

Automating Selective Dynamic Compilation

Markus Ulrich Mock

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

University of Washington

2002

Program Authorized to Offer Degree: Department of Computer Science & Engineering

University of Washington
Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Markus Ulrich Mock

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Co-Chairs of Supervisory Committee:

Susan Eggers

Craig Chambers

Reading Committee:

Susan Eggers

Craig Chambers

Carl Ebeling

Date:

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to Bell and Howell Information and Learning, 300 North Zeeb Road, Ann Arbor, MI 48106-1346, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature_____

Date_____

University of Washington

Abstract

Automating Selective Dynamic Compilation

by Markus Ulrich Mock

Co-Chairs of Supervisory Committee:

Professor Susan Eggers
Department of Computer Science & Engineering

Professor Craig Chambers
Department of Computer Science & Engineering

Run-time specialization of programs can potentially improve execution time by exploiting the (semi-) invariance of values. Several research prototypes have been developed that enable the user to apply run-time specialization using annotations in the source code. While they were a great improvement over previous manual systems, in reality their use has been limited because writing annotations that achieve good program speedups is a challenging task for humans, requiring a deep understanding of the program's characteristics and of the applicable run-time optimizations and their effects on the underlying computer architecture.

In this dissertation I show that the major obstacle to the general application of such systems can be eliminated by automating the generation of annotations that drive a dynamic compiler. I demonstrate that a judicious combination of a novel static program analysis with run-time information about the program obtained by value profiling can achieve the same speedups as annotations written by human programmers and in a fraction of the time. I evaluate the concepts developed in this dissertation with a prototype system named *Calpa*. The prototype comprises algorithms and tools to gather run-time information about programs and static analysis algorithms to generate annotations to drive the DyC compiler, previously developed at the UW.

Moreover, I also show how combining run-time information with static analysis can be useful

in other contexts, by demonstrating the benefits of using dynamic pointer information in software engineering and improving the effectiveness of a program slicing tool.

TABLE OF CONTENTS

List of Figures	iv
List of Tables	vi
Chapter 1: Introduction	1
1.1 Exploiting Run-time Information for Optimization	3
1.2 The Promise of Run-time Specialization	4
1.3 The Thesis	6
Chapter 2: Background and Related Work	9
2.1 Run-time Specialization	9
2.2 Feedback-directed Optimization	22
2.3 Run-time Binary Optimization	23
2.4 Adaptive Run-time Compilation with JITs	26
Chapter 3: An Overview of the Calpa System	28
3.1 Tumi	30
3.2 Calpa’s Annotation Selector Tool	31
3.3 The Cost-Benefit Model	33
Chapter 4: Instrumentation Infrastructure	36
4.1 Instrumentation Overview	37
4.2 Frequency Profile Information	39
4.3 Value Profile Information	39
4.4 Instrumentation Library and Runtime	49
4.5 Profiling Evaluation	53

4.6	Related Value Profiling Approaches	59
4.7	Possible Improvements to Tumi	60
Chapter 5:	The Calpa Annotation Selector Tool	61
5.1	Pointer Analysis	61
5.2	Candidate Static Variable Analysis	64
5.3	Candidate Divisions and Search Strategy	68
5.4	The Cost-Benefit Model	71
5.5	Performance Estimation	79
5.6	Selecting Procedures for Annotation	82
Chapter 6:	Experimental Evaluation	84
6.1	Workload	84
6.2	Methodology	86
6.3	CSV Set Growth	86
6.4	Annotations	89
6.5	Profile Sensitivity	98
6.6	Summary and Future Work	104
Chapter 7:	Exploiting Dynamic Pointer Information	106
7.1	Dynamic Points-To Sets	108
7.2	Workload	109
7.3	Program Slicing	109
7.4	Methodology	112
7.5	Results	114
7.6	Efficient Generation of Dynamic Points-to Sets for Function Pointers	126
7.7	Conclusions and Future Work	127
Chapter 8:	Conclusions	129
8.1	Future Work	130

Bibliography	132
Appendix A: Instrumentation Library Prototypes	143
Appendix B: Source Code for Annotated Procedures	145

LIST OF FIGURES

1.1	Specialization Example	7
2.1	Partial Evaluation Example	12
2.2	Overview of the DyC System	15
3.1	Overview of the Calpa System	29
3.2	CSV Example	31
4.1	Tumi Instrumentation Procedure	38
4.2	Target Object Descriptors	42
4.3	Instrumentation Example for Scalars	45
4.4	Instrumentation Example before Transformation	46
4.5	Instrumentation Example after Transformation	47
4.6	Instrumentation Example for Structures and Unions	50
4.7	Data Capturing Routines for Integers	52
5.1	CSV Set Algorithm	67
5.2	Generation of New Divisions	70
5.3	Estimating Instructions Counts	75
6.1	Kernel Speedups	92
6.2	Application Speedups	93
6.3	Conditional Specialization	102
7.1	Average Points-To Set Sizes	115
7.2	Percentage of Singleton Sets	116

7.3	Percentage of Optimal Points-To Sets	117
7.4	Improvements with Dynamic Data	119
7.5	Slice Size Improvements	121
7.6	Number of Data Dependences	123
7.7	Classification of Data Dependences	123
7.8	Classification of Slice Improvement	125
A.1	Function Prototypes of Data Capturing Routines	144
B.1	binary	146
B.2	chebychev	147
B.3	dinero	148
B.4	dotproduct	156
B.5	quake	157
B.6	m88ksim	158
B.7	pnmconvol	159
B.8	query	161
B.9	romberg	162

LIST OF TABLES

2.1	DyC Program Speedups	20
4.1	Instrumentation Slowdown	54
4.2	Profiling Time Breakdown	55
4.3	Instrumentation Slowdown with Unhooking	58
4.4	Effects of Instrumentation on Program Size	58
5.1	Opcode Cost	78
5.2	Mathematical Operator Cost	80
6.1	Workload	85
6.2	Program Inputs	87
6.3	Number of Candidate Divisions	88
6.4	Annotations Generated by Calpa	90
6.5	Annotations Written by Humans	91
7.1	Program Descriptions	110
7.2	Average Slice Size	120
7.3	Filtered Slice Sizes	126

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisors Craig Chambers and Susan J. Eggers. They have taught me how to do research and helped me improve my writing and communication skills. Thanks goes to Carl Ebeling, for serving on my reading committee, and for being such a diligent reader of this dissertation. Calpa would not exist without the DyC system, so I owe thanks to Brian Grant and Matthai Philipose for their efforts in the design and implementation of it, as well as the many enriching discussions about the concepts and techniques behind it.

I would also like to thank Manuvir Das for his help in getting the One-Level Flow algorithm implemented quickly. I benefitted greatly from the his knowledge in static pointer analysis. Working with him was not only a pleasure, but has really launched me on a new path of interesting research questions.

Without Darren Atkinson and his expertise in program slicing, the study on using dynamic points-to data for program slicing would have been a much more arduous task. I greatly appreciate his work on getting the Sprite slicing tool to accept dynamic points-to sets. I enjoyed working with him, and he never stopped impressing me with both his enthusiasm and ready availability.

I thank my fellow graduate students sharing *Chateau, Chambre 112* with me, Alex, Ashish, Igor, Ratul, Sarah, and Tammy, for making coming to the office so much fun: it was never boring with you. I am indebted to my friend Anhai Doan for the many frank discussions, moral support and his insights, which helped me a lot on the final stretch. And last but not least, I thank my wife Patricia, for her love, support and understanding in all those years, *muchas gracias*.

”Love, work and knowledge are the wellsprings of our life. They should also govern it.”

(Wilhelm Reich 1897-1957)

Chapter 1

INTRODUCTION

Traditional static compiler optimizations are becoming less effective because of several recent changes in software and hardware technology. Increasingly, software is written in languages that support and encourage the use of abstraction mechanisms that require interprocedural optimization for an efficient implementation. For example, in object-oriented languages, typically small accessor methods are used to read or write object fields. Since the run-time cost of calling such small methods is relatively high compared to the simple functions they perform, it is of paramount importance that such small methods be inlined whenever possible, eliminating the call overhead and potentially enabling additional optimizations. Inlining, however, is often made impossible, when only partial programs are available to the compiler. Unfortunately, this is now quite common, because of the modularization of software, and the heavy use of library code.

An additional challenge for static compilers arises because software is increasingly written in a generic way, i.e., it is written to be (re-) usable in multiple contexts, true to good software engineering practice. For example, a sorting routine might be capable of sorting arrays of integers, floating point numbers and strings, and a rendering routine in a graphics package may be capable of rendering images with multiple light sources, even though it might be used most frequently with only a single light source. At compile time the particular (run-time) use of such a generic routine will generally not be known, preventing the compiler from making optimizations that would be possible if the particular use context were known.

In addition to the changing ways software is written today, it is now also often deployed differently, for instance in the form of dynamically linked libraries or dynamically loaded classes. For example, code written in the increasingly popular Java language uses dynamic class loading to use code provided by standard Java libraries. As a consequence, only parts of the whole program

are available to the compiler at compile time, which limits the scope of optimizations that can be performed.

Another area where static optimization techniques are of limited effectiveness are program transformations to improve memory access times. Since most computers today use intricate memory hierarchies to store programs and data, predicting the effects of memory accesses on performance are very difficult for the compiler. Therefore, static compilers typically do not attempt to perform any sophisticated optimizations beyond scheduling heuristics for load and store instructions. Program transformations based on run-time profiles, on the other hand, have been shown to be very effective, for example, for improving cache locality [CKJA98].

For all these reasons, the effectiveness of static program optimization has decreased in practice. To improve program execution time and use available processor resources effectively, new approaches are required. One key to the success of these approaches lies in the exploitation of run-time information. Run-time information can be used to provide precise information where static analysis is too imprecise, and it can drive optimizations performed at run-time. An example of the former is feedback-directed optimization, where run-time profile information is used to guide the optimization process in a static compiler. By taking advantage of more precise run-time data instead of relying on compile-time heuristics, the effectiveness of many optimizations can be significantly improved. Programs can also be optimized at run-time, by exploiting certain run-time properties, for instance, the invariance of variables. Run-time specializers take advantage of this by producing optimized code for specific run-time values of variables and data structures; their optimizations depend on the availability of the actual run-time values, and can only be performed once the values become known at run-time.

Since run-time information can often provide an advantage over purely static techniques, it is no surprise that a variety of run-time optimization approaches have been developed; Section 1.1 provides a short overview. One approach that promises potentially large performance benefits is run-time specialization, discussed in Section 1.2. This section also examines why run-time specialization is currently not more widely used despite its potential benefits. The remedy to this situation is the topic of the thesis of this dissertation, presented in Section 1.3.

1.1 Exploiting Run-time Information for Optimization

Various optimizations have been devised that exploit run-time information to improve program performance. One approach that requires little change to an existing compiler is feedback-directed optimization. The idea behind it is to leverage run-time information at compile time. Typically, this works as a two-step process. First, the program is compiled with a profile-gathering option turned on in the compiler. After running the program at least once, the program is compiled a second time, using the run-time profile information to make better compile-time optimization decisions. Feedback-directed optimization has been particularly successful for optimizations that exploit locality of reference for a better use of the memory hierarchy [PH90, HKC97], and for procedure inlining [CL99].

Link-time optimization is another approach to compensate for the limited effectiveness of traditional compile-time optimization. To be effective, link-time optimization often has to reconstruct higher-level control and data flow information to ensure the correctness of its transformations, which can be expensive. Moreover, while some of the transformations (e.g., code positioning) can be conveniently and cheaply performed at link time, selecting which transformations to perform usually requires information obtained by profiling, which makes many link-time optimizations similar to feedback-directed optimizations except that the linker performs the actual program transformation rather than the compiler.

Since the Java programming language relies heavily on dynamic class loading and encourages the use of abstraction mechanisms such as polymorphism, a lot of attention has been devoted to improving the performance of Java programs by *Just-In-Time* (JIT) compilation. JIT systems typically start out by either interpreting or quickly compiling Java methods as they are executed without applying optimizations that require substantial analysis or transformation time. As the program executes, they try to identify *hot spots*, i.e., regions of code that are executed frequently, to focus optimization effort to regions where optimizations are likely to have the biggest impact and then compile them applying more time-consuming and potentially more beneficial code transformations. Deciding when such (re-) compilations should occur is one of the main challenges of JIT systems; many rely on hard-coded heuristics, some use an explicit cost-benefit model to make these decisions [AFG⁺00].

While JITs optimize programs represented in portable bytecode format, binary optimizers operate directly on the executable, either with or without hardware support. The advantage of this approach is that it is applicable even when the source code cannot be obtained (for instance, third-party code), or when the executable is not fully known at program start (for instance, dynamically loaded code modules). While optimization at the binary level is very flexible, the need to obtain all information required to ensure the correctness of code transformation at run time restricts the transformations to relatively simple ones whose performance improvement is also limited. In contrast, when source code is available at compile time, run-time transformations can be preplanned statically, thereby enabling more complex and effective transformations to be performed at run time. One example of a run-time transformation that can be preplanned at static compile time is *run-time specialization*.

1.2 The Promise of Run-time Specialization

It is particularly effective in removing interpretation or parameterization overheads from software with potentially large performance gains; for instance, programs optimized with the DyC system ran up to 4.6 times faster than statically compiled and fully optimized programs without run-time specialization [GPM⁺99]. Parameterization or interpretation overhead is found in many programs today, since good software engineering practice encourages writing code that can be reused in multiple contexts. Therefore, procedures are often coded to be more general than required by the context in which they are used.

Run-time specializers eliminate some of this generality by producing code that is specialized for specific values of variables or data structures, as those values become available at run time, and perform subsequent value-specific optimizations, such as constant folding, strength reduction, and full loop unrolling. Unlike compile-time specializers [JGS93], run-time specialization systems do not require prior knowledge of the values that will be used for optimization, a prerequisite which has typically limited the applicability of compile-time specialization.

Although run-time specializers can be applied more widely than compile-time specializers, they have to spend time transforming the program at run-time before executing it. Consequently, a judicious choice of where and what transformations should be applied is necessary to ensure that the

benefits resulting from the transformations outweigh the costs of performing them at run time. All run-time specialization systems to date have put this choice in the hand of their users, either by requiring the programmer to explicitly compose code templates that are used to produce specialized code at run time (e.g., the 'C system [EHK96]), or by requiring the programmer to put annotations in the program source code that are used by a subsequent analysis pass to determine where to specialize code, and which particular transformation to apply at run time [LL96, GMP⁺97, CHM⁺98].

An example illustrating the potential benefits of run-time specialization is shown in Figure 1.1. Figure 1.1(a) shows the code of procedure `rp`, which converts Cartesian coordinates (x, y) to polar coordinates $(radius, theta)$. Suppose it is called repeatedly with $y=0$. We can take advantage of this by producing a specialized version of `rp` for $y=0$ and executing it in lieu of the generic version, whenever `rp` is called with $y=0$. The specialized version (named `rp0` in Figure 1.1(b)) is produced by evaluating all computations whose result can be determined by knowing y alone. For instance, the multiplication $y*y$ and subsequent addition has been eliminated in `rp0`. Similarly, the `if`-test $y \geq 0$ is evaluated, and the false branch is eliminated along with several other computations. Because fewer computations are performed in `rp0`, it executes faster than the original routine.

In an annotation-driven run-time specializer, such as DyC, a programmer specifies what code should be specialized using annotations. To cause the transformations whose results are shown in Figure 1.1(b) (except for the elimination of argument y), the programmer puts one annotation in the source code (shown in Figure 1.1(c)). The `make_static(y)` annotation tells the DyC system to produce specialized versions of procedure `rp` for the values of variable y . Driven by this annotation, DyC produces executable code for `rp`, which, when invoked at run time, produces and invokes specialized versions of `rp` that will be executed in lieu of the original, unspecialized and slower version. While one simple annotation suffices in DyC to specialize this example, an imperative run-time specializer requires the user to explicitly code the transformations to be performed at run time by hand. This is typically both tedious and error-prone, which has confined the use of imperative run-time specializers to small programs.

Although annotation-drive systems are easier to use than imperative systems, they still require user intervention. Even though the users are relieved of the burden of explicitly programming the run-time transformations, they are still required to determine where such transformations might be beneficial to be able to write annotations that result in program speedups. Unfortunately, potential

speedups depend on a number of factors: the run-time characteristics of the program to be optimized, the run-time effects of the transformations performed by the run-time specializer and their effectiveness on the underlying computer architecture. Achieving an understanding of these factors for realistic applications requires considerable programmer effort. This burden has turned out to be a major hindrance to the widespread application of run-time specializers in practice.

1.3 The Thesis

The goal of my dissertation research was to overcome the main obstacle in the use of run-time specialization systems by obviating the need for user annotations. This goal is achieved by demonstrating the veracity of the following thesis statement:

Annotation-driven run-time specialization systems can be automated by generating annotations that are likely to improve program execution time, based on automatic static analysis and run-time value profiling alone.

In this dissertation I demonstrate that a judicious combination of static program analysis and run-time information – obtained by program profiling – eliminates the need for user annotations. To validate the concepts developed in this thesis, I designed and implemented a prototype system, named *Calpa*¹ [MBCE99, MCE00], which was used to automatically generate annotations for the DyC run-time specializer, previously designed and implemented at the University of Washington by several graduate students, including me. With *Calpa*, it is now possible to try to optimize larger programs than could previously be annotated by a human user of a run-time specializer.

The techniques implemented in *Calpa* are evaluated by comparing the annotations and the ensuing program speedups with annotations previously found to be the best by human programmers. In addition, *Calpa* is shown to successfully exploit optimization opportunities on some previously untried application, considered too large to be effectively annotated by hand. Since speedups achieved with run-time specializers are potentially input-dependent, I also evaluate *Calpa*'s sensitivity to changing program inputs when generating annotations and show how annotations can be made robust.

¹Calpa is the name of an ancient Inca oracle ritual that used the entrails of llamas and other camelids to predict the future. The *Calpa* system presented in this thesis uses run-time information and a cost-benefit model to predict the performance impact of possible annotations.

```

double rp(double x, double y, double *a) {
    double radius, theta;
    radius = sqrt(x*x + y*y);
    if (x == 0)
        theta = y >= 0? (PI / 2) : -(PI / 2);
    else {
        if (x > 0) theta = atan(y / x);
        else if (y >= 0)
            theta = atan(y / x) + PI;
        else
            theta = atan(y / x) - PI;
    }
    *a = theta;
    return radius;
}

```

Figure 1.1a: This example shows a routine that converts Cartesian coordinates (x, y) to polar coordinates $(radius, theta)$.

```

double rp0(double x, double *a) {
    double radius, theta;
    radius = sqrt(x*x);
    if (x == 0)
        theta = (PI / 2)
    else {
        if (x > 0) theta = 0;
        else      theta = PI;
    }
    *a = theta;
    return radius;
}

```

Figure 1.1b: Routine resulting after specializing `rp` for `y=0`.

```

double rp(double x, double y, double *a) {
    double radius, theta;

    make_static(y);

    radius = sqrt(x*x + y*y);
    if (x == 0)
        theta = y >= 0? (PI / 2) : -(PI / 2);
    else {
        if (x > 0) theta = atan(y / x);
        else if (y >= 0)
            theta = atan(y / x) + PI;
        else
            theta = atan(y / x) - PI;
    }
    *a = theta;
    return radius;
}

```

Figure 1.1c: Annotations used in the DyC system to produce specialized versions of `rp` for the values of `y` at run time.

While Calpa shows that run-time information can be successfully combined with static analysis to improve program performance, the application of run-time information is not limited to this domain. I also show how one specific type of run-time information, dynamic pointer information, can be exploited to improve certain software tools and also to obtain a bound on potential improvements arising from better static pointer analyses.

More specifically, this thesis makes the following contributions:

- It presents a program analysis that combines static analyses with value profiling to identify good annotation candidates and optimization regions for run-time specialization.
- It develops a multi-faceted cost-benefit model for run-time specialization that enables the automatic evaluation of the costs and benefits of dynamically compiling different annotation candidates.
- It demonstrates experimentally that these techniques produce annotations that are of comparable quality as those found by human programmers, but requiring only a small fraction of the time it takes human programmers to annotate a program.
- It shows that the usefulness of run-time information extends beyond applications in program optimization by showing how a software engineering method, program slicing, can be improved with dynamic pointer information.

The remainder of this dissertation is organized as follows. Chapter 2 provides background about dynamic program optimization in general, with a particular focus on the DyC system. Chapters 3 – 5 provide an overview of the Calpa system, discuss the instrumentation infrastructure that was used to obtain the run-time information used by Calpa, and give details of the algorithms used in Calpa and their implementation. Chapter 4 also includes a discussion of related instrumentation approaches for value profiling. Calpa’s design is evaluated experimentally in Chapter 6. Chapter 7 describes how run-time pointer information can be exploited to improve program slicing. And finally, Chapter 8 provides a summary of the contributions of this thesis, discusses some unresolved issues, possible improvements, and ideas for future work.

Chapter 2

BACKGROUND AND RELATED WORK

As mentioned in the introduction, various research efforts have been undertaken to achieve program speedups by run-time optimization. The goal of this thesis is to demonstrate that program performance can be improved with run-time specializers without the need for any human intervention. This is accomplished by showing how annotations that drive a run-time specializer can be generated automatically. To provide the necessary background for this work, this section discusses run-time specialization, in particular the DyC run-time specializer, which was used to validate the algorithms developed in this thesis.

Apart from run-time specialization, there are other ways to optimize programs at run time. This chapter also describes related work in run-time optimization that does not require any effort on the programmer's part. The existing approaches can be categorized into the following groups: (1) feedback-directed optimization systems; (2) binary run-time optimizers, such as Dynamo; and (3) adaptive run-time compilation systems, e.g., Just-In-Time Compilers (JITs) for the Java language.

The remainder of the chapter is organized as follows. Section 2.1 provides some background on run-time specialization with a focus on how the DyC run-time specializer works. Section 2.2 describes the ideas behind feedback-directed optimization, and Section 2.3 describes how binary optimizers work. Finally, Section 2.4 discusses some work on adaptive compilation in the context of Java JITs.

2.1 Run-time Specialization

Several systems have been designed that attempt to achieve program speedups by code specialization for specific run-time values of variables or data structures. These systems require the programmer either to explicitly specify how and what code should be generated at run-time (e.g., 'C [EHK96]),

or to control the run-time specialization by source code annotations (Tempo [CN96, CHM⁺98], DyC [GMP⁺00b], and Fabius [LL96]). Common to all run-time specialization systems is that they achieve their speedups by replacing a generic version of a procedure (or subsection thereof, in case of the DyC system) with a specialized version by partially evaluating the procedure.¹

Partial evaluation [JGS93] is a program transformation that takes a procedure p (or a part thereof) and a subset s of the procedure's inputs and replaces p by a procedure p' obtained by evaluating all computations of p that only depend on the values of s . For example, the partial evaluation of procedure `mul-add` in Figure 2.1(a) with parameter $a = 1$ and parameter $b = 0$ results in a procedure named `mul-add_1_0` in Figure 2.1(b), where the multiplication by a and addition of b have been evaluated and elided from the residual code. Partial evaluation achieves program speedups by either removing some computations from the original program, or by enabling the replacement of more expensive with cheaper computations. The latter occurs, for instance, if procedure p is partially evaluated with parameter $a = 2$. In that case, while the multiplication cannot be eliminated completely, it can be implemented with a shift instruction (as shown in Figure 2.1(c)), which in general is significantly faster than a multiplication instruction.

A major limitation of the partial evaluation transformation is that s , the inputs that are used to partially evaluate the procedure, have to be known before running the program. When that is not the case, users of partial evaluators resort to a technique commonly called "the trick" [JGS93]. The trick consists of producing partially evaluated programs for all possible values of the input set s . The obvious problem with this approach, however, is that it can lead to severe code blowup, resulting in an overall program slowdown due to decreased cache locality.

One solution that avoids code blowup is to defer partial evaluation until run time, when the input values are definitely known, and only produce those partial evaluations that are actually going to be used. When partial evaluation is performed at run-time, it can often lead to substantial program speedup, in particular for programs that are heavily parameterized (by producing specialized code for the parameter values that are actually used at run time), or that are interpretative in nature,

¹Partial evaluation and specialization are often used interchangeably. In this document, *partial evaluation* is used to denote the transformation that takes one version of code and transforms it into a specialized version by evaluating the computations that depend only on the inputs that are used in the partial evaluation step. The term *specialization* is used for any method that produces a specialized code version from a generic one, regardless of how this transformation is performed, including, for instance, data specialization [KR96]. *Dynamic compilation* refers to any kind of run-time optimization system that generates code at run-time, including but not limited to run-time specialization systems.

for instance, simulators or interpreters (because partial evaluation can remove the interpretation overheads).

As previously mentioned, run-time specialization systems can be either imperative or annotation-driven. 'C [EHK96] is an example of an imperative run-time specializer for C. To achieve run-time specialization, the programmer must write C code that performs the actual partial evaluation transformation at run time. Compared to annotation-driven systems, specializing programs with 'C is more tedious and complicated. Therefore, in practice, 'C appears to have been used only for relatively small programs, because for larger programs the burden for the programmer becomes too big. Annotation-driven systems on the other hand, have been used successfully on programs of over 10,000 lines of code.

The following section provides more details about one of these, the DyC system, and presents some details about its annotations, and its run-time compilation costs and benefits. The Tempo run-time specializer is compared briefly to DyC in Section 2.1.5; while there are some important differences between the systems, the comparison shows they are sufficiently similar, so that with some modifications to Calpa, it should also be able to produce annotations for Tempo. And finally, Section 2.1.6 discusses the Fabius system, a run-time specializer for a subset of the ML language.

```
int mul-add(int x, int a, int b) {  
    int r;  
    r = x * a + b;  
    return r;  
}
```

Figure 2.1a: The `mul-add` procedure.

```
int mul-add__2(int x, int b) {  
    int r;  
    r = x << 1 + b;  
    return r;  
}
```

Figure 2.1c: Evaluating `mul-add` with parameter `a=2` also improves performance because the multiplication by 2 can be replaced by a faster shift operation, yielding residual procedure `mul-add__2`.

```
int mul-add_1_0(int x) {  
    int r;  
    r = x;  
    return r;  
}
```

Figure 2.1b: Partial evaluation of procedure `mul-add` with parameters `a = 1` and `b = 0` results in procedure `mul-add_1_0` where the multiplication and addition have been eliminated, which saves execution time.

2.1.1 The DyC System

DyC (pronounced *dicey*) is an annotation-driven (declarative) run-time specializer for the C programming language that optimizes programs for the Alpha processor.² Its specialization directives are *selective*, i.e., arbitrary sub-regions of code rather than whole procedures can be optimized. DyC performs most of its program analysis at static compile time (i.e., its run-time optimizations are *staged* [PCE02]) to achieve very low run-time code generation costs, enabling speedups of up to 4.6 on medium-sized (< 15K lines of code) C programs [GPM⁺99]. Several techniques are used in DyC to achieve this level of performance: (1) a sophisticated form of partial-evaluation-style binding time analysis (BTA) that supports program-point-specific polyvariant division and specialization; (2) low-cost, dynamic versions of traditional global optimizations that include zero and copy propagation and dead-assignment elimination, and (3) dynamic peephole optimizations, such as strength reduction.

DyC's polyvariant division and specialization are powerful mechanisms, that make DyC's annotation language particularly expressive. Polyvariant division allows the same piece of code to be analyzed with different combinations of variables being treated as run-time constants; each combination is called a division. Polyvariant specialization allows multiple compiled versions of a division to be produced, each specialized for different values of the run-time-constant variables. Program-point-specific polyvariance commences at arbitrary points in programs, not just at function entries. Polyvariant specialization can result in complete loop unrolling by creating a specialized copy of a loop body for each set of values of the run-time-constant loop induction variables. Complete loop unrolling is unlike unrolling done by traditional static compilers in that the loop is eliminated rather than enlarged. For simple loops, such as those that merely increment a counter until an exit condition is reached, a linear chain of unrolled loop bodies results (which we call *single-way loop unrolling*). For more complex loops, one iteration may lead to several alternative loop iterations (e.g., if it contains branch paths that update the loop induction variables differently), or even return to a previously executed loop iteration, producing in general a directed graph of unrolled loop bodies (which we call *multi-way loop unrolling*).

²Annotations are put in the C source code, and program analysis is performed at the intermediate code level; the actual run-time specialization, however, is performed on program binaries and was implemented for the Alpha architecture.

To trigger dynamic compilation, programmers annotate their source code to identify *static* variables, which are variables for whose values DyC will generate specialized code at run time. Loads from memory can be annotated as static if their contents are run-time constant, and called procedures can be annotated as static if they return the same result whenever given the same arguments (and when they have no side effects). When procedures are annotated as static, they are called just once for a particular argument list, and the results are memoized.

DyC's BTA analyzes code downstream of the annotations, intraprocedurally, to separate those computations that depend solely on the annotated static variables, plus additional static variables that are derived from them (called *static computations*), from the computations that depend at least in part on other variables (called the *dynamic computations*). Static computations are executed just once, at dynamic compilation time. By delaying final compilation of the dynamic computations until run time, the results of the static computations can be treated as embedded constants in the dynamic computations. Since DyC's BTA is program-point-specific and flow-sensitive, a *dynamic region*, i.e., the program region for which specialized code is produced, can start and stop at any program point, and a variable may be static at some program points and not at others. For each dynamic region, DyC builds a customized dynamic compiler (also called a generating extension [JGS93]) that generates code at run time, using the values of the static variables once they become known. On entry to a dynamic region, a *dispatcher* selects the appropriate specialized version based on the values of the specialized variables, or invokes the dynamic compiler to produce a new version.

Optionally, the BTA can perform a *reachability* analysis, a flow-sensitive data flow analysis which computes control dependence expressions for every program point that specify under what conditions the program point is reachable. For instance, the branches of an `if`-statement will include the control condition of the `if` on the `true`-side of the `if`, and its negation on the `false`-side. In some cases this enables the BTA to determine that certain computations are static when it would otherwise have to conservatively conclude that they might depend on dynamic values. Therefore, with reachability analysis possibly more computations may be eliminated from the program resulting in larger specialization benefits.

When a dynamic region is entered at run time, the region's dispatcher checks an internal cache of previously dynamically generated code for a version that was compiled for the current values of the annotated variables. If one is found, it is executed. If not, the dispatcher invokes the region's

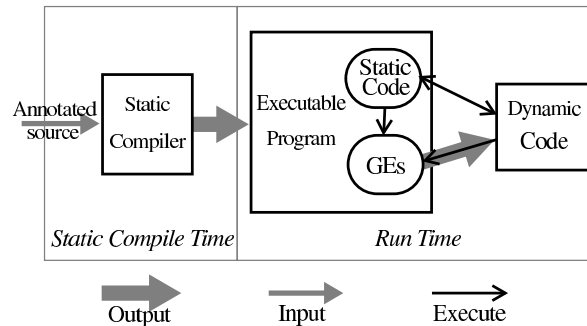


Figure 2.2: This graphs show how a program is optimized with the DyC system. The annotated C source code is fed into the DyC compiler which includes an optimizing static compiler. It produces an Alpha binary, which includes statically optimized code, labeled static code, and generating extensions (customized dynamic compilers), which, when invoked, produce specialized code at run time (dynamic code).

custom dynamic compiler to generate code that is specialized to the current values of the annotated variables. The custom compiler evaluates the static computations and emits machine code for the dynamic computations. When done, the newly generated code is saved in the dynamic code cache and then executed. Invoking the dynamic compiler and dispatching to dynamically generated code are the principal sources of run-time overhead. Figure 2.2 shows graphically the components that make up the DyC system and the steps involved in optimizing an application with it.

If a load is annotated as static, the dynamic compiler assumes that the contents of the referenced memory location remains the same for all invocations of the dynamic region. If the memory location is later updated, then the affected dynamically compiled code should be thrown away and re-dynamically compiled for the new value. In the current DyC system, whenever a store changes the contents of a memory location upon which code might be specialized, the program must invoke DyC's `invalidate()` operation, which flushes the compiled code caches of all affected functions, and resets the dynamic region's code pointer to point to the dynamic compiler. The dynamic compiler will then produce new code for the invalidated dynamic region, based on the new contents of the memory location. While the respecialization will be automatic after the code cache has been invalidated, it is the user's responsibility to insert the `invalidate()` directives at all program points

where they are required to ensure correctness.

2.1.2 DyC's Annotations

DyC is driven primarily by annotations that identify initial run-time constant variables and data structures. Optional policy annotations allow the programmer to specify whether specialization and division should be mono- or polyvariant, whether code downstream of conditional branches or switches should be dynamically compiled eagerly or lazily, and whether the dynamic code cache for each region should keep many, one, or zero code versions. A final annotation identifies program points where `invalidate()` calls should be inserted. The following paragraphs give a high-level description of DyC's annotations, concentrating on those that are relevant for the run-time overhead incurred by DyC. The full set of annotations is described in detail in [GMP⁺00b].

The `make_static` Annotation

DyC uses an annotation named `make_static` to declare the variables for which code should be specialized. For instance, the user would put `make_static(x, y)` in the code to specify that downstream of the annotation, code for the values of `x` and `y` should be specialized. Similarly, an annotation `make_dynamic` is used to declare that the downstream specialization for the named variables should no longer be used. Which instructions are affected by the annotation is determined by a data flow analysis; essentially, the `make_static` annotation for a variable is propagated along control flow edges until it reaches the end of the procedure or a corresponding `make_dynamic` annotation.

Policy Annotations

DyC provides various policy annotations to control several parameters of the specialization process. For run-time optimization behavior and its associated run-time cost the specialization and caching annotations are most important. The specialization policies control the number of specialized versions produced. To produce just one version (i.e., use monovariant specialization), the policy annotation `mono_specialize` is used. To produce multiple versions (i.e., use polyvariant specialization) the `poly_specialize` policy annotation is used.

The caching policies control how many code versions should be cached for an annotated variable

and whether the cache check to dispatch to the corresponding code version can be omitted. The `cache_all` and `cache_all_unchecked` policies cache as many code versions as necessary, with or without performing a cache check, respectively. The `cache_all_unchecked` policy is generally used for full loop unrolling when the iteration space can be completely determined at specialization time. For instance, a `for`-loop that is counting from 1 to n , can simply be replaced with the n versions of the loop body.

If only one code version per annotated variable should be produced at run time, the programmer uses the `cache_one` or `cache_one_unchecked` policy. In the former case, a cache check is performed and if it fails, code is re-specialized for the current value of the annotated variable and replaces the currently cached code. For the `cache_one_unchecked` policy the cache check is omitted, so that after producing the initial specialized code version, that single code version is executed in all future executions. For this to produce correct results, the variable must be guaranteed to have only one value during execution. It is the user's responsibility to ensure that the annotation is used only where it is appropriate.

The `invalidate()` Annotation

The `invalidate()` annotation is different from the other annotations because it applies only to the program point where the programmer inserts it. It causes DyC to throw away all specialized code for the procedure named in the annotation to cause a re-specialization to occur the next time any of the dynamic regions of that procedure is executed.

This annotation is particularly tedious to use for a human programmer, since it has to be put at all program points in the program where the specialized code might potentially become incorrect and has to be *invalidated*. For instance, if the programmer annotates loads from an array `a` of size `n` to be static, at every program point in the program where the contents might be modified, the programmer must insert a (possibly guarded) `invalidate()` annotation. For example, the statement `a[i] = x;` would have to be replaced by `a[i] = x; invalidate(f);`, assuming that the specialized code region is in procedure `f`. For every pointer store that might change the contents of `a`, a conditional check would be inserted; for example, `*p = 0;` would be replaced by `*p = 0; if ((a <= p) && (p <= & a[n-1])) invalidate(f);`, to invalidate the code cache only when the pointer store

would in fact change the contents of the array.

2.1.3 *Benefits and Costs of DyC's Transformations*

Since DyC optimizes programs at run time, the performed transformations must yield at least as much time savings as it costs to perform them. This section discusses how applications may benefit from DyC's transformations, and their associated run-time costs [GMP⁺00a].

Benefits

Like other run-time specialization systems, DyC obtains most of its optimization benefits from the elimination of static instructions from the specialized code. The dead code elimination it performs, is beneficial only indirectly by decreasing the memory footprint of the dynamically generated code. This, however, becomes important when polyvariant specialization is used, particularly when loops are fully unrolled. For DyC, strength reduction can sometimes provide noticeable additional benefit when multiplications or divisions can be eliminated from the instruction stream, since the Alpha processor's integer multiplication and division instructions, are very expensive (approximately 8 and 32 cycles respectively) [Gra01].

Costs

Optimizing code with DyC incurs the following run-time costs:³ (1) the *specialization cost*, i.e., the cost of generating the specialized code; (2) the *caching cost*, i.e., the cost of dispatching to and selecting the correct code version from the code cache; and (3) the *invalidation cost*, i.e., the cost of checking whether executing an `invalidate()` is necessary and the cost of the `invalidate()` operation itself.

Since DyC performs most of its analysis at static compile time, the cost of producing specialized code at run time is very low. Typically, it takes on the order of a hundred cycles to produce an instruction at run-time, with code generation costs ranging from 13 cycles to 823 cycles per instruction generated (on a 500MHz Alpha 21164 workstation). To a first approximation the specialization cost

³When not stated otherwise, cost in this and subsequent sections always refers to an execution time cost, i.e., the run-time cost it takes to perform a certain operation.

is proportional to the number of instructions generated. When polyvariant specialization is used, the code generation cost is incurred for each new instance of an instruction that is generated, i.e., for each distinct value a specialized code version is produced for the instruction.

The dominant cost component in DyC’s caching mechanism comes from the cache key construction and the subsequent hash table lookup to find the address at which the specialized code starts. The key construction cost is proportional to the key size, i.e., number of variables that form the key. The hashing cost is roughly constant.⁴ This cost ranges from 63 to 81 cycles. The remaining cost to actually branch to the specialized code is roughly constant, ranging from 10 to 15 cycles on an Alpha 21164 [Gra01]. This means that the unchecked caching policies are a lot cheaper (up to factor of 9 faster) than the checked policies.⁵

The costs of invalidation-based caching are much smaller as well, since only a simple check of a flag has to be performed (to test whether the code cache has been flushed). In his measurements, Grant reports a total cost of 8 to 15 cycles (including the branch to the specialized code) [Gra01]. This, however, does not include any costs incurred at invalidation points. When invalidation points are present, the cost of performing the `invalidate()` operation (which invalidates the code cache) as well as the cost to perform the run-time check that may guard it (which determines whether the code cache has to be invalidated) have to be taken into account. The cost of the latter will be dependent on the complexity of the guard condition.

2.1.4 Performance of the DyC System

Grant et al. [GPM⁺99] showed that DyC is able to speed up a variety of different applications, from interpreters to graphics applications to simulators. Some of its speedups are shown in Table 2.1. The first three columns list the name, description and size (in lines of C code) for each application. Column four lists the speedup for an application’s optimized dynamic region.⁶ Column five

⁴This assumes there are no or few collisions, which is generally the case if the hash table is sufficiently large and the hash function is good. The hash table size was chosen large enough to ensure this in practice [GMP⁺00a, Gra01].

⁵In his Ph.D. thesis [Gra01], Grant also explored an alternative implementation for the cache lookup, based on the idea of inline caching and polymorphic inline caching [HCU91], which, when applicable, can often significantly reduce the dispatch time.

⁶For application `viewerperf` there were two dynamic regions. The region speedup shown in the table is for the shader routine; the other region (routine `project&clip`) achieved a region speedup of 1.3.

Table 2.1: This table shows the dynamic region and whole-program speedups achieved for several medium-sized benchmarks.

Application	Description	Total size (lines)	Region speedup	execution %	Whole-programs speedup
dinero	cache simulator	3,317	1.7	49.9	1.5
m88ksim	Motorola 88000 simulator	12,531	3.7	9.8	1.05
mipsi	MIPS R3000 simulator	3,417	5.0	≈100	4.6
pnmconvol	image convolution	1,054	3.1	83.8	3.0
viewperf	renderer	15,006	1.2	41.4	1.02

is the percentage of execution time an application spends in the program dynamic region (in the unoptimized application). And the last column is the speedup for a whole program.

Whole-program speedups depend on a variety of factors; most noticeably, when the dynamically optimized region accounts for a large fraction of the overall execution time (almost 100% for `mipsi` and over 80% for `pnmconvol`), the improvements obtained from run-time specialization translate almost directly into considerable whole-program performance improvements. However, even for `m88ksim` or `viewperf`, the *dynamic region speedups* were substantial: 1.2 for `viewperf` and 3.7 for `m88ksim`.

2.1.5 The Tempo System

Tempo [CN96, CHM⁺98] is another example of an annotation-driven specializer for the C programming language. Both systems are quite similar and require the user to declare where and for which variables code should be specialized. Unlike DyC, Tempo performs interprocedural analysis including a pointer and side-effect analysis, enabling specialization across call boundaries. The pointer analysis makes Tempo easier to use for the programmer since Tempo identifies invalidation points automatically, whereas DyC requires the programmer to identify them. However, when the source code for some modules is not available to Tempo, the user is required to write a specification describing the potential side-effects and alias relationships for them, and Tempo depends on their correctness to guarantee the correctness of the specialization.

Both DyC and Tempo support polyvariant specialization and division; however, while DyC's

annotations enable the specialization of sub-parts of procedures, Tempo only supports whole procedure-level division and specialization. It also does not provide annotations to control the caching of specialized code; this is user's responsibility, who has to explicitly allocate and free buffers for specialized code. DyC also performs some run-time optimizations that Tempo does not support, including strength reduction, zero and copy propagation, and dead assignment elimination. DyC's greater versatility makes it applicable in contexts where Tempo cannot be used [GPM⁺99]; however, for applications they both can specialize, its speedups are comparable to those achieved with the DyC system [CN96, CHM⁺98].

Since both systems achieve their speedups by partial evaluation, support polyvariant specialization and division, they obtain the same kinds of specialization benefits. Similarly, they incur the same kind of run-time costs (specialization, code caching and invalidation costs). Consequently, Calpa's approach of automatically generating annotations for DyC should work for Tempo as well, requiring only a minor change in the basic cost parameters in the cost-benefit model used to evaluate possible annotations. To model Tempo's interprocedural specialization capability, however, Calpa's scope of analysis would need to be extended to an interprocedural level.

2.1.6 *The Fabius System*

Unlike Tempo or DyC which specialize code for the C programming language, the Fabius [LL96] system is a run-time specializer for a purely functional subset of ML. Since no side-effects are possible in this language, neither the user nor the specializer have to deal with code invalidation or pointer analysis to detect possible side-effects, which simplifies both the use and the implementation of the run-time specializer. Like Tempo and DyC, the user has to provide annotations to declare which variables should be specialized. In Fabius, the programmer uses a form of function currying to tell the system what function arguments should be specialized. Based on the currying annotation, Fabius performs all dynamic code generation and management automatically. However, the annotations provide no way of specializing sub-parts of functions, or of controlling the dynamic compilation and optimization process. In addition, Fabius performs little other cross-statement optimization besides register allocation.

2.2 *Feedback-directed Optimization*

Unlike run-time specialization, feedback-directed optimization tries to improve the effectiveness of standard static compilers by using run-time profile information to guide the (static) compilation process. Since it requires very little change to the static compiler – just the ability to read the output of a profiling tool and use it in the optimizer –, most commercial compilers for the C programming language today include at least rudimentary support for feed-back directed optimization. The advantages compared to run-time optimization systems are that no run-time compilation cost is incurred that has to be recouped by the optimizing transformation, and no special run-time optimizer has to be designed.

Feedback-directed optimizers for the C programming language typically use a standard basic block or procedure execution profile, which gives execution frequency counts for each basic block and procedure, respectively, to guide their optimization process. For example, in Compaq's compiler for the Alpha processor [CL99], basic block execution counts are used to improve several of its static optimization passes. First, it uses execution counts to help decide whether to inline a procedure. In the absence of profile information, the optimizer estimates how often a procedure might be called and gives preference to frequently executed procedures. In addition, procedures just called from one site are inlined since that will not negatively affect temporal locality in the instruction cache. With profile information, functions that are *almost always* called from one call site are inlined as well, and the execution data is used to select these procedures. Compared to the static heuristic, the profile-driven optimization improved performance on average by 10% on a 500MHz Alpha 21264-based workstation. Compared to a baseline without inlining, the static heuristic improved performance by about 3%, whereas the feedback-driven inlining improved performance by about 13% on average.

The observation that feedback-directed procedure inlining can produce good program speedups, is also confirmed by the work of Ayers et al. [ASG97], who implemented an aggressive feedback-directed procedure inlining and cloning framework that operates at the intermediate code level, which enables aggressive inlining across file boundaries. Using frequency profiles, their optimizer aggressively inlines and clones procedures, improving the performance of SPEC95 applications on a PA8000-based workstation by an average of 32%, with a maximum speedup of 1.8. They identified two principal sources for these large speedups: a significant decrease of data cache misses due to

eliminated caller and callee register saves, and the elimination of many branches. The aggressive inlining showed no clear trend on instruction cache performance; for some programs instruction cache misses increased, for others they actually decreased (due to a reduction in the total number of instruction cache accesses).

The second-most successful feedback-directed optimization in the Compaq study was code layout, which used a slightly modified version of the Pettis and Hansen [PH90] algorithm. On average the layout optimization based on profile data improved performance by 4%, with a maximum of 20% for the `vortex` SPEC benchmark.

The third optimization that sometimes benefited significantly from profile data was trace scheduling and superblock formation. On average it improved performance by 3%, with a maximum of 10% improvement for the `li` benchmark. Using profile data for other optimizations had little effect, improving performance on average by not more than 2% and for no application more than 6% (which was achieved for loop optimizations in the `jpeg` benchmark).

In summary, it appears that profile data (at least execution count data) is useful for code layout, inlining and trace scheduling optimizations. For others, like register allocation or loop optimizations, feedback-directed optimization does not appear to offer a significant advantage over purely static compilation and optimization methods.

2.3 Run-time Binary Optimization

Similar to run-time specialization systems, run-time binary optimizers attempt to improve performance by transforming the program at run-time. In contrast to run-time specialization systems they do not require any user intervention and instead work by automatically performing some run-time code transformation, typically based on the results of a prior phase of observation, where run-time characteristics are collected either by executing the code in an interpreter (e.g., Dynamo [BDB00]) or by a hardware-based mechanism (e.g., Replay [PL01]). In these systems both the scope and aggressiveness of optimization is limited to keep run-time optimization costs low. Their advantages, on the other hand, are that they do not require any user intervention, handle dynamically changing code naturally, and do not require the availability of source code. The following sections describe two representative binary optimization systems and their main characteristics.

2.3.1 *Dynamo*

Dynamo is a run-time optimization system that optimizes binaries for the PA-8000 processor on the HP-UX operating system fully transparently, requiring neither specialized compilation or preparation of the binary nor any user annotations to drive the optimization process. Instead it uses run-time monitoring of the application's behavior to detect optimization opportunities.

Dynamo starts executing the application using its built-in PA-8000 instruction interpreter. During this execution mode, Dynamo uses a heuristic, called *most recently executed tail (MRET)* to pick a hot instruction sequence or *trace*, i.e., execution paths through the program that are frequently executed, so that any improvements to the code on that trace would likely result in a noticeable improvement in overall execution time. Once such a hot path is detected, the corresponding code, which may include branches and calls to library functions, is stored in a *fragment cache* (a software data structure that stores PA-8000 instructions) as a *fragment*, a single-entry, multi-exit, contiguous sequence of instructions.⁷

Eliminating branches and calls, which are very expensive on the PA-8000 architecture, are the first important step performed to optimize the code in the fragment cache. Then Dynamo optimizes the fragment in single forward and backward passes, performing redundant branch elimination, redundant load removal, and redundant assignment elimination. It also performs some standard optimizations, like copy and constant propagation, strength reduction, loop invariant code motion and loop unrolling.

To detect when traces are no longer hot, loops in the fragment are disallowed except for backward branches to the start of the trace. All other loop-back branches are treated as exits from the fragment cache and relinquish control back to the Dynamo system, which updates the counters it uses to keep track of hot code paths. When the heuristic determines that a trace is no longer hot, Dynamo reverts to interpreting the code to possibly detect new hot traces. To mitigate the possible slowdown from interpretation when no opportunities for trace formation are found, Dynamo “bails out” from interpreting an application when the ratio of time spent in interpreting the code over the time spent in native execution in the fragment cache exceeds a preset threshold. At that point, the

⁷A fragment is similar to a superblock [HMC⁺93], except that it is a dynamic instruction sequence, and can cross static procedure call and return boundaries.

original unmodified application binary is executed directly on the PA-8000 processor.

With the combination of Dynamo's optimizations and its bailout strategy, Dynamo is able to achieve an average speedup of 9% for SPEC95 benchmarks over the moderately optimized static binaries (+O2 switch in the HP compiler). When more aggressive and feedback-directed optimizations have been used to produce a program binary (+O4 switch and +P for profile feedback), Dynamo is generally not able to achieve any additional speedups, because feedback-directed optimizations are able to realize the most beneficial of Dynamo's transformations (removing call boundaries via inlining). The performance of the Dynamo-optimized +O2-binaries, however, equals the performance of statically +O4-optimized binaries (and is slightly better for the +O4 binaries without feedback directed optimization). Therefore, Dynamo can be used to transparently achieve the same level of performance as the more intrusive feedback-directed optimizations, which can be a considerable usability advantage in some cases.

2.3.2 *Replay*

Replay [PL01, FBC⁺01] is a project similar in spirit to Dynamo, as it also optimizes arbitrary binary code. However, it uses special hardware support to monitor program behavior and a hardware frame cache where code belonging to hot paths is stored and optimized. The prototype is realized as an architecture description simulated using the SimpleScalar 3.0 architecture simulation tool kit that "implements" the optimization engine in hardware. It performs various standard optimizations, including dead code removal, constant propagation and common sub-expression elimination. Fahs et al. [FBC⁺01] report an average improvement of 13% in execution time (measured as total number of cycles it takes to execute the applications in the simulator) on their benchmark suite (consisting of SPEC and some other applications) over statically optimized binaries. Since the results are based on simulations alone and there is potentially a significant amount of imprecision in architecture simulations [DBK01], it remains to be seen whether the Replay architecture would be able to produce significant speedups in practice.

2.4 Adaptive Run-time Compilation with JITs

Java [GJS96] has become the de facto standard programming language for Web Page designers. This has resulted in a lot of research [CL97, CLS00, ATCL⁺98, YMP⁺99, AFG⁺00] aimed at improving the performance of implementations of the Java Virtual Machine (JVMs), which is a platform-independent mechanism for executing Java programs distributed in the Java bytecode format – the preferred way of providing functionality to users without giving away software source code.

Many of the performance optimizations implemented in these JVMs strive to remove some overheads particular to the Java language and its execution model. Synchronization removal optimizations [Ruf00, ACSE99] are very important for achieving good execution times. Since they result in potentially very large benefits compared to the benefits achievable with more traditional compiler optimizations, it is generally advisable to perform them. To balance the potential benefit of such run-time optimizations with the run-time compilation cost, JVMs typically use an adaptive optimization mechanism, whose goal is to identify Java methods that warrant higher compilation cost because of their high execution frequencies.

Consequently, IBM's Jalapeño [AFG⁺00], the Intel Microprocessor Research Lab Virtual Machine (MRL JVM) [CLS00], and Sun's Hotspot [hot99] include an adaptive optimization mechanism that selectively (re-) compiles Java methods to improve program performance. The work of Arnold et al. [AFG⁺00] on Jalapeño is closest to the automatic optimization approach taken in this dissertation, since they use an explicit cost-benefit model to decide when dynamic recompilation of a Java method should take place. In contrast, the MRL VM and Hotspot use a more ad hoc heuristic and execution frequency thresholds to decide whether to recompile a method.

In the Jalapeño JVM all Java methods are compiled lazily to native code as they are executed using a fast compilation level (baseline) that performs little optimization. The JVM includes an *adaptive optimization system (AOS)* that uses program instrumentation, hardware performance monitors and VM instrumentation to produce profiling data about the application and VM execution to decide whether methods would perform better if recompiled at a higher optimization level. Two separate threads, called *organizer threads*, periodically process and analyze the raw data, separating the profile gathering task, which is resource constrained, from the data analysis. One of the threads, the *hot methods organizer*, processes method samples (execution counts) and generates hot method events,

which are enqueued in an organizer event queue, which is used by a separate *controller thread* to decide whether recompilation or other optimizations should be performed.

The controller uses a cost-benefit analysis to decide whether it would be profitable to recompile a method. For a method currently compiled at level i , where $i = 0 \dots 3$ (baseline and 3 successively more aggressive optimization levels), the control estimates T_i , the expected time the program will spend executing method m , if m is not recompiled; C_j , the cost of recompiling method m at optimization level j , for $i < j \leq 3$; and T_j , the expected time the program will spend executing method m in the future, if m is recompiled at level j . Then the controller decides to recompile m at level j if $C_j < T_i - T_j$, i.e., when the benefit of executing the more highly optimized code exceeds the cost of the recompilation. Since the time the program will continue to execute is unknown, the controller simply estimates that the program will continue to execute for as long as it has already run. The time the program will spend in method m at level i (T_i) is then computed by multiplying the fraction spent in method m so far (as indicated by the profile data) by the total estimated remaining program execution time. The execution time at the higher optimization level is estimated using the formula $T_j = T_i * S_i / S_j$ where S_i and S_j are preset speedup estimates versus the baseline execution times. The recompilation cost C_j is estimated to be proportional to the method size, where a (fixed) distinct multiplicative constant is used for each optimization level.

In their experiments, Arnold et al. evaluated the adaptive multi-level recompilation strategy versus a number of other configurations, including fixed optimization levels and adaptive recompilation with restricted maximum optimization levels. While for some programs a particular strategy did better than the adaptive scheme, no single strategy performed consistently better. Since higher-optimization levels pay off in long-running programs but produce worse performance for short-running applications, only the multi-level adaptive strategy was able to achieve consistently good performance for both classes of programs. Moreover, for every application in their benchmark suite, the adaptive multi-level strategy performance was close to the performance of the optimal strategy for that application, indicating that the implemented heuristic works well in practice.

Chapter 3

AN OVERVIEW OF THE CALPA SYSTEM

The previous two chapters have demonstrated the increasing need for and utility of run-time optimization to achieve good program performance. They have also presented various methods of achieving program speedups at run-time. The purpose of this chapter is to give an overview of the Calpa prototype system and its algorithms that are developed in this thesis. The following chapters (Chapter 4 and Chapter 5) will then drop from this bird’s-eye perspective to provide the details of how the Calpa prototype system achieves the goal of making dynamic program optimization via run-time specialization a fully automatic program optimization.

To compute annotations for an annotation-driven run-time specializer, Calpa combines program analysis and profile information. It models the costs and benefits of run-time specialization systems and uses the profile information to evaluate and select the best annotation from a set of annotation candidates identified by a static analysis. Calpa consists of two components, an instrumentation tool (henceforth called *Tumi*¹) and an annotation selector tool.

Tumi’s task is to provide the selector tool with profile information from an application execution that will aid it in making good annotation choices. It does this by automatically instrumenting the application to track the execution frequencies of basic blocks and log the values of variables at all definitions and uses. Tumi performs a pointer analysis to compute a conservative approximation of the set of variables referenced by each load and store instruction, which is used to track the execution frequencies of store instructions and the targets they are updating, a necessary step in assessing the costs of code cache invalidations, as explained in Section 2.1.3.

The task of the Calpa annotation selector is to identify good program annotations. It first identifies, for each instruction in potential dynamic regions, the minimal set of static variables that cause the instruction that uses them to be static (recall that static instructions are instructions that depend only on the values of specialized variables and data structures). In order to increase the number

¹Tumi is the name of the ritual knife used in the Calpa oracle ritual.

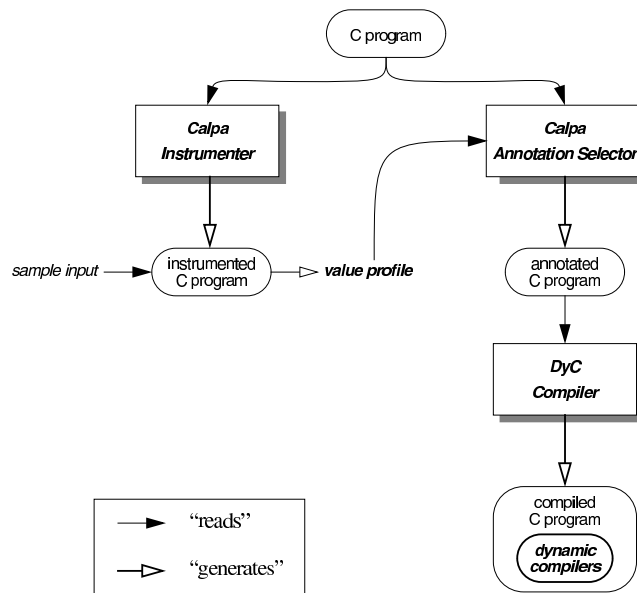


Figure 3.1: This graph shows the components of the Calpa system.

of simultaneously static computations, the selector tool combines the sets, and uses a cost-benefit model to evaluate the performance impact of each candidate set. The cost-benefit model models the costs of generating and caching code at run time and the benefits that result from code specialization and subsequent value-specific optimizations. It uses profile data to estimate the run time for each combination. If a new combination is predicted to achieve better program performance, it is retained as the current best choice. After all combinations have been considered, Calpa automatically generates the selected DyC annotations and outputs annotated C source code, which is then compiled with DyC. Figure 3.1 shows Calpa’s components and how they are used to optimize a program.

At the core of Calpa’s annotation selector is a cost-benefit model that predicts the effect of candidate annotations on the run time of a dynamically compiled application. The model’s cost function estimates the run-time overhead of dynamically generating and dispatching to specialized code. It is comprised of several sub-functions, all of whose parameters are derived from actual costs in DyC. Its cost function accounts for the cost of generating specialized code at run time by estimating the number of instructions that have to be generated, taking into account the effects

of code-expanding transformations, such as full loop unrolling. It also models the periodic code caching and dispatching costs, as well as the cost for checking whether respecialization is necessary due to code cache invalidations.

The benefit function predicts the execution-time savings when running the dynamically generated code. It takes into account the savings obtained by executing static instructions only once for each dynamic code version and executing the optimized, dynamically generated code instead of the original instructions. Since not all instructions eliminated from the dynamic instruction stream may contribute to performance on a wide-issue processor (like the Alpha 21164, on which DyC runs), instructions not on the critical path are ignored or their benefit is "downgraded" compared to instructions on the critical path. All instructions counted towards the benefit are weighted by their latency and the frequency of their execution to obtain an estimate of the overall execution time savings.

The following sections provide more information about each of Calpa's components. Section 3.1 provides an overview of Calpa's instrumentation tool Tumi. Section 3.2 discusses the annotation selector tool. An overview of the cost-benefit model is given in Section 3.3.

3.1 Tumi

To provide the necessary data for the cost-benefit analysis, Tumi instruments an application to collect information about program variables and program point execution frequencies during its execution. Tumi inserts code into the application to monitor and summarize three kinds of information: basic block frequencies (the number of executions, not execution time), variable definitions, and variable uses. When a variable or data structure is accessed via a pointer, Tumi uses the results of its pointer analysis to insert code that associates the pointer's value with a corresponding variable or data structure from its points-to set. This serves two purposes: first, it attributes the use or definition of a particular value to the variable or data structure that is defined or used; second, it identifies potential invalidation points of variables and data structures, so that Tumi can insert code to monitor the points' execution frequencies. To account for definitions that take place in library functions for which no source code is available, Calpa conservatively assumes that any variable or data structure that escapes to a library function is possibly modified by it.

An instrumented application logs values of definitions and uses as they occur, storing them as a

```

    i = 0                {}
L1:  if i >= size goto L2 {i,size}
    uelem = u[i]        {i,u[]}
    velem = v[i]        {i,v[]}
    t = uelem * velem   {i,u[],v[]}
    sum = sum + t       {i,sum,u[],v[]}
    i = i + 1           {i}
    goto L1             {}
L2:

```

Figure 3.2: An example illustrating how candidate static variable sets (CSV sets) are computed. The example computes the dot-product of two integer vectors u and v . The third column, shows the CSV sets for each instruction. For example, The CSV set for the comparison between i and $size$ is $\{i, size\}$, because the values of both i and $size$ have to be known to know the outcome of the comparison.

per-invocation histogram of values and their number of occurrences. Each value-occurrence pair is tagged by the invocation number of the procedure in which it was produced (to obtain more accurate cost information for loop indices that are dependent on procedure parameters (see section 3.1.2)). Data is kept in a hash table in memory up to a user-specifiable maximum memory usage. When necessary, the values for the least recently used variables or data structures are written out to disk to make room for new values. To limit overall log size, monitoring is dynamically disabled for variables and data structures for which more than a user-specified number of distinct values have been recorded. (Because of dynamic compilation overhead, it is unlikely that these data will be run-time constants.) Before the application terminates, the profile data is written to a log file. How applications are instrumented, what values were used for the configurable parameters, and what information is collected, is explained in detail in Chapter 4.

3.2 Calpa's Annotation Selector Tool

3.2.1 Candidate Static Variable and Candidate Division Sets

The first task in Calpa's annotation selector tool is the computation of the basic sets of variables which, if annotated as static, cause the operations that use them to become static. We call these sets

candidate static variable (CSV) sets. The dot-product code fragment shown in Figure 3.2 illustrates CSV sets; next to each statement is the CSV set that makes the statement static. In general, the CSV set for an instruction is the set of variables used as source operands to the instruction, or the empty set for an instruction without variable operands. For example, the comparison between `i` and `size` is static if and only if both variables are static; hence the statement’s CSV set is $\{i, \text{size}\}$. For memory loads, the contents of the memory must also be static for the load to be static. For example, the load of `u[i]` is static if and only if both `i` and the array `u` is static, so the statement’s CSV set is $\{i, u[]\}$, where `u[]` represents both the address `u` and its contents. Since DyC’s binding time analysis will determine that instructions are static if their source operands are computed by unique static instructions, Calpa treats operands with a single reaching definition specially. In this case, the static variables for the operand are those that correspond to the CSV set of the reaching definition (instruction). For example, in the statement `t=uelem*velem`, since `uelem` and `velem` each have exactly one definition, the CSV set for this statement is $\{i, u[]\} \cup \{i, v[]\} = \{i, u[], v[]\}$.

Calpa can build bigger dynamic regions with greater degrees of run-time optimization by merging CSV sets into larger sets of variables, called *candidate divisions (CDs)*. The set of static instructions for a CD is the union of the instructions whose CSV sets are subsets of the CD. In general, if there are n CSV sets, there are 2^n ways to combine them. However, since CSV sets tend to overlap, not all combinations will produce different sets. For instance, if we combine $\{u[], i\}$ and $\{u[], v[], i\}$, we obtain the same set as when we combine $\{v[], i\}$ and $\{u[], v[], i\}$. In the dot-product example above, we get the following set of possible CDs (called the CD set):

$$\{\{\}, \{i\}, \{i, \text{size}\}, \{i, u[]\}, \{i, v[]\}, \{i, u[], v[]\}, \{i, \text{size}, u[]\}, \{i, \text{size}, v[]\}, \{i, \text{size}, u[], v[]\}, \{i, \text{sum}, u[], v[]\}, \{i, \text{size}, \text{sum}, u[], v[]\}\}.$$

The CD set captures all possible combinations of annotated variables that result in distinct collections of static instructions. Other combinations of static variables never have to be considered, since they will produce the same results as some CD already in the CD set. For instance, $\{\text{sum}, i\}$ leads to the same instructions being static as $\{i\}$, which is already included in the CD set. Therefore, it suffices to evaluate only the CDs in the CD set to estimate the potential benefits that specialization can achieve, rather than evaluating all 2^k possible combinations of k variables; for instance, in the dot-product example, the CD set size is only 11, whereas there are 32 sets in the power set of the five variables.

Calpa's Search Strategy

For each procedure, Calpa first computes the CSV sets of all instructions. The CD set is then enumerated, using a gradient search strategy. Variables are first sorted by their number of distinct values, obtained from the profile. Then, beginning with the CSV sets that contain the more invariant variables, new candidate divisions are generated and their run-time benefit and cost estimated. The search process remembers the best candidate division choice and its estimated speedup. The search terminates if all choices have been enumerated (feasible for small applications, such as `dotproduct`), a set time quota has expired, or the gradient of improvement over the best choice so far drops below a preset threshold.

3.3 The Cost-Benefit Model

As outlined in Section 2.1.1, the DyC system incurs three different costs when dynamically compiling a program: a one-time specialization cost for producing a dynamically compiled region for particular static values, periodic dispatching costs which are paid each time a section of dynamically generated code is executed, and an invalidation check cost for variables and data structures for which invalidation-based caching is used.

3.3.1 Specialization Cost

The specialization cost is roughly proportional to the number of dynamic instructions generated for all code versions. Therefore, when computing its specialization cost estimate, Calpa uses the profile data to estimate the number of different code versions that will be produced. Its estimate of the total number of instructions generated at run time is proportional to the product of the number of specializations and the number of dynamic instructions generated for each.

When loops are completely unrolled, Calpa constrains code blowup by guarding the loop-unrolling annotation with a condition that is generated by the specialization cost model. It generates the expression $d * range(i) < limit$, where $range(i)$ is the actual number of different values observed for the loop induction variable i and $limit$ is a predetermined constant (modifiable by a Calpa command-line argument, d a multiplicative factor corresponding to the loop body size) to guard the unrolling. Hence, the loop unrolling will only be performed if the number of values of i

remains below the threshold of *limit* generated instructions. If the preset limit is too large (causing performance degradation due to cache effects) or too conservative, it can be changed (for the results reported in Chapter 6, a limit of 32K instructions was used). Automating this process, for instance, by observing the resulting cache behavior of the generated application and dynamically adjusting the limit, is the subject of future work.

3.3.2 Code Caching Cost

Using a simple binding-time analysis, Calpa identifies program points at which cache lookups for dynamically generated code are necessary. However, a cost need not be incurred at all cache lookup points. For example, DyC includes a caching policy (`cache_one_unchecked`) to specify that it is safe to omit a cache lookup for invariant static variables – in this case, specialized code is generated for the variable’s first encountered value, cached and used thereafter without a cache lookup. Alternatively, `cache_all_unchecked` is used for variables that step through a small sequence of values. When the cache lookup point is encountered, a new version of code is generated without a lookup. This policy is useful for complete (single-way) loop unrolling, where the induction variable is monotonically incremented.

Calpa uses profile information as a hint to decide whether an unchecked policy should be used; if a variable has only one value, it becomes a candidate for `cache_one_unchecked`; if it has only a few values and the variable is a loop index, then it may be suitable for `cache_all_unchecked`. To guarantee the safety of the `cache_one_unchecked` policy, the invalidation point analysis will ensure that the variable, once defined, will not be redefined. The `cache_all_unchecked` policy is always safe when specialization is performed on demand. Calpa can choose eager specialization if it can determine that the loop termination condition is static; if it cannot (e.g., for multi-way loop unrolling), it chooses the safe, lazy option.

When Calpa uses the more costly `cache_all` policy (which does a cache lookup), it assesses a per-lookup fee to allocate memory for a cache key and do a hash table lookup (85 cycles in the model), plus a small additional cost to construct the cache key from the set of variables whose values must be checked in order to dispatch to the correct version of code (5 cycles per variable). Since the caching cost is paid each time the cache check point is executed, it is multiplied by the execution

frequency of the point, obtained from the profile.

Invalidation Cost

When Calpa uses invalidation-based caching for a variable or data structure, it first computes its invalidation points. The cost of each invalidation point is the product of its execution frequency and some fixed cost. In the model, a value of 100 cycles is used, which is sufficiently large to account for the cost of a moderately complex guard condition that checks whether an address is within a certain range. The sum over all invalidation points is the variable or data structure's total invalidation cost.

3.3.3 The Benefit Model

Calpa's benefit estimation identifies all instructions that are made static by the particular CD being evaluated. For each procedure and choice of static variables, it runs a simple and fast binding time analysis (a simplified version of DyC's BTA [GMP⁺00b] described in Section 5.4.1) to compute the derived static variables and the division of variables and instructions into static or dynamic. (The BTA also computes the program points that require dynamically-generated-code cache lookups, important for the estimation of caching costs, described above.) In the current implementation, the BTA assumes that the variables are specialized throughout the procedure. In some cases, specializing a smaller region of the procedure may result in the same number of static instructions, but with a smaller specialization or caching cost. In such a case, the current implementation will tend to overestimate these costs. The benefit of specializing a procedure for a particular choice of static variables is computed by estimating the number of saved cycles that will result from their instructions becoming static. To avoid overestimating the benefits from specializing instructions that do not contribute to the execution time of the program on a wide-issue processor, for instance, instructions off the critical path, Calpa only considers qualifying instructions and weights their contribution. How instructions qualify and how they are weighted is explained in detail in Section 5.4.5.

Chapter 4

INSTRUMENTATION INFRASTRUCTURE

To estimate the cost or benefit of particular annotation choices, Calpa requires information about the run-time values of variables and data structures as well as the execution frequencies of instructions. To generate the profile information for Calpa, I designed *Tumi*, a profiling tool based on program instrumentation, and implemented it with the Suif Compiler Infrastructure, whose intermediate representation allows for an easy mapping between C source-level and IR-level information. *Tumi* instruments an application to capture basic block execution frequencies, the values of all definitions and uses of variables and data structures, and the frequency of invalidations, i.e., the frequency of changes to variables or data structures at pointer stores.

Traditional frequency profiles can be obtained either by sampling or by program instrumentation. In sampling-based profiling the processor's program counter is sampled at regular intervals (typically by interrupting the program using the processor's timer interrupt), and its value is used to find out which procedure is currently executed. In contrast, instrumentation-based approaches add instrumentation code to the program to record, for instance, how often each basic block is executed. The advantage of sampling-based approaches is that they typically result in less program slowdown than program instrumentation. On the downside, however, the information that can be obtained via sampling is typically more coarse-grained than the data that can be gathered by instrumentation. To gather the detailed information for Calpa's cost-benefit model, I therefore chose to use program instrumentation rather than a sampling-based approach. In addition, since DyC's annotations are at the C source code level, Calpa needs information at the same level. Consequently, program instrumentation at the source code level, or at an intermediate representation level that preserves both source level names and line numbers is required.

The rest of this chapter describes *Tumi* in more detail. Section 4.1 provides an overview of the implementation of the instrumentation tool. Sections 4.2 and 4.3 respectively discuss how frequency and value profile information is obtained. Section 4.4 discusses the data capturing routines

provided by the instrumentation library and the implementation of the value profile cache. Section 4.5 discusses the impact of instrumentation on run time and executable size, and also presents a performance optimization that is used to mitigate the slowdown. Finally, Sections 4.6 and Section 4.7 discuss related value profiling work and possible improvements to Tumi.

4.1 Instrumentation Overview

Since the Suif compiler infrastructure [WFW⁺94] provides both a C front- and backend and preserves C source level information throughout the compilation pipeline, it was used to implement the Calpa instrumentation tool. In addition, I used the control and data flow analysis libraries [HY97] that are part of the Machine Suif version of the Suif compiler. These libraries provide support for the construction and transformation of the control flow graph (CFG) of the program, and also implement some standard basic data flow analyses, such as reaching definitions. Suif and Machine Suif version 1.1.2 were used in the implementation.

The steps performed to instrument an application are outlined in Figure 4.1. The application is first converted to the Suif IR using the Suif frontend tools. If the application consists of multiple files, their symbol tables are made consistent by running the `linksuif` tool, which ensures that identical symbols in different files are assigned the same internal representation i.e., the same symbol id numbers. Then, a static points-to analysis is run on those files, which adds information about pointer targets to the IR. On this IR the Tumi instrumentation tool itself is run and adds instrumentation code. The resulting IR is converted back to C and compiled with a standard C compiler; in my experiments I used the Compaq vendor compiler (`cc`) whenever possible, since it typically runs faster than the GNU `gcc` compiler and produces code of slightly better quality. To produce an executable, the resulting object files are linked together with the Tumi instrumentation library that implements data capturing and other utility routines (described in Section 4.4.1). The executable can then be run in the same way as the original application.

In the following sections, I'll describe more details of capturing frequency and value profile information, and the supporting instrumentation library routines.

1. Convert C source code to SUIF IR; make the symbol tables consistent, if necessary, with the `linksuif` tool.
2. Run the points-to analysis on the IR and annotate the IR with points-to targets for all pointer-valued expressions, including calls through function pointers.
3. Add instrumentation code to each procedure to capture frequency and value profiles. Add a call to the runtime initialization routine in procedure `main` and a call to save value profiles at the end of `main` and before all calls to `exit`. For calls that may call `exit`, conditionally write profiles, but only when the program in fact exits.
4. Convert the instrumented IR back to C and compile with `cc` or `gcc`. Link with `libinstr.a`, the instrumentation runtime library, which implements data capturing and other bookkeeping routines for the profiles.

Figure 4.1: Steps performed to instrument an application with Tumi.

4.2 Frequency Profile Information

Calpa uses a frequency profile, in the form of a record of how often each basic block was executed during the profiling run, to weight the costs and benefits in its cost-benefit model. Capturing this frequency profile information is straightforward with Machine Suif's CFG library. For each procedure in the program to be instrumented, the CFG is constructed. A global array variable with elements of type `unsigned long` is created for each procedure, with the number of elements equal to the number of basic blocks of the procedure. At the top of each basic block, code is inserted that increments the array element corresponding to the basic block. The contents of all array variables are saved to disk before the program terminates.¹ Since the `unsigned long` data type is 64 bits long on the Alpha architecture, the basic block counters will wrap around only after $2^{64} \approx 1.8 * 10^{19}$ executions; therefore no overflow check was implemented.²

Instead of adding a counter to every basic block, a spanning tree algorithm could be used to place a minimal number of counters in the program and thereby mitigate the slowdown from frequency profile collection [BL94]. However, since most of the slowdown is incurred by value profiling, any improvements to frequency profiling will not have a significant impact on overall performance of an instrumented application. Therefore, only the straightforward counter-per-basic-block approach was implemented in Tumi.

4.3 Value Profile Information

To determine various parameters in its cost-benefit model, Calpa uses information about the values of variables and data structures during program execution. To obtain these values, the value profile records for each use or definition of a variable or data structure the value that is used or assigned, respectively. Depending on the data type of the variable for which we want to gather value profile information, slightly different instrumentation actions and run-time bookkeeping are required. Moreover, when data is accessed through an aliased name, an additional step associating the mem-

¹A call to an appropriate instrumentation library routine is inserted before each call to `exit` in the program to ensure the counters are written out before the program exits.

²Assuming that it takes one cycle to increment a counter, $1.8 * 10^{19}$ cycles on a 500MHz Alpha processor correspond to over 1000 years of execution time.

ory address with a unique name for the accessed variable may be necessary. In C, aliases arise only through pointers and in union data types, so I will first discuss the most common case of aliases through pointers in Section 4.3.1. Then, I'll discuss the particulars of the different data types; Section 4.3.2 discusses instrumentation and run-time bookkeeping for scalars, and Section 4.3.3 looks at array, structure and union types.

4.3.1 Matching Pointer Addresses with Compile Time Names

To associate the values that are assigned or used in pointer accesses with the corresponding variable or data structure, the pointer's run-time address has to be mapped to a compile-time name corresponding to the accessed variable or – in the case of heap-allocated data – memory allocation site. During the profiling run, this task is performed by an *address matching routine*, which matches a pointer address against the addresses of all objects that the pointer might possibly point to. The set of these potential pointer targets is computed before profiling by a static points-to analysis.

By default, Tumi uses the One Level Flow algorithm [Das00] (OLF) for its points-to analysis; other implemented choices are Steensgaard's [Ste96b] algorithm, or an extension thereof proposed by Shapiro and Horwitz [SH97b].³ These algorithms create one distinct compile-time representative for each memory allocation site, i.e., data structures allocated at different allocation sites are distinguished, while data structures allocated at the same malloc site are not. Consequently, multiple run-time addresses may correspond to a single compile-time representative. The same is true for local variables, which may be allocated at different stack addresses in different invocations. For global and static variables, on the other hand, the run-time addresses will not change during execution.

Therefore, depending on the compile-time representative, different instrumentation and run-time actions are necessary to map a run-time address to a compile-time name. The following sections present the *target object descriptor* data structure, which is used in address matching, and then discuss the instrumentation and run-time actions that are performed for global, static, and local variables, and for data structures allocated on the heap.

³In addition, Ghiya & Hendren's algorithm [GH96] can be run for pointers to heap-based data structures, after one of the previously mentioned points-to algorithms has been run.

Target Object Descriptors

To match run-time addresses with compile-time names, Tumi creates a target object descriptor (TOD) for each object in the static points-to set that the static pointer analysis computed for a pointer dereference in the program. Each TOD represents an association of a compile-time variable name or malloc site id with its run-time address (or addresses in the case of malloc sites). For each static points-to set that was created by the points-to analysis, Tumi creates an array of TODs, containing the associations for all the variables or malloc-sites present in the set; i.e., the array of TODs is used to implement a set of mappings from compile-time names to run-time addresses. This collection of TODs is used by the address matching routine to find the compile-time representative corresponding to a pointer address, by comparing the pointer address to the addresses stored in the TODs and selecting the TOD that contains a matching address.

The TOD data structure definition is shown in Figure 4.2. Field `kind` indicates whether the object represented by the TOD is a variable or a malloc site. Symbol names, including variable and procedure names, are uniquely identified in the Suif IR by an id triple of the form (`symbol-id`, `procedure-id`, `file-id`). Tumi uses a triple (`instruction-id`, `procedure-id`, `file-id`) to uniquely identify malloc sites. These id numbers are stored in fields `id{1,2,3}` in the TOD structure. `deref_level` is set to `address` (value = 0) if field `addr` contains the run-time address of the object the TOD represents, `indirect` (value = 1), if it is the address of an address variable where the object address can be found at run time (as explained below). The last field, `hook_counter` is used for performance optimization and is explained in Section 4.5.2.

Global Variables

In the C language, the addresses of global variables are fixed after the program has been loaded. Therefore, for a global variable `v`, Tumi simply initializes the TOD array with `v`'s fixed address:

```
object_addrs[] = {..., {&v, var, id1, id2, id3, address, 0}, ...}.
```

A tag address is used to indicate that the address contained in the `addr` field is the address of variable `v` itself (as compared to the cases described below).

```

typedef struct __tod_s {
    char** addr; // address
    int kind; // heap or var
    unsigned int id1; // variable or malloc-site id-triple
    unsigned int id2;
    int id3;
    int deref_level; // what is address? (address = 0 address of object, indirect = 1, address variable)
    int hook_counter; // used in unhooking of instrumentation calls
} tod_t;

```

Figure 4.2: For each object in a static points to set, at run-time a target object descriptor (TOD) is used to maintain its run-time address. Field `kind` identifies if the object is a variable or malloc-site, `id{1,2,3}` is the variable or malloc-site id triple. `deref_level` is 0 if `addr` contains the run-time address of the object, or 1 if it is the address of an address variable where the object address can be found.

Static Variables

Like global variables the addresses of static variables are fixed at run-time. However, due to scope rules, a static variable may not be visible at a program point that uses a pointer that points to it. Therefore, for each static variable `s`, Tumi creates a global variable `s_addr` that is initialized in the scope of `s` to hold the address of `s`. `&s_addr` is placed in the object address array:

```
object_addrs[] = {..., {&s_addr, var, id1, id2, id3, indirect, 0}, ...}.
```

The tag `indirect` indicates that `&s_addr`⁴ is not the object address but a pointer to the address variable that contains the address of `s`.

Local Variables

For each invocation of a procedure `foo`, the locals defined in it are likely to be allocated at different stack addresses. Therefore, for each local variable `l`, Tumi creates a global address variable `l_addr` that is initialized to hold `l`'s address by code inserted at the header of `foo`:

⁴The name `s_addr` is used only for the example. In reality, a new distinct name is created so that there never is a name clash for multiple static variables `s` in different scopes.

```

foo() {
  l_addr = &l;
  ...
}

```

`&l_addr`⁵ is then placed in the object address array:

`object_addrs[] = {..., {&l_addr, var, id1, id2, id3, indirect, 0}, ...}`, again with the `indirect` tag, since the object address is accessed indirectly via the address variable `l_addr`.

When the static points-to set indicates that a pointer used in procedure `p` may point to one of procedure `p`'s own local variables, Tumi passes the address of the local directly to the address matching routine (possible because the local variable is visible in `p`), instead of creating an address variable to hold the address of the local. This saves the overhead of assigning the address of the local variable to the address variable. For local variables that do require global address variables, Tumi creates one per local variable and sets its value at procedure entry. However, for recursive procedures multiple activations⁶ may be active at the same time, each with its own copy of the local variable, all of which may be accessible (via pointers). To handle this case, instead of creating a single address variable, a list data structure could be created to have access to the addresses of all activations. Fortunately, the applications targeted in this thesis did not require this (more expensive) mechanism, so it was not implemented.

Heap-allocated Data Structures

To be able to identify that an address was allocated at a particular `malloc` site, Tumi instruments each call to a memory-allocating function. For example, `p = malloc(n)` would be changed to:

```

temp = malloc(n);
__tumi_update_malloc(temp, id1, id2, id3);
p = temp;

```

When invoked at run time, `__tumi_update_malloc` hashes the `malloc` site id (the triple `(id1, id2,`

⁵Again, `l_addr` is just a name used for the example; in practice a fresh unique temporary name is generated to avoid name clashes.

⁶An *activation* refers to a stack frame for the execution of a procedure. For recursive procedures, several of its activations may be live at the same time. [ASU86]

id3)) to the root of a red-black tree maintained for each malloc site. The address is then added to the red-black tree. Using a red-black tree enables fast (logarithmic time in the number of memory blocks allocated at a site) matching of an address with the addresses allocated at a malloc site. In a TOD array the malloc site is represented by its unique id triple and tagged with a heap tag to distinguish it from variables: `object_addrs[] = {..., {0, heap, id1, id2, id3, 0, 0}, ...}`. The address matching routine hashes the malloc site id triple (id1, id2, id3) to obtain the root of the corresponding red-black tree, and a binary search is used to find the closest address. The contents of the `addr` and `deref_level` fields are not used in this case, so they are simply set to zero.

4.3.2 Value-profiling Scalars

Regardless of whether a variable or data structure is accessed directly or via a pointer, once it has been identified, the value for it needs to be recorded. This section discusses how this is done for scalars; Section 4.3.3 then looks at the instrumentation to capture values for composite data types.

Scalars Accessed by Name

Capturing values for scalars accessed by name is straightforward. The use of a variable is prefaced by a call to a data capturing routine that is specific to the type of the variable. This routine is passed the current value of the variable and the variable's unique id triple. A definition of a variable is broken up into three parts. First, the expression that computes the value that is assigned is computed into a fresh temporary of the corresponding type. Then a call to a data capturing routine that captures the temporary's value and records it for the defined variable is inserted. Then the original definition is replaced by an assignment from the temporary to the variable.

Figure 4.3 shows an example of the performed transformation. The uses of variables `x` and `y` in the multiplication are preceded by calls to the data capturing routine `__calpa_rt_update_int_var_def_or_use`. It takes six arguments: the first is a perfect hash code⁷ (of the id triple for which the value is recorded) that enables fast access to the list of values that

⁷The perfect hash code is constructed during instrumentation and used at run-time to index into a hash table. For scalar variables there will never be a collision at run time. However, accesses to any field of a structure or union and accesses to any element of an array, share the same index. Their offset from the base address is used to associate values with the particular field or array element (described in Section 4.3.3).

Original C code:

```
int foo(int x, int y) {
    int z;
    z = x * y;
    return z;
}
```

Instrumented C code:

```
int foo(int x, int y) {
    int z;
    int __calpa_mgt_temp_defs0;

    __calpa_rt_update_int_var_def_or_use(1u, 1, 1610612737u, 1u, -1, x);
    __calpa_rt_update_int_var_def_or_use(2u, 1, 1610612738u, 1u, -1, y);
    __calpa_mgt_temp_defs0 = x * y;
    __calpa_rt_update_int_var_def_or_use(0u, 0, 1610612739u, 1u, -1, __calpa_mgt_temp_defs0);
    z = __calpa_mgt_temp_defs0;
    __calpa_rt_update_int_var_def_or_use(3u, 1, 1610612739u, 1u, -1, z);
    return z;
}
```

Figure 4.3: Example illustrating instrumentation for directly accessed scalar variables. For easier readability, the effects of instrumentation are shown in C code, i.e., after converting the instrumented code back to C from the Suif IR, on which all transformation is performed.

have already been recorded for a particular variable; definitions and uses are recorded separately, so the hash code for variable z is different for recording the use and definition. The second argument identifies whether the variable is used (argument = 1) or defined (argument = 0). Arguments 3 – 5 are the symbol id, procedure id, and file id that uniquely identify the variable. The final argument is the value that is being recorded for the variable. The value used in the definition of variable z is captured by computing the value into a freshly created temporary (in the example named `__calpa_mgt_temp_defs0`), then calling the data capturing routine, and performing the original assignment.

Original C code:

```

int g;

void foo(int* p, int v) {
    int foo_l;
    *p = v;
    p = &foo_l;
    *p = v;
}

int main() {
    int l;
    foo(&l, 0);
    foo(&g, 0);
}

```

Figure 4.4: Example program to show instrumentation for scalar variables accessed via pointers. Procedure `foo` is called with `p` pointing to local variable `l` of procedure `main`, and to global variable `g`. The instrumented version is shown in Figure 4.5.

Value-profiling Scalars Accessed via Pointers

For accesses to scalars via pointers, the pointer address and the TODs representing the points-to set of the pointer are necessary for the address matching step. Therefore, Tumi inserts a call to a data capture routine that passes the value used or assigned, and additionally the pointer address and the base address of an array of TODs, which is constructed as explained in Section 4.3.1. Figures 4.4 and 4.5 show an example illustrating these steps. For the program shown 4.4, the static pointer analysis computes `l`, `g`, and `foo_l` as the points-to set for the dereference of pointer `p` in procedure `foo`. Since `l` is a local variable of procedure `main`, an address variable is created in the instrumented procedure (`__calpa_addr_var_1610612737_3_1` in Figure 4.5) and initialized to hold `l`'s address at the start of procedure `main`. The address of this address variable and the address of the global `g` are put into the TOD for `l` and `g`, respectively, and the two TODs are put into a TOD array (`__calpa_temp_var1`) that is used by the address matching routine.

A call to the instrumentation library function `__calpa_rt_update_int_mem_def_or_use` is used

Instrumented C code:

```

int g;
void *__calpa_addr_var_1610612737_3_1; /* address variable for main:l*/
struct __tod_s __calpa_temp_var1[2] = {
    { (char**) &g, 0, 1u, 0u, -1, 0, 0 },
    { (char**) &__calpa_addr_var_1610612737_3_1, 0, 1610612737u, 3u, -1, 1, 0 }
}; /* array containing the target object descriptors for pointer p */

void foo(int *p, int v) {
    int foo_l;
    int *__calpa_mgt_temp_defs0;
    int __calpa_mgt_temp_defs1;
    __calpa_mgt_temp_defs1 = v;
    __calpa_mgt_temp_defs0 = p;
    __calpa_rt_update_int_mem_def_or_use(1u, 0, __calpa_mgt_temp_defs0,
        __calpa_mgt_temp_defs1, 1610612737u, 2u, -1,
        __calpa_temp_var1, 2, 7, 2u, 2u, 0, &foo_l, 1610612739u, 2u, -1);
    *__calpa_mgt_temp_defs0 = __calpa_mgt_temp_defs1;
    __calpa_mgt_temp_defs0 = &foo_l;
    p = __calpa_mgt_temp_defs0;
    __calpa_mgt_temp_defs1 = v;
    __calpa_mgt_temp_defs0 = p;
    __calpa_rt_update_int_mem_def_or_use(1u, 0,
        __calpa_mgt_temp_defs0, __calpa_mgt_temp_defs1, 1610612737u, 2u, -1,
        __calpa_temp_var1, 2, 7, 6u, 2u, 0, &foo_l, 1610612739u, 2u, -1);
    *__calpa_mgt_temp_defs0 = __calpa_mgt_temp_defs1;
}

extern int main() {
    int l;
    __calpa_rt_initialize(); // initialize run-time value cache data structures
    __calpa_addr_var_1610612737_3_1 = &l;
    foo(&l, 0);
    foo(&g, 0);
    __calpa_rt_save_bb_counters(); // save frequency and value profile data
}

```

Figure 4.5: Example illustrating instrumentation for scalar variables accessed via pointers. `__calpa_temp_var1` contains the two target object descriptors for pointer `p` containing the address of global `g` and the address variable that holds the address of local variable `l` at run time.

to capture the value defined in the `*p` dereference. The first two arguments are the hash code and definition/use flag; the third argument contains the value of the pointer and the fourth argument the value that is assigned or used. The next three arguments uniquely identify the pointer used in the memory access, followed by the pointer to the base address of the TOD array (`--calpa_temp_var1`), followed by the array size (two in the example).

The example also illustrates how TODs are passed on the stack when a pointer may point to a local variable of the instrumented procedure. As mentioned in Section 4.3.1, addresses of local variables of the same procedure (`foo_1`, in the example) are passed as separate arguments to the matching routine rather than creating a global address variable for them and putting a TOD entry in the TOD array. To indicate how many such descriptors are passed on the stack, the first argument after the TOD array size indicates the number of arguments that follow. For each target object whose address and description is passed on the stack, a 5-tuple of integers is passed: an identification whether the object is a variable, the object address,⁸ and the unique id triple of the variable.⁸

Furthermore, a pair that uniquely identifies the definition site is passed before any TOD for definitions. Hence, in the example, a total of seven arguments are passed: the first two, (2, 2), identify the definition site, the following five are the TOD for local variable `foo_1`.

4.3.3 Arrays, Structures and Unions

Unlike scalars, arrays, structures and unions comprise multiple data items. Tumi generates value profile information that distinguishes different data items of the same array, structure or union using a (`<variable name>`, `offset`) pair, where `offset` uniquely identifies which data item of the composite data structure is referenced; `offset` is simply the offset of the data item from the base address of the structure, union or array. For union data structures, this also resolves the alias problem, since distinct field names that are aliased have the same offset from the base address of the union.

Arrays, structures and unions in C enable the definition of arbitrarily complex data types. However, the instrumentation library can only provide a fixed number of data capture routines which are specific to the particular data type whose value is being recorded. Tumi solves this problem by

⁸In the current implementation, malloc site descriptors are never passed in this way, so the argument indicating whether a variable or a malloc site id is passed, is actually redundant.

recursively breaking up a complex data type into its element types until it arrives at the fundamental scalar data types. An example of this is shown in Figure 4.6. The use of structure x in $foo(x)$, is broken up into uses of each of the structure's fields; in the case of nested structure definitions, this process would continue recursively until arriving at uses of scalar data types. Then the values for each field are captured with the corresponding data capturing routine, for instance, structure field v is captured using `__calpa_rt_update_char_mem_def_or_use`.

In Suif IR all accesses to structure or union fields are represented as memory accesses, so that the two distinct field access operators `.` and `->` are represented by the same IR. Consequently, capturing the values of structures and structure fields (and unions and union fields) is achieved by passing the address of the structure field (in the example for $x.v$ this address is computed into `__calpa_mgt_temp_uses2`) and the field value. Tumi creates a singleton points-to set for each field access that contains only the name of the structure or union variable. The base address of the structure or union variable is also passed to the instrumentation routine, along with the unique id triple that identifies the structure variable.

Array accesses are handled in a similar way. Recursively defined array types are broken down until data accesses of scalar types are reached. The address of the element accessed and the base address of the array are passed to the data capturing routine and the value used or assigned is associated with the corresponding offset from the base address.

4.4 Instrumentation Library and Runtime

Tumi's instrumentation library implements the data capturing routines, which are called from the instrumented application. The instrumentation libraries share a common runtime, which implements the hash table that contains the profiled values, and some other utility routines, like value cache initialization and input/output routines which read and write profile data from/to disk.

4.4.1 Instrumentation Library

The instrumentation library implements the data capturing routines which are inserted by Tumi into the application code to capture values of variables during execution. For each scalar data type two routines are provided: one for variables that are referenced directly, another for accesses through a

Original C code:

```

typedef struct {
    int v;
    char ch;
    int* p;
} stype;
void foo(stype s) {
    /* .. */
}
int main() {
    stype x;
    foo(x);
}

```

Instrumented C code:

```

int main() {
    stype x;
    int __calpa_mgt_temp_uses1, *__calpa_mgt_temp_uses2;
    char __calpa_mgt_temp_uses3, *__calpa_mgt_temp_uses4;
    int **__calpa_mgt_temp_uses5;
    __calpa_mgt_temp_uses1 = x.v;
    __calpa_mgt_temp_uses2 = &x.v;
    __calpa_rt_update_int_mem_def_or_use(0u, 1, __calpa_mgt_temp_uses2, __calpa_mgt_temp_uses1,
        1610612737u, 64u, -1, (void *)0u1, 0, 5, 0, &x, 1610612737u, 64u, -1);
    __calpa_mgt_temp_uses3 = x.ch;
    __calpa_mgt_temp_uses4 = &x.ch;
    __calpa_rt_update_char_mem_def_or_use(0u, 1, __calpa_mgt_temp_uses4, __calpa_mgt_temp_uses3,
        1610612737u, 64u, -1, (void *)0u1, 0, 5, 0, &x, 1610612737u, 64u, -1);
    __calpa_mgt_temp_uses2 = x.p;
    __calpa_mgt_temp_uses5 = &x.p;
    __calpa_rt_update_ptr_mem_def_or_use(0u, 1, __calpa_mgt_temp_uses5, __calpa_mgt_temp_uses2,
        1610612737u, 64u, -1, (void *)0u1, 0, 5, 0, &x, 1610612737u, 64u, -1);
    foo(x);
}

```

Figure 4.6: Example illustrating the instrumentation of structures and unions.

pointer. The procedure prototypes for all Tumi data capturing routines can be found in Appendix A.

Capturing values for direct accesses is straightforward. Each data capturing routine puts the value into a value data structure which consists of two fields: a tag field that identifies the data type of the captured value, and a value field, whose type is a union type of all scalar data types. An example showing the routine for the integer data types is shown in Figure 4.7. First, a tag for the hash table lookup is constructed from the id triple, the update type (definition or use), and the perfect hash code created during instrumentation. The integer value is then stored into the value data structure and the data type tag is set to `rt_type_int` to indicate an integer data item. Then the main value cache update routine `update_cache_entry` passes the tag, the value and a flag, that indicates whether the call to this data capture routine should be *unhooked*. Unhooking is a performance optimization, discussed in Section 4.5.2.

Capturing values for accesses via pointers performs the same steps as required for direct accesses. First, however, the pointer address needs to be matched with the corresponding variable or data structure to attribute the captured value correctly. Figure 4.7 shows the steps performed to capture integer values with procedure `__calpa_update_int_mem_def_or_use`. As a first step, if the update is for a definition, the two arguments that identify the invalidation site are read off the call stack. Then the address matching routine `try_match_address` is invoked, which returns in `result` the id triple of the target object that matches. For definitions, a call to `__calpa_update_ips` then updates the invalidation point statistics, followed by the construction of the tag for the hash table lookup as in the direct access case. The remaining code is for the unhooking optimization, discussed in Section 4.5.2.

4.4.2 Instrumentation Runtime

The instrumentation runtime implements the `update_cache_entry` routine, which is called from all data capturing routines to possibly add a new value to the run-time value cache. It adds the value provided by the data capturing routine to the values already profiled for a variable or data structure indicated in the `tag` argument, as shown in the example in Figure 4.6. Before a value is added to the hash table, however, it performs a check to determine if the variable or data structure should still be profiled by checking whether the number of distinct values for the variable or data structure exceeds

```

void __calpa_update_int_var_def_or_use(uint hashcode, int kind, uint
sid, uint prid, int fn, int value) {
    tag_t tag;
    value_t val;
    make_var_tag(&tag, hashcode, (update_kind) kind, sid, prid, fn);
    val.type_id = rt_type_int;
    val.u.int_val = value;
    update_cache_entry(&tag, &val, make_unhook_mode(kind, 1));
}

void __calpa_update_int_mem_def_or_use(uint hashcode, int kind, void* addr, int value,
    uint sid, uint prid, int fn, void* aaddr, int asize, int count, ...) {
    long ptr_diff = MAXLONG;
    int item = -1;
    boolean exact = false;
    int i;
    // first identify which of the possible objects we do actually update
    va_list ap;
    calpa_memory_obj_desc_t result;
    int good;
    uint ips_id = 0, ips_pi = 0;
    if ((kind == update_def)) {
        // grab ips ids first
        va_start(ap, count);
        ips_id = va_arg(ap, int);
        ips_pi = va_arg(ap, int);
        count -= 2;
    }
    if (count) {
        if (kind != update_def)
            va_start(ap, count);
        good = try_match_address(&result, &ap, count, addr, aaddr, asize);
    }
    else
        good = try_match_address(&result, NULL, count, addr, aaddr, asize);
    tag_t tag;
    make_ptr_tag(&tag, hashcode, (update_kind) kind, sid, prid, fn, &result);
    if (kind == update_def)
        __calpa_update_ips(ips_id, ips_pi, &tag);
    value_t val;
    val.type_id = rt_type_int;
    val.u.int_val = value;
    tod_t* sp = (tod_t*) aaddr;
    boolean updated = update_cache_entry(&tag, &val, make_unhook_mode(sp? sp[0].hook_counter : 0, good));
    // only use when all potential targets are in the array none on the stack for now
    if (!count) {
        if (!updated)
            sp[0].hook_counter++;
        else
            sp[0].hook_counter = 0; // reset whenever access to still-profiled data structure
    }
}

```

Figure 4.7: Data capturing routines for accesses to integer data, both direct and via pointers.

a specific (customizable) threshold. If this threshold is exceeded, `update_cache_entry` simply returns `false`, without any modifications to the value cache. Otherwise, the value cache is updated and `true` is returned. In practice, Tumi uses two different thresholds, one for scalar variables and one for composite data structures. This enables finer control over the number of distinct values to profile. In the default configuration, 500 values per data structure and 100 values for scalar variables are recorded.

The fast return without updating the value cache serves two purposes. Most importantly, it prevents the run-time value cache from growing unboundedly, thereby limiting adverse effects on the data cache performance of the instrumented application. Second, it reduces profiling time, since the access to the hash table and the costly check to determine if the value is a repetition of a previously seen value is eliminated.

4.5 Profiling Evaluation

4.5.1 Profiling Speed

Instrumentation for value-profiling with Tumi generally results in significant program slowdowns. Table 4.1 shows the execution times for four medium-size benchmarks (described in more detail in Section 6.1). The table shows the execution times of the unmodified programs and two configurations of instrumented programs measured by running the programs on the `train` input data sets provided with the Spec benchmarks (and the `022.li` input for `pnmconvol`). In the first configuration (labeled *low threshold*), profiling for a variable was turned off after 50 distinct values had been recorded in the value table, and for an array, structure or union after 250 distinct values; for instance, for an array of size 50, this might happen after every element has registered 5 distinct values or one element 250 distinct values. In the configuration labeled *high threshold*, profiling was turned off only after 100 distinct values for a scalar variable and 500 values for an array, structure or union had been recorded.

The table shows that while the original execution times ranged from fractions of a second to about 3 minutes, the profiling times are between two to three magnitudes larger, ranging from over 15 minutes to over 12 hours. The (geometric) mean slowdown for the high threshold configuration was 493; when profiling was turned off more quickly, the mean slowdown dropped to 396x, i.e.,

Table 4.1: This table shows the slowdown experienced by applications due to Tumi’s instrumentation.

Application	Original runtime [seconds]	Profiling with high threshold		Profiling with low threshold	
		runtime	slowdown	runtime	slowdown
m88ksim	0.4	15.4 min.	2313	10.1 min.	1516
dinero	1.3	3.7 min.	170	3.7 min.	170
pnmconvol	29.5	5.1 hrs.	628	4.5 hrs.	554
equake	185	12.4 hrs.	240	8.8 hrs.	172

performance improved by almost 25%.

Most of the slowdown is due to adding values to the value cache. A gprof [GKM82] profile of the profiling run for application `equake` shown in Table 4.2 illustrates this (profiles of other applications show the same trends). The profile shows the execution times for all procedures that accounted for at least 1% of overall execution time. Columns one and two show the procedure name and the percentage of execution time spent in it. Columns three through five show how many seconds were spent in the procedure itself (excluding procedures that it called), how often the procedure was called, and how much time was spent on average in each execution of a procedure (in milliseconds per call). Column six shows the total time spent on average in each procedure and its callees, and the last column shows the time spent in a procedure and all other procedures listed above it. With the exception of routines `main` and `smvp` all procedures are instrumentation procedures. We can see that over 53% of the total execution time is spent in the `update_cache_entry` routine, with 5 billion calls and an average of about 500 cycles per invocation to update the value cache. Almost 10% of the execution time is spent in the routine `find_closest_address`, which is the address matching routine that uses red-black trees to find addresses that correspond to malloc sites; per invocation, about 430 cycles are spent in this routine. Both `update_cache_entry` and `find_closest_address` were carefully examined and show little potential for further improvement. Therefore, the most promising way of improving profiling performance lies in eliminating calls to the instrumentation runtime whenever possible. One approach to achieve this goal is described in Section 4.5.2.

Table 4.2: This gprof profile of a profiling run for `equake` shows that most of the time is spent in the `update_cache_entry` routine.

Procedure name	%	self			total ms/call	cumulative seconds
		time	seconds	calls		
<code>update_cache_entry</code>	53.0	5054.75	5035997563	0.00	0.00	5054.75
<code>find_closest_address</code>	9.9	940.30	1093868424	0.00	0.00	5995.05
<code>smvp</code>	9.6	913.48	2709	337.20	2006.46	6908.52
<code>__calpa_rt_update_int_var_def_or_use</code>	9.4	898.51	10406788725	0.00	0.00	7807.04
<code>__calpa_rt_update_double_mem_def_or_use</code>	5.6	533.33	1524886250	0.00	0.00	8340.37
<code>main</code>	4.0	383.83	1	383830.08	9530183.59	8724.20
<code>try_match_address</code>	3.1	297.75	1525797317	0.00	0.00	9021.95
<code>__calpa_rt_update_double_var_def_or_use</code>	1.6	150.69	1679255177	0.00	0.00	9172.64
<code>make_ptr_tag</code>	1.5	141.66	1525797317	0.00	0.00	9314.30
<code>__calpa_rt_update_ips</code>	1.4	134.30	557012429	0.00	0.00	9448.61

4.5.2 Performance Optimization

Variables that have many distinct values during execution are unlikely to be good specialization candidates because of the specialization cost for polyvariant specialization. Therefore, eliminating them from the value profile and replacing information about them with conservative estimates (e.g., assuming their specialization cost will be prohibitive) will not change the annotations that Calpa will generate. Fortunately, it is possible to restrict profiling at run time to only those variables for which only few values have been seen by selectively turning off profiling for those variables for which more than a certain (configurable) number of values have been seen during instrumented execution.

I used a technique called *unhooking*, previously described by Traub et al. [TSS00]. Unhooking simply means ensuring that a call to an instrumentation library routine (e.g., `__calpa_rt_update_int_mem_def_or_use`) is no longer executed (either by overwriting it with a NOP or branching around the call), thereby saving the overhead of the call and the run-time cost of the profiling routine. To control whether a specific call to the instrumentation library routine should be unhooked, the Tumi run-time value cache keeps track of how many distinct values have already been recorded for a particular value, and no longer adds values to the value cache once the threshold is exceeded, as described in Section 4.4.2. Using the unhooking optimization, we can additionally

save the cost of calling into the instrumentation library by eliminating the call to the data capturing routine completely.

The Tumi runtime implements unhooking by overwriting the call to the data capturing routine with a branch around the call. The `jsr` cannot simply be overwritten, since it is followed by two instructions that rebuild the Alpha's global pointer (`gp`) register. The following assembly code is used to perform the unhooking:

```
ldah $1, -15392($31) # build br +2 ahead of updated pc
lda  $1, 2($1)
stl  $1, -4($10)    # replace jsr by br
call_pal 134      # IMB: make i-cache coherent
```

First, the opcode of a branch instruction that jumps ahead by 2 is computed into register 1. This instruction is written at the location of the `jsr` instruction (register 10 contains the data capturing routine's return address from which 4 has to be subtracted). Finally, the instruction cache is made coherent by flushing all entries with the IMB PAL code instruction. Sometimes, the two instructions that rebuild the `gp` register after a call are moved apart by the code scheduler. If that is the case, Tumi first rewrites the binary to place them immediately after the `jsr` instruction, and only then performs the unhooking.

One complication arises for instrumentation calls that may record values for multiple variables or data structures, i.e., for memory accesses via pointers. Such a call cannot be unhooked if the pointer might access some variable or data structure for which value profiling is still enabled. To track such accesses, the TOD structure contains a counter (the `hook_counter` field, shown in Figure 4.2), which is incremented each time an access to a specific target does not result in a cache update, because profiling for that target has already been turned off. How this is implemented, can be seen in the last five lines of the data capturing routine for integers (shown in Figure 4.7). Each time the access results in a cache update, the counter is reset to zero. As soon as the counter reaches a configurable threshold, the call is unhooked; by default, a hook counter threshold of 100,000 was used.⁹

⁹The test of `count`'s value checks whether any target object descriptors are passed on the stack. In that case, no unhooking is performed, since this would require determining which instructions preceding the call to be unhooked have arguments on the stack, which is generally impossible to know without information from the C compiler that was used to produce the binary. The code scheduler, for instance, may have moved instructions pushing arguments on the

This design trades off efficiency for value profile precision, since in a future access a variable that is still profiled may actually be accessed but that access will not be recorded because the call has been unhooked. An alternative design would use a flag for each target object and only when all flags indicate that none of the potential target objects are profiled anymore, the call would be unhooked. However, since the static points-to sets are likely to be imprecise and larger than the set of objects actually accessed at run-time, few calls would get unhooked in practice. Moreover, experiments (described in [MDCE01] and discussed in Chapter 7), showed that pointer dereferences in C typically access only a few objects at run time, many fewer than the static points-to sets indicate. Consequently, this design decision is unlikely to have any negative impact on value profile precision in practice.

Table 4.3 shows the impact of the unhooking optimization on performance degradation. The table shows the profiling run-times for the two configurations where profiling was turned off for 100 distinct values for variables and 500 values for data structures (high threshold), and for the low threshold configuration with 50 and 250 values respectively. The columns labeled "improvement" show the profiling run-time improvement relative to the profiling times without the unhooking optimization (shown in Table 4.1). With the exception of `m88ksim` in the high threshold configuration, unhooking resulted in improvement in profiling times of up to 4.3x. The (geometric) mean improvement was 1.45 for the 500/100 configuration, and 1.44 for the 250/50 configuration. In both cases the default hook counter threshold value of 100,000 was used. Using a smaller hook counter threshold of 1,000, did not result in any noticeable changes to the profiling times.

The behavior of `m88ksim` in the 500/100 configuration is interesting, since it actually shows that unhooking may sometimes result in a slowdown. This may happen when the cost of unhooking calls to data capturing routines (for the most part, the flushing of the instruction cache), is not amortized. This may happen when the program point from which the call was unhooked, is executed too infrequently after the unhooking, so that the cost savings from not performing the call is not reaped sufficiently often. When the configuration is changed to turn off profiling and start unhooking earlier (with only 250/50 values for data structures and variables, respectively). unhooking becomes effective for `m88ksim` as well.

Table 4.3: This table shows the profiling slowdowns when unhooking optimization is used. The columns labeled "improvement" show the improvement in profiling times relative to the corresponding configuration without unhooking. The (geometric) mean improvement was 1.45 for the high profiling threshold, and 1.44 for the low profiling threshold.

Application	Original runtime [seconds]	Profiling with high threshold		Profiling with low threshold	
		runtime	improvement	runtime	improvement
m88ksim	0.4	930.2	0.99	519.9	1.17
dinero	1.3	220.7	1.00	207.6	1.06
pnmconvol	29.5	18067.5	1.03	16267.9	1.00
equake	185	10177.5	4.45	9185	3.46

Table 4.4: This table shows the blowup in executable size incurred by instrumentation. On average (geometric mean), the executable size grows by a factor of 10.5, which includes the fixed-size instrumentation library of 665KB.

Application	Original size (KB)	Instrumented size (KB)	Blowup
m88ksim	227	2944	10.8
dinero	64	688	10.8
pnmconvol	72	544	7.6
equake	48	664	13.8

4.5.3 Space Blowup

For small kernel-size applications, the greatest increase in executable size from instrumentation comes from linking with the instrumentation library, which is approximately 665KB large. For medium-size and large applications on the other hand, the instrumentation code added to each procedure (e.g., calls to data capturing routines) is responsible for most of the code blowup. Table 4.4 shows the binary size expansion experienced by `m88ksim`, `dinero`, `pnmconvol`, and `equake`. In all cases the blowup was considerable, expanding binary size on average by a factor of 10.5 (including the statically linked instrumentation library). While the code growth is substantial, it is not prohibitive. Moreover, if instrumentation were applied more selectively (as outlined in Section 4.7), it would likely be a lot less severe.

4.6 Related Value Profiling Approaches

The idea of value profiling was introduced by Calder et al. [CFE97, CFE99]. The motivation for their work was the observation that some long latency instructions on the Alpha 21064 processor (e.g., division, which can take up to 60 cycles to execute) are frequently executed with the same arguments. For instance, in the `hydro2d` benchmark from the SPEC92 benchmarks suite, they found that 64% of the executed divide instructions had either a 0 for its numerator, or a 1 for its denominator. Consequently, optimizations based on constants (such as run-time specialization), might be beneficial for such applications as repeated long-latency computations could be memoized, and the results used instead of repeatedly performing the same computation.

To find out how often such opportunities arise in practice, they designed a value profiler that captures register values of binaries for the Alpha processor. Their value profiler is implemented using the Atom [SE94] binary instrumentation tool and keeps the values written by each instruction in a *Top N Value (TNV)* table for every profiled register. The TNV stores pairs of the form (value, number of occurrences), and uses a least frequently used replacement (LFU) policy when more than N values are written to a register during execution. To avoid the LFU policy preventing new values entering the table, the bottom half is periodically cleared. This is necessary because, once the table is filled with values that have been profiled more than once, new values might continually replace each other and effectively prevent any of them from getting higher frequency counts.

With this modified LFU strategy, Calder et al. were able to identify several opportunities where value-based optimizations, namely specialization, might be used profitably. However, since their profiling was on the level of processor registers, to optimize an application, they had to inspect the binary code, and map the instructions that exhibited a large invariance back to the corresponding source code statements and apply the code specialization optimizations manually [CFE99]. For instance, they rewrote the `alignd` routine in `m88ksim`, created a (compile-time) specialized version, and added a run-time check to select the hand-specialized version when possible during execution.

While their profiling infrastructure requires some manual mapping steps to identify good specialization steps, their more limited run-time data collection is quite a bit more efficient than Tumi's run-time profiling. On average, Calder's profiler slowed down the applications by a factor of 32.9.

For an adaptive version of the profiling tool that dynamically disables profiling for instructions whose TNV table has *converged*, i.e., reached a steady state, the slowdown dropped to a factor of 10.

Watterson and Debray [WD01] present a value profiling infrastructure very similar to Calder et al.'s. It is also based on the Atom tool, but they use a cost-benefit model to decide which instructions should be profiled. As soon as the cost-benefit estimation for a profiled instruction indicates that the instruction is unlikely to be a good candidate for the link-time specialization transformations they perform, profiling is stopped. This idea is similar in spirit to Tumi's unhooking optimization for variables for which a large number of values have already been profiled. Compared to profiling all load instructions for the entire duration of a program, their *goal-directed profiling* reduced the profiling cost from a slowdown of 10-28x down to only 2.4-5x.

4.7 Possible Improvements to Tumi

Calpa might benefit from implementing a goal-directed profiling strategy in Tumi as well. For instance, a cheap frequency profile could be gathered. Based on this frequency profile, CSV sets could be computed and their benefit evaluated. Variables that are of no or little potential benefit would then be marked as irrelevant and Tumi would ignore them in a subsequent instrumentation and profiling pass to gather a full value and frequency profile for the application.

An even simpler optimization might be used to improve Tumi's performance. Currently, every use of a variable is profiled. However, subsequent uses of a variable use the same value when there is no intervening definition. When this can be statically determined, only the first use of the variable could be instrumented. This could be implemented by computing def-use chains and instrumenting only the first use of a variable v that shares the same definition with other uses of v .

Chapter 5

THE CALPA ANNOTATION SELECTOR TOOL

Chapter 3 gave an overview of the different components that make up the Calpa system, Chapter 4 presented the instrumentation details on its instrumentation infrastructure. The focus of this chapter are Calpa’s annotation selector tool, its algorithms, and the cost-benefit model. It is organized as follows: Section 5.1 discusses the different pointer analysis algorithms that are used in the Calpa system. Section 5.2 describes the candidate static variable analysis in more detail. Section 5.3 gives a description of the strategy used to search the space of potential annotations. Section 5.4 describes Calpa’s cost-benefit analysis and its pertaining algorithms. Section 5.5 shows how performance estimates are computed for CSV sets. And finally, Section 5.6 looks at the algorithms used to select the procedures for which annotations should be computed.

5.1 Pointer Analysis

To identify potential definitions of variables and data structures (called *invalidation points*), Calpa uses a static pointer analysis algorithm. Several algorithms were implemented to assess the impact of the pointer analysis precision on the ability to derive good DyC annotations. A less precise pointer analysis algorithm will potentially produce more invalidation points, resulting in a higher invalidation checking cost estimate in the cost-benefit model, which in turn may prevent the specialization for some variables which in fact might be good candidates. A more precise algorithm may take more time to run, but could potentially result in annotations with a higher overall speedup.

To enable scaling Calpa to larger applications, I implemented three different flow- and context-insensitive pointer analysis algorithms: Steensgaard’s algorithm [Ste96b], an extension thereof proposed by Shapiro and Horwitz [SH97b], and a recent improvement to Steensgaard’s algorithm (called *One-Level-Flow (OLF)*) proposed by Manuvir Das [Das00].

These algorithms compute for each variable and expression in the program the sets of objects

(variables, heap-allocated data structures, or procedures) the variable may point to, regardless of where the variable or expression occurs in the program (i.e., they are flow-insensitive), and also irrespective of the calling context (i.e., they are context-insensitive). Flow- and context-insensitive algorithms have been shown to scale to programs of over a million lines of C code [Das00], which makes them the algorithms of choice for the analysis of large programs.

In addition, [Das00] showed that the One-Level Flow algorithm generally produces results as precise as Andersen’s well-known flow- and context-insensitive algorithm [And94], which, however, is considerably slower in practice. Andersen’s algorithm, in turn, has been shown [LH99a] to be of comparable precision as some other well-known flow- [PLR94] or context-sensitive [LH99a] pointer analysis algorithms, which suggests that in practice, the results obtainable with more expensive flow- or context-sensitive algorithms (when they are practical) may not be significantly better than what is achievable with the OLF algorithm.

In addition to the flow- and context-insensitive algorithms, I also implemented one context-sensitive algorithm designed to potentially disambiguate accesses to heap-allocated data structures [GH96]. However, for the benchmarks used in this thesis, this algorithm did not lead to higher precision, so it was not used in Calpa’s evaluation. For a different set of benchmarks, particularly for numeric programs with many heap-allocated matrices, it may be worthwhile to revisit this algorithm.

5.1.1 Invalidation Point Computation

Computing the invalidation points for a program requires identifying which abstract locations may be modified by each instruction in the program. This is done in two steps. First, one of the scalable pointer analyses is run, to identify which abstract locations may be modified by each pointer store instruction. In addition to the potential modifications that occur directly or via pointers, modifications that occur transitively via procedure calls need to be identified. Therefore, in a second step, a conservative approximation of the call graph (based on the results of the scalable pointer analysis of the first step) is computed. Then, the transitive closure of the call graph is computed using a standard algorithm [CLR90]. The invalidation points for a procedure call are then computed as the union of all invalidation points (direct or pointer stores) for all the transitively reachable procedures

for that call. To save memory in this step, a single representative is created for each invalidation point, which identifies its location in the program (procedure, file and instruction number). The invalidation point sets are represented compactly as bit sets, where each bit position represents one invalidation point in the program.

While computing the sets of possibly (transitively) modified abstract locations for each instruction, a hash table is used to record for each abstract location the associated set of invalidation points that can potentially modify the abstract location. This enables fast access to the set of invalidation points given an abstract location, which makes the computation of invalidation costs in the cost-benefit model efficient.

When Calpa's annotation selector decides to use the invalidation-based caching scheme for a particular variable or data structure, the result of the invalidation point computation is used to identify those program points where (conditional) calls to the DyC `invalidate()` operation have to be inserted. When the memory operation at an invalidation point is guaranteed to access only one variable or data structure, an unguarded call can be inserted; otherwise, the call is guarded with a test condition that compares the pointer address to the address of the specialized variable or data structure. If the variable or data structure is a composite data structure, the test potentially involves testing against an address range rather than a simple address comparison. Calpa currently only handles the simple case of variables not allocated on the heap for which an expression for the address range can be computed at run-time. For instance, for globally defined array variables, Calpa uses the declared array size to compute the address range that has to be tested; for structures it uses the total structure size. For heap-allocated data structures, it conservatively generates an unconditional call to the `invalidate` operation.

5.1.2 *Library Functions*

To correctly account for possible modifications to variables by calls to library functions, and to model their behavior on points-to sets correctly, the pointer and invalidation point analysis make the following assumptions:

- User-defined global variables are assumed not to be modified by any calls to a library routine. Since libraries are usually written without assuming specific global variables to be defined

by a program that uses the library, this will be sound in general. For libraries that violate this assumption, the user can provide a library skeleton function that models the effect of the library routine on the global variable, and add this code skeleton to the list of analyzed procedures.

- Library routines that take pointers as arguments are assumed to potentially modify any of the objects the pointer may point to. The standard C library routines, however, are modeled more precisely to avoid overly conservative results.
- Library routines are assumed not to call back into user code. If this assumption is violated, the user has to provide skeletons that model the behavior of the library routine correctly to ensure correct analysis results. For the benchmarks used in this thesis, only modeling of the C library routines that can call back to user code (e.g., `quicksort`) was necessary to ensure correct results.

5.2 Candidate Static Variable Analysis

After computing the invalidation points for the whole program, each procedure is analyzed to identify variables that might be good annotation candidates. Section 3.2 explained how the candidate static variable sets are identified for a basic block. This section first gives a more detailed rationale for CSV sets and then provides more details about CSV set computation for basic blocks. It also looks at how CSV sets are computed for whole procedures, and how the initial CSV sets are combined to form larger sets of annotation candidates.

5.2.1 Rationale

One of the major problems in writing good annotations is that there are so many annotation choices of variables for whose values specialized code could be generated. These choices, i.e., sets of specialized variables are called *candidate divisions* (cf. Section 3.2). With n variables there is $\binom{n}{0} = 1$ candidate division with no variables specialized, $\binom{n}{1} = n$ candidate divisions with one specialized variable, $\binom{n}{2} = \frac{n(n-1)}{2}$ with two specialized variables, etc. In total, there are $\sum_{i=0}^n \binom{n}{i} = 2^n$ candidate divisions, that is the number of candidate divisions grows exponentially in the number of the

variables accessible in a procedure (local variables and globals). Therefore, it will generally not be possible to simply try out all possible candidate divisions of variables. Fortunately, however, this is not necessary either.

The key insight is that different candidate divisions of variables may lead to the same instructions becoming static, i.e., to the same specialization benefits. Consequently, by generating and evaluating only those candidate divisions that result in distinct specialization benefits, we can potentially significantly reduce the time it takes to generate annotations without any loss in annotation quality.

To enumerate only candidate divisions that result in distinct specialization benefits, we start at the instruction level by computing the (minimal) set of variables that have to be static to make the instruction static, i.e., the CSV set for the instruction. By combining (set union) of the CSV sets of several instructions, we obtain a minimal set of variables that have to be specialized to make all the corresponding instructions static. All other candidate divisions that make the same instructions static have to be super sets of this minimal set. Since the cost of specializing for more variables is at least as high as the specialization cost for the minimal set (because the cache key will be potentially bigger, or there will be more invalidation points), by evaluating only the performance impact for the minimal sets we are guaranteed not to miss any better candidate division. These minimal candidate division sets are called *representative candidate divisions*, in short *RCDs*.

Evaluating just one representative for all candidate divisions that result in the same specialization benefit will in general considerably reduce the number of candidate divisions that have to be evaluated. Assume there are m instructions in a procedure, then to evaluate all possible distinct specialization benefits we only have to look at all possible combinations of the m associated CSV sets. This may not look like a win because m may often be bigger than n , the number of accessible variables. In practice, however, since the CSV sets are based on the actual computations performed in the procedure and their particular ways of using variables together, combining the m CSV sets result in far fewer distinct candidate divisions than the upper bound of 2^m . Section 6.3 presents some data that demonstrates that in practice the number of candidate divisions only reaches the theoretical maximum only when the number of variables is very small, so that the number of candidate divisions to evaluate remains manageable.

While in practice the number of representative candidate divisions grows slowly, it may still be

desirable to evaluate only a subset of all RCDs. If some RCDs are omitted, some good (maybe the best possible) annotation choices might be missed. However, by generating and evaluating RCDs in a particular order, the probability that this might occur can be minimized. Section 5.3 explains in detail the order in which Calpa generates and combines candidate divisions.

5.2.2 CSV Sets for Basic Blocks

The first step in computing candidate divisions consists of identifying (for each instruction) the minimal set that makes the instruction static, i.e., its candidate static variable set. The CSV sets for individual instructions are computed on a particular level of Suif IR that is very close to the original C expressions. Each C statement is represented by a *tree instruction*, consisting of an opcode that identifies the operation and operand sub-trees for the operands. Operands are therefore either references to source variables or constants, or the result of other instructions represented as trees, i.e., at this IR level there are no compiler-introduced temporaries.¹ The CSV set for an instruction is then computed as the set of variables on the leaves of the instruction.

The algorithm just described potentially requires more variables to be specialized than would be strictly necessary. This is because if the value of a variable is computed from the value(s) of some other variable(s) that are also specialized, DyC’s binding time analysis will derive that variable to be static even without it being annotated as a specialized variable. For example, if both variables x and y are specialized, and z is related to them by the (single) definition of $z = x + y$, z is derived to be static simply because both x and y are specialized variables. To account for this property of DyC’s BTA, a variable will not be added to the CSV set of an instruction if it has a single reaching definition. Instead, the CSV set of the definition is used. That is, when computing the CSV set for an instruction $z = x \text{ op } y$ where $v = \dots$ is the single reaching definition for variable x , and S the CSV set for that definition, then the CSV set will be computed as $S \cup \{y\}$. Figure 5.1 gives the complete algorithm in pseudocode.

To determine that there is a single reaching definition and identify the corresponding variables

¹The only exception to this rule are temporaries introduced for the return values of procedure calls and for string literals. There is no specialization opportunity for string literals, so they can be ignored. While DyC can memoize the return values of side-effect free procedures, this is currently only exploited in Calpa for the mathematical functions of the standard library `libmath.a`. Its procedures are automatically recognized by Calpa as side-effect free. Once recognized, Calpa treats them like elementary operations, so that calls to them are specialized when beneficial.

```

CSV(v) = {v} if v does not have a single reaching definition
        = CSV (def) where def is the unique reaching definition
          instruction defining v
CSV(const) = {}
CSV(dst = exp1 binop exp2) = CSV(exp1) U CSV(exp2)
CSV(dst = unop exp) = CSV(exp)

```

Figure 5.1: This pseudocode shows how the initial CSV sets are computed for each instruction based on the inductive base cases for variables and constants.

and their CSV sets, Calpa performs a reaching definitions computation prior to computing CSV sets; the implementation simply uses the reaching definitions analysis provided by the Machine Suif Dataflow Libraries [HY97]. Since this analysis library does not perform pointer analysis, global variables and variables that have their address taken are not included in the reaching definitions computation. Therefore, the CSV analysis conservatively assumes that such variables might have multiple reaching definitions.

5.2.3 CSV Sets for Procedures

Currently, Calpa generates a set of annotations only for whole procedures, i.e., no sub-regions with possibly distinct annotations are identified. In some cases, better speedups may be possible with different annotations in distinct parts of a procedure, for example, when a variable is semi-invariant in one part of the procedure but not in another, so that specialization would be beneficial for the former but not the latter procedure sub-region. In this case, Calpa would currently either decide not to specialize the variable, or its annotations are likely to result in lower speedups than possible with different annotations for the distinct regions. In our benchmark suite, this case never occurred. In theory, however, generating annotations only on a whole-procedure basis may miss some good annotations. The development of an algorithm to determine optimal sub-regions, therefore, is an interesting area for future work. Since Calpa currently computes only one annotation for an entire procedure, the CSV sets for a whole procedure are simply obtained as the collection (set union) of

all CSV sets computed for all (reachable) basic blocks.

5.3 Candidate Divisions and Search Strategy

As previously explained, the initial candidate divisions are the CSV sets obtained by analyzing all instructions in a procedure. Additional candidate divisions are obtained by combining candidate divisions via set union. This section first explains how a current set of candidate divisions is ordered with the goal of evaluating the costs and benefits for likely better divisions early to avoid missing good annotations in case not all RCDs can be evaluated. I then explain how new candidate divisions are generated once all current candidate divisions have been evaluated. And finally, the heuristic used to terminate the creation and evaluation of new candidate divisions is described.

5.3.1 Ordering of Candidate Divisions

To minimize the chances of missing good candidate divisions in case not all RCDs can be computed, the initial candidate divisions are evaluated in a specific priority order. This priority order is computed by assigning a *fitness score* for each candidate division. The fitness score is based on an invariance measure for each variable in the division. Variables with few values are likely to be better specialization candidates because their specialization cost will be lower. Each variable is assigned an invariance value based on the number of values recorded for it in the value profile. The invariance values for all variables of a division are then combined to an overall fitness score for the division. The divisions are then evaluated in order of their fitness scores.

To compute the fitness score, the invariance for all variables is computed first. The invariance of a variable is computed as the inverse of the number of distinct values that are reported by the value profile for the variable, e.g., for a variable with two distinct values, the invariance value is 0.5. If there is no information available for a variable, its invariance is set to 0. The higher the invariance of a variable, the lower its potential specialization cost is likely to be, since more distinct code versions would have to be generated for variables with more distinct values. The fitness score for a candidate division $\{v_1, \dots, v_n\}$ with invariance score of variable i being k_i is then computed as: $fitness(\{v_1, \dots, v_n\}) = \sum_{i=1}^n -1/k_i$, that is, the negative sum of the number of distinct values found for each variable. The higher this value (less negative) the lower the expected specialization cost for

the whole division is likely to be.

After the fitness score for all divisions has been computed, they are evaluated in ascending fitness score order. Whenever a division has been evaluated, it is marked to prevent it from being evaluated more than once (since it may be produced again in a future combination step). Moreover, the cost-benefit performance evaluation that was computed for it, is stored to guide the creation of future combinations as described below.

5.3.2 *Creating New Representative Candidate Divisions*

Once all current candidate divisions have been evaluated, new divisions are created by combining current divisions using the set union operation. The goal is to produce new candidate divisions that result in better overall cost-benefit evaluations earlier than candidate divisions that yield worse performance estimates. To achieve this goal the following heuristic is used. The candidate division that had the best performance estimate is selected as the *generator candidate division*. It is combined with all other divisions to create new candidate divisions. The motivation behind using the division that had the best performance estimate so far is to enlarge the specialization benefit (i.e., create a larger candidate division) based on a candidate division that is (as evidenced by the cost-benefit model) already good, i.e., has a good ratio of specialization benefit to specialization and caching cost.

Once all combinations of the generator candidate division with all other candidate divisions are computed, the fitness scores for the new divisions are computed, and the divisions are then evaluated in fitness score order. If the combination of the generator candidate division with all other divisions fails to produce any new candidate divisions, the candidate division that had the second-best performance estimate is selected as the generator candidate division, and the generation step is repeated. This continues until new candidate divisions are produced, or all candidate divisions have been tried as generators. When all candidate divisions have been tried as generators without producing new combinations, all possible RCDs have been generated and the search can stop. Figure 5.2 shows graphically how the fitness scores and performance estimates are used to generate new divisions. The next section describes the criteria that are used to stop the search before all RCDs have been generated.

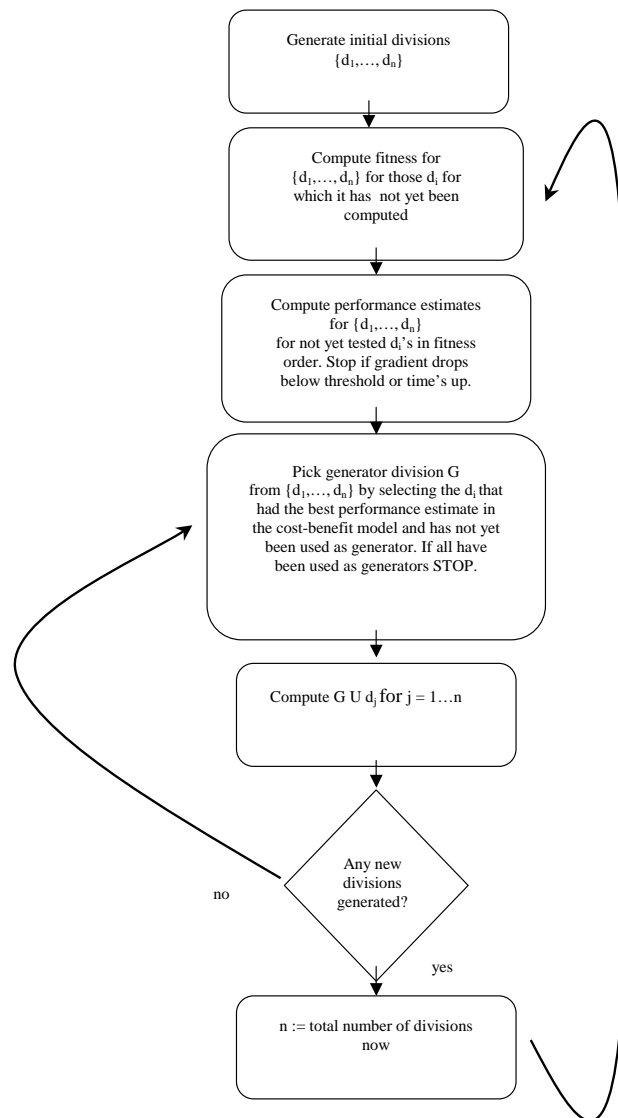


Figure 5.2: This graph shows how new divisions are generated using the fitness scores and performance estimates computed by the cost-benefit model. The process stops when the gradient of improvement drops below a preset threshold, the time quota is up, or when no new divisions can be generated.

5.3.3 Search Termination

To generate annotations faster, Calpa allows the evaluation of candidate divisions to finish before all possible RCDs have been evaluated. The heuristics described in Section 5.3.1 and Section 5.3.2 are used to evaluate likely good candidate divisions earlier than later. To decide when to terminate the search, a *gradient search strategy* is used. Whenever a candidate division is found whose performance estimate is better than the previously best found candidate division, the gradient of improvement, i.e., the ratio of the new (better) performance estimate and the previously best performance estimate is computed. If this gradient drops below a preset threshold (`min_gradient`) and a minimum number (`min_tests`) of candidate divisions have been evaluated, the search is terminated. The default values used for in the experiments in Chapter 6 were 1.01 and 1,000 respectively, i.e., the search was terminated when the improvement fell below one percent of the previously best performance estimate, and at least one thousand candidate divisions had been tried.

This gradient search strategy terminates the search after a minimum number of alternatives have been evaluated and no sufficiently better annotation candidates have been identified, i.e., the decision to terminate the search is based exclusively on characteristics of the analyzed program (its RCD sets), and the candidate division generation heuristic. The search times resulting from a particular `min_gradient` value are therefore hard to predict for a user. To enable a better control of how much time is spent in annotating procedures, an additional criterion to terminate the search is provided.

This criterion is based on a time quota, where analysis time is measured as the product of CFG size (number of nodes) and the number of tested candidate divisions. Since the time it takes to compute the performance estimate for an annotation is roughly proportional to the CFG size, this provides a way of bounding the annotation generation time. Unless reported otherwise, for the experiments in Chapter 6 a time quota of 100,000 was used, e.g., for a procedure with 50 CFG nodes the search was terminated after evaluating 2000 candidate divisions (unless it was terminated earlier based on the gradient criterion, or the lack of new RCDs).

5.4 The Cost-Benefit Model

The purpose of the cost-benefit model is to compute a performance estimate for each candidate division in order to find a good annotation choice. The overall design of the cost-benefit model was

described already in Section 3.3. This section presents in more detail how the costs and benefits assessed by the model are computed. Both the benefit estimation and the cost estimation require the results of the BTA for the candidate division that is being evaluated, so some relevant details of the BTA are discussed first. Then, Sections 5.4.2 to Section 5.4.4 discuss how the different cost components are computed; Section 5.4.5 explains how the benefit estimate is computed.

5.4.1 *Binding Time Analysis*

The BTA is implemented as an iterative data flow analysis. It is a simpler version of the BTA that is implemented in the DyC system [GMP⁺00b] inasmuch as no reachability analysis (cf. Section 2.1.1) is performed. Reachability analysis enables some operations to be considered static that are otherwise determined to be dynamic. The difference in operation classification has two consequences for the cost-benefit model. First, the benefit of a particular candidate division might be underestimated, since fewer instructions are tagged static than actually will be made static by DyC's BTA. Second, the number of dynamic instructions that have to be generated at run-time might be overestimated. Both effects make Calpa's cost-benefit model somewhat more conservative in estimating any performance advantage for a particular candidate division.

Identifying Cache Check Points

To estimate the caching costs for variables that use the key lookup caching strategy, Calpa's BTA needs to compute where in the code DyC would insert cache check points. In practice, since Calpa's BTA does not include reachability analysis, it will produce a conservative approximation to the set of cache check points computed by DyC.

Cache check points are necessary at every point where a specialized variable is assigned a dynamic value (called *promotion points* in [GMP⁺00b]); this includes the procedure entry for all *extended procedure parameters*, i.e., for the procedure's parameters plus any global variables referenced in the procedure.² In addition, cache check points are required at control flow merges with multiple reaching definitions of a specialized variable (called *discordant merges* in [GMP⁺00b]).

²Since Calpa does not identify dynamic regions that are sub-regions of a procedure, the `make_static` annotations are placed at procedure entry. If sub-regions of procedures are identified for specialization in a future system, the region entry will be the respective cache check point.

Calpa's BTA uses the reaching definitions data flow analysis to identify such merge points; since it does not perform reachability analysis, it potentially identifies some merges as cache check points that do not have cache checks in the DyC system. As a consequence, Calpa's cost-benefit model may assess a higher cache cost than necessary.

5.4.2 *Specialization Cost*

To the first order, the specialization cost in DyC is roughly proportional to the number of generated instructions. Therefore, as a first step an estimate of the number of instructions that need to be generated for a particular candidate division is computed. The number of instructions that are generated for a dynamically optimized application depend on three factors:

1. the number of instructions that are dynamic, based on the chosen candidate division;
2. the number of different instances that are generated for each of the dynamic instructions, which is based on the number of distinct values that occur at run time for the specialized variables;
3. and finally, on the number of different instances of a dynamic instruction that are generated due to full loop unrolling.

Loop-free Code

Assuming that none of the variables in a candidate division is a loop induction variable, the number of instructions that can be generated at worst for n variables v_1, \dots, v_n , with variable v_i having c_i distinct values is given by the following expression: $\prod_{i=1}^n c_i$. This estimate assumes that all variables of the candidate division can vary independently. If this is not the case, for instance, if one variable's value can be derived from another variable, the product expression will overestimate the number of distinct instances that will be generated for an instruction.

To obtain a tighter estimate, we can omit variables that are not extended parameters from the product. The remaining local variables are either used as loop induction variables or for expressions that are computed based on extended parameters and will therefore not vary independently. Considering only extended parameters in the product, appears to produce a better estimate of the

number of generated instructions in practice than using all variables in the candidate division. The modified product given by $\prod_{i=1}^n c'_i$, where $c'_i = c_i$ if v_i is a parameter and $c'_i = 1$ otherwise, is called the *parameter factor*.

Instruction Estimation for Loop Unrolling

As a first step in estimating how many instances of a dynamic instruction are generated when loop unrolling is used, the natural loops of a procedure are computed using Machine Suif's data flow library [HY97], which implements the standard algorithm for loop identification described in [ASU86]. Using the identified natural loops, the *loop nesting tree* is computed. The loop nesting tree is a data structure where nodes represent loops and edges represent loop inclusion relationships; an edge (l_1, l_2) connects two loops l_1 and l_2 if and only if l_1 is the immediately enclosing loop of l_2 . The loop nesting tree is the basis for computing the *normalized loop frequency* for each loop based on the frequency profile data.

The normalized loop frequency is an estimate of the average number of trips around the loop normalized by how often the loop was executed either because of nesting or because of multiple calls to the procedure containing the loop. It is used as a multiplier to compute how many instances of each dynamic instruction in the loop will be produced. It is computed by dividing the execution frequency of the loop header (obtained from the frequency profile) by the execution frequency for the parent loop's header. Finally, to account for multiple invocations of the procedure, the value is further divided by the execution frequency of the procedure.

After computing the normalized loop frequency for all loops, the loops that will be unrolled based on the annotations are identified. DyC will completely unroll loops for which there is a static loop exit condition. If one is present, the normalized loop frequency is used to compute the number of instructions resulting from loop unrolling as described above.

After accounting for multiple instances of dynamic instructions by loop unrolling, the final instruction estimate is obtained by multiplying by the parameter factor, to take the multiple versions for distinct values of specialized variables into account. If the number of generated instructions exceeds a predefined threshold, the number of instructions is set to ∞ (implemented by the maximum double floating point value) to effectively prevent specialization. The rationale behind this

1. Compute the parameter factor: $pf(proc) = \prod_{i=1}^n c'_i$, where $c'_i = c_i$ if v_i is a parameter and $c'_i = 1$ and c_i the number of distinct values for parameter i .
2. Compute the loop nesting tree.
3. For each loop compute the *normalized loop frequency* as $nlf(loop) = freq(loop)/freq(parent(loop))/freq(proc)$, where $freq(loop)$ is the execution frequency of the loop header.
4. Compute the number of instances for each dynamic instruction as: $pf * nlf(loop)$ if *loop* is unrolled (has static exit condition), otherwise as pf .
5. If the total number of instructions exceeds `max_allowable_instructions` (32K by default), set the number of instruction to ∞ .

Figure 5.3: This figure shows the steps performed to estimate the number of instructions generated for a particular candidate division. For loop-free code the parameter factor (pf) is used to estimate the number of instances generated for each dynamic instruction. When a loop is unrolled, this estimate is multiplied by the normalized loop frequency to account for the multiple versions of the loop body.

heuristic is to avoid generating more instructions than would fit in the processor's instruction cache. The current model uses a 32K threshold, ensuring that never more than 32,768 instructions will be generated. A summary of the steps performed to estimate the number of generated dynamic instructions, can be found in Figure 5.3.

5.4.3 Caching Cost

The caching cost is computed for each candidate division by first identifying *cache check points*, i.e., program points where cache checks have to be inserted. Cache check points within a procedure are identified by the BTA (as described in 5.4.1); the procedure entry basic block is an additional point where a cache lookup takes place. For some variables a cache check may be omitted, i.e., a

`cache_one_unchecked` or `cache_all_unchecked` policy may be possible. A policy analysis (described below) is used to determine this. For each variable for which a cache check has to be performed, a fixed check cost per cache check point is assessed. Based on empirically observed values for DyC, five cycles per variable are used. This cost per cache check point is then multiplied by the execution frequency of the cache check point and the costs for all cache check points are summed up.

Policy Analysis

A *policy analysis* is used to determine whether cache checks can be omitted for some variables thereby lowering the caching costs. Currently, a simple heuristic is used to decide whether a particular (unsafe) policy should be considered to reduce the caching cost. The `cache_one_unchecked` or `promote_one_unchecked` policies are selected for variables whose invariance is equal to one, i.e., for which the value profile indicates that there was only one value during execution.

The `cache_all_unchecked` and `promote_all_unchecked` policies are selected only for variables that are loop induction variables and for which no more than a configurable threshold of distinct values were seen in the value profile (a threshold of 10,000 was used for the experiments reported in Chapter 6). A standard algorithm for loop induction variable identification is used to test whether a variable is a loop induction variable [Muc97].

The user can select whether unsafe annotations should be generated at all. By default, only safe annotations are generated, and presently no sophisticated analysis is used to try to determine whether a variable whose profile indicated that it had only one value is guaranteed to exhibit this property across all inputs. The details of such an analysis is the subject of future work.

5.4.4 Setup Cost

In general, the number of generated instructions is a very good approximation to the specialization cost. However, in some cases few instructions may be generated at a higher than average specialization cost because of the cost incurred by the evaluation of the static instructions, called the *setup cost*. This might potentially lead to overly optimistic specialization cost estimates, particularly for large candidate divisions that have many static instructions.

To evaluate whether adding a setup cost component to the original Calpa design [MBCE99, MCE00] would result in better annotation choices, I added setup cost estimation to the prototype implementation. The setup cost represents the time required to evaluate the static computations during code specialization. Each static instruction has to be evaluated once. To account for repeated setup cost due to loop unrolling, the setup cost for each instruction in an unrolled loop is multiplied by the normalized loop frequency. To further account for polyvariant specialization, the setup cost is also multiplied by the parameter factor.

While in theory, the setup cost makes Calpa’s cost-benefit model more precise, in practice the basic specialization cost estimate, i.e., without setup cost component, appears to be sufficient. For the benchmarks used in this dissertation, including the setup cost in the cost-benefit model had no effect on annotation generation (cf. Section 6.4.2).

5.4.5 Estimating Benefit

The benefit estimation algorithm is fairly straightforward. For every instruction that was made static by the particular candidate division choice, a specific benefit value is computed as the product of the instruction’s execution frequency and the *instruction cost*. The scaled cost savings for all instructions are added to compute the overall benefit for the candidate division.

The instruction cost that is used to compute the benefit value for each static instruction is computed as follows. Depending on the opcode that identifies what operation is performed, a specific basic instruction cost is used. The opcodes and their associated costs are listed in Table 5.1. To avoid being overly optimistic, instructions that are not on the critical path, are treated specially, in some cases by ignoring them completely and setting their cost to zero (Section 5.5 explains in detail when and how they are taken into account). The critical path is computed by using the instruction latencies (equal to the instruction costs) by constructing the data dependence graph for each basic block. When there are multiple critical paths, an instruction that is on any of them is assumed to be critical only instructions that are on none of the critical paths are treated as not being critical.

Some instructions are treated specially because their potential benefit depends on how they are used; load, call and data type conversion (Suif IR opcode `cvt`) instructions fall into this category. Instructions that load floating point values, are assumed to yield no benefit, because a floating point

Table 5.1: This table shows the cost (in machine cycles) that is used in the cost-benefit model for each of the Suif IR opcodes.

Cycle Cost	Opcode	Description
1	cpy	copy
1	add	add
1	sub	subtract
1	neg	negate
8	mul	multiply
32	div	divide
32	rem	remainder
32	mod	modulus
1	min	minimum
1	max	maximum
1	not	bitwise not
1	and	bitwise and
1	ior	bitwise inclusive or
1	xor	bitwise exclusive or
1	abs	absolute value
1	asr	arithmetic shift right
1	lsl	logical shift left
1	lsr	logical shift right
1	rot	rotate
1	cvt	convert
2	lod	load
1	str	store
1	seq	set equal
1	sne	set not equal
1	sl	set less than
1	sle	set less than or equal
1	btrue	branch if true
1	bfalse	branch if false
1	jmp	unconditional jump
1	cal	call
1	ret	return
1	mbr	multi-way branch
32	divfloor	divide with floor
32	divceil	divide with ceiling
2	memcpy	memory-to-memory copy

value cannot be built in place. Instead the floating point (run-time) constants are constructed in and loaded from memory; consequently, there is no benefit derived from eliminating loads of floating point values.

Call instructions are treated specially if they call mathematical library functions. This is done to account for the savings obtained from memoizing calls to them (as described in Section 2.1.1). The execution time for a mathematical function is highly data-dependent on its inputs. Therefore, the values used in the model (shown in Table 5.2) are averages, which were obtained experimentally by calling the each library function on a series of random numbers and measuring the execution time with the Alpha processor's cycle counter (cycle times were measured on a 500MHz Alpha 21164 based workstation).

The Suif IR produces data type conversion instructions which in some cases will not result in actual data type conversion instructions on the Alpha processor, such as when the `cvt` IR instruction is used to convert an integer value to a pointer value. This is done to make sure that instructions that require pointers (e.g., load instructions) always operate on quantities of pointer type. In actual machine code, however, no conversion takes place. Therefore, if the `cvt` opcode is simply used to produce a pointer-typed value, its instruction cost is set to 0 to avoid assessing a benefit that would never materialize in practice.

5.5 Performance Estimation

To estimate the overall performance impact of a candidate division on a procedure, an execution time estimate for the optimized and un-optimized procedure is computed; together with an estimate of the fraction of overall execution time spent in the procedure, their ratio is used to estimate overall program speedup. For the annotated program, the execution time estimate comprises the time for specialization, cache and invalidation checking, the execution time for the unspecialized code minus the execution time saved by specialization.

This section describes how an execution time estimate for the baseline (unoptimized) procedure is obtained, and how the costs and benefits computed in the cost-benefit model are used to compute an execution time estimate for a particular candidate division and procedure.

Table 5.2: Calls to mathematical routines from the standard library of mathematical functions `libmath.a` are treated as pseudo-ops with the cost estimates shown in the table. The cost in cycles was obtained experimentally using random inputs and the Alpha cycle counter for measurement.

Library function	Cycle Cost
<code>acos</code>	133
<code>asin</code>	118
<code>atan</code>	82
<code>atan2</code>	157
<code>ceil</code>	31
<code>cos</code>	78
<code>cosh</code>	73
<code>exp</code>	66
<code>fabs</code>	8
<code>floor</code>	33
<code>fmod</code>	58
<code>frexp</code>	22
<code>ldexp</code>	35
<code>log</code>	61
<code>log10</code>	60
<code>modf</code>	27
<code>pow</code>	118
<code>sin</code>	80
<code>sinh</code>	72
<code>sqrt</code>	72
<code>tan</code>	85
<code>tanh</code>	41

5.5.1 *Estimating Baseline Execution Times*

To estimate the execution time for a program, Calpa implements three distinct execution time estimators, which are called `cost-cycles`, `critical-cycles` and `pixie-cycles`. The first two are used for performance estimation of the annotated and baseline codes; all three are used to select which procedures should be annotated, which is described in detail in Section 5.6.

The `cost-cycles` estimator simply uses the instruction costs listed in Table 5.1 and Table 5.2 and the execution frequencies of each instruction from the frequency profile to compute an execution time estimate for each instruction. The estimate for a procedure is obtained by summing the times of all of its instructions (each weighted by frequency), and the whole program time estimate is obtained by adding up the times for all procedures. This essentially obtains an execution time estimate for a processor without multiple functional units, assuming that all memory instructions hit in the cache.

The `critical-cycles` estimator works like the `cost-cycles` estimator, except that instructions that are not on any critical path are not counted. This estimator produces an execution time estimate for a superscalar processor with infinite resources assuming that all memory instructions hit in the cache.

Since the Alpha processor is a superscalar machine with multiple (but not infinite) resources, the `cost-cycles` estimator provides an upper bound and the `critical-cycles` estimator a lower bound on execution time (always assuming that instructions hit in the cache), with the actual execution time lying somewhere in between. Therefore, the arithmetic mean of the two estimators is used to get an execution time estimate for a procedure (or program) without annotations.

The `pixie-cycles` estimator is based on the algorithm used by the well-known `pixie` binary instrumentation tool [MIP90]. It assumes a uniform instruction cost of one cycle per instruction and that all memory instructions hit in the cache. It is only used for selecting which procedures to annotate (cf. Section 5.5.2).

5.5.2 *Estimating Execution Times for Candidate Divisions*

To estimate the total execution time for a particular candidate division, the cost of each instruction is estimated and multiplied by its execution frequency.

The execution time for an instruction is computed as the arithmetic mean of the `cost-cycles`

and `critical-cycles` estimators, disregarding the cost for static instructions (static instructions on the critical path for the `critical-cycles` estimator), i.e., the benefits from eliminating static instructions that are not on any critical path is essentially counted only half. The motivation for this approach is as follows: the `cost-cycles` estimator produces an estimate for a single-scalar processor, so it will tend to overestimate the time savings arising from the elimination of static instructions on the dual-issue Alpha 21164, whereas the `critical-cycles` estimator computes the critical bath based on a processor with infinite resources and consequently tends to underestimate any time savings. Therefore, the mean of the two estimates will in general be a better estimate of the actual savings than each of the estimators alone.

To this basic execution time, the caching cost for all cache check points is added (which is also scaled by the execution frequency of each cache point). Then the specialization cost and invalidation check cost are computed and added to the overall execution time estimate, i.e., the estimate of execution time T is computed as:

$$T(CD) = \frac{\text{costcycles}(CD) + \text{criticalcycles}(CD)}{2} + sc(CD) + ic(CD) + cc(CD)$$

where sc , ic , and cc are the specialization (including setup), cache check, and invalidation check costs, respectively.

Computing Performance Estimate Values

The speedup for a candidate division is simply the estimated execution time compared to the unannotated baseline version. It is computed as the ratio of the baseline execution time estimate and the total execution time estimate for the candidate division:

$$\frac{\frac{\text{costcycles}(base) + \text{criticalcycles}(base)}{2}}{\frac{\text{costcycles}(CD) + \text{criticalcycles}(CD)}{2} + sc(CD) + ic(CD) + cc(CD)}$$

Values of greater than one indicate a speedup, values lower than one a slowdown.

5.6 Selecting Procedures for Annotation

To speed up the annotation computation, Calpa restricts it to a subset of all procedures. This can be done without forsaking any potential speedups by taking advantage of Amdahl's law [Amd67],

which states that improving the performance of a computation that accounts for a fraction f of total execution time, will improve total execution time by at most a factor of $1/(1 - f)$. Conversely, if we want to speed up a program by at least a factor of f' , we should not waste time on procedures that account for less than $\frac{f'-1}{f'}$ of the total execution time. Therefore, as a first step before annotating a program, the execution time of each procedure and its fraction of total execution time is estimated, using the execution time estimators presented in Section 5.5.1. For the results presented in Chapter 6, the following criteria were used to select a procedure p for annotation:

1. Compute the cycle estimate for procedure p and for the whole program using the cost-cycles, pixie-cycles, and critical-path cycles estimators. If, for either of these three estimates, the execution time percentage of procedure p is at least 3%, then annotate the procedure.
2. If, for either of the three estimators, p was among the top 20 procedures (ranked by their execution time estimates), and its execution time percentage was not lower than 1%, then annotate the procedure.

Chapter 6

EXPERIMENTAL EVALUATION

This chapter demonstrates that Calpa’s approach to automating selective dynamic compilation works well in practice. This claim is validated by experimentation: Calpa is used to derive annotations for a set of programs that had previously been dynamically optimized with DyC. In addition, Calpa is run on a previously untried program and shown to be able to identify optimization opportunities in it. It is also shown that annotations can be generated rapidly (within half an hour) once value profiles are available.

The remainder of this chapter is organized as follows. Section 6.1 presents the workload and Section 6.2 discusses the methodology used to evaluate the Calpa system. Section 6.3 shows that in practice the number of RCDs is small. Section 6.4 presents the annotation generation results for the workload used in this thesis, with a focus on annotation time and quality, and the influence of several parameters of the cost-benefit model on them.

Section 6.5 studies how sensitive the generated annotations are to changes in program input for the value profiles, and presents a technique to control sensitivity where needed. Finally, Section 6.6 presents a summary of the results of this chapter and discusses possible future work.

6.1 Workload

To evaluate Calpa’s effectiveness in annotation generation, I chose the benchmarks previously used to evaluate the DyC system [GPM⁺99] (for which a ”gold standard” of human annotations was available), and additionally a benchmark from the SPEC2000 benchmark suite. Table 6.1 contains the applications and a short description of their characteristics. `dinero` is a cache simulator that takes a description of a cache configuration and simulates the cache on a MIPS binary. `m88ksim` is an application from the SPEC95 benchmark suite; it simulates the Motorola 88000 processor. `pnmconvol` is an application from the `netpbm` library and performs image convolution, given a bitmap image and

Table 6.1: This table shows some characteristics of the workload used to evaluate the Calpa system. For the applications, their origin is listed. The kernels are small synthetic programs that perform a particular task; they have previously been used in the evaluation of DyC and other run-time specializers.

Program	Description	Size (Lines of Code)	Source
Applications			
dinero	cache simulator	3,317	University of Wisconsin
equake	seismic simulation	1200	SPEC2000
m88ksim	Motorola 88000 simulator	12,531	SPEC95
pnmconvol	image convolution	1,054	netpbm package
Kernels			
binary	binary search	147	
chebychev	polynomial function approximation	145	
dotproduct	dot-product of two vectors	134	
query	database query	149	
romberg	numerical integration	158	

a convolution matrix. `equake` is an application from SPEC2000 that performs seismic wave propagation computations and is the only application not previously optimized with the DyC system.¹ The `binary` kernel performs a binary search on an array of key-value pairs. `chebychev` computes a Chebychev polynomial approximation of a function. `dotproduct` computes the dot-product of two integer vectors. `query` performs a set of database queries, selecting a record from a database based on a query consisting of a conditional expression testing record properties. `romberg` performs numerical integration by iteration, using the Romberg numerical integration algorithm. For more details on the implementation of the programs, their source code can be found in Appendix B.

¹Absent from this table are two applications that were successfully sped up by DyC: `mipsi` a MIPS instruction set simulator and `viewperf`, an Open GL graphics application. Unfortunately, these applications could not be run through Calpa, due to limitations in the Suif frontend tools (the conversion tool to Suif IR and `linksuif`), which failed during the conversion process, or resulted in incorrect IR that could not be converted back to compilable C code.

6.2 Methodology

To test Calpa’s ability to generate good annotations, as a first step a value profile for the programs was generated by instrumenting the programs with Tumi, and then running them on a sample input. For the SPEC applications, the training input data sets were used for this purpose; for the kernels, the inputs that were used are described in Table 6.2. Then Calpa was run on the applications with this profile data to automatically generate annotations. The annotated programs were then compiled with the DyC compiler and speedups were measured by running the programs on all available inputs (for the SPEC applications this comprises the test, training and reference inputs). Table 6.2 lists and briefly describes all input sets used for the execution time measurements.

Assuming that Calpa’s cost benefit model correctly predicts the performance impact of annotations, the timing run on the training input should not result in any slowdowns, and should potentially yield some speedups. The execution times on the other inputs gives an indication of how input-dependent DyC’s program speedups are. All timings and speedup measurements were performed on a 500Mhz Alpha 21164-based workstation with 512MB of RAM. The speedups were measured by running both the unoptimized and optimized applications 51 times (11 times for whole program speedups), taking the median execution times to compute the speedup value.

6.3 CSV Set Growth

To compute its annotations, Calpa searches in the space of representative candidate divisions. In theory, the total number of RCDs may be equal to the number of combinatorially possible candidate divisions, which is of exponential size in the number of variables. This section presents some data that shows that this exponential blowup occurs in practice only when the number of variables is small (10 or fewer).

Table 6.3 lists the number of all possible representative candidate divisions for application *dinero*, and for the kernels.² For each analyzed procedure, the table contains the number of variables, the number of RCDs, the theoretically possible number of divisions, and the percentage of

²For the other applications, the exhaustive generation of all RCDs ran out of memory before all candidate static variable set combinations had been tried. In each case, only a moderate number of RCDs had been generated by the time memory was exhausted.

Table 6.2: This table shows the input data sets that were used for profiling and speedup measurements.

Program	Input data
dinero	Memory references for the <code>008.espresso</code> , <code>022.li</code> , and <code>047.tomcatv</code> SPEC benchmarks. The <code>022.li</code> input was used as the training input.
equake	SPEC-provided test, training and reference data sets.
m88ksim	SPEC-provided test, training and reference data sets, and one input set (named <code>brpt</code>) that sets 4 breakpoints in the <code>dcrand.lit</code> program.
pnmconvol	One common bitmap image and a 9x9 convolution matrix for the test case with 79% zeros; an 11x11 convolution matrix with 95% zeros for the training case; and a 11x11 convolution matrix with 50% zero entries.
binary	Arrays of size 15, 1023, and 4095 for the test, training and reference inputs which are searched 300, 3,000, and 30,000 times, respectively.
chebychev	Degree of Chebychev polynomial of 5, 10, and 20 for the test, training, and reference inputs respectively, running 10, 100, and 1,000 iterations of the computation, respectively. The function being approximated is the <i>cosine</i> function.
dotproduct	Test input: computation of 50,000 dot-products of two vectors of size 10 that are 90% zero-filled; training input: computation of 500,000 dot-products of two vectors of size 20 that are 90% zero-filled; reference input: computation of 50,000 dot-products of two vectors of size 100 that are 90% zero-filled
query	Query size of 7, 13, and 21 for test, training, and reference input, respectively, for a database of 1,000 elements, and 200,000 iterations of the query.
romberg	Numerical integration that divides the integration interval into 2, 4, and 8 sub-intervals for the test, training, and reference input, performing 1,500, 3,000, and 15,000 integrations, respectively.

Table 6.3: This table shows the number of candidate divisions for dinero and the kernels. For each application, data for all Calpa-analyzed procedures are shown, listing the number of variables, the number of RCDs, and the theoretical maximum number of divisions. The last column shows the ratio of the number of RCDs and the theoretical maximum as a percentage.

application	# variables	# divisions	theoretical max.	percentage of max.
dinero	6	8	64	12.5
	4	7	16	43.8
	12	512	4096	12.5
	12	1024	4096	25.0
	8	80	256	31.3
	3	8	8	100.0
	6	32	64	50.0
	5	12	32	37.5
	40	81920	$2^{40} \approx 10^{12}$	0.0
binary	7	24	128	18.8
chebychev	10	105	1024	10.2
dotproduct	7	32	128	25.0
query	6	32	64	50.0
romberg	10	512	1024	50.0

theoretically possible divisions that are representative. The table demonstrates that while in theory, there are more 2^{40} candidate divisions for one of dinero's procedures alone, the total number of representative candidate divisions is much smaller; the procedure with the largest number of RCDs has 81,920 RCDs versus more than 10^{20} possible candidate divisions. Only for small procedures, including some of the kernels, does the number of RCDs reach the theoretical limit. However, since in those cases the total number of variables is small, the total number of RCDs remained small as well, and never exceeded 1024, making the exhaustive evaluation of all RCDs feasible in practice in many cases.

6.4 Annotations

Table 6.4 gives a description of the annotations that were generated by Calpa for each application and the corresponding speedups achieved by DyC with those annotations. The annotations were generated using the One-Level flow algorithm to compute invalidation points, the profiles were generated using the training inputs, and these value profiles were used for annotation generation. The table also lists the time it took Calpa to generate the annotations. For comparison, the best annotations previously found by human annotators (if any), are shown in Table 6.5 along with their speedup values.

6.4.1 Annotation Evaluation & Speedups

Table 6.4 clearly demonstrates that, given value profiles, Calpa can produce annotations in very reasonable times, ranging from just a second for some of the kernels, to minutes for the larger applications, with a maximum of just over 27 minutes for `m88ksim`. In contrast, when we evaluated DyC’s capabilities in [GPM⁺99], it took us several weeks to derive the best annotations for the whole suite of applications, in a tedious trial-and-error process, trying out a new annotation, keeping it if it was better than what we had previously found, and tossing it out if it did not improve performance. For the non-kernel applications, this also involved manual inspection of the run-time values of program variables to determine which might be good annotation candidates. In contrast, Calpa requires no human intervention other than supplying an input data set for the profiling run, running the profile and then the annotation tool .

Ultimately, however, Calpa is only useful if is capable of deriving annotations that result in speedups; at the same time, it should not produce annotations that result in slowdowns, at least for the inputs that were used to derive the annotations. In addition, we can compare Calpa’s annotations to the ”gold standard” of the best annotations previously found by human programmers, to determine where Calpa makes conservative choices when in fact more aggressive optimizations might be possible. The following paragraphs look at these two yard sticks for each application in turn.

The speedups for all inputs and all applications are shown in Figure 6.1 for the kernels, and in Figure 6.2 for the applications. For the kernel applications and also for `pnmconvol` for which most of the time is spent in the dynamic region, asymptotic speedups for the dynamic region are

Table 6.4: This tables shows the annotations that were computed by Calpa for the benchmarks, the resulting speedups, and the time it took to generate the annotations. All annotations were computed based on the training profiles, with speedups measured for the training inputs as well. Speedups numbers are asymptotic speedups for the optimized routines. For `m88ksim` also whole program speedups are shown (marked as *w.p.*), since the optimized procedure accounted only for a small fraction of overall execution time.

Program	Annotations	Speedup	Annotation Time [min.:sec.]
dinero	Annotated procedure: <code>mainloop</code> ; the pointer to the cache data structure <code>cachep</code> , the pointer to the cache policy data structure <code>polycp</code> , and loop induction variables <code>stackdepth</code> and <code>stackdepth2</code> .	1.3	4:52
equake	No annotations (for the training profile)	1.0	7:54
m88ksim	Annotated procedure: <code>ckbrkpts</code> ; the pointer to the breakpoint array <code>bp</code> , the flag that indicates whether breakpoints are enabled <code>brkenabled</code> , and the loop induction variable iterating over the breakpoint array <code>cnt</code> .	3.6 (1.01 w.p.)	27:04
pnmconvol	The convolution format parameter <code>format</code> , the maximum pixel value parameter <code>maxval</code> , the convolution matrix for red, green, and blue <code>rweights</code> , <code>gweights</code> , <code>bweights</code> , induction variables <code>crow</code> , <code>ccol</code> and <code>ccolso2</code> used to loop over rows and columns, and the row and column bound parameters <code>crow</code> and <code>ccols</code> .	6.3	1:20
binary	No annotations.	1.0	0:01
chebychev	The interval bounds <code>xa</code> , and <code>xb</code> , the number of sub-intervals <code>n</code> , and loop induction variables <code>j</code> and <code>k</code> ; in addition, the call to the mathematical library function <code>cos</code> , is annotated as static.	12.3	0:01
dotproduct	both input vectors <code>u</code> and <code>v</code> , the vector size (<code>size</code>), and the loop induction variable <code>i</code> .	9.03	0:01
query	The pointer to the query data structure <code>qe</code> and the query size <code>n</code> .	2.1	1:00
romberg	The integration interval bounds <code>a</code> and <code>b</code> , loop induction variables <code>i</code> , <code>m</code> , and <code>n</code> and <code>M</code> , a parameter that determines the number of sub-intervals for the Romberg integration.	1.3	0:45

Table 6.5: This table shows the best annotations found by human annotators for the applications and their corresponding speedups. Speedups numbers are asymptotic speedups except for `m88ksim`, for which also whole program speedup (marked as w.p.) is shown.

Program	Annotations	Speedup
dinero	Annotated procedure: <code>mainloop</code> ; the pointer to the cache data structure <code>cachep</code> , the pointer to the cache policy data structure <code>policyp</code> , the pointer to the cache control data structure <code>ctrlp</code> , and loop induction variables <code>stackdepth</code> and <code>stackdepth2</code> .	1.3
equake	not previously annotated	n/a
m88ksim	Annotated procedure: <code>ckbrkpts</code> ; the pointer to the breakpoint array <code>bp</code> , and the loop induction variable iterating over the breakpoint array <code>cnt</code> .	4.3 (0.97 w.p)
pnmconvol	The convolution format parameter <code>format</code> , the maximum pixel value parameter <code>maxval</code> , the convolution matrix for red, green, and blue <code>rweights</code> , <code>gweights</code> , <code>bweights</code> , induction variables <code>crow</code> and <code>ccol</code> as well as <code>crowso2</code> and <code>ccolso2</code> used to loop over rows and columns of the convolution matrix, and row and column bound parameters <code>crow</code> s and <code>ccol</code> s.	6.7
binary	The data array <code>x</code> , its size <code>n</code> , <code>l</code> and <code>r</code> the lower and upper bound indices in the binary search, and <code>u</code> the index of the current data array element; the latter three are used to completely (multi-way) unroll the search loop.	2.48
chebychev	The interval bounds <code>xa</code> , and <code>xb</code> , the number of sub-intervals <code>n</code> , and loop induction variables <code>j</code> and <code>k</code> ; in addition, the call to the mathematical library function <code>cos</code> , is annotated as static.	5.
dotproduct	both input vectors <code>u</code> and <code>v</code> , the vector size (<code>size</code>), and the loop induction variable <code>i</code> .	6.9
query	The pointer to the query data structure <code>qe</code> and the query size <code>n</code> .	4.0
romberg	The loop induction variables <code>i</code> , <code>m</code> , and <code>n</code> and <code>M</code> , a parameter that determines the number of sub-intervals for the Romberg integration.	1.1

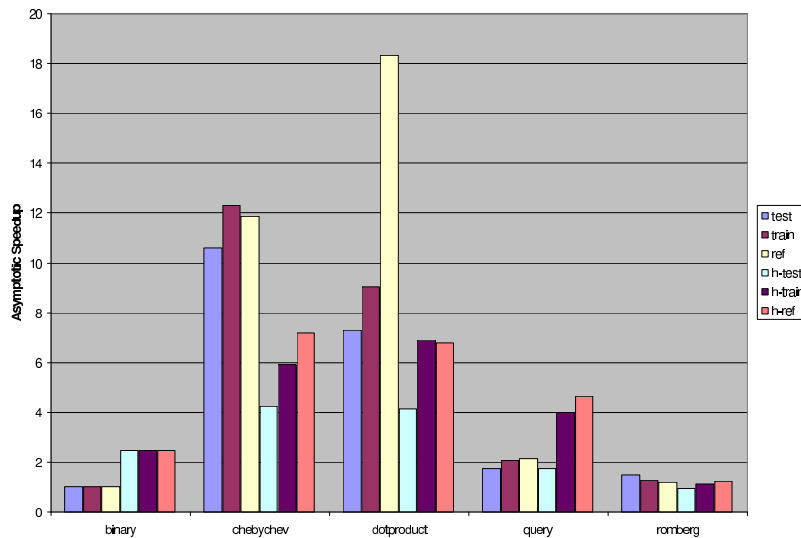


Figure 6.1: Speedups for the kernels across different inputs. The input labels starting with h- show the speedups achieved by the best, though sometimes unsafe, human annotations. The Calpa annotations were generated based on the training input value profile.

shown; for the applications `equake` and `m88ksim` whole program speedups are shown. With the exception of `equake`, the training inputs were used to generate the Calpa annotations; the speedups shown for `equake` are based on annotations generated by Calpa when run with the test input profile.

dinero

In `dinero`, the procedure `mainloop`, which is the main cache simulation loop, is specialized by annotating the (invariant) data structures that describe the simulated cache (`cachep` and `polycyp`). `cachep` contains, for instance, the cache block size and cache associativity; `polycyp` contains, e.g., the prefetch policies of the simulated cache. The values loaded from these data structures are used frequently in the simulation, and specializing for them eliminates the load overhead and enables subsequent constant propagation and folding optimizations. Loop induction variables `stackdepth` and `stackdepth2` are used in loops that can be completely unrolled since their upper bound is a run-time constant parameter loaded from the `cachep` data structure.

In addition to these, the human annotator also annotated the `ctrlp` data structure for structure

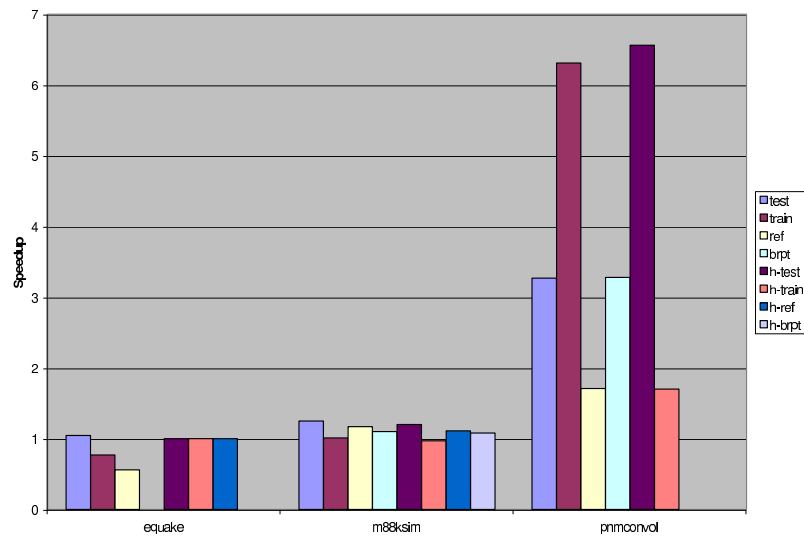


Figure 6.2: Whole program speedups for `m88ksim` and `equake`, and asymptotic region speedups for `pnmconvol`. For `equake`, annotations were only generated for the test input (the reasons for this behavior are discussed in Section 6.5); these annotations were used for the speedup measurements.

`field output`, which contains a cache configuration run-time constant that controls the output mode on the memory bus. The other fields in this data structure, however, are updated during the simulation, typically with many different distinct values. Since Calpa’s pointer analyses do not distinguish fields of a structure, any update of a structure field is treated as an update of the whole structure. Consequently, Calpa’s analysis concluded that many different specialized code versions would be necessary to specialize this data structure and chose not to annotate it. Every definition of one of the non-constant structure fields would have been an invalidation point for the whole structure. Since these updates occur in the frequently executed main loop, the invalidation cost computed by Calpa for this annotation was much higher than any additional benefit from eliminating the load of the structure field `output` and the subsequent constant propagation and folding optimizations.

If Calpa used a pointer analysis that distinguished structure fields, it would be possible for it to detect the specialization opportunity for `field output` as well. Such a more precise pointer analysis would produce no additional invalidation points if only the (run-time constant) `field output` was specialized. As mentioned in Section 4.3.3, Tumi already distinguishes updates of different structure

fields, therefore with the more precise static pointer analysis, Calpa would be able to determine that no invalidations of the structure field occur at run-time.

Unfortunately, due to a bug in the DyC system that we were unable to fix, we could not measure the speedup for the Calpa-generated annotations. However, since there are only a few computations that depend on the `output` field, the speedup is likely to be very close to the manually annotated version (therefore, the speedup listed in Table 6.4 should be taken as an estimate only).

equake

For the training input, Calpa did not produce any annotations for `equake`. However, when run on the test input, annotations were derived; the speedups and reasons for this sensitivity, are discussed in Section 6.5.

m88ksim

For `m88ksim`, Calpa chose to specialize routine `ckbrkpts`, the same routine that was previously identified as an annotation candidate by human annotators. The routine iterates over an array of breakpoints (which is empty for the training input), and checks whether the current address (an argument to the routine) is found in the array of breakpoints; if the breakpoint is found, it sets a flag and returns -1, otherwise it returns 0. Since the breakpoint array does not change for the input program that `m88ksim` simulates, the unrolling of the loop that iterates over the breakpoints will simply result in a NOP since no breakpoints are present, i.e., all the specialized routine does is return 0. The loop unrolling is achieved by annotating the loop induction variable `cnt`.

Before the loop is executed, the `ckbrkpts` routine checks, whether breakpoints are enabled at all (by testing the value of `brkenabled`). If not, the routine returns right away, returning 0. Calpa chose to additionally specialize this flag, with a resulting speedup of 3.55.

The annotations with the best speedups written by human annotators, achieve a speedup of 4.29. This is because they used the unsafe key lookup mechanism in the annotations: Calpa uses the invalidation-based caching scheme for variable `bp`, the pointer to the breakpoint array. Using the unsafe key lookup, which that checks only the value of the pointer but not the contents of the breakpoint array, results in less overhead and therefore higher speedup. However, since Calpa only

generates safe annotations, it does not choose this cheaper caching alternative.

The whole-program speedups across all inputs are shown in Figure 6.2. Interestingly, the Calpa-annotated program achieves slightly better speedups despite worse asymptotic speedups in the annotated `ckbrkpts` routine. Most notably, for the training input the Calpa-specialized version still achieves a 1% speedup, while the human-annotated version slows down by 3%. The reason for this un-intuitive behavior lies in negative instruction cache effects.

For all inputs, the instruction cache footprint of `m88ksim` was reduced by running the `cord` procedure reordering tool on the binary. `cord` performs a Pettis-Hansen-style [PH90] reordering of the procedures in the binary to improve instruction cache behavior. Without this reordering, the speedups achieved for the standalone `ckbrkpts` routine do not materialize in the execution of the full `m88ksim` program. The reason lies in the code size increase caused by the presence of dynamic compilers in the binary,

The particular procedure reordering that `cord` performs is based on the output of a special profiling run, performed by instrumenting the (un-annotated) `m88ksim` code with the `pixie` tool [MIP90], and running the instrumented binary on some input. The input used in this case consisted of the collection of the test, training, and reference input. It appears that the procedure reordering produced by this ordering works much better on the test and reference input than on the training input, which experiences the worst instruction cache performance degradation.

pnmconvol

For `pnmconvol`, Calpa derived a subset of the best annotations that had previously been found by hand. In contrast to the human annotator, Calpa chose to unroll loops with induction variables `crow` and `ccol`, and `ccolso2`, but not for induction variable `crowso2`, since it estimated that in this case too many instructions would be generated.

The difference however, resulted in only a minor difference in performance: the speedup on the training input dropped from 6.6 to 6.3. Part of this reduced speedup is due to the higher caching cost in the Calpa-annotated program: Calpa chooses the safe invalidation-based caching for the color weights arrays `rweights`, `gweights`, and `bweights`, whereas the human annotation used an unchecked cache policy.

binary

The kernel benchmark `binary` is an interesting case, since Calpa was not able to find the annotations previously found by human annotators. The reason lies in the complicated execution path of the routine. The annotations chosen by the human annotator caused the search loop that performs the binary search over the data array to be completely unrolled. However, the path through the loop is data-dependent on the (dynamic) value of the `key` argument, the value which is searched in the array, and therefore multiple paths through the loop depending on the key values are possible. However, all the distinct possible paths through the loop can be produced beforehand (eagerly), which is what the annotations written by the human annotator do.

Calpa's analysis is not capable of determining that this is possible, because it can only identify standard induction variables whose values progress in a linear fashion ($ai + b$, where a and b are constants, and $i = 0, 1, 2, \dots, n$). Calpa's cost-benefit model conservatively assumes that the search loop in `binary` can only be lazily unrolled, with much higher associated costs than occur with eager loop unrolling. This effectively prevents any annotations from being generated. The human annotator causes the loop unrolling by specializing for the values of `n`, `l`, `r` and `u`. Moreover, the contents of the data array `x` is specialized with an unsafe unchecked cache policy. With these annotations, DyC produces a speedup of 2.48, regardless of the input set.

chebychev

For `chebychev`, Calpa computed a superset of the annotations found by the human annotator. In addition to the number of sub-intervals for which the Chebychev polynomials were computed (`n`), the loop induction variables `i`, `k`, and the static call to the trigonometric function `cos`, Calpa also selected to specialize for the lower and upper interval bounds `xa`, `xb`, for which the polynomials were computed. In general, one might not want to do this, since the Chebychev routine may be called for a large number of distinct intervals. However, the driver program that was used in the experiments to call `chebychev` only used the same interval bounds throughout the execution, so that specialization for them is feasible.

Calpa's choice of specializing for the interval bounds on top of the variables chosen by the human annotator, resulted in significant performance improvements compared to the speedups with

the human annotations. For the training input the speedup improved from 5.9 to 12.3, for the test inputs from 4.2 to 10.6, and for the reference input from 7.2 to 11.9. The large speedup improvement is mostly the result of eliminating all calls to the function being approximated with polynomials (the cosine function). Since Calpa recognizes this as a pure function and the static interval bounds cause the arguments to the user function to be static as well, the cost-benefit model computes a large benefit from memoizing the result of the calls to the cosine function. If the Chebychev routine were not repeatedly called with the same interval bounds, or if the user function were not recognized by Calpa as side-effect free, the same annotations that were chosen by the human annotator would be generated.

dotproduct

`dotproduct` shows a similar phenomenon as `chebychev`: Calpa specializes all the variables identified by a human annotator, but in addition it also chooses to specialize for the second of the two integer vectors passed to the `dotproduct` routine `v`. Again, the reason is that the driver routine that calls `dotproduct` passes the same vectors `u` and `v` to the routine, whereas the human annotator had assumed that only vector `u` would be constant at run-time. Specializing for both vectors and unrolling the loop over the vectors completely results in replacing the computation in the `dotproduct` return by the final result value, which produces an even higher specialization benefit than eliminating the multiplications by zeros and the additions of zero that are performed for 90% of the vector elements (the input set consists of 90% zero-filled vectors, as described in Table 6.2).

Specializing both vectors instead of just one, as done by the human annotator, improves the speedups by up to a factor of 2.7. For the training input, the speedup improves from 6.9 to 9.0 and the largest speedup is seen for the reference input, which achieves a speedup of 18.3.

query

For the kernel `query`, Calpa found the same annotations as the human annotator. It specialized the query array data structure, and completely unrolled the query loop that iterates over the query array. However, unlike the human annotator, it chose the safe invalidation-based caching for the data base query array.

The different caching policies resulted in different speedups. The Calpa-derived annotations achieved a speedup ranging from 1.7 to 2.2. The best human annotations achieve speedups from 1.7 to 4.6, about twice as large as what is achievable with safe annotations on these inputs.

romberg

For `romberg` Calpa finds a superset of the annotations that were found by the human annotator. It chose to unroll the same loops and also specialized for parameter M , which determines the number of sub-intervals used in the Romberg integration. Again, as in `chebychev`, the driver routine calls `romberg` only with one fixed set of interval bounds a, b ; Calpa therefore chose to specialize them as well. The additional annotation of the interval bounds resulted in small speedup improvements, for instance, for the training input the speedup rose from 1.1 to 1.3.

6.4.2 *Effects of Setup Cost Estimation*

As mentioned in Section 5.4.4, estimation of the setup cost was added to the cost-benefit model to prevent an artificially low specialization cost estimate when the specialization of many variables would leave few dynamic instructions. While in many cases such annotations are likely to be suppressed because of excessively high caching or invalidation costs, when there are only few caching or invalidation points, not taking into account setup costs may produce bogus annotations. To check the effects of setup cost estimation in practice, I reran the annotation tool on the applications without the setup cost estimation. As it turns out, excluding the setup cost from the cost-benefit model resulted in the same annotations for all the applications in our benchmark suite. This means that the original cost-benefit design is sufficiently accurate.

6.5 *Profile Sensitivity*

Calpa computes its annotations based on profile information. Profiles are generated by running an instrumented application on a particular sample input. Consequently, the value and frequency profiles that Calpa uses to compute its annotations are input-dependent, and profiles obtained with distinct inputs may potentially result in distinct program annotations. While traditional feedback-directed optimizations also suffer from this problem, since run-time specialization uses value-specific

optimization to improve program performance, a variation in profiles may potentially have a larger impact on overall program performance. We would therefore like to prevent, whenever possible, overly tuning an application to a specific profiling input (with potential performance degradation for other inputs), and at the same time avoid missing optimization opportunities that are likely to be present across many, if not all, inputs.

6.5.1 Influence Factors

Profile information impacts Calpa’s generation of annotations in several ways:

1. the selection of the sets of procedures which should be annotated, based on their contribution to overall execution time;
2. the number of distinct values for a variable, which is used to estimate the number of code versions that have to be generated when the variable is specialized;
3. the value of loop bounds of loops that are considered for full loop unrolling, which determines the estimate of how many instances of the loop body will be generated by DyC.

Influence on Procedure Selection

The first of these factors is benign: if Calpa overestimates the contribution of a procedure it subsequently manages to improve, the overall performance impact will at worst be diminished. If it underestimates the contribution of a procedure, Calpa might miss an annotation opportunity. However, this is unlikely to have a significant performance impact, since Calpa analyzes all procedures that account for at least 3% of execution time in any of its execution time estimates.

Influence on Specialization Cost for Polyvariant Specialization

The second factor that directly influences specialization cost estimates may either result in excessive specialization when the profiling sample input shows significantly fewer distinct values than actually occur during execution of the DyC-optimized program on other inputs. However, while the actual values of a variable during execution may change across execution with different inputs, the property

that a variable takes only one, or a few values appears to be much more stable. This was the case for our benchmark suite, and was also observed to be the case by other researchers [CFE99]. Given this property, observing few values for a variable in one value profile would be a good predictor that it will also take on few values in another profile, which would make Calpa’s specialization cost estimate stable across inputs.

6.5.2 Influence on Code Size in Full Loop Unrolling

The specific value of a specialized variable can have an enormous impact on the specialization cost (and its estimate), when it determines the number of instruction instances that are generated for a loop body in full loop unrolling. As mentioned in Section 5.4.2, Calpa sets the specialization cost to ∞ , when the estimate for the number of generated instructions exceeds a predefined bound. If this bound is not exceeded for the profile input, but will be exceeded when executing the DyC-optimized program on another input, a program slowdown is likely to result because of deteriorated instruction cache behavior.

Conditional Specialization

While not currently implemented, one way to address this problem is *conditional specialization*, where the annotation causing the loop unrolling is guarded by a condition. As an example of how guard conditions might be generated automatically, consider the example shown in Figure 6.3. Figure 6.3(a) shows the annotation that specializes the `dotproduct` routine for the contents of vector `u`. By annotating both the loop induction variable `i` and the loop bound `size`, DyC is directed to fully unroll the loop, producing `size` copies of the loop body. This is likely to be beneficial only up to a certain point; once the number of instructions generated exceeds the processor’s instruction cache size, a performance degradation is almost certain to result.

In Calpa, DyC’s capability of conditional specialization can be used to prevent this from happening. In conditional specialization, a condition is used to guard an annotation, This causes DyC to produce two divisions downstream of the guarded annotation: one in which the variables mentioned in the annotation are specialized, and one in which they are not. Figure 6.3(b) shows how such a guard might look like for the `dotproduct` routine. The `make_static` annotation is guarded with a

test that compares the value of `size` with a threshold value; only when the vector size is below this threshold, the annotation is "executed". At run time, run-time specialization will only be performed when `size` is below this threshold, otherwise, the dynamic compiler will not be invoked and the generic dot-product routine will be executed. The actual value of `THRESHOLD` will depend both on the underlying processor and the size of the specialized code.

Since Calpa's cost-benefit model already estimates the number of generated instructions, it can be used to determine a good threshold value. Figure 6.3(c) shows the `dotproduct` in intermediate code representation. If `u`, `i`, and `size` are specialized, at most three instructions will be generated per loop body: the load of `v[i]`, the subsequent multiplication, and the add to `sum`. Therefore, we know that the number of instructions c is bounded by $3 * size$, i.e., $c \leq 3 * size$. Suppose our instruction cache size is 32K, i.e., it can hold up to 8192 instructions. We also want to give some part of the instruction cache to the dynamic compiler itself, and allocate at most, say, 75% of the instruction cache (= 6144 instructions) to dynamically generated code. Then we should set the threshold so that $c \leq 3 * size \leq 6144$, or $size \leq 2048$. Using this simple formula, Calpa can generate the bound for conditional specialization automatically, at least for the case of fully unrolled counting loops, which have been the most frequently unrolled loops in DyC's applications suite.

```

int dotproduct(int u[], int v[], int size) {
    int sum = 0;
    int i;

    make_static(u, size,i);
    for (i=0; i<size; i++)
        sum += u @ [i] * v[i];
    return sum;
}

```

Figure 6.3a: Unconditionally specialized routine; if `size` is big, unrolling the loop will produce too much code.

```

int dotproduct(int u[], int v[], int size) {
    int sum = 0;
    int i;

    if (size < THRESHOLD)
        make_static(u, size,i);
    for (i=0; i<size; i++)
        sum += u @ [i] * v[i];
    return sum;
}

```

Figure 6.3b: Guarded specialization: the runtime specializer will be invoked only when `size` is below `THRESHOLD`.

```

sum = 0
i = 0
L0: if i >= size goto L1
    t1 = u[i]
    t2 = v[i]
    t3 = t1 * t2
    sum = sum + t3
    goto L0
L1:
    return sum

```

Figure 6.3c: The routine in three address code representation, without the guard condition.

6.5.3 Sensitivity in the Benchmark Suite

For all the programs in our benchmark suite except `equake`, Calpa generated the same annotations regardless of the input that was used to generate the value profile. For `equake`, Calpa generated annotations only when the value profile was generated with the test input data set but not for the others.³

For the test input, Calpa chose to annotate the three procedures `phi0`, `phi1` and `phi2`, each of which is estimated to account for about 3% of overall execution time. Calpa decided to annotate the single argument `t` to the phi-routines as well as one other (global) simulation parameter value (`t0`). Based on these, each of the phi-functions computed a trigonometric expression. For instance, `phi0` looks as follows:

```
double phi0(double t) {
    double value;
    double t0 = Exc.t0;
    if (t <= t0) {
        value = 0.5 / PI * (2.0 * PI * t / t0 - sin (2.0 * PI * t / t0));
        return value;
    } else return 1.0;
}
```

By specializing `t` and `t0`, the call to `sin` can be memoized and the whole expression that computes `value` can be computed at specialization time. For the training and reference inputs, the expression `t <= t0` evaluated to false most of the time, so that the routine simply returned 1. The time savings obtained when the other path through the routine was taken was not enough to recoup the cache key check cost for `t` and `t0`, so that Calpa decided not to specialize.

This decision is justified by the actual execution times on the different inputs. Running `equake` annotated with the annotations generated from the test input on the test input, achieves a whole program speedup of 4.5% (the phi procedures account for a total of about 9% total execution time,

³For the training and reference input, only about 0.8% of the total execution time is spent in those routines, so Calpa by default does not even consider them for annotation. If, however, the annotation selector tool is run on these procedures, no annotations are produced for the training and reference input, i.e., the lack of annotations for these inputs is the result of different performance estimates for these profiles compared to the test input profile.

so the speedup cannot exceed 9.8%). However, when the annotated program is run on the training or reference inputs, it slows down by 23% and 44%, respectively.

6.6 Summary and Future Work

In this chapter I have evaluated the Calpa system by running it on a set of benchmark applications ranging from small kernels to mid-size applications of several thousand lines of code. With the exception of the `binary` kernel, Calpa was able to find specialization opportunities in all programs where such opportunities had been previously found by human programmers. In several, Calpa's annotations produced higher speedups than the best human annotations. In the case of `dotproduct` this was clearly due to a bad driver program that invoked the specialized routine in an atypical way, creating more specialization opportunities than would normally be found in practice. At the same time, however, this highlights Calpa's capability of detecting specialization opportunities based on actual program behavior, as evidenced in the profile information. Where Calpa's annotations differed from annotations written by humans the chief reason was safety: Calpa always chooses annotations that are guaranteed not to change program semantics regardless of input. Only in the case of `binary` Calpa's limited ability in recognizing opportunities for eager multi-way loop unrolling prevented it from producing any annotations at all.

While Calpa's annotations potentially depend on the inputs used for profiling, this does not necessarily occur in practice. The only application for which we found input sensitivity was `equake`. However, for the default training input no annotations were generated, which demonstrates that with sufficiently representative profile inputs, no overly specific annotations will be generated. For the kernels, overspecialization only occurred when overly simplistic driver routines were used.

An automatic way of generating conditional specialization guards is desirable to avoid program slowdowns in those cases where annotations are sensitive to the value profiles used to generate them. We have shown a method for automatic generation of guard conditions for simple counting loops that are unrolled completely. The generalization of this case with guard conditions for more complex loop structures and multi-way loop unrolling, is an interesting area for future research.

As mentioned in Section 6.4, Calpa would benefit from a pointer analysis that distinguishes structure fields. This would result in field-specific invalidation points for structures enabling, for

instance, the specialization of one field even though other fields of the same structure are frequently updated at run-time; `dinero` would benefit from such an improvement and with it, Calpa should be able to derive the full set of annotations that the human annotator was able to identify. Steensgaard [Ste96a] proposed an extension to his original algorithm that distinguishes fields. Even though in theory it can be considerably slower than the original algorithm, for typical programs it runs only insignificantly slower than the almost linear time original algorithm, so that it may be a good choice for a future version of Calpa, enabling, for instance, the generation of annotations for the `output` field that was missed in the annotation of `dinero` by the current version of Calpa.

Chapter 7

EXPLOITING DYNAMIC POINTER INFORMATION

Automatic dynamic compilation with Calpa and the underlying DyC system take advantage of the invariance or semi-invariance of variables and data structures to perform code specialization and value-specific optimizations that can achieve significant code speedups. There are, however, other run-time properties of programs that might be harnessed for run-time code improvement. One particular property that might be exploited beneficially is aliasing.

Two expressions are said to be *aliased* if they refer to the same storage location. In the C programming language aliases typically arise due to pointers; for instance, if pointer p points to global variable x , the expressions $*p$ and x are aliased. In the presence of aliases, the compiler has to make conservative assumptions when optimizing code, for instance, a variable may not be allocated to a register because of the use of a pointer that might potentially modify it, or alternatively, if the variable is allocated to a register, the register has to be reloaded from memory each time a store instruction may have changed the variable's value.

To ensure that correct code is generated, traditional compilers use static pointer analysis to determine when variables may be aliased. These analyses necessarily produce conservative approximations to the true aliasing relationships because of the approximate nature of standard data flow analyses, and because of additional imprecision caused by C's weak type system. For example, variables of integer type may contain pointer addresses which may cause the pointer analyses to assume aliasing relationships that are impossible in practice. Unfortunately, the conservative results of static pointer analyses may prevent the compiler from performing optimizations that would in fact be sound given the actual run-time pointer behavior.

To overcome this shortcoming, Bernstein et al. [BCM94] developed an approach targeted at inner loops over arrays whose accesses are reported to be aliased by a static pointer analysis. As a first step, their approach identifies the condition under which the aliasing occurs. Then, their algorithm produces two versions of the loop, one which is optimized assuming no aliasing, which

typically enables more aggressive optimization, and one version optimized assuming aliasing. At run time, the aliasing condition is evaluated to select the more aggressively optimized version when possible. They restrict their transformation exclusively to inner loops where optimization benefits are likely to be high, and they apply it only when the aliasing condition is loop-invariant, which ensures a low run-time cost for the selection test. With this approach, they were able to achieve speedups only on a few numerical SPEC92 benchmarks, although the achieved speedup was higher than a factor of two in one case.

While Bernstein et al. showed that there are potentially large benefits for exploiting run-time pointer information, it is not clear how often such opportunities arise in practice. If existing pointer analysis algorithms produce few false positive classifications of potential aliases, there may be only limited opportunity for run-time pointer information. To assess this potential, I compared the results of some well-known scalable pointer analyses with the actual run-time behavior of pointers, observed via instrumentation [MDCE01]. The results show that at run time pointer dereferences typically access only a few objects, often just one, whereas the static points-to sets computed by scalable static points-to analyses are typically one to three orders of magnitude larger. This gap indicates that there may be significant opportunity for Bernstein-style optimizations that take advantage of actual pointer behavior rather than suppressing any aggressive transformations a priori based on static analysis alone.

Another area where run-time pointer information may be useful in practice are programming understanding tools, such as program slicers. These tools have been of limited usefulness in practice, partly because of the imprecision caused by static pointer analysis, which they require for the detection of pointer-induced data dependences. To determine the potential improvement these tools may experience with better pointer information, I used Tumi to generate dynamic pointer information, and then used it in a slicing tool [MAEC02]. The results demonstrate that for a certain class of programs, slice size can be reduced significantly with dynamic pointer information.

The remainder of this chapter is organized as follows: Section 7.1 introduces the notion of dynamic points-to sets, and discusses how Tumi was changed to generate them. Section 7.2 discusses the programs for which dynamic points-to sets were generated and which were subsequently used in the slicing experiments. Section 7.3 presents some background on slicing and the Sprite slicing tool that was used in the experiments. Section 7.4 describes the methodology that was used in the

experiments, in particular, how dynamic points-to sets were generated, and the options and criteria that were used to generate slices. Section 7.5 presents our experimental results for the comparison of static and dynamic points-to sets and the slices obtained with them. Section 7.6 shows how to efficiently generate dynamic pointer information for function pointers, and Section 7.7 presents conclusion and an outlook on future work.

The work on dynamic points-to sets was done jointly with Manuvir Das. The application of the dynamic points-to sets to program slicing was done jointly with Darren Atkinson. For the latter, Atkinson implemented the required changes to the Sprite slicing tool, and the study design and experiments were done jointly.

7.1 Dynamic Points-To Sets

Often pointer analysis algorithms are formulated as points-to analyses, which compute for each pointer-valued expression the set of objects (variable, procedure symbol, or memory-allocation site) the expression may point to. To assess how close the results of static analysis algorithms are to actual run-time behavior (and consequently the potential for any transformations based on run-time pointer information), I modified the instrumentation framework presented in Chapter 4 to collect only information about the targets of pointer-valued expressions, i.e., for every pointer dereference, the objects that are accessed during execution are recorded. The whole set of objects accessed during program execution at a particular dereference point is called the *dynamic points-to set* for that dereference point. Any sound static points-to analysis must contain at least the objects in the corresponding dynamic points-to set, i.e., the dynamic points-to sets provide a lower bound on the (in general, not computable) optimal points-to set for a pointer dereference. Dynamic points-to sets are unsound in general; however, if input coverage is good, they are likely to be close to optimal. The gap between dynamic points-to set size and the size of static points-to sets produced by a static pointer analysis, conveys an indication of the precision of the static pointer analysis algorithm.

To compute the dynamic points-to sets, Tumi was changed to record which variable or data structure was identified by the address matching step (cf. Section 4.3.1) that is performed for each pointer access, instead of storing the value that was loaded or stored (as for value profiling). Instead of a value table, a simple set data structure is created for each program point that performs a pointer

load or store, or a call through a function pointer. The target id returned by the address matching routine is added to this set whenever the pointer operation is performed. Before the program exits, the contents of all these sets are written to disk, and in an offline pass, their target objects descriptors (cf. Section 4.3.1) are used to map the target ids back to the corresponding compile-time variable or procedure names, or malloc site. The dynamic points-to sets can then be compared to their static counterparts directly.

7.2 Workload

For the comparison of static with dynamic points-to sets and their application in program slicing, we chose to use applications from the SPEC 2000 benchmark suite, since SPEC benchmarks are of considerable size, cover a wide range of tasks (e.g., simulations, group theoretic computations, graphics, databases, word processing) and they are actually used in practice.¹ In addition, we used several benchmarks that have been employed by other researchers in the evaluation of their slicing studies [HRB90, LH99a, LH99b]. Table 7.1 shows the programs with their sizes and the number of executable lines (i.e., lines that actually perform some computation). The slices computed by the Sprite slicing tool that was used in our experiments include only executable lines of code; therefore, the slice sizes reported in Section 7.5 refer to the number of executable lines included in the slice. The last five programs use function pointers heavily (discussed in detail in Section 7.5.2), and are therefore listed together. The column labeled “slicing criteria” lists the number of criteria that were used for slicing an application; how these criteria were generated is discussed in Section 7.4.3.

7.3 Program Slicing

Since the dynamically observed points-to sets are generally much smaller than the results of scalable static points-to analyses, any program analysis that depends on pointer information might benefit from the reduced points-to set size. To exploit them for program optimization, it is necessary to ensure that any transformations performed on the potentially unsound dynamic data are correct, which will typically require a run-time check. Program analysis, however, is also used in software

¹Applications are submitted to the SPEC consortium and selected based on their relevance and representativity of actual computing practice.

Table 7.1: Sizes and descriptions of the programs used in the experiments. An executable line is any line of source code that performs a computation during runtime. In particular, declarations, blank lines, and comments are excluded. Italicized programs use function pointers heavily and are therefore listed together.

	Source Lines	Executable Lines	Reachable Functions	Executed Functions	Slicing Criteria	Origin	Description
art	1,270	545	22	18	837	SPEC 2000	image recognition, neural networks
equake	1,513	670	24	19	1,111	SPEC 2000	seismic wave propagation simulator
mcf	1,909	635	24	21	880	SPEC 2000	combinatorial optimization
bzip2	4,639	1,246	63	21	1,579	SPEC 2000	compression
gzip	7,757	1,864	62	26	1,546	SPEC 2000	compression
ispell	8,020	2,742	107	33	1,617	GNU (v3.1.20)	spell checking
parser	10,924	4,414	297	230	6,223	SPEC 2000	word processing
diff	11,755	3,285	110	27	2,110	GNU (v2.7)	file comparison
ammp	13,263	5,614	161	46	5,146	SPEC 2000	molecular dynamics
vpr	16,973	5,954	255	163	7,993	SPEC 2000	circuit placement and routing
less	18,305	4,371	328	117	1,879	GNU (v358)	text file viewing
twolf	19,748	11,304	167	104	13,816	SPEC 2000	placement and global routing
vortex	52,633	23,245	643	518	31,324	SPEC 2000	object-oriented database
<i>grep</i>	13,084	3,674	108	39	3,520	GNU (v2.4.2)	pattern matching
<i>find</i>	13,122	3,004	96	37	740	GNU (v4.1)	filesystem searching
<i>mesa</i>	49,701	21,069	770	130	7,270	SPEC 2000	graphics
<i>burlap</i>	49,845	16,608	189	123	5,293	FELT (v3.05)	finite element solver
<i>gap</i>	59,482	19,998	826	356	15,245	SPEC 2000	group theory interpreter

engineering tasks, some of which do not require soundness, for instance, *program understanding* tasks, that only try to help the user better understand program behavior.

One particular technique that has been suggested to aid programmers in such tasks is *program slicing*. A program slicer takes a *slicing criterion*, i.e., a pair of the form (variable, statement) (called the *criterion variable* and *criterion statement*, respectively) and computes either the set of statements that may have determined the value of the variable at that statement, called a *backward slice*, or the set of statements that may be affected by a change to the criterion statement, called a *forward slice*. Backward slicing may be used for debugging; other applications of slicing are software maintenance, testing, and reverse engineering [BLW96, FR01, GL91, TD01, Wei84]. With such applications in mind, a variety of program slicing tools have been developed, many of these for the widely used C programming language [AG98, BLW96, Gra, HC98, Tip95].

Although in theory program slicers make several program understanding tasks easier, their usefulness in practice has been limited, because existing program slicers frequently produce slices that are quite large. Various reasons have been proposed to explain this phenomenon. For example, some slicers use context-insensitive data-flow analyses, which analyze procedures equally regardless of their call site. As a consequence, data-flow information for multiple calls to the same function is shared, resulting in more imprecise analysis results, which may lead to a larger slice size.

Another culprit may be the pointer analysis algorithms used in the slicing tool. Flow-sensitive and context-sensitive algorithms potentially produce the most precise results, but due to their time complexity ($O(n^3)$ or worse, where n is the number of lines of code) they generally do not scale well, limiting their applicability to relatively small programs. Ironically, slicing would be most beneficial for large programs that cannot be easily understood without the use of tools. Therefore, existing slicers typically use pointer analyses that trade precision for better performance to enable slicing of complex programs. These analyses are typically not fully flow- or context-sensitive [Das00, SH97b, Ste96b, WL95].

The pointer imprecision problem is particularly severe for C programs, which use pointers extensively to simulate call-by-reference semantics in procedure calls, to emulate object-oriented dispatch via function pointers, to avoid the expensive copying of large objects, to implement list, tree, or other complex data structures, and as references to objects allocated dynamically on the heap. Therefore, imprecise pointer analyses will result in conservative assumptions about data dependences in a pro-

gram, contributing to a larger slice size.

The pointer analysis used by existing slicers is usually implemented as a points-to analysis, and the computed points-to sets are used by the subsequent data-flow analysis to resolve pointer dereferences in slice computation. In practice, the dynamic points-to sets are significantly smaller than the sets produced by scalable points-to analyses (cf. Section 7.5.1). Moreover, they are a lower bound for the results of any sound static analysis, so that they can be used to obtain an upper bound on the potential improvement of slice sizes that might be achieved by using more precise pointer analysis algorithms in slicing.

7.4 Methodology

This section describes how the dynamic points-to sets were generated, how slices were computed for static and dynamic points-to sets, and what slicing criteria were used for the slicing experiments.

7.4.1 Generating Dynamic Points-To Data

To obtain the dynamic points-to sets for the applications in this study, we used the SPEC-provided test inputs, which are meant to exercise the programs' functionality. We chose to use the test inputs, since they allow us to gather the points-to sets faster. We also found that running the applications on the larger reference data sets produced virtually unchanged points-to data, since the reference inputs appear to be executing the same parts of the application, only more often. For the non-SPEC programs, we either used the examples and test suites provided with the applications or performed representative tasks, such as searching through all files in a directory hierarchy or scanning a large volume of text. The dynamic points-to sets are generated for each program point that performs a pointer-dereference or call through a function pointer, i.e., they are flow-sensitive. For some of our slicing experiments, we produce flow-insensitive dynamic points-to data by merging the dynamic points-to sets for a pointer across all its dereference points.

Tumi's instrumentation slowed down the applications by one to two orders of magnitude, causing the test inputs to finish within minutes or hours, well within the time scale of computing actual slices. Moreover, since the points-to data can be reused across different slices of the same application, the cost of generating the dynamic points-to data can be amortized.

7.4.2 Slicing Methodology

We modified version 3.0 of the *Sprite* program slicing tool [Atk99, AG98] to compute slices either with dynamic or static points-to data. *Sprite* is a research prototype developed for slicing C programs. Currently, *Sprite* computes only backward program slices.

Sprite first constructs the control-flow graph (CFG) of the program. The CFG consists of basic blocks of three-address statements, each of which represents a single computation such as a simple addition or pointer dereference. Steensgaard's static points-to analysis [Ste96b] is then performed over the CFG to compute equivalence classes of memory locations that are used as points-to sets during slice computation. Although *Sprite* can perform a slight modification of the points-to analysis that distinguishes the fields of structures, this modification was not used since previous results found it yielded little improvement in program slices and negatively impacted performance during slicing [AG98]. To compute a program slice, *Sprite* computes a maximum-fixed-point solution to the data-flow equations given in [AG98] using an iterative, convergence algorithm. After slice computation is complete, *Sprite* reports the number of source code lines included in the slice and can highlight the included lines within a user interface.

Most default options to *Sprite* were used during the experiments. The sole exception was that the names of functions that performed custom memory allocation (e.g., `xmalloc` in `find`) were specified to increase the precision of the points-to analysis. We generally computed context-insensitive slices in our experiments, because previous work had shown that context-sensitive slices are not much smaller and require significantly more time to compute [AG96, BAG00]. However, to rule out context-sensitivity as a factor influencing our results, we also computed some context-sensitive slices.

7.4.3 Slicing Criteria

Ideally, slicing criteria, i.e., pairs of the form `(statement, variable)`, would be chosen that might be used by a software engineer during debugging (since *Sprite* computes a backward program slice). However, since we are unfamiliar with the benchmarks, we instead elected to exhaustively generate slicing criteria for each program. This ensures results that are not biased because of a

particular choice of slicing criteria.² We then restricted the initial slicing criteria to come from only those functions that were actually executed during some instrumentation to ensure the availability of dynamic pointer information.

Table 7.1 shows the number of possibly reachable functions in each program and the number of functions executed during the instrumentation runs, along with the number of criteria used in the experiments. The percentage of executable functions of the total (statically) reachable functions varies widely for the programs, indicating that the code coverage of the inputs is sometimes quite poor (e.g., for `mesa` for which only about 17% of the reachable functions were executed), indicating that the test cases provided with the applications need improvement. The set of reachable functions was constructed using a call graph extractor that uses Steensgaard’s [Ste96b] points-to analysis to account for the effects of function pointers.

7.5 Results

This section first compares the dynamic points-to sets generated for the applications in our benchmarks suite to their static counterparts, and then discusses the effects of their application in our program slicing experiments.

7.5.1 Points-To Sets Results

In our dynamic points-to experiments, we compared the dynamic points-to sets to the static points-to sets produced by two well-known, scalable points-to analyses, Steensgaard’s [Ste96b] almost linear-time algorithm, and Das’ [Das00] improvement of it, the One-Level Flow algorithm. Figure 7.1 shows the average static and dynamic points-to set sizes for the applications.³ The averages were computed as arithmetic means over all pointer dereference points in the program that were executed. The figure demonstrates that in general, the One-Level Flow algorithm (labeled OLF) is considerably more precise than Steensgaard’s algorithm; only for `mcfl` the two produce almost identical results. In addition, the dynamic points-to sets are typically one to three orders of magnitudes

²Other slicing work has typically glossed over this point; for instance, [LH99a] and [SH97a] do not describe in detail which criteria were used in their experiments.

³For application `bur1ap`, the average across all executed dereference points for the Steensgaard algorithm is missing because of a bug in Suif’s C backend that occurred during instrumentation.

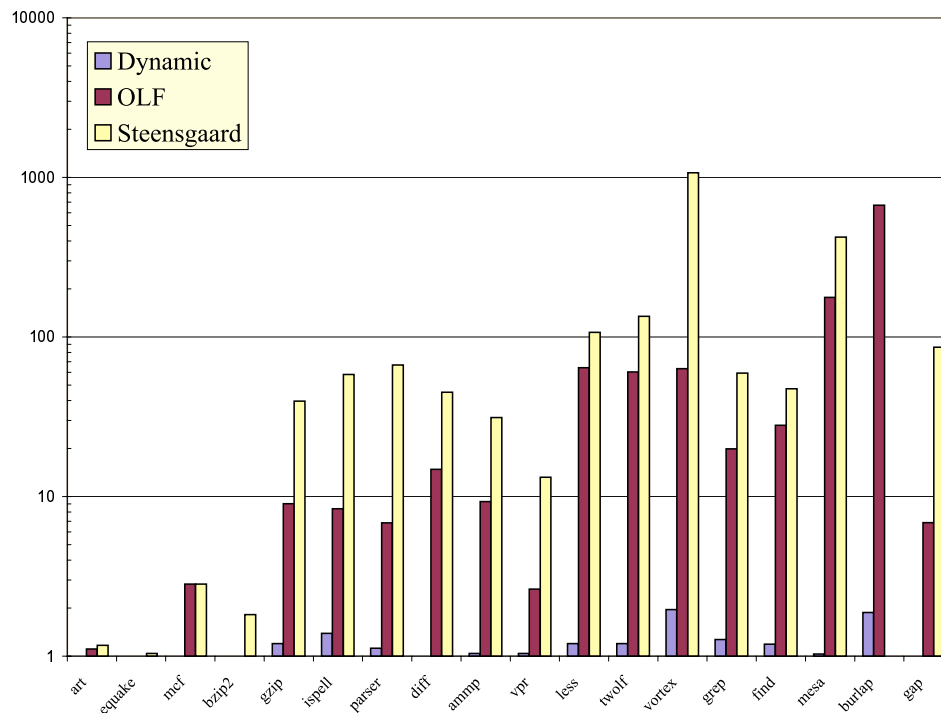


Figure 7.1: This graph shows the average points-to set sizes for the One-Level Flow (labeled OLF) and Steensgaard’s algorithm and for the dynamic points-to sets. Generally, OLF is significantly more precise than Steensgaard’s algorithm, and the dynamic points-to sets are up to two orders of magnitude smaller than their static counterparts, with average sizes close to one.

smaller than their static counterparts, with the dynamic averages being close to one.

Since the average dynamic points-to set sizes are close to one, there must be a large number of singleton sets; their percentage of all points-to sets is shown in Figure 7.2.⁴ For the applications `art`, `equake`, `mcf`, `bzip2`, and `diff` all dynamic points-to sets were singletons; for the other applications the percentage of singleton sets ranged from 77% for `ispell` to 99.9% for `gap`.

While the dynamic points-to sets are in general much smaller than the static points-to sets produced by the One-Level Flow algorithm, for some applications the OLF algorithm produces sets that are almost as small. Since dynamic points-to sets are a lower bound on the results of any sound

⁴Note that in the case of a malloc site one compile-time representative may correspond to multiple run-time instances.

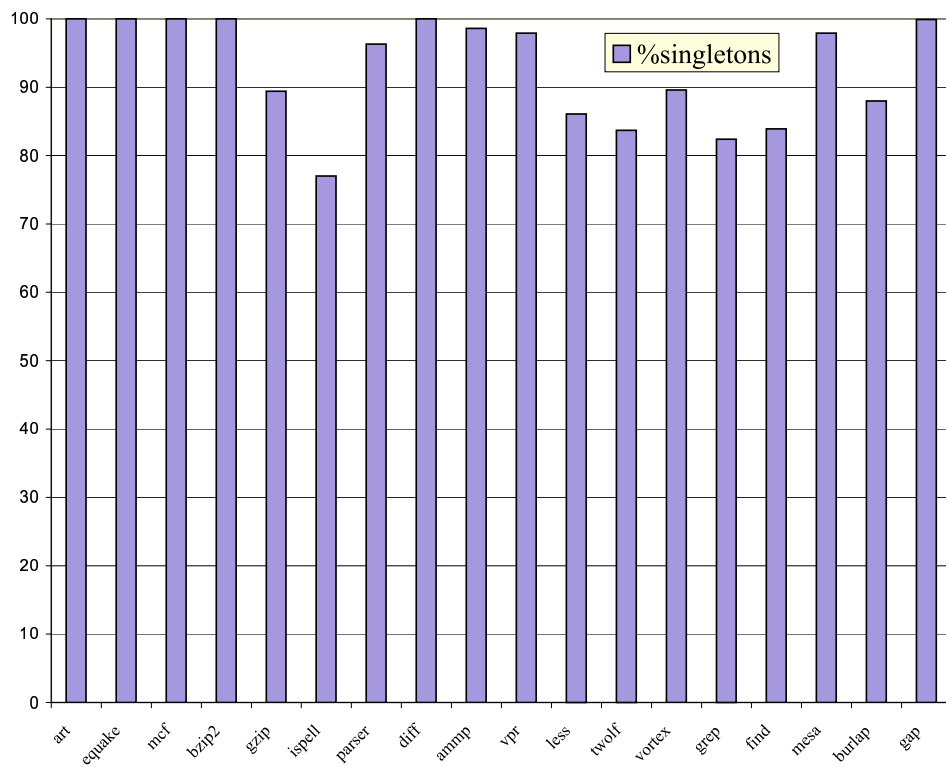


Figure 7.2: This graph shows the percentage of dynamic points-to sets that were singletons.

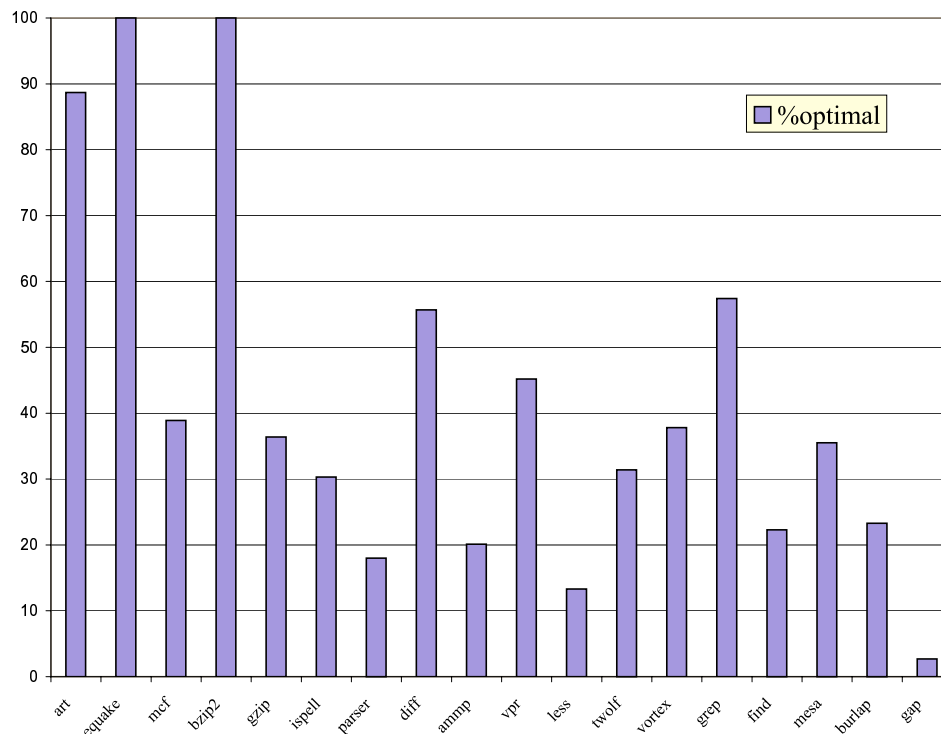


Figure 7.3: This graph shows the percentage of dereference points where the static points-to sets computed by the OLF algorithm were equal to the dynamic points-to sets, i.e., where the static points-to sets must be optimal.

points-to analysis, we can infer that a points-to set is in fact optimal when the static and dynamic sets are identical. Figure 7.3 shows the percentage of dereference points for which the points-to sets computed by OLF algorithm were identical to their dynamic counterparts. For `equake` and `bzip2` the OLF algorithm produces optimal results for nearly all dereference points. This is because they use pointers only to create and access arrays, not to create and manipulate complex structures. For the other applications, the number of definitely optimal sets ranges from 20% to 90%, showing that the quality of the results of a static points-to algorithm greatly varies from program to program. For all programs, the percentages of optimal static points-to sets have to be read as lower bounds, since the dynamic points-to sets may be unsound, so that some static sets may in fact be optimal despite being larger than their dynamic counterparts.

7.5.2 Slicing Results

Data-Flow Analysis

We measured two quantities during slice computation: dereference size and data-flow set size. The dereference size is the size of a pointer's points-to set at the time of its dereference. When a statement of the form $*p = x$ or $x = *p$ is visited during slicing, the size of the dereference set of p is recorded by Sprite. (Since Sprite uses an iterative algorithm, a single program point may be visited many times; however, the points-to set and hence the dereference size do not change.) We used this quantity to measure the number of data-flow facts (e.g., variables) present at a program point. We also measured the size of the incoming data-flow set of a basic block. Upon visiting each block, the size of the set is recorded by Sprite. This quantity gives an indication of the amount of data-flow information being propagated during analysis.

We then computed averages over all slices of these two quantities for each program in our test suite. Figure 7.4 shows the improvements in each of these measurements from using the flow-insensitive dynamic points-to data. The improvements from using flow-sensitive points-to data are similar (within 1–5% for the dereference size and 1–2% for the data-flow set size) and are not shown.

The reduction in dereference size is typically an order of magnitude or more, ranging from a factor of only 1.4 for `equake` to a factor of close to 700 for `mesa`. However, the reduction in data-flow set size is less, sometimes substantially so. This drop implies that although we have introduced fewer data-flow facts into the analysis at statements involving pointers, that decrease does not yield an equal decrease in the amount of data-flow information being propagated during analysis.

Slice Size

Table 7.2 presents the average slice size for each program, including the results obtained using flow-sensitive dynamic points-to data. The data shows that using the flow-sensitive points-to data instead of the flow-insensitive data has virtually no effect on slice size. Since for the majority of the dereference points the flow-sensitive dynamic points-to sets were identical to the flow-insensitive sets, this was unsurprising. Furthermore, this indicates that (at least for the applications in our benchmark suite) there may be limited benefits of using flow-sensitive pointer analysis in general, which appears to be consistent with the way pointers are typically used in C programs (passing

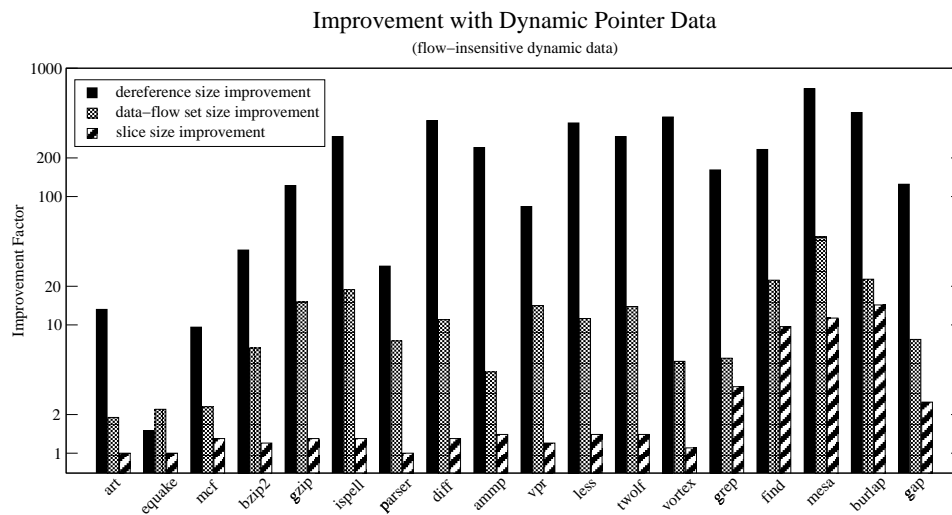


Figure 7.4: Improvement in average dereference size, set size, and slice size for slices computed using dynamic data.

pointers to large structures, for instance).

Figure 7.4 presents the improvement in average slice size when using dynamic pointer data. The data shows that our applications fall into two categories. For the first category, the improved pointer data results in only insignificant improvement. All of the applications in the second category, however, which comprises all the applications that use function pointers heavily (`grep`, `find`, `mesa`, `burlap`, and `gap`), showed a considerable reduction in slice size.

Slices of Programs With Little Function Pointer Use

In order to explain the lack of improvement for programs with little function pointer usage, we wanted to isolate and eliminate factors, such as context-sensitivity and control dependences, that might influence slice size. As discussed in Section 7.4.2, previous work indicated that increasing context-sensitivity yields only a small improvement in slice size. However, context-sensitivity and points-to set size are not orthogonal, and any such improvement could be magnified by using more precise points-to data. Therefore, we performed slices with increased context-sensitivity as prac-

Table 7.2: Average number of lines in a slice for slices computed using the static and dynamic points-to sets.

	Static	Dynamic (flow-insensitive)	Dynamic (flow-sensitive)
<i>art</i>	59.6	57.1	57.1
<i>quake</i>	168.4	164.8	164.8
<i>mcf</i>	56.8	45.3	45.3
<i>bzip2</i>	73.0	58.5	58.5
<i>gzip</i>	54.0	42.0	42.0
<i>ispell</i>	242.2	185.5	185.5
<i>parser</i>	195.9	186.9	186.7
<i>diff</i>	228.3	171.2	171.2
<i>ampp</i>	339.0	247.0	247.0
<i>vpr</i>	117.0	100.5	100.3
<i>less</i>	536.9	394.3	393.8
<i>twolf</i>	335.6	237.9	237.9
<i>vortex</i>	3,449.3	3,240.3	3,240.3
<i>grep</i>	527.8	183.2	183.2
<i>find</i>	460.8	47.4	45.7
<i>mesa</i>	3,267.3	288.3	288.3
<i>burlap</i>	5,291.6	369.6	369.4
<i>gap</i>	7,758.1	3,133.5	3,006.7

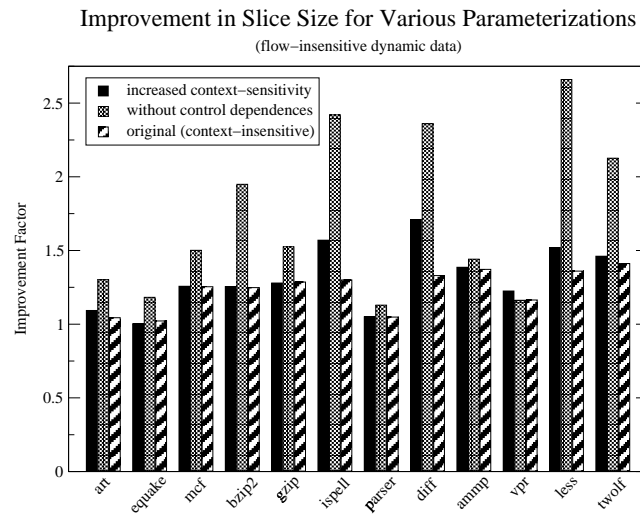


Figure 7.5: Improvement in average slice size for slices with increased context-sensitivity, for slices computed without control dependences, and for context-insensitive slices.

tical.⁵ Comparisons between the improvements in slice size for our original (context-insensitive) slices and the slices with increased sensitivity are shown in Figure 7.5. As the figure demonstrates, increasing context-sensitivity has little effect on the improvement in slice size. In fact, the slices with static data and slices with dynamic data generally improved about the same when context-sensitivity was enabled.

Another possible factor that might explain the general lack of improvement in slice size for the programs in the first category are control dependences. For example, more precise points-to data might eliminate a data dependence between two statements A and B , but A may still be included in the slice because B is control-dependent upon A . To assess the impact of control dependence on our slices, we modified Sprite to ignore control dependences when computing slices. The effects of ignoring control dependences on slice size are shown in Figure 7.5. The graph demonstrates that the slice size reduction resulting from slicing with dynamic pointer data was only slightly higher

⁵We were able to perform fully context-sensitive slices only for the smaller applications. For the medium-sized applications we were able to compute 2-CFA results (i.e., context-sensitive for call depths of up to 2 [Shi91]). For *vortex* even computing 1-CFA results ran out of memory.

when ignoring control dependences. Five of the applications (`bzip2`, `ispell`, `diff`, `less`, and `twolf`) show significant improvement; however, the factor of improvement is still much less than what might be expected when using points-to data that is 10 to 100 times better. Therefore, although control-dependences have some effect on any improvement that can be gained from better pointer information, they cannot totally account for any lack of substantial improvement in all of the applications in the first category.

Since neither context-sensitivity nor the effects of control dependence explain the limited improvements gained by using dynamic points-to data for the applications with little function pointer use, we decided to look at the data dependences that are present regardless of the quality of pointer information. Therefore, we modified `Sprite` to construct a data-dependence graph that could be used to compute a program slice. (Ordinarily, `Sprite` uses an iterative algorithm to compute a maximum fixed point solution as discussed in Section 7.4.2.) In the data-dependence graph, an edge links a use of a program variable to definitions that reach that use. Once the graph is constructed, a program slice can be computed (ignoring control dependences) by simply performing graph reachability [HRB90]. Smaller points-to sets should lead to fewer dependences between statements and therefore fewer edges in the graph. Figure 7.6 shows the number of edges in the data-dependence graph computed using both the static and dynamic points-to data.⁶ The number of dependences decreases significantly for most programs. Only `art` and `equake` show little reduction, which is not unexpected given that they use pointers only to create and access arrays, not to create and manipulate complex structures.

Not all edges in the data-dependence graph are due to the effects of pointers. Direct dependences are those dependences that are *not* induced by pointer dereferences. These edges are always present regardless of the precision of the points-to sets. Figure 7.6 also shows the number of direct dependences between statements. Figure 7.7 shows this same data but with edges classified as direct edges, dynamic pointer edges, and edges present only when using the static data.

As the figures indicate, for the smaller programs such as `gzip`, the majority of the dependences are direct. For the medium-sized programs such as `vpr`, approximately 33% of the dependences are direct. Consequently, any benefits from the more precise points-to data are immediately dimin-

⁶`vortex` is not shown in the figure because the computation of the data dependence graph ran out of memory. It is also omitted from Figure 7.7 for the same reason.

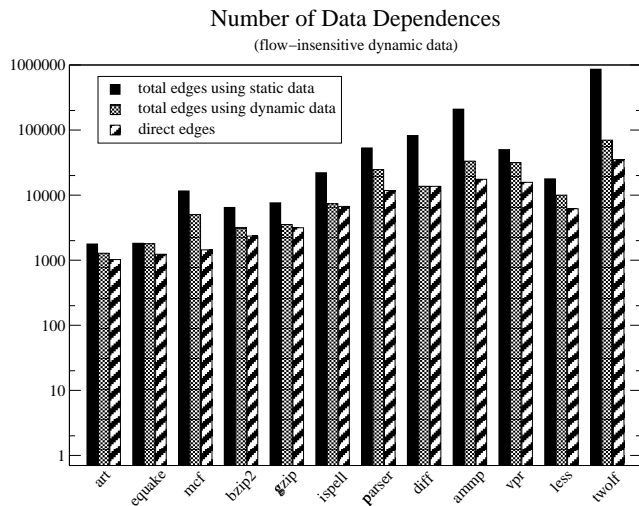


Figure 7.6: Number of data dependences computed using static and dynamic points-to data. Also shown is the number of direct (non-pointer-induced) dependences.

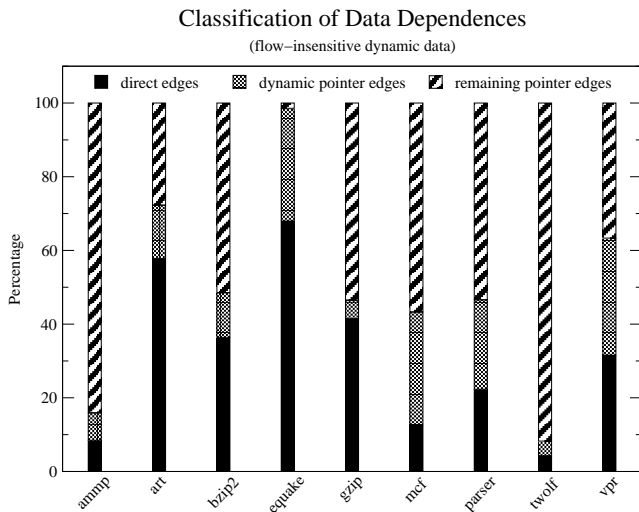


Figure 7.7: Classification of data dependences showing the number of direct edges, dynamic pointer edges, and remaining pointer edges (additional pointer edges present using the static points-to data but not present using the dynamic points-to data).

ished by Amdahl’s Law [Amd67]. Amdahl’s Law states that regardless of how much a part of a program that accounts for a fraction f of the execution time is improved, the overall speedup will never exceed a factor of $1/(1 - f)$. Similarly, regardless of how much we improve pointer-induced data dependences, this improvement will never exceed the limit imposed by the fraction of direct dependences present in the program. For instance, for `vpr` the data dependence edge improvement could be at most a factor of 3.2 even though its average dynamic points-to set is 100 times smaller than the static points sets. Since the data dependence edges largely determine the final slice, slice size improvement through better pointer information is ultimately limited by the fraction of direct dependences present in the program. Our results show that for the C programs in our benchmark suite, which we believe to represent “typical” C programs, direct dependences make up a large fraction of all data dependences. Consequently, even our optimal (or optimistic) pointer information improved slice size only insignificantly.

Slices of Programs with Heavy Function Pointer Use

For the programs in our benchmark suite that use function pointers heavily, we found that slices with dynamic data decreased by a factor of 2.5 for `gap` to 14.3 for `burlap`. To ascertain that this improvement is in fact due to improved function pointer data, we applied dynamic pointer information selectively in the following way. In addition to the slices with dynamic data for all pointers, we computed slices where we used dynamic data only for the function pointers and static data for the pointer variables, and slices where we used static data for the function pointers but dynamic data for variables. As shown in Figure 7.8, using the dynamic data only for the function pointers accounts for 48% to 91% of the improvement achieved by using dynamic data for all pointers. On the other hand, using the dynamic data only for the variables achieves only little reduction in slice size, demonstrating that most of the benefit derives from the improved call graph.

To estimate how much of the additional improvement is due to the optimistic nature of our dynamic pointer data, we used the following simple static techniques to obtain a better bound for the possible improvement due to better function pointer data. First, we enabled the filtering of the points-to sets for function pointers based on the their prototypes, i.e., all procedures in a points-to set whose prototype do not match the required prototype at the call site are eliminated from the

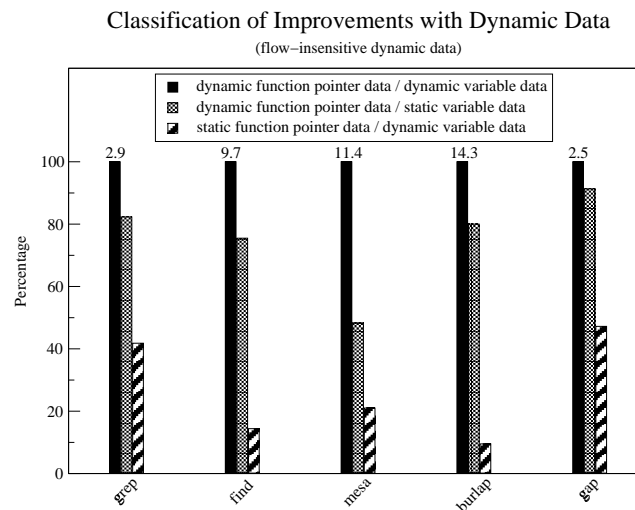


Figure 7.8: Classification of improvements in slice size for slices using dynamic data. For each application, the factor of improvement using dynamic data for both function pointers and variables is shown.

static points-to sets.⁷ For cases in which prototype filtering failed to reduce the points-to set size, we examined the source code of each application by hand to determine the approximate points-to sets for function pointers. To specify this information to Sprite, we specified a lexical pattern for filtering the static points-to data. For example, for `find`, we specified a pattern indicating that any call through a function pointer named “`parse_function`” resolved to any function whose name began with “`parse_`”. Table 7.3 shows the resulting slice sizes using prototype or lexical filtering and the slice sizes for applying dynamic data universally, and selectively to only function pointers.

With the exception of `gap`, for which we were unable to come up with good lexical filters because of the complexity of and our unfamiliarity with the program, we found that the slice sizes obtained with dynamic function pointer data were generally closer to the sizes resulting from filtered static points-to data than to the much larger slices obtained by using the static points-to data alone. For instance, for `burlap` the average filtered slice size was 1,128, whereas the static slice size was 5,292, and the slice size with dynamic function pointer data was 461, i.e., the slices with static data were

⁷This is sound for programs obeying the ANSI C rules. In general, however, it may be unsound.

Table 7.3: Slice sizes for slices with static data, prototype or lexically filtered static data for function pointers, dynamic data applied to function pointers only, and dynamic data applied to both function pointers and variables.

	Static	Filtered	Dynamic (func. ptrs.)	Dynamic (all)
grep	527	223	222	183
find	460	253	63	47
mesa	3,267	639	596	288
burlap	5,292	1,128	461	370
gap	7,758	7,747	3,433	3,133

on average 4.7 times larger than the filtered slices, but the slices with dynamic function pointer data were only a factor of 2.4 too optimistic. For `grep` and `mesa` the dynamic slices were particularly close to the results obtained with filtering, indicating that the dynamic slices are not too optimistic. For `find` the slices with dynamic function pointer information turned out to be very optimistic. The reason is the poor code coverage of the test cases provided with the `find` tool (Table 7.1 shows that only 37 of the 96 reachable functions were executed), so that of the potentially called functions at call sites only a few were exercised when gathering the dynamic points-to data. Since good test cases that exercise all parts of a program should be part of any sound software development practice, we expect that using dynamic points-to data for function pointers will work well in practice, as long as good test cases for the pointer data generation are available.

7.6 Efficient Generation of Dynamic Points-to Sets for Function Pointers

Since the results in Section 7.5.2 show that slices of programs that include many calls through function pointers can be considerably improved by using dynamic points-to data for the call sites, we would like to be able to gather points-to sets for function pointers with minimal slowdown, much lower than the one to two orders of magnitude degradation generally caused by Tumi’s instrumentation. Fortunately, unlike run-time addresses of variables, procedure addresses do not change at run time. Therefore the expensive mapping from run-time addresses to compile-time names (performed while the program is executing) is not necessary. Instead, it suffices to capture only the addresses of

the functions that are invoked at call sites that use function pointers.

To instrument these call sites, I created a lightweight instrumentation version of Tumi that collects only the run-time addresses of function pointers and does not perform any mapping from addresses to compile-time names. Instead, it stores only the addresses of function pointers in a per-call-site hash-table. When the program finishes, the contents of the hash tables are saved to disk, and later mapped to procedure names (using, for instance, the Unix tool `nm`) to obtain the points-to sets for the executed call sites.

This lightweight instrumentation resulted in much smaller slowdowns, ranging from 0.6% for `mesa` to 30% for `gap`, with a geometric mean of 10.3%. These degradations are comparable to the slowdowns imposed by standard profiling tools such as `gprof` or `pixie`. With this technique, therefore, function pointer data can be collected efficiently with minimal run-time overhead, making it a practical technique for the collection of dynamic function pointer information.

7.7 Conclusions and Future Work

In this chapter, I have presented a comparison of points-to information produced by static analyses and actual dynamically occurring behavior. We found that while static points-to sets are on the order of tens or hundreds of objects per dereference, even for the best scalable algorithm, the actual dynamically occurring sets are much smaller, with 97% of the sets being singletons, and average sizes close to one. The large gap between run-time pointer behavior and the results of scalable static pointer analyses presents an opportunity for the exploitation of run-time pointer information.

While Bernstein et al. [BCM94] have already looked at exploiting dynamic alias information to improve program performance for inner loops, a more thorough study is required to determine how much optimization opportunity for run-time pointer information arises in practice. A limit study to determine this opportunity, is an interesting area for future research.

This chapter also looked at applying run-time pointer information in software engineering tasks, by modifying a program slicing tool to work with dynamic points-to sets. We found that the effects of more precise dynamic points-to information on slice size were bimodal. Programs with many calls through function pointers experienced significant improvements in slice size, while applications with no or little function pointer usage showed only minor slice size reductions.

Since programs with heavy function pointer use experience significant slice size reductions, and we also show that dynamic function pointer data alone can be collected with little overhead, using dynamic points-to data in program slicing may be a practical technique to improve slices for programs that use function pointers frequently. Moreover, this technique may be useful for slicing of object-oriented programs as well, since object-oriented dispatch shares some of the characteristics of function pointer calls in C. Since programs are increasingly written in object-oriented languages, exploring the usefulness of dynamic points-to sets in slicing for these languages, may be a fruitful area for future research.

Since little improvement was found for programs with few calls through function pointers, advances other than improved pointer analysis may be necessary in order to improve the quality of slices. Since algorithmic improvements like context-sensitive slicing or optimal points-to information showed little general improvement of slice sizes, requiring the user to make certain assertions about program properties, e.g., that the source code observes ANSI type rules, may turn out to be the best (maybe only) way towards practically useful program slicing in this case.

Chapter 8

CONCLUSIONS

As traditional compile-time optimizations are becoming less effective because of changes in software and hardware technology, many research efforts have explored various forms of run-time optimization to either achieve high performance or to enable the use of programming languages and paradigms that would incur a significant run-time overhead in the absence of special run-time transformations.

One promising run-time transformation that can both improve the performance of existing programs and encourage the use of abstraction in programs is run-time specialization. Unfortunately, while it can sometimes improve the performance of programs considerably, its adoption as a standard program optimization technique has been impeded by the considerable effort that run-time specialization systems typically require from their users.

This thesis has demonstrated how to overcome this major obstacle and has made several contributions to the field of dynamic program optimization and analysis. Foremost among these contributions, it has shown how annotation-driven run-time specializers can be automated, obviating the need for any user intervention apart from running the applications on sample inputs. Automation is achieved by a combination of run-time value-profiling, which produces crucial run-time information about the application, and a compile-time analysis that uses the run-time information to evaluate optimization opportunities for their costs and benefits, choosing the overall best optimization strategy.

The viability of this approach is validated with a concrete prototype implementation that produces annotations very close to what a human might choose, but in a fraction of the time. At its heart lies a cost-benefit model, the first detailed model of run-time specialization costs and benefits. Another pivotal element is its candidate static variable set analysis, which effectively reduces the number of optimization choices that have to be evaluated with the cost-benefit model without any loss in annotation quality.

While exploiting run-time information is crucial for automating run-time specialization, its use-

fulness goes much further. In this thesis, I extended the idea of exploiting run-time information to the software engineering domain by showing how program slicing can benefit from dynamic pointer information. By leveraging our original profiling infrastructure, we have been able to identify which class of programs might benefit from improved pointer information and have provided an upper bound on the reduction in slice size achievable with it.

8.1 Future Work

This section describes avenues for future research, starting with addressing some of Calpa's current shortcomings, and then discussing some more open-ended directions.

8.1.1 Addressing Calpa's Shortcomings

There are several shortcomings of the current prototype system that need to be addressed in order to improve its general usability. Reducing value profiling times would greatly improve the applicability of our techniques. As a first step, the profiling slowdown could be reduced by eliminating redundant calls to data capturing routines. Currently, Tumi inserts calls to data capturing routines for variable uses even when static analysis can determine that the variable's value has not changed since the previous instrumentation point where its value was captured last.

In addition to speeding up Calpa, improving the quality of the annotations it generates would be beneficial as well, potentially resulting in higher speedups for code that is currently annotated or identifying specialization opportunities that are currently missed. One opportunity for improving the quality of Calpa's annotation can be found in more precise pointer analysis, namely by using an analysis that distinguishes structure fields. Another would improve the accuracy of the cost-benefit model. Currently, instruction costs are fixed in the model, in particular, memory operations are always assumed to hit in the cache; when they don't, Calpa's cost-benefit model may underestimate the benefit obtainable from the elimination of static instructions. This might be improved by exploiting the Alpha processor's performance counters during profiling to obtain empirical measures for the latencies of load instructions in the application, and using the observed values in the cost-benefit model instead of the hard-wired constants. This would also improve the identification of critical paths that may in part depend on the instruction latencies of memory instructions.

8.1.2 *Architectural Support for Run-time Specialization*

Since run-time specialization performs its program transformation during program execution, it is only beneficial when the benefits of its transformations outweigh their run-time costs. Consequently, lowering the cost of any run-time operation can potentially increase the applicability and improve the performance of run-time specialization. One particular way to tackle this cost-benefit tradeoff, consists of making necessary run-time operations, such as code cache management operations, or invalidation check costs, cheaper by providing hardware support for them.

To lower invalidation checking costs, for instance, the processor architecture might provide a set of special address range registers, which would be loaded (by compiler-generated code) with the address range of data structures used in dynamic optimization. Whenever the address of a store instruction falls into the indicated address range, the processor would set a special status flag, which could be tested by the dynamic compiler to trigger code generation. This would be much cheaper than current software mechanisms that rely on the insertion of checking code at every potential modification point (cf., Section 5.1.1). Exploring this idea and other mechanisms for an efficient management of the run-time code cache, has the potential making run-time specialization profitable where its use is currently prevented by excessively high run-time costs.

8.1.3 *Dynamic Program Analysis*

Another interesting, though more speculative, line of research is the investigation of how static and dynamic analysis might be joined to overcome restrictions to optimization imposed by imprecise results of purely static analysis. The goal would be to find out how general data flow analyses could be complemented by the dynamic observation of data flow facts, in such a way that program transformations would be less restricted. I envision a combined static-dynamic program analysis that aggressively optimizes code based on dynamically observed data flow facts, yet maintains correctness by generating simple run-time checks to fall back to more conservative transformations when aggressive code optimization would violate program semantics.

BIBLIOGRAPHY

- [ACSE99] Jonathan Aldrich, Craig Chambers, Emin Gun Sirer, and Susan Eggers. Static analyses for eliminating unnecessary synchronization from Java programs. In Agostino Cortesi and Gilberto Filé, editors, *Static Analysis*, volume 1694 of *Lecture Notes in Computer Science*, pages 19–38. Springer, 1999.
- [AFG⁺00] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 47–65, Minneapolis, MN, USA, October 2000.
- [AG96] Darren C. Atkinson and William G. Griswold. The design of whole-program analysis tools. In *Proceedings of the 18th International Conference on Software Engineering*, pages 16–27, Berlin, Germany, March 1996.
- [AG98] Darren C. Atkinson and William G. Griswold. Effective whole-program analysis in the presence of pointers. In *Proceedings of the 6th ACM International Symposium on the Foundations of Software Engineering*, pages 46–55, Lake Buena Vista, FL, USA, November 1998.
- [Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the AFIPS 1967 Joint Computer Conference*, pages 483–485, Atlantic City, NJ, USA, April 1967.
- [And94] Lars O. Andersen. *Program Analysis and Specialization for the C Programming Language*. Ph.D. dissertation, University of Copenhagen, DIKU, May 1994.
- [ASG97] Andrew Ayers, Richard Schooler, and Robert Gottlieb. Aggressive inlining. In *Pro-*

ceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation, pages 134–145, Las Vegas, NV, USA, June 1997.

- [ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, USA, 1986.
- [ATCL⁺98] Ali-Reza Adl-Tabatabai, Michał Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast, effective code generation in a just-in-time Java compiler. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 280–290, Montreal, Quebec, Canada, June 1998.
- [Atk99] Darren C. Atkinson. *The Design and Implementation of Practical and Task-Oriented Whole-Program Analysis Tools*. Ph.D. dissertation, University of California, San Diego, Department of Computer Science & Engineering, April 1999.
- [BAG00] Leeann Bent, Darren C. Atkinson, and William G. Griswold. A comparative study of two whole programs slicers for C. Computer Science Technical Report CS2000-0643, University of California, San Diego, Department of Computer Science & Engineering, 2000.
- [BCM94] David Bernstein, Doron Cohen, and Dror E. Maydan. Dynamic memory disambiguation for array references. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 105–111, San Jose, CA, USA, November 1994.
- [BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 1–12, Vancouver, BC, Canada, June 2000.
- [BL94] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4):1319–1360, 1994.

- [BLW96] Leo Beltracchi, James R. Lyle, and Dolores R. Wallace. Using a program slicing CASE tool for evaluating high integrity software systems. In *Proceedings of the 1996 American Nuclear Society International Topical Meeting on Nuclear Plant Instrumentation, Control, and Human-Machine Interface Technologies*, pages 1033–1039, University Park, PA, USA, May 1996.
- [CFE97] Brad Calder, Peter Feller, and Alan Eustace. Value profiling. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 259–269, Research Triangle Park, NC, USA, December 1997.
- [CFE99] Brad Calder, Peter Feller, and Alan Eustace. Value profiling and optimization. *Journal of Instruction Level Parallelism*, 1, March 1999.
- [CHM⁺98] Charles Consel, L. Hornof, R. Marlet, G. Muller, S. Thibault, and E.-N. Volanschi. Tempo: specializing systems applications and beyond. *ACM Computing Surveys*, 30(3es):19–es, September 1998.
- [CKJA98] Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. Cache-conscious data placement. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 139–149, San Jose, CA, USA, October 1998.
- [CL97] Michał Cierniak and Wei Li. Just-in-time optimizations for high-performance Java programs. *Concurrency: Practice and Experience*, 9(11):1063–1073, November 1997. Special Issue: Java for computational science and engineering — simulation and modeling II.
- [CL99] R. Cohn and P. Lowney. Feedback directed optimization in Compaq’s compilation tools for alpha. In *Proceedings of the 2nd Workshop on Feedback-Directed Optimization*, Haifa, Israel, November 1999.

- [CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.
- [CLS00] Michal Cierniak, Guei-Yuan Lueh, and James M. Stichnoth. Practicing JUDO: Java under dynamic optimizations. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 13–26, Vancouver, BC, Canada, June 2000.
- [CN96] Charles Consel and François Noël. A general approach for run-time specialization and its application to C. In *Conference Record of POPL '96: 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 145–156, St. Petersburg, FL, USA, January 1996.
- [Das00] Manuvir Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 35–46, Vancouver, BC, Canada, June 2000.
- [DBK01] Rajagopalan Desikan, Doug Burger, and Stephen W. Keckler. Measuring experimental error in microprocessor simulation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 266–277, Göteborg, Sweden, June 30–July 4, 2001.
- [EHK96] Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In *Conference Record of POPL '96: 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 131–144, St. Petersburg, FL, USA, January 1996.
- [FBC⁺01] Brian Fahs, Satarupa Bose, Matthew Crum, Brian Slechta, Francesco Spadini, Tony Tung, Sanjay J. Patel, and Steven S. Lumetta. Performance characterization of a hardware mechanism for dynamic optimization. In *Proceedings of the 34th Annual Symposium on Microarchitecture*, pages 16–27, Austin, TX, USA, December 2001.

- [FR01] Margaret A. Francel and Spencer Rugaber. The value of slicing while debugging. In *Proceedings of the 7th International Workshop on Program Comprehension*, pages 151–169, Pittsburgh, PA, USA, May 2001.
- [GH96] Rakesh Ghiya and Laurie J. Hendren. Connection analysis: A practical interprocedural heap analysis for C. *International Journal of Parallel Programming*, 24(6):547–578, December 1996.
- [GJS96] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. Addison-Wesley, Reading, MA, USA, 1996.
- [GKM82] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126, Boston, MA, USA, June 1982.
- [GL91] Keith B. Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, August 1991.
- [GMP⁺97] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and S. Eggers. Annotation-directed run-time specialization in C. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM-97)*, volume 32, 12 of *ACM SIGPLAN Notices*, pages 163–178, New York, NY, USA, June 12–13 1997. ACM Press.
- [GMP⁺00a] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. The benefits and costs of dyc’s run-time optimizations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(5):932–972, 2000.
- [GMP⁺00b] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. DyC: An expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, 248(1-2):147–199, 2000.

- [GPM⁺99] Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers, and Susan J. Eggers. An evaluation of run-time optimizations. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 293–304, Atlanta, GA, USA, May 1999.
- [Gra] GrammaTech, Inc. Codesurfer user guide and reference manual.
- [Gra01] Brian K. Grant. *Benefits and Costs of Staged Run-time Specialization*. Ph.D. dissertation, University of Washington, Department of Computer Science & Engineering, 2001.
- [HC98] Mary Jean Harrold and Ning Ci. Reuse-driven interprocedural slicing. In *Proceedings of the 20th International Conference on Software Engineering*, pages 74–83, Kyoto, Japan, April 1998.
- [HCU91] U. Hölzle, Craig Chambers, and Dave Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP'91 Conference Proceedings*, Geneva, Switzerland, 1991.
- [HKC97] Amir H. Hashemi, David R. Kaeli, and Brad Calder. Efficient procedure mapping using cache line coloring. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 171–182, Las Vegas, NV, USA, June 1997.
- [HMC⁺93] Wen-mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The superblock: An effective structure for vliw and superscalar compilation. *The Journal of Supercomputing*, 7(1):229–248, January 1993.
- [hot99] The Java Hotspot performance engine. April 1999. White paper available at URL <http://java.sun.com/products/hotspot/whitepaper.html>.

- [HRB90] Susan Horwitz, Tom Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.
- [HY97] Glenn Holloway and Cliff Young. The flow and analysis libraries of machine SUIF. In *Proceedings of the 2nd SUIF Compiler Workshop*, Stanford, CA, USA, August 1997.
- [JGS93] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, New York, NY, USA, 1993.
- [KR96] Todd B. Knoblock and Erik Ruf. Data specialization. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 215–225, Philadelphia, PA, USA, May 1996.
- [LH99a] Donglin Liang and Mary Jean Harrold. Efficient points-to analysis for whole-program analysis. In *Proceedings of the 7th European Software Engineering Conference and ACM Symposium on the Foundations of Software Engineering*, pages 199–215, Toulouse, France, September 1999.
- [LH99b] Donglin Liang and Mary Jean Harrold. Reuse-driven interprocedural slicing in the presence of pointers and recursion. In *Proceedings of the 1999 International Conference on Software Maintenance*, pages 421–432, Oxford, England, August 1999.
- [LL96] Peter Lee and Mark Leone. Optimizing ML with run-time code generation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 137–148, Philadelphia, PA, USA, May 1996.
- [MAEC02] Markus Mock, Darren Atkinson, Susan J. Eggers, and Craig Chambers. Improving program slicing with dynamic points-to data. In *Proceedings of the 10th ACM Symposium on the Foundations of Software Engineering*, Charleston, SC, USA, November 2002. To appear.

- [MBCE99] Markus Mock, Mark Berryman, Craig Chambers, and Susan J. Eggers. Calpa: A tool for automating dynamic compilation. In *Proceedings of the 2nd Workshop on Feedback-Directed Optimization*, pages 100–109, Haifa, Israel, November 1999.
- [MCE00] Markus Mock, Craig Chambers, and Susan J. Eggers. Calpa: A tool for automating selective dynamic compilation. In *Proceedings of the 33rd Annual Symposium on Microarchitecture*, pages 291–302, Monterey, CA, USA, December 2000.
- [MDCE01] Markus Mock, Manuvir Das, Craig Chambers, and Susan J. Eggers. Dynamic points-to sets: A comparison with static analyses and potential applications in program understanding and optimization. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 66–72, Snowbird, UT, USA, June 2001.
- [MIP90] MIPS Computer Systems, Sunnyvale, CA. *UMIPS-V reference manual (pixie and pixstats)*, 1990.
- [Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.
- [PCE02] Matthai Philipose, Craig Chambers, and Susan J. Eggers. Towards automatic construction of staged compilers. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 113–125, Portland, OR, USA, 2002.
- [PH90] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 16–27, White Plains, NY, USA, 1990.
- [PL01] Sanjay J. Patel and Steven S. Lumetta. rePLay: A hardware framework for dynamic optimization. *IEEE Transactions on Computers*, 50(6):590–608, June 2001.

- [PLR94] Hemant D. Pande, William Landi, and Barbara G. Ryder. Interprocedural def-use associations for C systems with single level pointers. *IEEE Transactions on Software Engineering*, 20(5):385–403, May 1994.
- [Ruf00] Erik Ruf. Effective synchronization removal for java. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 208–218, Vancouver, BC, Canada, June 2000.
- [SE94] Amitabh Srivastava and Alan Eustace. ATOM: a system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 196–205, Orlando, FL, USA, June 1994.
- [SH97a] Marc Shapiro and Susan Horwitz. The effects of the precision of pointer analysis. In *Proceedings of the 4th International Symposium on Static Analysis*, pages 16–34, Paris, France, January 1997.
- [SH97b] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 1–14, Paris, France, January 1997.
- [Shi91] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. Ph.D. dissertation, Carnegie Mellon University, School of Computer Science, May 1991.
- [Ste96a] Bjarne Steensgaard. Points-to analysis by type inference of programs with structures and unions. In Tibor Gyimothy, editor, *Compiler Construction, 6th International Conference*, volume 1060 of *Lecture Notes in Computer Science*, pages 136–150, Linköping, Sweden, 24–26 April 1996. Springer.
- [Ste96b] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 32–41, St. Petersburg, FL, USA, January 1996.

- [TD01] Frank Tip and T. B. Dinesh. A slicing-based approach for locating type errors. *ACM Trans. on Software Engineering and Methodology*, 10(1):5–55, January 2001.
- [Tip95] Frank Tip. A survey of program slicing techniques. *Journal of programming languages*, 3(3):121–189, September 1995.
- [TSS00] Omri Traub, Stuart Schechter, and Michael D. Smith. Ephemeral instrumentation for lightweight program profiling. Technical report, Harvard University, Department of Electrical Engineering and Computer Science, 2000.
- [WD01] Scott Watterson and Saumya Debray. Goal-directed value profiling. In *Proceedings of the 10th International Conference on Compiler Construction (CC2001)*, Genova, Italy, April 2001.
- [Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.
- [WFW⁺94] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12), December 1994.
- [WL95] Robert P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 1–12, La Jolla, CA, USA, June 1995.
- [YMP⁺99] Byung-Sun Yang, Soo-Mook Moon, Seongbae Park, Junpyo Lee, SeungIl Lee, Jinpyo Park, Yoo C. Chung, Suhyun Kim, Kemal Ebcioglu, and Erik Altman. LaTTe: A Java VM just-in-time compiler with fast and efficient register allocation. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation*

Techniques (PACT '99), pages 128–138, Newport Beach, CA, USA, October 12–16, 1999.

Appendix A

INSTRUMENTATION LIBRARY PROTOTYPES

```

void __calpa_update_char_var_def_or_use(uint hashcode, int kind, uint sid, uint prid, int fn, char value);
void __calpa_update_double_var_def_or_use(uint hashcode, int kind, uint sid, uint prid, int fn, double value);
void __calpa_update_float_var_def_or_use(uint hashcode, int kind, uint sid, uint prid, int fn, float value);
void __calpa_update_int_var_def_or_use(uint hashcode, int kind, uint sid, uint prid, int fn, int value);
void __calpa_update_ldouble_var_def_or_use(uint hashcode, int kind, uint sid, uint prid, int fn, long double value);
void __calpa_update_llong_var_def_or_use(uint hashcode, int kind, uint sid, uint prid, int fn, long long int value);
void __calpa_update_long_var_def_or_use(uint hashcode, int kind, uint sid, uint prid, int fn, long value);
void __calpa_update_ptr_var_def_or_use(uint hashcode, int kind, uint sid, uint prid, int fn, void* value);
void __calpa_update_short_var_def_or_use(uint hashcode, int kind, uint sid, uint prid, int fn, short value);
void __calpa_update_uchar_var_def_or_use(uint hashcode, int kind, uint sid, uint prid, int fn, unsigned char value);
void __calpa_update_uint_var_def_or_use(uint hashcode, int kind, uint sid, uint prid, int fn, unsigned int value);
void __calpa_update_ullong_var_def_or_use(uint hashcode, int kind, uint sid, uint prid, int fn, unsigned long long value);
void __calpa_update_ulong_var_def_or_use(uint hashcode, int kind, uint sid, uint prid, int fn, unsigned long value);
void __calpa_update_ushort_var_def_or_use(uint hashcode, int kind, uint sid, uint prid, int fn, unsigned short value);

void __calpa_update_char_mem_def_or_use(uint hashcode, int kind, void* addr, char value, uint sid,
uint prid, int fn, void* aaddr, int asize, int count, ...);
void __calpa_update_double_mem_def_or_use(uint hashcode, int kind, void* addr, double value, uint sid,
uint prid, int fn, void* aaddr, int asize, int count, ...);
void __calpa_update_float_mem_def_or_use(uint hashcode, int kind, void* addr, float value, uint sid,
uint prid, int fn, void* aaddr, int asize, int count, ...);
void __calpa_update_int_mem_def_or_use(uint hashcode, int kind, void* addr, int value, uint sid,
uint prid, int fn, void* aaddr, int asize, int count, ...);
void __calpa_update_ldouble_mem_def_or_use(uint hashcode, int kind, void* addr, long double value, uint sid,
uint prid, int fn, void* aaddr, int asize, int count, ...);
void __calpa_update_llong_mem_def_or_use(uint hashcode, int kind, void* addr, long long int value, uint sid,
uint prid, int fn, void* aaddr, int asize, int count, ...);
void __calpa_update_long_mem_def_or_use(uint hashcode, int kind, void* addr, long value, uint sid,
uint prid, int fn, void* aaddr, int asize, int count, ...);
void __calpa_update_ptr_mem_def_or_use(uint hashcode, int kind, void* addr, void* value, uint sid,
uint prid, int fn, void* aaddr, int asize, int count, ...);
void __calpa_update_short_mem_def_or_use(uint hashcode, int kind, void* addr, short value, uint sid,
uint prid, int fn, void* aaddr, int asize, int count, ...);
void __calpa_update_uchar_mem_def_or_use(uint hashcode, int kind, void* addr, unsigned char value, uint sid,
uint prid, int fn, void* aaddr, int asize, int count, ...);
void __calpa_update_uint_mem_def_or_use(uint hashcode, int kind, void* addr, unsigned int value, uint sid,
uint prid, int fn, void* aaddr, int asize, int count, ...);
void __calpa_update_ullong_mem_def_or_use(uint hashcode, int kind, void* addr, unsigned long long value, uint sid,
uint prid, int fn, void* aaddr, int asize, int count, ...);
void __calpa_update_ulong_mem_def_or_use(uint hashcode, int kind, void* addr, unsigned long value, uint sid,
uint prid, int fn, void* aaddr, int asize, int count, ...);
void __calpa_update_ushort_mem_def_or_use(uint hashcode, int kind, void* addr, unsigned short value, uint sid,
uint prid, int fn, void* aaddr, int asize, int count, ...);

```

Figure A.1: Function prototypes of data capturing routines.

Appendix B

SOURCE CODE FOR ANNOTATED PROCEDURES

```
/* array from 0..n-1, return position or -1 */
int search(int x[], int n, int key) {
    int l, u, r;
    l = 0;
    u = n-1;
    r = n / 2;

    for(; l <= u ; r/=2 ) {
        int p;
        p = u - r;
        if(x[p] == key) {
            return p;
        } else if(x[p] > key) {
            u = p-1;
        } else {
            l = p+1;
        }
    }
    return -1;
}
```

Figure B.1: Source code for the binary kernel.

```
void chebychev(float c[MAX], int n, float xa, float xb) {
    int k, j;
    float xm, xp, sm;
    float f[MAX];
    xp = (xb + xa) / 2;
    xm = (xb - xa) / 2;
    k = 1;
    for(; k <= n; k++) {
        f[k] = cos(xp + xm * (float)cos DYC_STATIC_CALL(PI * (k - 0.5) / n));
    }
    j = 0;
    for(; j <= n-1; j++) {
        sm = 0.0;
        for(k = 1; k <= n; k++) {
            sm = sm + f[k] * (float)cos DYC_STATIC_CALL(PI * (float)j *
((float)k - 0.5) / (float)n);
        }
        c[j] = (float)(2.0 / (float)n) * sm;
    }
    return;
}
```

Figure B.2: Source code for the chebychev kernel.

```
mainloop(CACHETYPE *cachep, POLICYTYPE *policyp, CTRLTYPE *ctrlp, METRICTYPE *metricp) {
    extern int dumpaddr();
    extern int dumpstate();
    extern int fetch();
    extern int outputmetric();
    extern int update();
    extern int flushcache();
    DECODEDADDRTYPE decodedaddr;
    int fetch_return;

    for (;;) {
        {
            register DECODEDADDRTYPE *dap = &decodedaddr;
            int addr;
            int thelabel;
            static int flushcount = -2;
            if (flushcount===-2) {
                flushcount = ((ctrlp ->Q<1) ? -1 : 0);
            }
            if ((flushcount>=0) && (flushcount++ >= ctrlp ->Q)) {
                flushcount=1;
                flushcache(cachep,ctrlp,metricp);
            }

            if (readfrominputstreamfromfile(&thelabel,&addr,ctrlp)==(-1)) {
                fetch_return = (-1);
            }
            else {
                if (thelabel == XFLUSH) flushcache(cachep,ctrlp,metricp);
            }
        }
    }
}
```

Figure B.3a: Source code for dinero.

```

{
    register int labl = thelabel;
    unsigned int theaddr;
    unsigned int theblocksize, thenumUorDsets, thenumIsets;
    theaddr = addr;
    theblocksize = cachep ->blocksize;
    thenumUorDsets = cachep ->numUorDsets;
    thenumIsets = cachep ->numIsets;
    dap->address = theaddr;
    dap->accesstype = labl;    {
        int blockaddr;
        int dap_block;
        int subblocksize;
        int stripped_label;

        blockaddr = theaddr / theblocksize;
        stripped_label = (labl & 0x7);
        if (stripped_label < XINSTRN) {
            dap->set = blockaddr % thenumUorDsets;
            dap->tag = blockaddr / thenumUorDsets;
        } else if (stripped_label == XINSTRN) {
            if (thenumIsets != 0) {
                dap->set = (blockaddr % thenumIsets) + thenumUorDsets;
                dap->tag = blockaddr / thenumIsets;
            }
        } else {
            dap->set = blockaddr % thenumUorDsets;
            dap->tag = blockaddr / thenumUorDsets;
        }
    }
}

```

Figure B.3b: Source code for dinero (continued).

```

{
    STACKNODETYPE *stackupdate();
    STACKNODETYPE *findnth();
    register DECODEDADDRTYPE *dap = &decodedaddr;
    int setnumber;
    int stackdepth;
    int elements_per_set;
    int miss;
    int blockmiss;
    STACKNODETYPE *preptr;
    STACKNODETYPE *ptr;
    STACKNODETYPE *replacedptr;

    setnumber = dap->set;
    elements_per_set = cachep ->assoc;
    {
        int addrtag = dap->tag;
        int setsize = elements_per_set;
        int stacknum = setnumber;
        extern STACKNODETYPE *stack;
        int found_flag=0;
        preptr = &stack[stacknum];
        ptr = stack[stacknum].next;
        stackdepth = 1;
        for (; ((ptr != 0L)
                && (ptr->tag != addrtag)
                && (stackdepth <= setsize));
            stackdepth++) {
            preptr = ptr;
            ptr = ptr->next;
        }
    }
}

```

Figure B.3d: Source code for dinero (continued).

```

if (!(ptr != 0L) && (ptr->tag == addrtag)) {
    if ((ptr != 0L)
        &&(stack[stacknum].tag - setsize >= 8))
        putonfreelist(stacknum,preptr);
    preptr = ptr = 0L;
    stackdepth = 0;
}
}
blockmiss = (stackdepth==0 || (stackdepth - elements_per_set)>0);
miss = blockmiss || ((dap->validbit & ptr->valid)==0);
if ( (policyp ->fetch-DEMAND!=0) && ((dap->accesstype==0) || (dap->accesstype==2))) {
    {
        STACKNODETYPE *stackptr = ptr;
        unsigned int prefetchaddr;
        extern long random();
        if (policyp ->abortprefetchpercent > 0 ) {
            if (random()/MAXINTPERCENT < policyp ->abortprefetchpercent) {
                goto PREFETCH_END;
            }
        }
        prefetchaddr = dap->address + cachep ->prefetchdisplacement;
        switch (policyp ->fetch) {
        case ALWAYS_PREFETCH:
            push_addrstack((PREFETCH+dap->accesstype),prefetchaddr);
            break;
        case LOADFORWARD_PREFETCH:
            if (
                (dap->address/cachep ->blocksize)==(prefetchaddr/cachep ->blocksize)
            ) {
                push_addrstack((PREFETCH+dap->accesstype),prefetchaddr);
            }
            break;

```

Figure B.3e: Source code for dinero (continued).

```

case SUBBLOCKPREFETCH:
    if (
        (dap->address/cachep ->blocksize) != (prefetchaddr/cachep ->blocksize)
    ) {
        prefetchaddr = prefetchaddr - cachep ->blocksize;
    }
    push_addrstack((PREFETCH+dap->accesstype),prefetchaddr);
    break;

case MISSPREFETCH:
    if (miss) {
        push_addrstack((PREFETCH+dap->accesstype),prefetchaddr);
    }
    break;

case TAGGEDPREFETCH:
    if ((miss) || ((dap->validbit & stackptr->reference)==0)) {
        push_addrstack((PREFETCH+dap->accesstype),prefetchaddr);
    }
    break;

case DEMAND:
default :
    printf("\n---Error in prefetch policy:%c\n",policyp ->fetch);
    exit(1);
}
PREFETCH_END;;
}
}

```

Figure B.3f: Source code for dinero (continued).

```

if (stackdepth!=1) {
  if ((policyp ->writeallocate-WRITEALLOCATE==0)
      || (dap->accesstype!=1)) {
    ptr = stackupdate(stackdepth,dap->tag,dap->blockaddr,
                     policyp ->replacement,setnumber,elements_per_set,ptr,preptr);
  }
  else {
    if ((stackdepth>1) && (stackdepth<=elements_per_set)) {
      ptr = stackupdate(stackdepth,dap->tag,dap->blockaddr,
                       policyp ->replacement,setnumber,elements_per_set,ptr,preptr);
    }
  }
}
if ((policyp ->writeallocate==NOWRITEALLOCATE)
    && (dap->accesstype==1)
    && (miss)) {
}
else {
  if (blockmiss) {
    ptr->reference = 0;
    ptr->dirty = 0;
    ptr->valid = 0;
  }
  ptr->valid |= dap->validbit;
  if ((policyp ->write-COPYBACK==0) && (dap->accesstype==1)) {
    ptr->dirty |= dap->validbit;
  }
}

```

Figure B.3g: Source code for dinero (continued).

```

if ((policyp ->fetch-TAGGEDPREFETCH==0) && (
                                (dap->accesstype==2) ||
                                (dap->accesstype==0) ||
                                (dap->accesstype==1) ||
                                (dap->accesstype==3) )) {
    ptr->reference |= dap->validbit;
}
}
metricp->fetch[dap->accesstype]++;
if (miss) {
    metricp->miss[dap->accesstype]++;
    if (blockmiss) {
        metricp->blockmiss[dap->accesstype]++;
    }
}
if (blockmiss) {
    {
        int stacknum = setnumber;
        int n = (elements_per_set+1);
        extern STACKNODETYPE *stack;
        int stackdepth2;
        register STACKNODETYPE *ptr;
        ptr = stack[stacknum].next;
        if (stack[stacknum].tag < n) {
            ptr = 0L;
        }
        else {
            stackdepth2 = 1;
            for (; stackdepth2 < n; stackdepth2++) {
                ptr = ptr->next;
            }
        }
        replacedptr = ptr;
    }
}
}

```

Figure B.3h: Source code for dinero (continued).

```
int dotproduct(int size, int u[], int v[]) {  
    int i;  
    int res;  
    int uelem, velem;  
  
    res = 0;  
    for (i=0; i<size; i++) {  
        uelem = u[i];  
        velem = v[i];  
        res = res + uelem*velem;  
    }  
    return res;  
}
```

Figure B.4: Source code for the dotproduct kernel.

```

struct excitation {
    double dt;      /* time step */
    double duration; /* total duration */
    double t0;     /* rise time */
};
extern struct excitation Exc;

double phi0(double t) {
    double value, t0 = Exc.t0;

    if (t <= t0) {
        value = 0.5 / PI * (2.0 * PI * t / t0 - sin(2.0 * PI * t / t0));
        return value;
    }
    else
        return 1.0;
}

double phi1(double t) {
    double value, t0 = Exc.t0;

    if (t <= t0) {
        value = (1.0 - cos(2.0 * PI * t / t0)) / t0;
        return value;
    }
    else
        return 0.0;
}

double phi2(double t) {
    double value, t0 = Exc.t0;

    if (t <= t0) {
        value = 2.0 * PI / t0 / t0 * sin(2.0 * PI * t / t0);
        return value;
    }
    else
        return 0.0;
}

```

Figure B.5: Source code for the phi-routines of the quake benchmark.

```
int ckbrkpts(unsigned int addr, int brktype) {
    struct brkpoints *bp; /* = brktable; */
    int cnt;

    if(!brkenabled)
        return(0);
    bp = brktable;
    cnt = 0;
    for(; cnt < TMPBRK; cnt++, bp++)
        if(bp ->code && ((bp ->adr & BRK_MASK) == addr))
            break;
    if(cnt == TMPBRK)
        return(0);
    if(!(bp ->code & brktype))
        return(0);
    if(bp ->code & BRK_CNT)
        if(++bp->cnt != bp ->limit)
            return(0);
    if((curbrkaddr = addr) != IP)
        curbrkaddr = IP;
    brkpend = 1;
    return(-1);
}
```

Figure B.6: Source code for the ckbrkpts routine of the m88ksim benchmark.

```

void do_convol(xel** xelbuf, xel** outbuf, int rows, int cols, int crows,
              int ccols, int crowso2, int ccolso2,
              float** rweights, float** gweights, float** bweights,
              xelval maxval, int format) {
    xelval g, r, b;
    float gsum, rsum, bsum, weight;
    int crow, row, rowbase, colbase, ccol, col;
    xel x;

    for ( row = 0; row < crowso2 ; ++row ) {
        for ( col = 0; col < cols; ++col )
            outbuf[row][col] = xelbuf[row][col];
    }
    for ( ; row < rows - crowso2; ++row ) {
        for ( col = 0; col < ccolso2; ++col )
            outbuf[row][col] = xelbuf[row][col];
        rowbase = row-crowso2;
        /* Convolve */
        if ( PNM_FORMAT_TYPE(format) == PPM_TYPE ) {
            for ( ; col < cols - ccolso2; ++col ) {
                colbase = col-ccolso2;
                rsum = gsum = bsum = 0.0;
                crow = 0;
                for ( ; crow < crows; ++crow ) {
                    ccol = 0;
                    for ( ; ccol < ccols; ++ccol ) {
                        x = xelbuf [rowbase+crow] [colbase+ccol];
                        r = xelbuf [rowbase+crow] [colbase+ccol]. r;
                        weight = rweights [crow] [ccol];
                        rsum += PPM_GETR( x ) * weight;
                        weight = gweights [crow] [ccol];
                        gsum += PPM_GETG( x ) * weight;
                        weight = bweights [crow] [ccol];
                        bsum += PPM_GETB( x ) * weight;
                    }
                }
            }
        }
    }
}

```

Figure B.7a: Source code for pnmconvol.

```

    if ( rsum < 0.0 ) r = 0;
    else if ( rsum > maxval ) r = maxval;
    else r = (int) (rsum + 0.5);
    if ( gsum < 0.0 ) g = 0;
    else if ( gsum > maxval ) g = maxval;
    else g = (int) ( gsum + 0.5);
    if ( bsum < 0.0 ) b = 0;
    else if ( bsum > maxval ) b = maxval;
    else b = (int) ( bsum + 0.5 ) ;
    PPM_ASSIGN( outbuf[row][col], r, g, b );
}
} else { /* g is for grey */
for ( ; col < cols - ccolso2; ++col ) {
    colbase = col-ccolso2;
    gsum = 0.0; crow = 0;
    for ( ; crow < crows; ++crow ) {
        ccol = 0;
        for ( ; ccol < ccols; ++ccol ) {
            x = xelbuf [rowbase+crow] [colbase+ccol];
            weight = gweights [crow] [ccol];
            gsum += PNM_GET1( x ) * weight;
        }
    }
    if ( gsum < 0.0 ) g = 0;
    else if ( gsum > maxval ) g = maxval;
    else g = (int) (gsum + 0.5);
    PNM_ASSIGN1( outbuf[row][col], g );
}
}
for ( ; col < cols; ++col)
    outbuf[row][col] = xelbuf [row] [col];
}
/* Now copy the remaining unconvolved rows in xelbuf. */
for ( ; row < rows; ++row ) {
    for (col = 0; col < cols; ++col)
        outbuf[row][col] = xelbuf[row][col];
}
}

```

Figure B.7b: Source code for pnmconvol (continued).

```

typedef enum { AA, BB, CC, DD, EE, FF, GG } fieldtype;
typedef enum { LT, LE, GT, GE, NE, EQ } bool_op;
struct query {
    fieldtype record_field;
    unsigned val;
    bool_op bool_op;
};
struct record {int a; int b; int c; int d; int e; int f; int g;};
typedef int (*iptr)(struct record *r);

int sq(struct record *d, struct query *q, int n) {
    int i,j;
    int field = 0;
    struct query *qe, *qf;

    qe = q;
    qf = qe + n; /* pointer arithmetic */
    for (; qe < qf; qe++) {
        switch (qe ->record_field) {
            case AA: field = d->a; break;
            case BB: field = d->b; break;
            case CC: field = d->c; break;
            case DD: field = d->d; break;
            case EE: field = d->e; break;
            case FF: field = d->f; break;
            case GG: field = d->g; break;
        }
        switch (qe ->bool_op) {
            case LT: if (field >= qe ->val) return 0; continue;
            case EQ: if (field != qe ->val) return 0; continue;
            case LE: if (field > qe ->val) return 0; continue;
            case GT: if (field <= qe ->val) return 0; continue;
            case NE: if (field == qe ->val) return 0; continue;
            case GE: if (field < qe ->val) return 0; continue;
        }
    }
    return 1;
}

```

Figure B.8: Source code for the query kernel.

```
extern float f(float x);
void romberg(float r[10][10], float a, float b, int M) {
    int n, m, i, N;
    float h, s;

    h = b - a;
    r[0][0] = (f(a) + f(b)) * h / 2.0;
    n = 1;
    for (; n <= M; n++) {
        h = h / 2.0;
        s = 0.0;
        i = 1;
        N = 1 << (n - 1);
        for (; i <= N; i++) {
            s = s + f(a + (2.0 * i - 1) * h);
        }
        r[n][0] = r[n-1][0]/2.0 + h * s;
        m = 1;
        for(; m <= n; m++) {
            r[n][m] = r[n][m-1] +
(1.0/((1 << (2*m))-1)) * (r[n][m-1] - r[n-1][m-1]);
        }
    }
}
```

Figure B.9: Source code for the romberg kernel.

VITA

Markus Ulrich Mock was born in Aulendorf, Germany, and grew up in Bad Schussenried, in Southwestern Germany. In 1992 he received a M.S. Degree in Computer Science (Informatik Diplom) from the University of Karlsruhe, Germany. After working there as a staff member, he moved to Seattle in 1994 to pursue a Ph.D. degree in Computer Science at the University of Washington. He received a M.S. degree in Computer Science in 1997, and a Ph.D. degree in August 2002.