--** <number> **--**
<LeftSucc>	::=	**SL** <number>
<RightSucc>	::=	**SR** <number>
<PredList>	::=	**P** *<Preds>* **-1**
<Preds>	::=	<number>
	::=	*<Preds>* <number>
<NumInstructs>	::=	**NI** <number>
<G_Rec>	::=	**G.**<func_num> <name> [**0** | **-1**] <alignment> <init_flag> <EOLN> *<SizeRec>*
	::=	**G.**<func_num> <name> <size> <alignment> <init_flag>
<SizeRec>	::=	**G.0 ^size^** <size>
<LocalInf>	::=	**LC** <name> <offset> <size>
<LoopIters>	::=	**LI** <loop_iters>

Where,

<EOF>	–	end-of-file marker
<EOLN>	–	end-of-line marker
<rw>	–	tells whether this access is a read (`r`) or a write (`w`)
<name>	–	name of local or global variable
<finum>	–	instruction number where found, relative to current function, starting from 0

<offset>	–	offset into current function activation record
<type>	–	type of memory reference: `B`, `W`, `R`, `F`, or `D`
<drefnum>	–	overall data reference number, relative to beginning of each file, starting from 0
<annulled>	–	1 if this memory reference is in an annulled delay slot; 0 otherwise
<address>	–	expression that gives the base address of the reference
<stride>	–	stride of the loop (length of each subitem)
<min_iters>	–	minimum number of loop iterations necessary to go through array
<max_iters>	–	maximum number of iterations needed to go through array
<loop>	–	block number of the header block of the loop containing the rtl with this memory reference
<caseflag>	–	special cases of data references in annulled slots are recognized by the compiler to facilitate dealing with them in the static simulator; this is the number of the special case, or 0 if the data ref is not in a delay slot or was not recognized.
<gl>	–	tells whether the following name is a global (`g`) or a local (`l`)
<basereg>	–	`s` if this local name is relative to the stack pointer and `f` if it is relative to the frame pointer
<indreg>	–	register used as induction variable
<loopnum>	–	number of the loop containing this induction register; the same number used in the *path* file
<initval>	–	expression that gives the initial value of this induction variable
<amount>	–	amount of space reserved on the stack for the activation record for this function
<number>	–	an integer number
<func_num>	–	number of function containing this global declaration
<alignment>	–	a number giving the alignment requirements of this global variable, i.e., 8 if the variable must be aligned on an 8-byte boundary
<size>	–	number of bytes required to store this global/local
<init_flag>	–	1 if variable is initialized, 0 otherwise
<loop_iters>	–	maximum number of iterations of loop in which the current block is contained; will only appear in blocks that are loop headers

And the tokens are,

LV Local Variable. It is used to indicate that this particular local reference is relative to the stack pointer, `r[14]`.

LA Local Argument. It indicates that this local is relative to the frame pointer, `r[30]`.

GV Global Variable. Indicates that this is a global reference consisting of just a global variable name that must be looked up in the global records list.

GR Global Reference. This is a global reference of the form <name> + <offset>, where <offset> is an integer number.

CR Calculated Reference. This will be a reference for which there will not be a simple single address. There will be one or two lines following this line giving more information about the range of addresses that may be accessed.

BA Stands for Base Address. This is basically the same thing as the CM line below but with additional information. This line still gives a base address in the form of an address string, but it also gives a stride and minimum and maximum number of iterations. The range of addresses that this calculated reference may access as well as the order in which they will be accessed can be computed using this information.

CM Stands for Code Motion. An expanded address string for a given data reference is stored during code motion in a special information list. If later in the compilation process, the compiler is not able to fully expand a data reference (see above) this list will be searched. If an earlier (and better) address expansion for this data reference is found on the list, it is output on the **CM** line.

OE Stands for Original Expansion. The compiler attempts to expand all data references that it determines not to be simple local or global variables. This expansion happens after all optimizations, including the filling of delay slots, have occurred. If this expansion is not successful it searchs for information about this data reference that may have been stored earlier in the compilation process as a side effect of code motion. If it finds this information a **CM** or a **BA** line (see below) will be output. If not, then the original expansion is output on the **OE** line.

AW This line means that there was no way to tell the order and manner in which each element of the memory reference may be accessed. Hence, when this reference is encountered it is assumed that the particular element at the time may be anywhere within the address space of the reference. A global or local base address of the reference is given.

IV Induction Variables. This line lists each induction register found for the loop in which the calculated reference given in the preceding BA or CM line was found. For each induction register preceded by a **1** in the list, this line gives the stride, minimum and maximum number of loop iterations, and the initial value in address string form. The **1** means that this information should be used to calculate a range of virtual addresses that may be accessed by this calculated reference. However, if the register is preceded by a **-1**, only the initial value is given, since it is to be used directly in place of the register in the address string in the preceding BA or CM line to compute a single virtual address. The **-1** is used to handle a special case in which access to the first element of an array may have been moved outside of a loop due to optimizations.

FN Function Name. This line gives the name of a new function, signaling that all of the following control-flow and data reference information is for program constructs and data references within this function.

SP Stack Space. This precedes a number giving the amount of space that will be used by the activation record for this function. It is used when computing relative offsets for local data.

FC Function Call. This is control-flow information that shows that a call to the named function will happen at the point in the program where this line occurs.

PM Actual Parameter Information. This line gives the number of and the best possible expansion of all actual parameters to the function called in the preceding `FC` line.

`--` Block Number Marker. Each basic block is marked by a number between these markers.

SL Left Successor. This precedes the basic block number of the left successor block of the current basic block. Used for fall-through.

SR Right Successor. This precedes the basic block number of the right successor block of the current basic block. Used for branching.

P Predecessors. This is the list of predecessor blocks of the current basic block. It is terminated by a **-1**.

NI Number of Instructions. This precedes a number giving the total number of instructions in this basic block.

G. Global Record. This begins a line that gives information collected in the front-end of the compiler about static data.

LC Local Information. This line gives the name and relative offset within the activation record of all local data objects within the current function.

LI Loop Iterations. This line gives the number of iterations of the loop in which the current block is the header. It will only be in blocks that are loop headers.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[2] R. Arnold. Bounding instruction cache performance. Master's thesis, Dept. of Computer Science, Florida State University, December 1996.

[3] R. Arnold, F. Mueller, D. B. Whalley, and M. Harmon. Bounding worst-case instruction cache performance. In *IEEE Symposium on Real-Time Systems*, pages 172–181, December 1994.

[4] M. E. Benitez and J. W. Davidson. A portable global optimizer and linker. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 329–338, June 1988.

[5] J. W. Davidson and D. B. Whalley. Quick compilers using peephole optimizations. *Software Practice & Experience*, 19(1):195–203, January 1989.

[6] J. W. Davidson and D. B. Whalley. A design environment for addressing architecture and compiler interactions. *Microprocessors and Microsystems*, 15(9):459–472, November 1991.

[7] M. Harmon, T. P. Baker, and D. B. Whalley. A retargetable technique for predicting execution time. In *IEEE Symposium on Real-Time Systems*, pages 68–77, December 1992.

[8] C. A. Healy. Predicting pipeline and instruction cache performance. Master's thesis, Dept. of Computer Science, Florida State University, December 1995.

[9] C. A. Healy, D. B. Whalley, and M. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *IEEE Symposium on Real-Time Systems*, pages 288–297, December 1995.

[10] C. A. Healy, D. B. Whalley, and M. Harmon. Worst-case timing analysis of instruction caches with wrap-around fill. TR 96-111, Florida State University, Department of Computer Science, 1996.

[11] Y. Hur, Y. H. Bea, S.-S. Lim, B.-D. Rhee, S. L. Min, Y. C. Park, M. Lee, H. Shin, and C. S. Kim. Worst case timing analysis of risc processors: R3000/R3010 case study. In *IEEE Symposium on Real-Time Systems*, pages 308–319, December 1995.

[12] S. Kim, S. L. Min, and R. Ha. Efficient worst-case timing analysis of data caching. In *IEEE Real-Time Technology and Applications Symposium*, pages 230–240, June 1996.

[13] L. Ko, C. Healy, E. Ratliff, R. Arnold, D. B. Whalley, and M. Harmon. Supporting the specification and analysis of timing constraints. In *IEEE Real-Time Technology and Applications Symposium*, pages 170–178, June 1996.

[14] L. Ko, D. B. Whalley, and M. Harmon. Supporting user-friendly analysis of timing constraints. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems*, volume 30, pages 99–107, November 1995.

[15] Y.-T. S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *IEEE Symposium on Real-Time Systems*, pages 298–307, December 1995.

[16] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *IEEE Symposium on Real-Time Systems*, pages 254–263, December 1996.

[17] S.-S. Lim, Y. H. Bea, G. T. Jang, B.-D. Rhee, S. L. Min, Y. C. Park, H. Shin, and C. S. Kim. An accurate worst case timing analysis for risc processors. In *IEEE Symposium on Real-Time Systems*, pages 97–108, December 1994.

[18] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Architectural Support for Programming Languages and Operating Systems*, pages 62–73, October 1992.

[19] F. Mueller. *Static Cache Simulation and its Applications*. PhD dissertation, Dept. of Computer Science, Florida State University, July 1994.

[20] F. Mueller. Generalizing timing predictions to set-associative caches. In *EuroMicro Real-Time Workshop*, June 1997. (accepted).

[21] C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–61, March 1993.

[22] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1(2):159–176, September 1989.

[23] J. Rawat. Static analysis of cache analysis for real-time programming. Master's thesis, Iowa State University, 1995.

[24] SPARC International Inc. *The SPARC Architecture Manual*, 1992. (Version 8).

[25] Texas Instruments. *TMS390S10 Integrated SPARC Processor*, February 1993.

[26] R. White, F. Mueller, C. Healy, D. Whalley, and M. Harmon. Timing analysis for data caches and set-associative caches. In *IEEE Real-Time Technology and Applications Symposium*, June 1997. (accepted).

[27] M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.

[28] N. Zhang, A. Burns, and M. Nicholson. Pipelined processors and worst case execution times. *Real-Time Systems*, 5(4):319–343, October 1993.

Bibliographical Sketch

Randall White was born in Orlando, Florida on April 17, 1965. He graduated as valedictorian from Apalachicola High School in Apalachicola, Florida in June 1983. He received his Bachelor of Science degree from the Florida State University in Computer Science in May 1987 and his Master of Science degree from FSU in Computer Science in December 1991. He will receive his PhD in Computer Science in April 1997. He is currently employed as a programmer and consultant with a local web development company. He is a member of the Association for Computing Machinery and the Upsilon Pi Epsilon Honor Society for the Computing Sciences. His research interests include programming languages, compiler theory, computer architecture, and parallel and distributed computing.