FAST, EFFECTIVE PROGRAM ANALYSIS FOR OBJECT-LEVEL PARALLELISM

by

William Christian Benton

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

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2008

For W.F.B., W.J.B., W.D.B., and W.T.B.

Was du ererbt von deinen Vätern hast,
Erwirb es, um es zu besitzen.
— Johann Wolfgang von Goethe

ACKNOWLEDGMENTS

It is customary for authors of academic books to include in their prefaces statements such as this: "I am indebted to ... for their invaluable help; however, any errors which remain are my sole responsibility." Occasionally an author will go further. Rather than say that if there are any mistakes then he is responsible for them, he will say that there will inevitably be some mistakes and he is responsible for them....

Although the shouldering of all responsibility is usually a social ritual, the admission that errors exist is not — it is often a sincere avowal of belief. But this appears to present a living and everyday example of a situation which philosophers have commonly dismissed as absurd; that it is sometimes rational to hold logically incompatible beliefs.

— David C. Makinson (1965)

This dissertation and the work it documents owe a great deal to the assistance, support, and advice of many people.

I could not have asked for a better advisor, and I am deeply indebted to Charles Fischer for his role in my graduate career. Charles is an excellent teacher and mentor and a tremendous resource. I have benefited greatly from his advice, encouragement, support, and multidisciplinary knowledge over these last five years. Thank you, Charles.

I am grateful to my thesis committee for their willingness to serve (and their willingness to get up early on a very cold Monday morning for my defense). I am indebted to each of them in particular for the ways that they have helped me to improve my dissertation work: to Susan Horwitz, for her precision, attention to detail, and consistently excellent and prompt feedback; to Ben Liblit, for his regular and helpful advice at both the micro- and macro-levels of research, development, and writing; to Marvin Solomon, for his careful and precise feedback and for his depth and breadth of knowledge; and to Paul Wilson, for his enthusiasm and perspective.

I have always benefited from excellent teaching. I would like to recognize, in chronological order, Stephen Sesko at Lawrence Livermore National Lab, who introduced me to recursion when I was eight as part of a summer Logo course; Steve McKelvey and Dick Brown, of the Mathematics department of

St. Olaf College, with whom I have enjoyed an extended conversation about computing and pedagogy; and John Poling and Rick Fairbanks, both formerly of the Philosophy department at St. Olaf, who taught me a great deal about thinking critically and writing well.

I have been fortunate to be a part of a collegial and vibrant programming languages research community here at Wisconsin. I am thankful to the members of the pl-chat mailing list in particular for good discussions, references, and camaraderie. I have also benefited from our department's strong computer architecture and systems groups, and would like to thank in particular the following present and former members of the department for their friendship and advice: Ras Bodik, Eli Collins, Jesse Davis, Tim Denehy, Allison Holloway, Steve Jackson, Mike Marty, and Philip Wells.

I am thankful to Bart Miller for his support early in my graduate career and for advice, mentoring, and exposure to many great ideas that I would not have otherwise encountered. Bart taught me a great deal about how (and how not) to do research and how to present ideas clearly. While working on the Paradyn project as a junior grad student, I was able to learn from senior students, and would specifically like to recognize the guidance and friendship of Phil Roth and Vic Zandy.

Dorian Arnold is one of my oldest friends at Wisconsin. I am grateful for his friendship first of all, for the absurd workout and lunch schedules we were able to maintain for many years, and for far too many conversations and well-fought arguments to count. I am also thankful for Dorian's feedback on my work, for the discussions we've been able to have about our research, and for his regular, jocular reminders that my margins and leading are as comically large as my taste in Carribean music is out-of-date.

I am thankful to Matt Allen for his friendship, regular and sound advice about matters ranging from fantasy football to computer architecture, and for our discussions about our research programs and about broader issues of concurrency and systems over the last few years. In particular, many nomenclatural improvements in my discussion of object-level parallelism are due to conversations with Matt.

I am glad to count Nick Kidd as a close friend and am indebted to him for his consistently excellent and subtle feedback on my work (in particular, on my VMCAI '09 and PPDP '07 papers); for thought-provoking discussions; and for his willingness to share advice on static analysis and model checking.

I am thankful to Anne Mulhern for her friendship, for many productive conversations about types and logic, for her feedback on my PPDP '07 paper, and for her excellent seminar on theorem-proving in Coq.

Having good research infrastructure has been an unexpected joy. I am thankful to all of the people and groups whose fine work I have been able to employ and extend. Vítor Santos Costa of the Universidade do Porto deserves thanks for the amazing Yap Prolog environment, his very quick responses to bug reports, and his guidance on modifying Yap to run under 64-bit OS X. I am also indebted to Vítor for his enthusiastic, expert advice on logic programming in general and the details of tabling in particular. The Soot group at McGill University, and especially Soot maintainer Eric Bodden, have my thanks for an excellent research compiler infrastructure; for helpful advice on the proper use and care of Soot; and for extremely quick response to issues and review of patches. The Jikes RVM community is very strong and has generously provided helpful, patient advice over a period of many years. In particular, I am grateful to Ian Rogers, Steve Blackburn, and Eliot Moss for their prompt and friendly responses to my questions.

I am grateful to Rebecca Hasti and Perry Kivolowitz, who supervised me in my various teaching appointments throughout my graduate career, taught me a great deal about teaching, and advocated for me as a teaching assistant.

I would like to thank the following people who, while they did not directly contribute to my dissertation work, have certainly made my time in Madison more pleasant: Dan Gibbons, Tim Glenn, Greg Jones, Taryn Okuma, Kim Huth, and Mark Hanson; Drs. David Jarrard, David King, and Glenn Liu; and my friends in the Madison Opera chorus.

I literally could not have completed this endeavor without my parents, W. David and Karen Benton, and anything I could say about them here will be inadequate. I am grateful for their encouragement and guidance throughout my childhood, for the way they have always challenged me to do my best, and for their moral, spiritual, and financial support during my graduate career. I am also thankful to my father for showing me at a very early age how exciting it could be to think about computation and to both my parents for continuing to offer advice until I was old and wise enough to accept it.

Finally, I am blessed with an amazing wife and son. Andrea is my best friend, a brilliant coach, and an excellent editor. Her encouragement, support, and love have sustained me throughout all of the challenges and triumphs of the past decade. Thank you, sweetheart, for everything.

This dissertation is dedicated to my son, William Thomas Benton, and to some other William Bentons: my father, my grandfather, and my great-grandfather.

Thomas, I hope you will possess the gifts your fathers have given you and make them your own.

S.D.G.

William C. Benton
December 2008

TABLE OF CONTENTS

Table of Contents vii

List of Tables ix

List of Figures xi

Abstract xiii

1 Introduction 1
- *1.1 Background* 1
- *1.2 Our contributions* 5
- *1.3 Synopsis* 7

2 Exposing latent object-level parallelism 9
- *2.1 Preliminaries* 11
- *2.2 Purity and safety* 17
- *2.3 The PIMA model* 19
- *2.4 Related work* 22

3 Declaring bytecode processors and analyses 29
- *3.1 Overview of the system* 32
- *3.2 Using DIMPLE$^+$* 33
- *3.3 The DIMPLE$^+$ IR* 37
- *3.4 The DIMPLE$^+$ Analysis Language* 55
- *3.5 Query engines* 58
- *3.6 Case study: Andersen's analysis* 63
- *3.7 Case study: effects inference* 68
- *3.8 Related work* 74

4 Effect inference for safe parallel execution 81
- *4.1 Mostly-functional behavior in Java programs* 82
- *4.2 Objects and effects: Background and motivation* 83
- *4.3 A lightweight object-oriented effects system* 84
- *4.4 Initializers and initialization effects* 90
- *4.5 Quiescing field inference* 94

4.6 Degrees of purity 97
4.7 Related work 98

5 Runtime evaluation and support 105
5.1 Experimental environment & evaluation methodology 106
5.2 Method invocations as potential tasks 111
5.3 Runtime support 116
5.4 Related work 118

6 Conclusions and future work 121

A Conventions 125
A.1 Naming conventions 125
A.2 Typographical conventions 126

B Relations in the DIMPLE IR 129

Colophon 135

References 137

LIST OF TABLES

1.1 Relative prevalence of implementation languages for software projects 4

3.1 DIMPLE$^+$structures representing Java values and constants 39
3.2 DIMPLE$^+$ structures representing Java arithmetic and value expressions 39
3.3 DIMPLE$^+$ structures representing Java comparison, bitwise, and logical expressions 40
3.4 DIMPLE$^+$ structures representing Java object and array expressions 41
3.5 DIMPLE$^+$ structures representing heap locations 41
3.6 DIMPLE$^+$ structures representing Java bytecode statements 43
3.7 Statements modeled by legal DIMPLE$^+$ assignment structures 44
3.8 DIMPLE$^+$ structures representing program structure and metadata 45
3.9 DIMPLE$^+$ structures representing method structure and control flow 46
3.10 DIMPLE$^+$ structures representing analysis-domain entities 46
3.11 Performance results for Andersen's analysis with the tabled query engine 68

4.1 IR for a simple effects system 86

5.1 Counts of methods that represent more than 0.01% of total method invocations. 112

A.1 Conventions for metavariable domains 125

LIST OF FIGURES

2.1 Sequential execution vs. execution exhibiting object-level parallelism 10
2.2 Example of minimal field access paths. The method foo2 is equivalent to the method foo, but it does not feature access paths of a length greater than 1. 14
2.3 Example of inherited classes that share field names 15
2.4 Memory diagrams illustrating state changes effected by field operations. 15
2.5 Typical method layout, with virtual function table pointer. 20
2.6 Proxy for a mutator method, with vtable pointer and updated delegate invocation queue. 22
2.7 Proxy for an accessor method, with vtable pointer, synchronization point, cleanup code, and original method body. 22

3.1 High-level overview of the DIMPLE$^+$ system's architecture. 34
3.2 Java listing for a simple cons-cell class 48
3.3 Metadata for the `Cell` class 49
3.4 Field and method declarations for the `Cell` class 50
3.5 Java bytecodes for `cons(int, Cell)` and `length()` methods 51
3.6 Truth table for the ternary majority function 61
3.7 Decision tree for the ternary majority function 61
3.8 Reduced ordered binary decision diagram for the ternary majority function 62
3.9 Andersen's analysis: select statement processing rules (rules treating arrays and exceptions are omitted) 65
3.10 Andersen's analysis: statement processing rules used by interprocedural transfer functions 66
3.11 Andersen's analysis: complete analysis rules 67
3.12 A simple `Point` class 69
3.13 `foo` exposes a shortcoming of the simple analysis 71
3.14 Select statement processing rules for parameterized effects inference (WRITE effect rules are omitted) 73
3.15 Select analysis rules for parameterized effects inference (rules treating WRITE effects are omitted) 74

4.1 Rules for reconstructing effects annotations in the straightforward effects system 89
4.2 Inference rules for initialization methods and initializer effects 93
4.3 Static and dynamic prevalence of final and quiescing fields in select DaCapo benchmarks 96
4.4 Percentage of all instance methods that are externally-pure or externally-read-only; INIT effects and effects on quiescing fields have been masked. 98
4.5 Example of a pure method that is not identified as such by our simple system 102

5.1 Per-thread state maintained by instrumentation code 109
5.2 Example instrumentation code as inserted for three kinds of basic blocks 110
5.3 Method invocation durations for `antlr` 113
5.4 Method invocation durations for `bloat` 113
5.5 Method invocation durations for `eclipse` 114
5.6 Method invocation durations for `jython` 114
5.7 Method invocation durations for `luindex` 115
5.8 Method invocation durations for `pmd` 115
5.9 A more efficient proxy implementation. 117

A.1 An example Java program listing 126
A.2 An example Prolog program listing 126

FAST, EFFECTIVE PROGRAM ANALYSIS FOR OBJECT-LEVEL PARALLELISM

William Christian Benton

Under the supervision of Professor Charles N. Fischer
At the University of Wisconsin-Madison

Multicore and multithreaded processors are ubiquitous. Applications in several domains, such as internet servers, scientific simulations, and high-end media creation tools, are generally capable of exploiting the concurrent contexts presented by these processors. However, the vast majority of client applications present sequential workloads that will not directly benefit from parallel hardware. In order for such applications to achieve high performance on such processors, they must be reimplemented as parallel applications.

Unfortunately, writing parallel software in mainstream programming languages is notoriously difficult. The thesis of this dissertation is that real-world client applications exhibit implicit thread-level parallelism because methods on distinct objects are often independent. We show that it is possible to identify this *object-level parallelism* statically via efficient, novel type-based program analyses to identify methods that are good candidates for safe parallel execution. In addition, we demonstrate that many methods that can be safely executed through object-level parallelism are substantial enough to constitute realistic parallel tasks.

Charles N. Fischer

ABSTRACT

Multicore and multithreaded processors are ubiquitous. Applications in several domains, such as internet servers, scientific simulations, and high-end media creation tools, are generally capable of exploiting the concurrent contexts presented by these processors. However, the vast majority of client applications present sequential workloads that will not directly benefit from parallel hardware. In order for such applications to achieve high performance on such processors, they must be reimplemented as parallel applications.

Unfortunately, writing parallel software in mainstream programming languages is notoriously difficult. The thesis of this dissertation is that real-world client applications exhibit implicit thread-level parallelism because methods on distinct objects are often independent. We show that it is possible to identify this *object-level parallelism* statically via efficient, novel type-based program analyses to identify methods that are good candidates for safe parallel execution. In addition, we demonstrate that many methods that can be safely executed through object-level parallelism are substantial enough to constitute realistic parallel tasks.

1 INTRODUCTION

The challenge of programming multi-core processors is real, but it is not a technical challenge. It is a purely sociological challenge. Technically, we have known since the 1980s how to program multi-core processors (in the guise of shared memory multiprocessors) and how to write programs for them (in terms of parallel algorithms).

— Peter Van Roy (2008)

Despite Van Roy's playful suggestion above, writing parallel software in mainstream programming languages is notoriously difficult. The thesis of this dissertation is that real-world client applications exhibit implicit thread-level parallelism because methods on distinct objects are often independent; furthermore, it is possible to identify this *object-level* parallelism statically via type-based program analysis and exploit it with lightweight runtime support.

1.1 BACKGROUND

For many years, the most dramatic improvements in software performance correlated closely with improvements in superscalar processor design and implementation. Improvements in manufacturing processes within particular microarchitecture generations led to smaller chips that could run at faster clock frequencies. Independently, each subsequent generation of chips devoted additional transistors to implementing ever more aggressive mechanisms for exploiting implicit instruction-level parallelism, some of which are described in a survey paper by Smith and Sohi (1995): branch predictors to enable speculatively fetching a longer stream of instructions; logic to enable the simultaneous issue of multiple independent instructions; register renaming units to eliminate false dependences between instructions; and other, more sophisticated speculation techniques.

Olukotun et al. (1996) argued — and time seems to have borne out — that continual performance improvements from exponential clock rate improvements and increasingly aggressive speculative hardware are unsustainable; this is the case, in large part, due to the power requirements of such improvements. Rather, they claimed, it makes more sense to use additional transistors and chip real estate to provide multiple, simpler chips in a single package. *Single-chip parallel processors* (Nayfeh et al. 1996; Tullsen et al. 1995) enable the simultaneous issue of instructions from multiple tasks on a single chip. *Multithreaded* designs extend the superscalar concept by allowing multiple threads to compete for hardware resources, simultaneously issuing instructions to different functional units. *Multicore* designs have multiple distinct execution units on a single die, which communicate by means of a shared cache. Still other designs combine aspects of multicore and multithreaded processors.

In the last decade, single-chip parallel processors have gone from being a hotly-discussed research topic to a ubiquitous component of computer systems, with multicore and multithreaded processors currently enjoying widespread use in workstation, server, and portable computers. These processors even appear in price-critical applications like game consoles: as an example, consider Microsoft's Xbox 360 platform (Andrews and Baker 2006), which was introduced in 2005 and whose CPU features 3 in-order cores running at 3.2GHZ with two hardware threads per core.

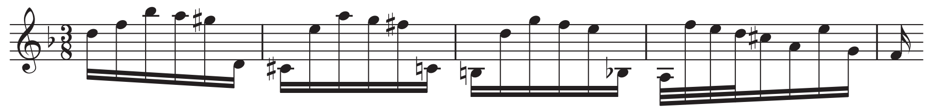
Because of the shift from aggressive superscalar processor designs to designs with more plentiful and less aggressive cores, applications can no longer rely on ever-faster uniprocessor performance. Instead of focusing chip resources on the highest-possible single-threaded performance, architects are building processors that can provide greater performance for concurrent workloads. As a consequence, future application speedups — indeed, even effective processor utilization — will come from software that can take advantage of concurrency.

Implicit concurrency is abundant in some kinds of programs, including those corresponding to server, scientific, and streaming-media workloads. Furthermore, there are known means to manually or automatically exploit such concurrency in many of these problem domains. (In addition, many of these problems inspire the sort economic incentive necessary to devote additional engineering effort to ensuring high utilization.) Server workloads are often explicitly multithreaded or multiprogrammed, and those that are not generally enjoy a straightforward path to explicit multithreading (e.g. by handling client requests with one of a pool of independent server threads). Automatically extracting threads from loop bodies in scientific, numerical, and streaming programs is a well understood problem, and there is ample library and language support to facilitate purpose-built parallel software in these domains.

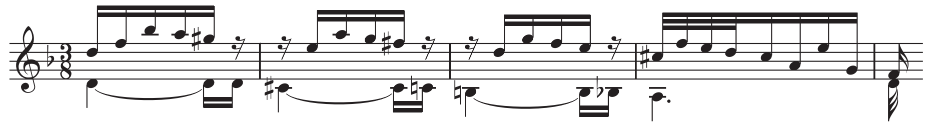
Unfortunately, the vast majority of end-user applications present serial integer workloads, with pointer-based data structures and irregular data accesses. Such programs cannot natively exploit hardware parallelism (unless, of course, two or more run at once via multiprogramming), and classical techniques to extract loop-body or array parallelism are not applicable for them. Many of these programs could be rewritten to exploit multicore or multithreaded processors, but only after heroic development (and maintenance) effort and potentially great cost.

Of course, not all programs or problems are able to take advantage of hardware concurrency. As a real-world example, even though it takes two hours for a pound of bread dough to rise in a bowl, one cannot raise the same amount in twelve minutes by dividing it among ten bowls. However, many applications that are written out as serial programs implicitly encode several independent, concurrent tasks.

The following short excerpt from J.S. Bach's d minor Chaconne for solo violin (from BWV 1004) serves as an analogy for a serial program with implicit concurrent tasks:



Although the writing in the above excerpt is monophonic (that is, only one note sounds at a time), Kennan (1987) showed that it implies polyphony (that is, multiple independent voices sounding concurrently). In listening to this example, one can clearly hear at least two parts in the single melodic line. We might write these out separately, with individual parts denoted by the direction of their note-stems, as follows:



In at least one way, sequential programs are no different from monophonic music. That is, a program written in the sequential style may nevertheless imply several independent tasks that can complete concurrently. The challenge is that programmers must expose these tasks (1) without changing the meaning of the program, (2) without introducing undue engineering or maintenance effort, and (3) without encoding dependences on the accidental features of particular microarchitecture implementations. Of course, there are performance costs associated with task creation and intertask coordination, so programmers have the added challenge of producing a parallel decomposition that performs at

	Projects	
Language	Count	Share
C	9181	23.56%
Java	6183	15.87%
C++	5139	13.19%
PHP	4490	11.52%
Perl	3788	9.72%
Python	3207	8.23%
JavaScript	1194	3.06%
Shell script	1096	2.81%
SQL	564	1.45%
Tcl	512	1.31%
Ruby	504	1.29%
Objective-C	388	1.00%
All other languages	2725	6.99%
All dataflow languages	1061	2.72%

Table 1.1: Relative prevalence of implementation languages for software projects listed on freshmeat.net; all languages with at least 1% of the total share of projects are listed.

least as well as the serial version.

In the position paper quoted at the beginning of this chapter, Van Roy (2008) playfully claims that the problems of multicore programming are solved and that the challenge posed by multicore processors is a merely *sociological* one: namely, that most programmers are unwilling or unable to program in dataflow languages in which the technical problems of exposing and exploiting implicit parallelism are well-understood.

It is difficult to dispute Van Roy's implication that few dataflow languages are truly ubiquitous; Table 1.1 shows the relative prevalence of the twelve most popular implementation languages on the popular software-tracking web site freshmeat.net as of September 2008, as well as the count and share of programs implemented in dataflow languages.[1] Only the top twelve languages

[1]We defined "dataflow" languages generously, including languages with only single-assignment variables (Erlang, Haskell, and Prolog), declarative query languages (SQL), and

have at least a 1% share of projects; these languages together account for 93% of projects. These numbers are not intended to represent the state of all software development — they do not account for internal development, in which releases are not announced in public, and likely undercount the prevalence of some languages as a result — but they represent a reasonable approximation of the implementation-language ecosystem for end-user applications.

Imagining the utopia in which dataflow languages enjoy well-deserved ubiquity makes for a pleasant daydream, but procedural and object-oriented languages with mutable state are far more widespread. In Table 1.1, all of the top seven languages feature mutable state, and the only one of the top twelve languages that we considered to be a "dataflow" language — SQL — is neither general-purpose nor Turing-complete. (SQL also accounts for over half of all projects implemented at least in part in "dataflow" languages.) Most of the imperative languages in the top twelve are object-oriented or support object-oriented features, and many run in managed environments or require nontrivial runtime support. Even if a language that makes it easy to exploit implicit parallelism were to immediately achieve widespread popularity tomorrow, we would still have tens of thousands of legacy applications written in languages like these in the top twelve, that are less amenable to known techniques.

1.2 OUR CONTRIBUTIONS

The focus of this dissertation is on technical advances to sidestep the sociological challenges. We have designed a parallel evaluation model, analyses and effect systems, and runtime support to help bridge the gap between the language climate we have now — dominated by managed object-oriented languages with mutable state and relatively unrestricted aliasing — and the language climate we hope to have in the future, which we expect will make designing and programming for pervasive concurrency far more tractable. The goal of this work is to present novel techniques, inspired by those for finding implicit parallelism in dataflow and mostly-functional languages, that can effectively identify a particular kind of implicit parallelism in a mainstream object-oriented language. We develop and evaluate our techniques in the context of the Java language and virtual machine, since Java is ubiquitous and we believe it to be representative of a useful class of languages.

mostly-functional languages with some impure features (Lisp, OCaml, Scheme, and Standard ML).

We assume Van Roy is being at least somewhat tongue-in-cheek when he proposes that the challenge of programming for multicore processors is merely that of bringing the delights of dataflow languages to a benighted and weary cohort of programmers. Industry inertia alone would seem to preclude this as a viable solution; furthermore, some problems do not admit straightforward solutions in dataflow languages. If we are (at least for the foreseeable future) stuck with programs written in imperative languages and programs that are difficult to reason about and automatically decompose, then it is desirable to bring some of the benefits of dataflow and mostly-functional languages to the languages that most working programmers use and to an enormous body of extant programs.

Given arbitrary side effects and unrestricted aliasing, general-purpose automatic parallelization is intractable. However, we exploit the engineering properties of object-oriented software — namely, that effects are typically confined to the receiver object — to treat some objects as independent processes, which synchronize with one another at certain method call boundaries. In this respect, our overall approach is similar to Multilisp (Halstead 1985), in which expressions can become processes, or to dataflow parallelism, in which an arbitrary number of evaluations can proceed independently once their dependences are satisfied and until their results are required.

The most salient difference between our contributions and the solved "technical challenge[s]" that Van Roy identifies in the context of dataflow and functional languages is that we are not operating within pure or mostly-pure languages with single-assignment variables. Rather, we are dealing with languages and runtimes in which side-effects are not only allowed but expected and idiomatic. As a consequence, we have developed analyses and an effect system to identify interference and independence between objects. Our analyses present concise summaries of the effecting behavior of methods and identify methods whose effects are confined to their receiver object; the runtime support we propose for our model is able to make use of dynamic aliasing information to precisify these summaries and enforce data dependence constraints.

We expect that, in the future, languages and programming models will provide better support for explicit parallel programming, and for parallel programming in-the-large. It would be foolish to speculate as to precisely what form these models and languages will take, but models that eliminate or minimize mutable shared memory between processes (including message passing between imperative processes and dataflow programming) have a great deal to recommend them over traditional shared-memory multithreaded programming. Since the techniques that we propose in this dissertation are applicable to extracting implicit parallelism from idiomatic imperative object-oriented programs, we expect that they would still be applicable to extracting implicit, fine-grained parallel tasks from explicitly-specified, isolated tasks consisting of idiomatic object-oriented parallel programs. In this way, we hope that our contributions can bridge the gap between the dataflow-parallelization techniques of the past and the burdens of finding ever more tasks in an explicitly parallel future.

1.3 SYNOPSIS

In the remainder of this document, we:

1. Introduce *object-level parallelism* (OLP), a measure of potential implicit parallelism analogous to instruction-level parallelism, and PIMA, a programming model designed to exploit OLP (in Chapter 2);
2. Present the DIMPLE$^+$ declarative analysis framework for flexible, interactive, and scalable analysis of Java bytecode programs (in Chapter 3), and which we use for implementing the analyses we present in Chapter 4;
3. Define and evaluate a novel object-oriented type-and-effect system, inference rules, and client analyses to identify a large class of run-time constant fields and object instance methods that are good candidates for parallel execution because their side effects are confined to individual objects (in Chapter 4); and
4. Evaluate dynamic opportunities to exploit latent OLP in Java programs and identify the runtime support necessary to do so (in Chapter 5).

We shall then conclude by presenting opportunities for further investigation, identifying the insights that we have acquired from imposing a new parallelism model onto already-written programs, and considering what these might mean for future programming models for purposefully-designed parallel programs.

The goal of bridging the gap between theories developed in the context of dataflow languages and practice in the world of imperative, object-oriented languages guides each of our contributions. The contributions in Chapters 2 and 4 bring parallelism techniques inspired by those for declarative languages to object-oriented languages by presenting a measure of implicit parallelism demarcated by module boundaries, a model for object-oriented parallel programming, and an analysis to determine when the effects of methods can be guaranteed independent in object-oriented programs. Chapters 4 and 5 characterize dynamic executions of object-oriented programs, showing substantial "mostly-functional" behavior, for which declarative parallelism is appropriate. Finally, the work described in Chapter 3 brings the benefits of declarative programming to an object-oriented language more indirectly, by presenting a logic programming environment for rapid, interactive development of new program analyses.

2 EXPOSING LATENT OBJECT-LEVEL PARALLELISM

A programming language is low level when its programs require attention to the irrelevant.

— Alan J. Perlis (1982)

Extensions for parallel execution have been a part of object-oriented languages for nearly as long as there have been object-oriented languages.[1] To date, most proposals for combining concurrency and objects have either required substantial program restructuring, have allowed the introduction of race conditions, or both. In this chapter, we present PIMA, a parallelism model designed to enable programmers and program transformations expose and exploit latent *object-level parallelism* (OLP) in idiomatic object-oriented programs. Much like *instruction-level parallelism* (ILP), which can be exposed by superscalar hardware that issues independent instructions simultaneously, OLP can be exposed by a virtual machine that activates independent methods on distinct objects concurrently. PIMA is a parallelism model that is designed to exploit OLP, motivated by the modular nature of object-oriented programs, capable of ensuring race-free executions, and likely to result in parallel executions without contention between threads. Furthermore, PIMA is more flexible than most extant models and is thus suitable for *automatic* transformation of serial programs to parallel ones.

Intuitively, the idea behind object-level parallelism is that the instance methods of certain objects may execute asynchronously without changing the result of a program, so that the work of a single *explicit* (that is, programmer-specified) thread may be split among a *main thread* and a pool of *delegate threads*. The PIMA model is a tool for exposing and exploiting latent OLP, just as microarchitectural techniques like speculation are tools for exploiting latent instruction-level parallelism. In contrast to hardware techniques for ILP, which operate dynamically, PIMA features cooperating static and dynamic components. Under PIMA, an object may be *demarcated* by a static annotation on a variable that refers to that object. (This annotation may come from an application programmer or an automatic program transformation.)

Method invocations on demarcated objects may be assigned to particular delegate threads by support code executed in the main thread, so that a

[1] Simula 67 provided a mechanism to associate an object with its own thread of control and execute method invocations asynchronously (Dahl et al. 1970).

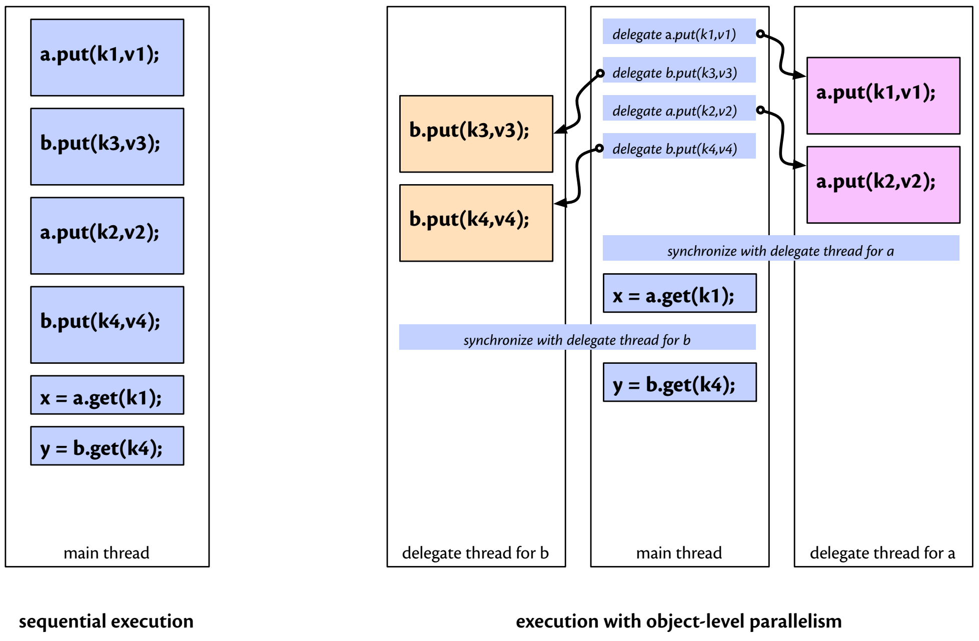


Figure 2.1: Sequential execution vs. execution exhibiting object-level parallelism

delegated method invocation M may execute concurrently with the program execution that immediately succeed M. (We shall refer to the remainder of dynamic execution after some method invocation M as the *continuation* of M.) The main thread will synchronize with the appropriate delegate threads when the result of a delegated method is required by dynamic program execution.

As an example, Figure 2.1 shows a pair of representations of dynamic program executions that manipulate and access dictionary objects. On the left, methods on the objects referred to by a and b are executed serially, in the main thread. On the right, the execution exploits the object-level parallelism latent in the serial execution. With object-level parallelism a and b each have their own delegate threads; instead of invoking methods on a and b, the main thread sends messages to the delegate threads for a and b.

In the remainder of this chapter, we will define our terms more precisely (Section 2.1), discuss necessary safety properties (Section 2.2), and describe the operation of a program running with OLP (Section 2.3). We then close by placing PIMA and OLP in the context of related work (Section 2.4). We will discuss static analyses to help ensure safety properties and runtime support for exposing OLP in Chapters 4 and 5, respectively.

2.1 PRELIMINARIES

Before we discuss object-level parallelism and PIMA, we will cover the terms we will use in discussing object-oriented language features, some of the relevant particulars of the Java language, and some of terms and concepts involved in identifying safe parallel executions.

Basic definitions

Object-oriented languages like Java typically provide at least three kinds of methods. *Instance methods* operate on a particular object and can access its private state. Instance methods must be invoked via a reference to a *receiver object* (that is, the object the method will be operating on); the receiver object is accessible within the method body via a special *implicit reference parameter* (typically called this or self). For example, using Java syntax, the statement x.f() invokes the f() instance method; the object referred to by x is the receiver. In contrast, *class methods* (also known as static methods) do not operate on a particular receiver object and are shared among all instances of a particular class. *Constructors* serve to set up an object's state; a given object lifetime will have exactly one associated constructor invocation prior to any instance method invocations. (Java, like other safe object-oriented languages, enforces this constraint.)

Two types of instance methods are worthy of particular attention. Informally, *accessor methods* inspect the private state of an object without changing it. *Mutator methods* modify the private state of an object (whether or not they inspect it). We shall refine these definitions shortly, after discussing the sorts of computational effects Java methods may exhibit.

Effects in Java

We treat effects in much greater detail in Chapter 4; here our goal is to present enough background and terminology in order to clarify our concepts of purity and safety. We begin by stating our intuition about effects: a *side effect* consists of a read or write to mutable state. As in Lucassen and Gifford's pioneering work on effect systems (1988), our broad goal in employing effects is to identify noninterfering computations that may execute concurrently.[2] To do so, we ascribe *effect signatures* to statements and method bodies. Just as a type signature is a representation of the range of possible values that a variable might assume or of the values that a function might operate upon and return, an effect signature is an approximation of the range of possible effects that a statement or method body may exhibit.

As we shall see, our intuition – merely defining side effects as reads or writes to mutable state – is too restrictive. Some effects, like incrementing a for loop induction variable, are not typically observable outside of the method in which they occur and thus cannot introduce data dependences between method invocations. As a step towards ignoring unobservable effects, we first identify two kinds of state:

Definition 2.1 *Ephemeral state* exists for a fixed, lexically-scoped lifetime and may be inaccessible for some subset of its lifetime.

Definition 2.2 *Durable state* may persist beyond the scope in which it was created; for example, dynamically-allocated heap objects are durable.

Stack-allocated storage, for example, has a lifetime corresponding to a method activation and is only available when the activation record in which it lives is at the top of the stack. At a lower level, compiler-generated temporaries or data on the Java value stack are also ephemeral. By contrast, durable state must be *sharable*: if a datum may persist beyond the scope in which it was created, then it must be possible to have multiple references in separate method activations that refer to it and alias one another.

For the purposes of ascribing effect signatures to a method, we can ignore effects that are only visible within that method. Since the lifetime of ephemeral state is necessarily bounded by the dynamic lifetime of a method or code block, we need not track effects on some ephemeral location E if we can guarantee that E cannot be modified implicitly. That is, if E is modified via assignments to a particular local variable – and only via assignments to that variable – then it suffices to track data dependences on E through standard dataflow analysis. As we shall see, in Java, modifications of local variables can only affect ephemeral state, and modifications of durable state can be syntactically differentiated from modifications of ephemeral state.

[2] In contrast to the system presented by Lucassen and Gifford, we do not consider allocation of heap objects as an effect.

In Java, all structures and arrays exist only in the heap as durable state and are accessed via opaque references. Local variables may be scalar-valued or may hold a reference to durable state, are stored on the stack, copied in assignment, and passed by value to other methods. It is impossible to create a reference to a stack value; changing the value of one local will not affect the value of a local with a different name. By contrast, it is possible to have several reference variables that alias the same heap object. Changing the value of one of these variables will change its referent and will not change the referents of any aliasing variables, but changing the contents of an object through a reference will, of course, affect any code that accesses the same object through an aliased reference.

Local variables are mutable, but any data dependence introduced by modifying local variables is *explicitly* transmitted by copying values to other variables or passing values as parameters to other methods, and can thus be tracked with program dataflow. On the other hand, computational effects that modify the state of heap objects are of special concern since these represent data dependences that are communicated *implicitly* via aliasing. Put another way, a data dependence between expressions involving the values of local variables will be obvious (since the location that stores the value of a local variable may be referred to by exactly one name), but a data dependence between expressions that read and write heap data may be harder to find, since several names may be aliases for the location of a value in the heap.

The first important detail about our treatment of effects is that we ignore effects on *ephemeral state* when constructing effects signatures for methods or statements. Since ephemeral data in Java must be copied or passed explicitly to other locations, effects on one ephemeral datum will be independent of effects on any other. Furthermore, since ephemeral data do not persist beyond the lifetime of a given method invocation, it is impossible for the ephemeral effects of two distinct method invocations to interfere.

We cannot ignore effects on durable state, and so we need some way to characterize effects on heap objects. In the discussion that follows, we shall assume that Java programs have been translated into a form so that any field access is through a minimal path, or one with exactly one field name in it; it is trivial to do this translation by introducing temporaries, as in Figure 2.2.[3] We further assume that each assignment statement either has a field reference on

[3]Note that this translation is done automatically by Java compilers as part of the translation to bytecode, and the analyses presented in Chapter 4 operate on bytecode. We mention it here to facilitate our presentation of the concepts of effects in the context of Java source code,

```
public class Tree {
   private Tree left, right;
   private Object data;

   // other methods

   public Tree foo() {
      return this.left.left.right;
   }

   public Tree foo2() {
      Tree t1 = this.left;
      Tree t2 = t1.left;
      Tree t3 = t2.right;
      return t3;
   }
}
```

Figure 2.2: Example of minimal field access paths. The method foo2 is equivalent to the method foo, but it does not feature access paths of a length greater than 1.

the left hand side or the right hand side, but not both. (That is, every statement is either a load or a store, but no statement is a load and a store.) Finally, we treat array accesses in a similar fashion: we require that each statement has at most one array access, and we assume that accesses to multidimensional arrays have been translated, via temporaries, to a sequence of accesses to one-dimensional arrays.

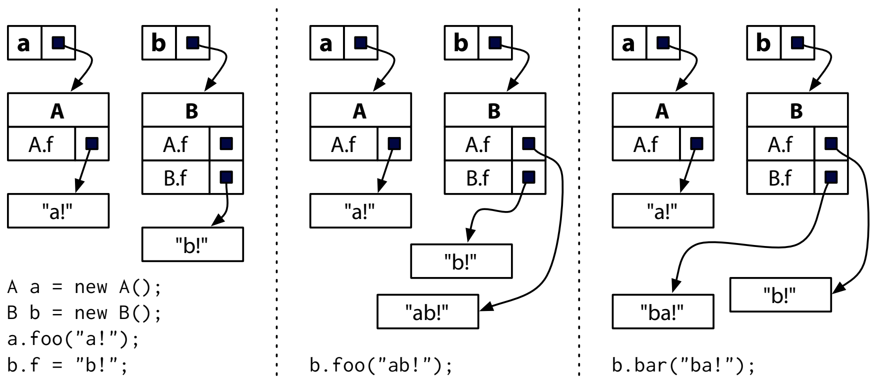
Like most object-oriented languages, Java supports *inheritance*; a *derived class* may extend a *base class* by adding fields and adding or overriding methods. Multiple classes may declare fields with the same name, even if one of these classes extends the other. Figure 2.3 shows two example classes in such a relationship. A is extended by B; both declare fields named f. The Java language specifies that the particular field accessed by a field access expression depends on the type of the reference to the containing object. The bar method in B demonstrates this by writing, through aliasing references of different types, to

which is more familiar to most readers than Java bytecode.

```
public class A {
   Object f;
   public void foo(Object o) {
      this.f = o; // assigns to A.f
   }
}

public class B extends A {
   Object f;
   public void bar(Object o) {
      A a = (A)this;
      a.f = this.f; // writes to A.f, reads from B.f
      this.f = o;  // writes to B.f
   }
}
```

Figure 2.3: Example of inherited classes that share field names



```
A a = new A();
B b = new B();
a.foo("a!");
b.f = "b!";
```

```
b.foo("ab!");
```

```
b.bar("ba!");
```

Figure 2.4: Memory diagrams illustrating state changes effected by field operations.

both the f field declared in A and to the f field declared in B.[4] Figure 2.4 presents memory diagrams of example instances of A and B after various operations take place.

A field reference as it appears in Java source code – for example, x.f – is ambiguous out of context. We need to know the type of x in order to determine which memory locations might be affected. (As Figure 2.4 shows, an instance of B has distinct fields A.f and B.f.) When we discuss the effects of instance methods, we will avoid this ambiguity by describing fields with an *abstract field reference* containing both a field's name and the name of its declaring class.

Definition 2.3 An *abstract field reference* is a 3-tuple $\langle \rho, \kappa, \nu \rangle$ where ρ is a region containing the object that owns the field, either a set of abstract region names or ρ_ω, the universal region; κ is the name of the class declaring the field being accessed; and ν is the name of the field as declared in κ.

Given our simplified form for Java statements, each statement that does not invoke a method may have exactly one effect: it may exhibit a READ or WRITE effect by loading from or storing to an abstract field reference, respectively. (We treat array loads and stores with a special abstract field reference containing the class of the array and a distinguished field name corresponding to "any array element.") A statement containing a method invocation I includes all the effects of every statement of every method body that could be selected by I. Calculating the effects of a method body is straightforward: if the effect of some statement S is given by $\mathsf{fx}(S)$ and the set of statements in a method M are given by $\mathsf{stmts}(M)$, then the effects of a method M are defined as: $\mathsf{fx}(M) = \{\mathsf{fx}(S_i) : S_i \in \mathsf{stmts}(M)\}$

An abstract region describes some part of durable state in which an effect may occur. We say that an effect or abstract field reference *implicates* the set of abstract regions in which it may occur. We defer our primary discussion of the representation of abstract regions until Chapter 4. However, we shall note two details about regions at this time:

First, with the exception of the universal region ρ_ω (which aliases all other regions), regions may not alias one another. As a consequence, any ascription of regions must ensure that two statements whose effects implicate disjoint sets of regions must not interfere. Put another way, if the sets of regions implicated

[4]Note that this is unidiomatic Java; typically, an instance field is declared as private, indicating that it may only be accessed or manipulated from within methods in its declaring class.

by two effects do not intersect, then two statements exhibiting those effects will not access the same concrete memory location. (We term an ascription of regions that satisfies this property *sound*.)

Furthermore, it is possible to parameterize method summaries on receiver objects. That is, a method summary may describe effects on the receiver object by implicating a special ρ_{this} region rather than by implicating the regions containing every possible receiver object explicitly. (This is analogous to universal polymorphism over receiver objects or to an object-sensitive analysis.) We can thus instantiate a method summary at a call site by replacing ρ_{this} with a set of regions containing every possible receiver object.

Identifying effects on fields of the receiver object is straightforward, since accesses to fields of this are, in most cases,[5] statically distinguishable from accesses to fields of other objects. By parameterizing effects on receiver objects, we can ascribe more precise effect signatures to method invocations; as we shall see, we can also use this information to identify methods whose effects are strictly confined to the receiver object.

2.2 PURITY AND SAFETY

With the definitions from Section 2.1, we can now define more clearly the notions of *accessor* and *mutator* methods.

Definition 2.4 An *accessor method* is a method that exhibits at least one READ effect on a field of the receiver object and returns a value.

Definition 2.5 A *mutator method* is a method that exhibits at least one WRITE effect on a field of its receiver.

Note that a method may be both an accessor and a mutator. Good style dictates that a method that changes mutable state will not return a value and that methods that return a value will not change state.[6]

Methods may also be *pure*. The classic definition identifies a method that exhibits no effects on durable state as pure. However, this definition is rather too

[5] The local variable associated with the receiver object is always local 0 at the bytecode level; since the Java language does not permit assignment to this, we know that any field access to the object referred to by local 0 is accessing a field of this. If another local aliases this, this assumption is imprecise but sound.

[6] This principle was named *command-query separation* by Meyer (1988). The PIMA model accounts for programs that do not exclusively exhibit good style, as we shall see in section 2.3.

restrictive, since it doesn't admit idempotent methods that create and modify durable state in order to complete their work. A better definition – due to Leavens et al. (1998, 2006) and applied for static analysis by Salcianu and Rinard – characterizes a method as pure if and only if it does not modify any state that exists immediately before method entry. This less-restrictive definition of purity captures a notion of method purity as the absence of potential interference with other code: a method may have effects on durable state that it creates – and that is not available to other methods until after it exits.

We can also identify some methods as *read-only* – these are methods that may have READ effects (but not WRITE effects) on durable state that exists before method entry. Note that all pure methods are also read-only methods.

If we are to characterize the purity of methods in typical object-oriented programs, we may wish to characterize instance methods by the effects that they have on durable state that exists outside of the receiver object. As a consequence, we develop notions of pure and read-only methods that are sensitive to whether effects occur to state within the receiver object or not:

Definition 2.6 An *externally-pure* method is one whose effects on durable state occur only to the receiver object (that is, in the this abstract region) or on state that did not exist immediately before method entry. Put another way, an externally-pure method is pure with respect to all state outside of the instance it is operating upon. All pure methods are also externally-pure.

Definition 2.7 An *externally-read-only* method is one whose WRITE effects on durable state occur only to the receiver object or to state that did not exist immediately before method entry. Put another way, an externally-read-only method is read-only with respect to all state outside of the instance it is operating upon. All externally-pure methods are also externally-read-only.

Intuitively, externally-pure methods M_i can execute asynchronously on some receiver object O provided:

1. they execute in the order in which they would be invoked in a sequential execution, and
2. the main thread synchronizes with the delegate thread for O, ensuring that all pending method invocations on O have completed, before either reading the state of O or modifying the state of O without using a mutator method on O.

The former constraint can be met by replacing method invocations with support code to insert the invocation into a queue of pending asynchronous calls in the delegate thread. If all method invocations are enqueued for the delegate thread, then they will execute in program order. The latter constraint can be satisfied statically, by demarcating only objects whose fields are strictly accessible through instance methods, or dynamically, by placing code in field read or write barriers to block until all pending activations complete if a load or store is to a field of a demarcated object.

If these conditions are met, then PIMA can guarantee a race-free execution with *sequential execution equivalence*, which we define as follows:

Definition 2.8 *Sequential execution equivalence* (SEE) is a property of a parallel execution that holds when every dynamic read sees the value written by the same dynamic write that it would in a serial execution with the same inputs.

Many methods are externally-pure, but many interesting methods are externally-read-only or may even modify state contained in objects other than the receiver. PIMA would be of limited utility for real Java programs if it were restricted to exploiting OLP only in externally-pure methods, especially since the effects of many methods that are not externally-pure are unlikely to interfere with those of methods executing on other objects. For example, an externally-read-only method might read only data that will not change during its continuation. A method that is not externally-pure might only modify objects that are completely encapsulated by its receiver. It should be clear that there are methods that are not externally-pure but that can safely execute in parallel without violating SEE, and these can often be identified by programmers. In Chapter 4, we define more clearly the sorts of effects that will not cause a method invocation to violate SEE and present analyses that automatically identify a broad range of methods that are suitable for safe object-level parallelism.

2.3 THE PIMA MODEL

PIMA extends the Java language with support for asynchronous method invocations in order to exploit OLP. As a consequence, it introduces parallelism by executing some instance method invocations concurrently with their continuations, synchronizing when the result of a computation is required by dataflow in the main thread. In this respect, PIMA is similar to the *future* abstraction of

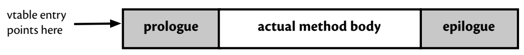


Figure 2.5: Typical method layout, with virtual function table pointer.

Baker and Hewitt (1977) and Halstead (1985); the *wait-by-necessity* concept of Caromel (1993); and the *lazy wait* construct of Meyer (1993). PIMA takes the flexibility of *future*, which can be applied to any expression, and presents an abstraction that is applicable to object-oriented programs, in which the "result of a computation" may be a change to the durable state associated with an object and not merely the irreducible value of an expression.

Other extant methods for parallelizing object-oriented programs designate objects for parallel execution based on class membership; for example, methods on instances of classes that extend `Thread` in Java (Lindholm and Yellin 1999) or on instances of classes annotated **separate** in Eiffel (Meyer 1993) may execute in a separate thread. PIMA, on the other hand, identifies potential concurrency in a manner orthogonal to class membership: by annotations on an instance. Whether or not an object's methods may execute asynchronously with PIMA is simply a property of the instance and the context of the method invocations. As a result, PIMA enables programmers and tools to opportunistically exploit implicit parallelism that library developers and application programmers may not have anticipated.

Logically, PIMA associates one *delegate thread* with each object. A delegate thread is a process that exists to execute asynchronous method invocations. More than one asynchronous method invocation may be in flight on a particular delegate thread at any one time; the delegate thread maintains a queue of invocations to execute in program order. While each object has its own logical delegate thread, only a small number of delegate threads will be active (that is, currently executing on virtual processors in the virtual machine or runtime) at any given time.

Before we discuss the operational details of PIMA, we shall review some details of runtimes for object-oriented languages. A standard layout for a compiled method consists of a *prologue*, which sets up the activation record and ephemeral state for the method invocation; the method body, or actual instructions of the compiled method; and the *epilogue*, which cleans up after the method. In order to support receiver polymorphism, each object has a *virtual function table* (or *vtable*), which stores the addresses of methods with certain names; this allows methods to be dispatched dynamically based on the class of the receiver. (Each object logically has its own vtable, but one physical vtable may be shared among all instances of a class.) Figure 2.5 shows the layout of a method and the destination of a virtual function table pointer referring to the beginning of that method.

An object is demarcated for asynchronous execution by a special annotation. When the runtime encounters this annotation on a reference R, it does several things; we first provide a brief overview before discussing each task in greater detail:

1. First, the runtime ensures that R is currently a suitable candidate for parallel execution. Specifically, it ensures that a virtual processor is available to execute methods on R. If one is not available, the annotation is ignored and execution proceeds serially. If one is available, the runtime continues with the next task.
2. The runtime then creates a delegate thread for the object referred to by R, if one does not already exist, and binds this thread to an available virtual processor.
3. Finally, it replaces the instance methods of the object referred to by R with special *proxy methods* that control communicating information about asynchronous invocations to the delegate thread and synchronizing with the delegate thread when its results are required by the main thread.

The first task that the runtime must perform with a demarcated object is ensuring that it is a good candidate for parallel execution. This has two components: safety and profitability. In a simple implementation of PIMA, safety is the responsibility of the programmer or compiler – it is not checked by the runtime – and ensuring profitability is limited to ensuring that there is a virtual processor available to execute the methods of a given object asynchronously. If effects annotations are available, then the runtime can use those in order to guarantee that executing a method asynchronously will not violate SEE. (Chapter 4 presents a system for automatic inference of computational effects.)

The runtime must then create a special delegate thread to execute asynchronous method invocations for a given object and schedule this thread to run on an available virtual processor. This thread simply repeatedly dequeues method invocations – consisting of method addresses and "environments" of actual parameter values. It then executes method invocations by setting up a synthetic prologue – that is, a stack frame with the actuals for a particular

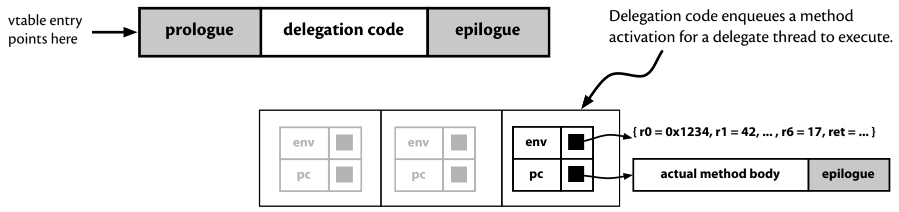

vtable entry points here
prologue
delegation code
epilogue
Delegation code enqueues a method activation for a delegate thread to execute.
env
pc
env
pc
env
pc
{ r0 = 0x1234, r1 = 42, ... , r6 = 17, ret = ... }
actual method body
epilogue

Figure 2.6: Proxy for a mutator method, with vtable pointer and updated delegate invocation queue.

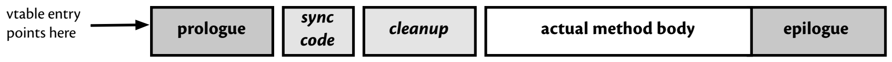


Figure 2.7: Proxy for an accessor method, with vtable pointer, synchronization point, cleanup code, and original method body.

invocation and a return address that points back to the delegate thread loop – before jumping to the beginning of the actual method body.

In order to ensure that instance methods are actually delegated, the runtime must replace instance methods on a delegated object with *proxy methods*. Since mutator methods may execute asynchronously, their proxies, as shown in Figure 2.6, enqueue a method invocation record to the delegate thread. If only the main thread is delegating method invocations, then changes to object state will occur in program order. Accessor methods, on the other hand, must wait for all pending state changes to complete before they inspect the state of a demarcated object. As a consequence, their proxies are rather more complicated (see Figure 2.7): they block, waiting for all pending delegated invocations to complete; they optionally "clean up" by reverting the object to an un-demarcated object; and finally, they execute the accessor method body in the main thread. Obviously, some methods both modify object state and return values; PIMA generates accessor proxies for these.

2.4 RELATED WORK

Many other parallelism models have been proposed for constructing parallel object-oriented programs, or for adding concurrent execution to otherwise serial programs with minimal effort. In this section, we place PIMA and OLP in context by examining the most relevant related work.

There is a long history of languages that support the deliberate construction of parallel programs – as we have mentioned, Simula 67 (Dahl et al. 1970) provided a mechanism to associate objects with separate threads of control. A great deal of effort has gone towards developing models and abstractions that make it easier to write programs that make progress and avoid data races, often by integrating parallel execution or synchronization with other language features like modules or scope.

Concurrent mostly-functional evaluation

ALGOL 68 (van Wijngarden et al. 1975; §3.3) provided *collateral clauses*, which were sequences of statements with an unspecified evaluation order; the clause would complete when all of the statements it contained had completed. ALGOL 68 also provided *parallel clauses*, or collateral clauses with semaphore-based synchronization. By treating routine parameter bindings as collateral or parallel clauses, ALGOL 68 enabled the possibility of concurrent parameter evaluation and recursion parallelism.

Friedman and Wise present opportunities for parallelism available in a purely applicative (i.e. side-effect free) language with "suspended *cons*." When *cons* is evaluated in their language, it returns a structure consisting of *suspensions* of its arguments. A suspension is like a *promise* in Scheme (Abelson et al. 1998) or a call-by-need thunk: a procedure that is evaluated at most once; it can be *coerced*, or evaluated and replaced with its result. In a language in which every *cons* is suspended, idle processors may be devoted to coercing suspensions that are soon to be required by the main computation. The approach Friedman and Wise describe is transparent to the programmer and relies on cooperation between the compiler and the runtime system.

Baker and Hewitt (1977) coin the term *call-by-future* to describe a calling convention in which each formal parameter of a subprogram is bound to a process that will evaluate the expression passed to that parameter. (Unlike the concurrent parameter evaluation available in ALGOL 68, Baker and Hewitt's approach is implicit.) They also introduce the notion of a *future*, a 3-tuple of: (1) a process to evaluate an expression E, (2) a memory location to store the result of evaluating E, and (3) a queue of processes waiting for E's result. Baker and Hewitt describe using futures for eager evaluation in applicative languages; thus, the main focal point of their work is on garbage collection of processes that are evaluating expressions whose results are not required by program dataflow.

The Multilisp language (Halstead 1985) generalizes call-by-future with the future annotation. Binding a variable x to (future v) spawns a process to evaluate v and binds x to a token representing its eventual result. Internally, variable accesses perform a touch operation, which checks the run-time type tag of an object to see whether or not it is a suspended future. If it is, then the accessing process blocks until the calculation of the future is complete. In programs written in mostly-pure languages like Multilisp, there is often a substantial amount of implicit parallelism. By annotating a program with future, a programmer may indicate to a compiler which tasks are most profitable to execute in parallel.

Moreau (1996) developed a semantics for Scheme with future; his semantics treats side effects and first-class continuations. He establishes future as a semantically-transparent annotation by way of several successive abstract machines, each with its own operational semantics. A formalism for future that includes side effects is a necessary step towards reasoning about real programs with future. However, the treatment of side effects in Moreau's semantics is quite conservative, requiring synchronization upon any access to shared memory if a process computing the value of a suspended future has accessed shared memory.

A future-like construct is part of the current Java concurrency library (Lea 2004), although Java futures are opaque placeholders for eventual values rather than transparent promises. Several groups have sought to increase the applicability and convenience of Java futures. Notably, Pratikakis et al. (2004) developed type-safe *transparent proxies* for Java future objects. Welc et al. (2005) demonstrated the feasibility of using software transactional memory to guarantee the safety of Java futures.

Concurrent objects

The future model is flexible enough to expose a great deal of parallelism in mostly-functional languages with few or no side effects, but may be lacking for contemporary imperative object-oriented languages, in which the result of a computation is as likely to be an update to the state of some object as it is to be a simple value. Furthermore, imperative languages with shared state require some form of mutual exclusion between threads, whether by synchronization at any change to sharable state (as Moreau's approach), or by using optimistic concurrency control (as Welc et al.).

The *monitor* abstraction (Hoare 1974) unifies mutual exclusion and lexical scope, making it more difficult for programmers to accidentally fail to acquire (or release) a lock. Monitors find widespread application today in the Java language, standard library, and virtual machine (Lindholm and Yellin 1999), which provide support for multiple concurrent threads of execution and a limited form of monitors. In Java, the notion of concurrency is tied to the notions of inheritance and subtyping. A Java thread is an object, and it appears to the programmer as an instance of either a subtype of `java.lang.Runnable` or a subclass of `java.lang.Thread`. Such classes are treated specially by the virtual machine, and contain methods to spawn new lightweight processes.

Two proposals to extend Eiffel with concurrency brought future-like concurrency more fully into the object-oriented world. In both *Eiffel//* (Caromel 1993) and *SCOOP* (Meyer 1993), the results of an asynchronous computation may be some change to the state of a particular object. The major differences between these two proposals relate to how parallelism is introduced and when results are required.

Meyer's (1993) SCOOP system integrated concurrency naturally into Eiffel's *design-by-contract* mechanism. In SCOOP, objects and classes may be declared as separate.[7] Methods invoked on separate objects or instances of separate classes will execute asynchronously on a separate virtual processor. Synchronization and ordering constraints on methods of separate objects are enforced by the precondition assertions required by design-by-contract, and asynchronous method invocations are queued by the separate object and executed in program order. The client of some method of a separate object will block waiting for its result (and the completion of any pending methods on the base object); the semantics here are akin to spawning a future and immediately executing a touch operation. As a result, a client of a separate object will only block when it requires a value from that object, and no sooner. Meyer calls this property *lazy wait*. Lazy wait is possible because Eiffel distinguishes between *commands*, or methods that update state but return no value, and *queries*, or idempotent methods that inspect state.

Caromel (1993) presents *Eiffel//*, a method for integrating concurrency into Eiffel that differs from Meyer's in several important respects. The main and most important difference is that Eiffel// distinguishes between *passive objects* and *active objects*. Active objects are those that have an associated independent thread of control, and references to them may be shared between

[7]It is important to note that the separate designation is orthogonal to the inheritance mechanism; a separate class may inherit from a non-separate class and vice versa.

processors. By contrast, methods on passive objects execute in the same thread as the caller; furthermore, passive objects have thread-specific visibility and cannot be shared between processors. Another difference between Eiffel// and SCOOP is that the concurrency model in Eiffel// is integrated into the inheritance mechanism: all active objects inherit from a class *PROCESS*. Finally, Eiffel// treats the results of asynchronous method calls more like futures than SCOOP does. Whereas SCOOP synchronizes when the result of an asynchronous method call is *assigned* to some variable, Eiffel// synchronizes when the result is *used*. Caromel terms this property *wait-by-necessity*.

PIMA *in context*

Our goal in designing PIMA was to develop a model for parallel construction and execution of object-oriented programs that was flexible, imposed little cognitive burden on the programmer, and was amenable to automatic safety checking and automatic parallel task extraction.

future and related models like ALGOL 68's collateral clauses integrate well with existing language features (i.e. expression evaluation or block structure) and present a reasonable mechanism for evaluating independent expressions concurrently. However, these models are not well-suited for concurrent execution of object-oriented programs, which exhibit many computational effects and in which the result of a subprogram call may well be state changes and not merely eventual values. A future-like construct — confusingly, also called `Future` — appears in the Java standard library as of Java 5, but it suffers from the same limitation: rather than exposing the capability to execute arbitrary application code in parallel, it simply presents a mechanism to execute a task and block when its final value is required.

The Eiffel// and SCOOP models, due to Caromel and Meyer, are a better fit for purposeful construction of object-oriented programs, and impose little cognitive burden on the programmer, since they are integrated with well-understood language features: Caromel unifies parallelism and inheritance (all active objects are instances of classes that extend *PROCESS*), while Meyer unifies parallelism and class membership (all objects that may run in separate threads are instances of classes that are designated separate, but this designation is orthogonal to inheritance). However, this mechanism is somewhat inflexible, since a programmer must decide early on that either *all* instances of some class will execute in their own threads or *none* of them will.

Standard Java threading unifies parallelism with subtyping and offers monitor-based concurrency control. Java threads are always available to Java programmers, and the lexically-scoped monitors eliminate some common errors. Furthermore, Java's standard library commendably provides synchronized data structures. However, the Java approach is *inflexible*, in that it requires a particular structuring of a class hierarchy and demands that programmers encode some machine-specific information – about the number of threads that they expect to use and the size of parallel tasks – into any parallel decomposition. This inflexibility impacts not only programmers but also tools that seek automated parallel decompositions. Furthermore, the standard pitfalls of parallel programming – race conditions, starvation, deadlocks, and priority inversion – are all just as possible when programming in Java threads as in other models. While a great deal of fine work has sought to present restricted models for race-free programming (Boyapati et al. 2002; Permandla et al. 2007) or to find races in standard Java programs (Choi et al. 2002; Naik et al. 2006; Naik and Aiken 2007), it is clear that unrestricted Java threads are not a suitable model for automated program decomposition and are also not readily amenable to automated safety checking.

PIMA imposes little cognitive burden on the programmer by identifying implicit object-level parallelism, finding concurrent tasks from methods on individual demarcated objects. By supporting demarcation annotations on individual objects, PIMA provides the flexibility of future-like models (in which any arbitrary expression may execute in parallel) with the suitability for object-oriented programming of Eiffel// and SCOOP. Finally, it offers a clear path to safety properties via SEE; as we shall discuss in Chapter 4, establishing the safety of preexisting annotations and even adding safe demarcation annotations can be fully automated.

3 DECLARING BYTECODE PROCESSORS AND ANALYSES

For this shall never be proved, that the things that are not are; and do thou restrain thy thought from this way of inquiry.

— Parmenides of Elea (5th C. bce, trans. Burnet)

It is possible, also with the old conception of logic, to give at the outset a description of all "true" logical propositions.

Hence there can never *be surprises in logic.*

— Ludwig Wittgenstein (1917, trans. Ogden)

In this chapter, we introduce the DIMPLE$^+$ program analysis framework for Java bytecodes. DIMPLE$^+$ (pronounced "successor dimple") extends our earlier work on the DIMPLE framework for interactive, declarative, scalable program analysis (Benton and Fischer 2007) by adding several major features: static single assignment (SSA) form, explicit domains for analysis entities, support for multiple analysis solver backends, and an interface for annotating Java bytecodes with analysis results. This chapter is self-contained and does not assume prior knowledge of the original DIMPLE system.

Program analyses provide static answers to questions about the dynamic behavior of a program. A researcher who wishes to develop a new program analysis must engage in two separate activities: devising a specification for the analysis and engineering a suitably efficient implementation. Developing a correct specification can be difficult and error-prone, as a correct analysis specification must properly treat all of the features and subtle corner cases in a given language. However, even given a correct specification, the effort necessary to produce an efficient implementation may be substantial or prohibitive. The implementer's task is difficult due to the disconnect between a formal analysis specification and an executable implementation.

Analysis designers typically specify analyses in a declarative style: whether as a system of constraints (Andersen 1994; Fähndrich and Aiken 1997; Rountev et al. 2001; Kodumal and Aiken 2005; Milanova et al. 2005), as type inference rules (Palsberg 2001; Steensgaard 1996; Diwan et al. 2001; Aldrich et al. 2002), or as a fixed-point of a system of dataflow equations. Many analyses also admit natural specifications as reachability queries on context-free languages or as logic programs (Reps 1998; Esparza and Podelski 2000; Sridharan et al. 2005;

Tomb and Flanagan 2005; Whaley and Lam 2004; Lam et al. 2005). General solvers for these sorts of problems have benefited from advances in logic programming language implementation; several are quite efficient. However, most existing solvers are far from ideal for rapid prototyping of new analyses.

Consider an idealized version of the process a researcher undertakes when developing a new program analysis. The designer:

1. Decides which abstract domains are interesting, and how best to model program properties as mathematical structures;

2. decides how individual kinds of program statements contribute to the analysis results;

3. writes a preprocessor to extract analysis relations from program source code; and finally

4. develops some sort of solver (or uses a preexisting specialized solver) to answer queries about the generated relations.

Since the analysis itself is likely to be specified declaratively, it would be ideal to use an analysis framework that could execute such a specification with little or no modification. Unfortunately, almost all existing analysis frameworks that support declarative specifications only aid users with the final step of this process: namely, providing a solver for analysis rules in a particular formalism. Many require extensive preprocessing or time-consuming automated optimizations before running an analysis, thus discouraging an interactive development style, casual experimentation, and rapid prototyping of new analyses.

We have developed DIMPLE$^+$, a fully-featured, declarative, and extensible analysis framework for Java. It is *fully-featured* in that it provides functionality for every step of the analysis design and development process. It is *declarative* in two ways: first, user-specified rules are written in a declarative language; also notably, DIMPLE$^+$ itself is primarily implemented in the Yap Prolog system (Costa et al. 2000). Finally, the major components of DIMPLE$^+$ are *extensible*: statement preprocessors may depend on arbitrary, user-defined Prolog procedures; and DIMPLE$^+$ is designed to support multiple, user-definable solvers to generate answers to analysis queries.

DIMPLE$^+$ consists of a typed intermediate representation of Java bytecodes and a domain-specific language that allows users to specify program analysis implementations in an essentially declarative fashion, much as they might write an analysis specification. Unlike most program analysis frameworks, DIMPLE$^+$ represents program texts, derived relations, and analysis rules uniformly – all as relations in a Prolog database. (If users choose to use the built-in tabled Prolog solver, then analysis results are also represented as Prolog clauses.) DIMPLE$^+$ thus enables analysis designers to develop declarative specifications for every stage of analysis development and evaluation: deciding which domains are under consideration and which program statements are relevant, extracting relations from relevant program statements, and answering queries based on a system of relations and analysis rules.

DIMPLE$^+$ is essentially different from other analysis frameworks in that it offers two important capabilities:

1. DIMPLE$^+$ provides a total, round-trip solution to analysis design and evaluation. That is, an analysis designer may use DIMPLE$^+$ to produce a declarative specification for every component of an analysis. DIMPLE$^+$ allows the user to define procedures for any necessary preprocessing of the program text in Prolog. Given declarative, user-supplied specifications, DIMPLE$^+$ then automatically generates a statement processing routine that generates relations from the program text and the rules that govern analysis results. Finally, DIMPLE$^+$ supports generating analysis solutions either via interfacing with an external solver or via a built-in solver based on tabled Prolog.

2. DIMPLE$^+$ provides a relational, declarative model for specifying program analyses. An analysis designer may thus execute an analysis specification that is very similar to the sort of specification that might appear in a technical paper.

There exist several excellent solvers that have been used to great effect for analysis problems. However, these tools require that the analysis designer use some external tool to develop a preprocessor that translates from program text to relations. Some analysis frameworks, such as those available within research compilers, enable users to develop preprocessors as well as solvers within the same tool, but DIMPLE$^+$, like its predecessor DIMPLE, enables round-trip analysis development in a declarative style.

In the DIMPLE$^+$ framework, program texts (in an intermediate representation of bytecode instructions), analysis rules, and analysis results are stored in a database of Prolog relations. Since analysis results are in the same format as the program to be analyzed, it is possible for analysis designers to reuse the results of costly analyses as program annotations – and, in fact, to store these annotations directly in the same database as the program text instead of recalculating the analysis results later.

The DIMPLE$^+$ program analysis framework features:

1. a system for encoding an intermediate representation of Java bytecodes as a database of facts,

2. a declarative framework for program analysis, which enables analysis users to prototype, develop, and debug new program analysis specifications by generating analysis implementations directly from specifications, and

3. a mechanism for debugging program analysis specifications that combines the efficient execution of tabled evaluation with the tracing capacity of conventional Prolog evaluation.

The combination of these features in one system enables rapid and seamless analysis design, development, and evaluation.

3.1 OVERVIEW OF THE SYSTEM

DIMPLE$^+$ comprises three parts. The front-end translates from Java bytecodes to the DIMPLE$^+$ intermediate representation (IR): a set of Prolog relations that fully describe the input application and library classes. The back-end provides a framework of rules and a domain-specific analysis language to implement program analyses as declarative queries on a database of relations; it translates from user-specified statement processing and analysis rules into code that implements a bytecode preprocessor and the actual analysis. The query engine actually generates an exhaustive solution to the analysis problem. In the original DIMPLE system, the query engine was part of the back-end and implemented in tabled Prolog; DIMPLE$^+$ enables users to plug-in different external solvers to meet different application requirements.

We have implemented the DIMPLE$^+$ front-end as a whole-program transformation that extends the Soot compiler framework. Soot (Vallée-Rai et al. 1999) converts from stack-based bytecode to a typed three-address representation called Jimple and generates a conservative method call graph. Our transformation then translates from Jimple's abstract syntax to the concrete syntax of a database of DIMPLE$^+$ IR relations. As we have stated, methods are modeled as sets of statements and all intraprocedural control flow is modeled by an explicit control-flow graph. It is therefore possible to use DIMPLE$^+$ to implement either flow-sensitive or flow-insensitive analyses; the analysis developer need merely decide whether or not to ignore control flow when declaring analysis rules.

We have implemented the DIMPLE$^+$ back-end in the Yap Prolog system (Costa et al. 2000). The back-end consists of:

1. a *domain-specific language* for specifying analyses,
2. a *library of relations and rules* that are available to client analyses and describe various aspects of Java's semantics and type system,
3. a *code generator* that translates from the domain-specific language into programs that implements a statement processing routine and an analysis evaluation routine, and
4. an *interface to a query engine* that executes the generated code to produce exhaustive analysis results.

The DIMPLE$^+$ back-end supports multiple query engines and enables users to develop support for their own query engines. However, it also includes a built-in query engine that makes use of the Yap system's support for *tabled execution*. Tabled Prolog (Chen and Warren 1996; Rocha et al. 2005) supports a *table* annotation on certain predicates. The results of a tabled predicate are memoized; thus, each is evaluated at most once for each tuple of arguments. Furthermore, a tabled Prolog system can automatically find a finite fixed point of recursive and mutually recursive tabled predicates. This property enables analysis designers to specify many relations that naturally admit a left-recursive definition in a straightforward and direct way, without considering nonterminating evaluation.

3.2 USING DIMPLE$^+$

We shall now present a high-level view of DIMPLE$^+$ and discuss a typical user's interactions with the DIMPLE$^+$ system while developing a new analysis. Figure 3.1 provides an overview of DIMPLE$^+$'s architecture; we shall refer to each component in the following overview.

A DIMPLE$^+$ user is likely to begin by creating databases of DIMPLE$^+$ IR relations for several interesting programs, in order to evaluate a new analysis on representative inputs. This task requires the DimpleGenerator application, which processes Java class files. DimpleGenerator extends the Soot compiler

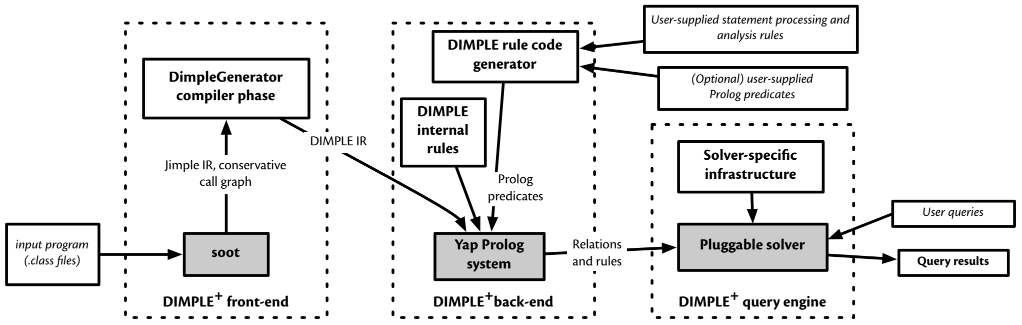

DimpleGenerator compiler phase
Jimple IR, conservative call graph
DIMPLE IR
input program (.class files)
soot
DIMPLE+ front-end
DIMPLE rule code generator
DIMPLE internal rules
Prolog predicates
Yap Prolog system
DIMPLE+ back-end
Relations and rules
User-supplied statement processing and analysis rules
(Optional) user-supplied Prolog predicates
Solver-specific infrastructure
Pluggable solver
User queries
Query results
DIMPLE+ query engine

Figure 3.1: High-level overview of the DIMPLE$^+$ system's architecture. Shaded items indicate infrastructure developed by other groups; italics indicate user-supplied inputs

framework and produces a DIMPLE$^+$ IR database with relations corresponding to every statement in a given application and in each of its libraries. The DIMPLE$^+$ IR also includes relations describing the application and library class hierarchies and relations describing a conservative approximation of the method call graph. (We discuss the DIMPLE$^+$ IR further and provide some example relations in Section 3.3.)

The user may then load the IR database into the Prolog system and begin asking queries about the program. This sort of interactive experimentation can often be a productive prelude to developing the actual statement processing and analysis rules. By exploring the sorts of statements and relations that may be relevant to a particular analysis in advance of specifying the analysis, the user may be able to note subtle details that might have otherwise been overlooked. After some experimentation, the user will begin to use the DIMPLE$^+$ analysis language to specify a statement processing routine and the analysis. (We discuss the DIMPLE$^+$ analysis language in Section 3.4.)

Most analyses will only need a subset of the relations in the DIMPLE$^+$ IR database. All analyses will benefit from simplifying the program relations by preprocessing the input program and deriving analysis-specific relations from the DIMPLE$^+$ IR. (At the very least, doing so will result in a clearer presentation.) Consider the following DIMPLE$^+$ IR relation, which corresponds to an assignment from one local variable to another:

```
stmt(unit(83),
   assignStmt(local(local(12), method(62806), primtype(long)),
              local(local(14), method(62806), primtype(int)))).
```

DIMPLE$^+$ creates explicit domains for most program entities of interest – like statements, local variables, fields, and methods – and ascribes a positive integer to each that uniquely identifies it in that domain throughout the whole program. The clause above describes the statement at the globally-unique program counter 83; it is of the form `stmt(unit(PC), S)`, where `PC` is in the domain of program counters (that is, unique identifiers for statements) and `S` is a structure representing a particular statement. The particular statement structure in this example is an `assignStmt`, which represents an assignment between locals. Locals, in turn, are represented as structures with elements from a domain of local identifiers, methods, and types. In this case, local number 12 is receiving a value from local number 14; both locals are contained in a method with unique number 62806.

The original DIMPLE IR used names instead of numbers (so all domains were implicit); DIMPLE$^+$ has only retained names for types. Using names instead of explicit numbered domains may aid readability in some cases, but it does so at the expense of verbosity.[1] Employing explicitly-numbered domains not only makes the IR database smaller, but it also facilitates imposing an ad hoc typing discipline on user-specified analysis and processing routines, enabling DIMPLE$^+$ users to develop more robust code.

The clause above includes a great deal of detail about a particular bytecode statement representing a particular assignment; not all of it may be useful for every analysis application. It may be more convenient for the analysis designer to derive a relation that omits some details that are not relevant to a particular analysis. Perhaps the analysis designer is interested in knowing that an assignment occurred between two locals, but some details, like the program counter, method name, and variable types, are not relevant for a particular analysis problem. In that case, the designer could preprocess the above statement to a much simpler relation, like this one:

```
assign(12, 14).
```

Of course, the derived relations can be more involved than simple projections of structures into simpler structures. The *statement processing routine* decides which statements are interesting for a particular analysis, and it may use arbitrarily complex criteria to do so. To give two realistic examples, particular derived relations may select only those statements that involve assignments to locals of reference types, or only those statements that might throw unchecked exceptions. A DIMPLE$^+$ user will specify a statement processing routine in

[1]The DIMPLE$^+$ representation contains no less information than the original DIMPLE IR.

terms of declarative rules in the DIMPLE$^+$ analysis language; these rules may use user-defined predicates on DIMPLE$^+$ IR statements, DIMPLE$^+$'s internal rules (which include facts and relations dealing with Java's type system and semantics), or any Prolog code. By combining arbitrarily complex selection and projection rules, analysis users can develop extremely powerful preprocessors in a declarative style. (We discuss the part of the DIMPLE$^+$ analysis language that defines statement processing routines in Section 3.4).

Once the analysis designer has declared statement processing rules, DIMPLE$^+$ can preprocess the input program. DIMPLE$^+$ affords several options here: for prototyping, when a user may be interested in having the entire program available, the statement processing routine will simply assert derived relations into memory. DIMPLE$^+$ users who are using external solvers as query engines, or who are using the internal tabled Prolog solver but demand maximum performance, will use DIMPLE$^+$ as a preprocessor. In so doing, they will write derived relations to a file, discarding the IR database so that an analysis will operate strictly on a database of derived relations instead of on a database that contains detailed information about every bytecode statement in the program.

If a DIMPLE$^+$ user chooses to use the built-in tabled Prolog query engine, it is possible to keep the entire IR database in memory. Doing so incurs a cost in heap usage, but has two key benefits: if one is prototyping an analysis, one may decide that new or different derived relations might be useful. Since the whole program is available, one may add these effortlessly. In addition, having access to the whole program offers an advantage when debugging analyses: if the whole program is available, DIMPLE$^+$ allows users to see precisely which IR statements contributed to each derived relation.

The analysis designer may then prototype analysis rules interactively, by issuing queries to the underlying tabled Prolog system. The user may then commence developing analysis rules by writing them in DIMPLE$^+$'s rule definition language. (See Section 3.4 for more information on the rule definition language.) DIMPLE$^+$'s code generator takes statements in the rule definition language and converts them to tabled Prolog rules. The code generator also generates support code to aid debugging analysis rules. The analysis designer will not invoke this support code directly; rather, DIMPLE$^+$ uses it to present derivations of why particular analysis results hold. (See Section 3.5 for more information on this capability.)

By default, generated analysis rules are asserted into the Prolog database. The code generator can also write the generated rules and compiler directives to a file, which can be compiled and loaded. Each of these is appropriate for different situations, and DIMPLE$^+$ gives users the opportunity to decide which to use at a given stage in the analysis development process. The former leads to slightly slower execution times but affords greater flexibility for prototyping, since asserted rules may be retracted or abolished. The latter option is most appropriate for production analyses, since it offers the fastest possible execution, but does not offer the opportunity to interactively change the analysis.

We shall now discuss the individual DIMPLE$^+$ components in greater detail.

3.3 THE DIMPLE$^+$ IR

We have developed a typed intermediate representation (IR) for Java bytecodes. Our representation is based on the Jimple IR (Vallée-Rai et al. 1999) but has an essentially declarative flavor: programs are represented as sets of relations, analyses are specified in a domain-specific language that adds rules and relations to the database, and analyses are executed by satisfying queries on generated rules and relations. Java classes are represented in terms of subtyping relationships, sets of method declarations and sets of field declarations. Methods are represented as sets of statements. All intraprocedural control flow is explicit and is modeled by a control-flow graph for each method. The abstract syntax of our representation is loosely based on the Jimple IR for Java bytecodes, which is a component of the Soot compiler framework. We call our representation DIMPLE$^+$ to reflect its *declarative* character and as an homage to prior intermediate representations like Simple (Hendren et al. 1993) and Jimple (Vallée-Rai et al. 1999).

DIMPLE$^+$, like Jimple, is based on three-address code or register quadruples. Each statement consists of variables corresponding to operands, an operation, and a variable in which to place the result. Each statement has a simple, "flat" form: all operands are either local variables, immediate values, or addresses, not complex expressions. Object fields and array elements are treated in a "load/store" fashion: they are loaded from memory, operated on as locals, and stored back to memory. As a consequence, there is no single DIMPLE$^+$ instruction, for example, to transfer a value from one heap location to another, or to increment the value of an object field.

In contrast to the three-address code of DIMPLE$^+$ and Jimple, the Java virtual machine executes bytecode instructions that operate on a stack. Since the stack is untyped, Java provides separate bytecode instructions for each potential operand type. (Lindholm and Yellin 1999) As an example, there are eight different bytecode instructions to load a value from an array element, depending on the element type: one for object references and one for seven of Java's eight primitive types.[2] There are also many "special case" bytecode instructions that incorporate immediate values in their opcodes. Of course, the wide variety of bytecode instructions – and the very nature of stack-based evaluation – greatly complicates analysis of unmodified bytecode. By basing DIMPLE$^+$ on an existing intermediate representation of bytecode that features fewer kinds of operations than does bytecode, it is possible to express analyses in terms of a simpler set of rules than it would if DIMPLE$^+$ were to operate on bytecode directly.

While the structure of the DIMPLE$^+$ IR owes a great deal to the statements and expressions of the Jimple IR, DIMPLE$^+$ has the capacity to be more flexible than a non-declarative representation. This flexibility is the primary advantage of DIMPLE$^+$ over Jimple and other non-declarative representations, and it comes from the fact that DIMPLE$^+$ represents programs as databases and is implemented in Prolog. Prolog is a *homoiconic* language in which data and code share the same representation. Furthermore, Prolog goal resolution does not differentiate between the *extensional database*, or clauses and structures asserted as facts, and the *intensional database*, or clauses and structures derived from the application of rules. In DIMPLE$^+$, the extensional database consists of the DIMPLE$^+$ IR produced from a Java program and the built-in relations describing Java's type system and semantics. However, DIMPLE$^+$ users may easily build a custom representation in the intensional database by defining rules that derive new statements, replace old ones, and generally coalesce and discriminate from the input IR, leaving only relations tailored for a specific analysis problem.

We shall now discuss the individual DIMPLE$^+$ IR relations.

Program-domain values and expressions

The first kinds of DIMPLE$^+$ IR structures we shall examine are those that correspond to Java values and expressions.

The Java language features several kinds of values: constants and local variables of primitive (i.e. scalar) and reference (i.e. object or array) types. The DIMPLE$^+$ IR models these kinds of values with fairly straightforward structures; the simplest of these are the reference and constant structures (Table 3.1),

[2]The `baload` instruction is used for loading from arrays of `byte` values as well as from arrays of `boolean` values.

Relation name	Brief description	Page
`caughtExceptionRef/1`	A reference to a caught exception object (in a Java exception handler)	129
`doubleConstant/1`	A `double`-valued constant	129
`floatConstant/1`	A `float`-valued constant	130
`intConstant/1`	An `int`-valued constant	130
`local/3`	Associates a member of the local domain with its containing method and type	131
`longConstant/1`	A `long`-valued constant	131
`nullConstant/1`	A constant `null` reference	131
`parameterRef/2`	An actual parameter value for the current method	132
`stringConstant/1`	A reference to a constant `String`	132
`thisRef/2`	A reference to the receiver object for the current method	132

Table 3.1: DIMPLE$^+$structures representing Java values and constants

Relation	Brief description	Page
`addExpr/2`	Integer or floating-point addition	129
`divExpr/2`	Integer or floating-point division (depending on the types of operands)	129
`mulExpr/2`	Integer or floating-point multiplication	131
`negExpr/1`	Unary negation	131
`phiExpr/1`	ϕ-function (for SSA)	132
`remExpr/2`	Integer modulus (remainder)	132
`shlExpr/2`	Integer bitwise shift left	132
`shrExpr/2`	Integer bitwise shift right	132
`subExpr/2`	Integer or floating-point subtraction	132
`ushrExpr/2`	Integer unsigned shift right	133

Table 3.2: DIMPLE$^+$ structures representing Java arithmetic and value expressions

Relation	Brief description	Page
`andExpr/2`	Bitwise or logical AND	129
`cmpExpr/2`	Numeric comparison	129
`cmpgExpr/2`	Floating-point comparison (cf. `dcmpg` bytecode)	129
`cmplExpr/2`	Floating-point comparison (cf. `dcmpl` bytecode)	129
`eqExpr/2`	Value equality	130
`geExpr/2`	Greater-than or equal to	130
`gtExpr/2`	Strictly greater-than	130
`leExpr/2`	Less-than or equal to	131
`ltExpr/2`	Strictly less-than	131
`neExpr/2`	Value inequality	131
`orExpr/2`	Bitwise or logical OR	131
`xorExpr/2`	Bitwise exclusive OR	133

Table 3.3: DIMPLE$^+$ structures representing Java comparison, bitwise, and logical expressions

which describe constants of various types (e.g. `intconstant/1`), local variables (`local/3`). Table 3.1 also shows several structures that represent placeholders for values that are set up outside of the current method, such as actuals for each formal parameter (`parameterRef/2` and `thisRef/2`), and thrown exception objects in a a `catch` block (`caughtExceptionRef/1`). Of the structures in Table 3.1, all can appear on the right-hand side of an assignment, but only `local/3` can appear on the left-hand side of an assignment. (DIMPLE$^+$ also models heap stores and loads as assignments; we shall discuss heap locations shortly.)

Expressions in Java source code evaluate into simple values before they are used (either as the right-hand side of an assignment or as an actual parameter to a method call). As a consequence, the values of expressions are explicitly transmitted to their destinations. By contrast, the bytecode statements that implement Java expressions in the JVM pop operand values from the stack and push their results, thus implicitly communicating their results as an operand for some other statement. For example, the Java-language expression `2 + 4`

Relation	Brief description	Page
`castExpr/2`	Cast a variable to a compatible type	129
`instanceOfExpr/2`	Reference-type compatibility test	130
`interfaceInvokeExpr/2`	Invoke an instance method with interface-style virtual dispatch	131
`lengthExpr/1`	Array length inspection	131
`newArrayExpr/2`	Allocate and initialize a new array	131
`newExpr/2`	Allocate and initialize a new object	131
`newMultiArrayExpr/2`	Allocate and initialize a new multidimensional array	131
`specialInvokeExpr/2`	Invoke an instance method with static dispatch (e.g. constructors, `private` methods, etc)	132
`staticFieldRef/1`	A reference to a particular static field	132
`virtualInvokeExpr/2`	Invoke an instance method with standard virtual dispatch	133

Table 3.4: DIMPLE$^+$ structures representing Java object and array expressions

Relation	Brief description	Page
`arrayRef/2`	A reference to a particular element of a particular array	129
`instanceFieldRef/2`	A reference to a particular instance field of a particular object	130
`staticFieldRef/1`	A reference to a particular static field	132

Table 3.5: DIMPLE$^+$ structures representing heap locations

evaluates to 6. Were this expression to appear on the right-hand side of an assignment, 6 would be explicitly transmitted to the location named by the left-hand side of the assignment. The expression `2 + 4` compiles into the following sequence of bytecode instructions: `bipush 4; bipush 2; iadd;`. Unlike the result of the expression, which must be explicitly transmitted, the bytecode statement implicitly transmits its result via the stack, leaving its result as an operand for some future statement.

The DIMPLE$^+$ structures that model Java-language expressions do so by modeling the bytecode statements that implement these expressions. The level of abstraction in the DIMPLE$^+$ IR is closer to that of bytecode than that of Java source; however, unlike the JVM the DIMPLE$^+$ IR is not stack-based, so the values of expressions are transmitted explicitly via assignment to locals. Tables 3.2 and 3.3 show the structures that model operators over scalar values, including arithmetic, logical, and comparison operators. (`eqExpr/2` also models object identity checking.)

Tables 3.4 and 3.5 cover structures that deal with expressions that operate on references, including allocation (e.g. `newExpr/2`) and access paths from a reference to a particular field or array value (e.g. `arrayRef/2` and `instanceFieldRef/2`). It also shows the four `InvokeExpr/2` structures that model the three types of method invocation expressions in Java:

1. `static` method invocations (`staticInvokeExpr/2`),
2. virtual method invocations (`interface`— and `virtualInvokeExpr/2`), and
3. statically-dispatched instance method invocations (`specialInvokeExpr/2`).

The latter corresponds to the `invokespecial` bytecode; it represents invocations of constructors and `private` methods.

Statements and program metadata

The second kinds of DIMPLE$^+$ structures we shall examine are those modeling the contents and structure of programs. Informally, Java programs consist of classes, which may extend other classes or implement interfaces; each class contains zero or more method and field declarations. Each method contains a set of statements and an intraprocedural control-flow graph. Statements in DIMPLE$^+$ are at lower level of abstraction than Java source language statements, but are at a higher level of abstraction than bytecode statements. We have

Relation name	Brief description	Page
`assignStmt/2`	Local assignment, heap load (with heap location as right-hand side), or heap store (with heap location as left-hand side)	129
`enterMonitorStmt/1`	Entering an object's monitor	130
`exitMonitorStmt/1`	Exiting an object's monitor	130
`gotoStmt/1`	Unconditional branch	130
`identityStmt/2`	Assignment to "special" (single-assignment) locals: formals, this, and caught exception references	130
`ifStmt/2`	Conditional branch	130
`invokeStmt/1`	Invocation of a `void` method (or method with ignored return value)	131
`returnStmt/1`	Return a value	132
`returnVoidStmt/1`	Return from a `void` method	132
`stmt/2`	Associates a program counter with a particular statement	132
`throwStmt/1`	Raise an exception	132

Table 3.6: DIMPLE$^+$ structures representing Java bytecode statements

already mentioned one example of this level of abstraction: DIMPLE$^+$ expression structures (Tables 3.2, 3.3, and 3.4) correspond to independent bytecode statements but may appear as constituents of DIMPLE$^+$ assignment statement structures.

Table 3.6 describes the DIMPLE$^+$ structures that model statements. Each statement is in three-address form; so statements typically operate on locals or immediate values. Most statement types are self-explanatory, but the two kinds of assignment statements are worthy of special interest. DIMPLE$^+$ identity statement structures (`identityStmt/2`) serve two purposes: to identify locals whose values are set up outside of the current method body (i.e. those corresponding to method actuals, to `this`, and to caught exception references), and to identify their values.

DIMPLE$^+$ assignment statements (`assignStmt/2`) model standard assignments as well as heap loads and stores; these are notable because expressions

Left-hand side	Right-hand side	Description
`local/3`	`local/3`	Local-to-local assignment
`local/3`	*any heap reference, viz.:* `arrayRef/2`, `staticFieldRef/1`, or `instanceFieldRef/2`	Heap load
any heap reference	`local/3`	Heap store
`local/3`	*a constant or any arithmetic, logical, comparison, invocation, or reference operation*	Expression evaluation and assignment

Table 3.7: Statements modeled by legal DIMPLE$^+$ assignment structures

may occur as their constituents. An `assignStmt/2` has two constituent structures, corresponding to the left-hand and right-hand sides of a Java language assignment. At least one side must be a local variable, but one side may be a heap location (for heap loads and stores) or a simple expression; Table 3.7 details the possible kinds of assignment statement.

Analysis entities and domains

The final group of DIMPLE$^+$ IR structures we examine, shown in Table 3.10, represent entities in analysis domains. These are structures that uniquely identify types, classes, methods, fields, statements, and variables. DIMPLE$^+$ places every one of these entities in an explicitly numbered domain. This has several benefits:

1. It reduces the verbosity of the representation without reducing the information content. A DIMPLE$^+$ user may make any clause more verbose via Prolog's `portray/1` mechanism, which enables users to change the depiction of structures so that, for example, a method structure might be replaced with a string

Relation name	Brief description	Page
`analyzed/1`	True for a method that has been processed by the front-end; enables DIMPLE$^+$ analyses to make conservative assumptions about open programs	129
`cg_main/1`	The `main` method; the root of the call graph	129
`class/2`	Associates an entity in the class domain with a class name	129
`concreteClass/1`	True for “concrete” classes (i.e. those that can be instantiated)	129
`containsStmt/2`	Indicates that a particular method contains the statement at a particular program counter	129
`field_decl/7`	Metadata relating to a field declaration	130
`final/1`	True for methods or fields declared `final`	130
`immedExtends/2`	Indicates the immediate subclassing relation	130
`immedImplements/2`	Indicates the immediate subtyping relation	130
`interface/1`	True if a “class” is really a Java interface	131
`mainclass/1`	Identifies the class containing this program’s `main` method	131
`method_decl/7`	Metadata relating to a method declaration	131
`publicClass/1`	Indicates that a particular class is declared as `public`	132
`stmtContainedBy/2`	Inverse of `containsStmt/2`	132
`unitActual/3`	Indicates that a particular local is passed as an actual parameter at a particular invocation site	133
`unitMaySelect/2`	Indicates that a method invocation site may select a particular method body	133

Table 3.8: DIMPLE$^+$ structures representing program structure and metadata

Relation name	Brief description	Page
`branches/1`	True for branch statements	129
`fallsThrough/1`	True for statements that may transfer control to their immediate successor (contrast with `branches/1`)	130
`offset/2`	Describes a bytecode offset into a particular method	131
`param_type/3`	Indicates the type of a formal for a given method	132
`return_type/2`	Indicates the return type of a given method	132
`succ/2`	Indicates that one statement may succeed another in normal control-flow	132
`stmt_offset/2`	Indicates the bytecode offset of a particular statement	132

Table 3.9: DIMPLE$^+$ structures representing method structure and control flow

Relation name	Brief description	Page
`arraytype/2`	Describes the type of a Java-language array of a given dimension and base type	129
`class/1`	Denotes a member of the class domain	129
`field/1`	Denotes a member of the field domain	130
`local/1`	Denotes a member of the local domain	131
`method/1`	Denotes a member of the method domain	131
`primtype/1`	Denotes a primitive Java-language type	132
`reftype/1`	Describes a Java-language reference type	132
`subsig/1`	Contains a method *subsignature*; that is, its name and an encoding of its parameter types	132
`unit/1`	Denotes a member of the program counter domain, which uniquely identifies statements	133
`value_unit/2`	Used with ϕ-expressions to denote pairs of values and predecessor statements	133

Table 3.10: DIMPLE$^+$ structures representing analysis-domain entities

representation of that method's signature.

2. It greatly simplifies operating with external solvers like BANSHEE (Kodumal and Aiken 2005) and BDDBDDB (Whaley and Lam 2004; Lam et al. 2005), which require explicit domains.

3. It enables DIMPLE$^+$ users to program defensively and impose an ad hoc typing discipline on their analyses. It is trivial to check whether or not a given identifier is a valid member of any particular domain.

An extended example

We conclude our discussion of the DIMPLE$^+$ IR with an extended example of a small Java program translated into bytecode and into the DIMPLE$^+$ IR. As we shall see in Sections 3.6 and 3.7, the DIMPLE$^+$ IR is only a starting point; the DIMPLE$^+$ system makes deriving analysis-specific representations quite straightforward.

Figure 3.2 shows the Java code for a simple "cons cell" class. Each instance of `Cell` contains an `int` field called `car` and a reference to another `Cell` called `cdr`. The `car` and `cdr` fields can be inspected by instance methods of the same names, but the `Cell` class makes no provision for modifying either. The DIMPLE$^+$ representations of the basic metadata for and structure of `Cell` class are given in Figures 3.3 and 3.4; these are fairly straightforward. Note that the DIMPLE$^+$ representation includes details that are implicit in the Java source, such as the fact that `Cell` extends `Object`.

Figure 3.5 shows the bytecode translations of two methods: `cons(int, Cell)` and `length()`. These bytecode listings are in the format produced by the decompiler in the standard `javap` tool: each instruction is prefaced by its offset in the method's code array, and immediate operands that are encoded into the instruction are made explicit. In Java, immediate operands typically correspond to entries in a class' *constant pool*, which contains numeric, string, method signature, and class name constants. An example statement featuring immediate operands is at offset `0` of `cons(int, Cell)`, which creates a new instance of the Cell class. The class to instantiate is given by the entry of a class constant in the constant pool for the `Cell` class; in this case, the constant representing the `Cell` class is at position 4 in the pool.

Recall that bytecode is stack-based and no values are transmitted explicitly; rather, instructions take their operands from the stack. So, the `ireturn`

```
/** Represents a cons cell. */
public final class Cell {
    private int car;
    private Cell cdr;

    public Cell(int car) { this.car = car; }

    public int car() { return car; }

    public Cell cdr() { return cdr; }

    public static Cell cons(int car, Cell cdr) {
        Cell hd = new Cell(car);
        hd.cdr = cdr;
        return hd;
    }

    public int length() {
        return (cdr == null) ? 1 : 1 + cdr.length();
    }

    public void print() {
        System.out.println(car);
        if (cdr != null) cdr.print();
    }

    public static void main(String[] args) {
        Cell ls = new Cell(5);
        for(int i = 4; i > 0; i--) ls = Cell.cons(i, ls);
        ls.print();
    }
}
```

Figure 3.2: Java listing for a simple cons-cell class

```
% class metadata
mainclass(class(0)).

class(class(0), 'Cell').
publicClass(class(0)).
concreteClass(class(0)).

% class(4) is java.lang.Object (omitted here)
immedExtends(class(0), class(4)).
```

Figure 3.3: DIMPLE$^+$ metadata for the `Cell` class

bytecode at offset 20 in `length()` will return whatever `int` is atop the stack when it is executed. A brief examination of this method reveals that two statements may execute before the `ireturn`: the `goto` at offset 8, in which case the constant 0 is atop the stack (from offset 7); or the `iadd` at offset 19, in which case the sum of 1 (from offset 11) and the result of invoking `length()` on this cell's `cdr` (from offsets 12, 13, and 16) is atop the stack.

Intermediate representations aim to simplify low-level representations by eliminating implementation details. In the case of bytecode, details like the constant pool and the value stack may make a Java virtual machine simpler to implement, but they make bytecode programs more cumbersome to analyze. The DIMPLE$^+$ representation makes explicit many details about bytecode and about program structure and is intended to be, as much as is practical, locally coherent, meaning that individual statements should make some sense to a human reader even out of context.

The DIMPLE$^+$ representation of a method body begins by indicating that the method has been seen by the DIMPLE$^+$ front-end and giving a table of its local variables, as in the `cons(int, Cell)` method (which corresponds to `method(11316)`):

```
analyzed(method(11316)).

local(local(34473), method(11316), primtype('int')).
local(local(34474), method(11316), reftype('Cell')).
local(local(34475), method(11316), reftype('Cell')).
```

The `analyzed/1` clause indicates that the front-end has processed the method numbered 11316; this is important because it allows user analyses to

```
% field declarations
field_decl(field(5221), 'car', primtype('int'), 'Cell.car',
    class(0), private, instance).
field_decl(field(5222), 'cdr', reftype('Cell'), 'Cell.cdr',
    class(0), private, instance).

% method declarations
method_decl(method(0), '<Cell: main([Ljava/lang/String;)V>',
    class(0), subsig(2823), public, static, ['concrete']).
param_type(method(0), 0, arraytype(reftype('java.lang.String'), 1)).
return_type(method(0), type(void)).

method_decl(method(11313), '<Cell: <init>(I)V>', class(0),
    subsig(220), public, instance, ['concrete', 'ctor']).
param_type(method(11313), 0, primtype('int')).

method_decl(method(11314), '<Cell: car()I>', class(0),
    subsig(5171), public, instance, ['concrete']).
return_type(method(11314), primtype('int')).

method_decl(method(11315), '<Cell: cdr()LCell;>', class(0),
    subsig(5172), public, instance, ['concrete']).
return_type(method(11315), reftype('Cell')).

method_decl(method(11316), '<Cell: cons(ILCell;)LCell;>',
    class(0), subsig(5173), public, static, ['concrete']).
param_type(method(11316), 0, primtype('int')).
param_type(method(11316), 1, reftype('Cell')).
return_type(method(11316), reftype('Cell')).

method_decl(method(11317), '<Cell: length()I>', class(0),
    subsig(98), public, instance, ['concrete']).
return_type(method(11317), primtype('int')).

method_decl(method(11318), '<Cell: print()V>', class(0),
    subsig(5174), public, instance, ['concrete']).
return_type(method(11318), type(void)).
```

Figure 3.4: DIMPLE$^+$ structures representing field and method declarations for `Cell`

```
public static Cell cons(int, Cell);
  Code:
   0:   new    #4; //class Cell
   3:   dup
   4:   iload_0
   5:   invokespecial  #5; //Method "<init>":(I)V
   8:   astore_2
   9:   aload_2
   10:  aload_1
   11:  putfield       #3; //Field cdr:LCell;
   14:  aload_2
   15:  areturn

public int length();
  Code:
   0:   aload_0
   1:   getfield       #3; //Field cdr:LCell;
   4:   ifnonnull      11
   7:   iconst_1
   8:   goto   20
   11:  iconst_1
   12:  aload_0
   13:  getfield       #3; //Field cdr:LCell;
   16:  invokevirtual  #6; //Method length:()I
   19:  iadd
   20:  ireturn
```

Figure 3.5: Java bytecodes for `cons(int, Cell)` and `length()` methods

make conservative assumptions in the face of open programs; all methods that may be transitively referenced by a program have corresponding entities in the methods domain, but `analyzed/1` will only hold for those whose bodies have been processed by the front-end.

The `local/3` clauses essentially represent declarations of local variables: these associate entites in the locals domain with the method that contains them and their types. Locals may correspond to formals or explicit local variables in the Java source, but they may also correspond to temporaries that only exist as an artifact of compiling to bytecode. Since temporaries do not have names and Java class files do not necessarily include the names even for declared local variables, DIMPLE$^+$ does not include names for locals by default. (However, the DIMPLE$^+$ front-end features an option to associate local-domain entities with variable names, if these are available as debugging information in the input bytecodes.)

The next part of a DIMPLE$^+$ representation of a method body is a set of clauses that relate particular entities in the program counter domain (`unit/1`; the `unit` parlance is borrowed from Jimple) to bytecode offsets into a method:

```
stmt_offset(unit(883), offset(method(11316), 0)).
stmt_offset(unit(884), offset(method(11316), 5)).
stmt_offset(unit(885), offset(method(11316), 11)).
stmt_offset(unit(886), offset(method(11316), 15)).
```

This set of clauses shows that four statements in the DIMPLE$^+$ representation of `cons(int, Cell)` correspond to bytecode statements and they correspond most closely to the bytecode statements at offsets 0 (an object allocation), 5 (a constructor invocation), 11 (a heap store), and 15 (returning a reference). By making these offsets are available to analysis developers, DIMPLE$^+$ facilitates communicating analysis results back to class files as bytecode annotations.

The DIMPLE$^+$ representation of the method body is terser than the bytecode representation because it does not need to encode expressions as statements. However, the DIMPLE$^+$ representation of the method also includes statements that are implicit in bytecode; specifically, statements corresponding to the method preamble:

```
containsStmt(method(11316), unit(881)).
fallsThrough(unit(881)).
stmt(unit(881), identityStmt(local(local(34473), method(11316),
   primtype('int')), parameterRef(method(11316), 0))).

containsStmt(method(11316), unit(882)).
fallsThrough(unit(882)).
```

```
stmt(unit(882), identityStmt(local(local(34474), method(11316),
   reftype('Cell')), parameterRef(method(11316), 1))).
```

The six clauses above represent two statements in the preamble of `cons(int, Cell)`. The `containsStmt/2` clauses relate program counters to the methods that contain them. Since there is no branching in method preambles, each of these methods "falls through" and transfers control to the next statement (by convention, the next statement is the one with a program counter that is the successor of the current statement). Both of these statements, at program counters 881 and 882, are "identity statements." An identity statement is a special kind of assignment that sets up locals whose values come from outside the method scope. In this case, locals 34473 and 34474 corresponds to the first and second formals of the method.

We shall now examine the statements that correspond to the actual method body, beginning with the allocation from the bytecode at offset 0:

```
containsStmt(method(11316), unit(883)).
fallsThrough(unit(883)).
stmt(unit(883),
   assignStmt(local(local(34475), method(11316), reftype('Cell')),
            newExpr('Cell.java:13', reftype('Cell')))).
```

The preceding statement assigns the result of an object allocation to local 34475 (which corresponds to `hd` in the original Java source). The `newExpr/2` structure represents allocation and has two constituents: an atom that represents the allocation site if line number information is available (it is in this example), and a structure representing the type of the allocated object. The `new` operator in the Java language combines object allocation, initialization, and constructor invocation (Gosling et al. 2000). In contrast, DIMPLE^{+}, like Java bytecode, separates allocation and initialization from constructor invocation. The next statement in `cons(int, Cell)` models the `invokespecial` bytecode that corresponds to constructor invocation:

```
containsStmt(method(11316), unit(884)).
unitMaySelect(unit(884), method(11313)).
unitActual(unit(884), this,
   local(local(34475), method(11316), reftype('Cell'))).
unitActual(unit(884), 0,
   local(local(34473), method(11316), primtype('int'))).
fallsThrough(unit(884)).
stmt(unit(884), invokeStmt(specialInvokeExpr(method(11313),
   [local(local(34473), method(11316), primtype('int'))]))).
```

This sequence of clauses is rather more interesting than those for the simple statements we have already examined. Note the `unitMaySelect/2` relation, which encodes a conservative approximation of the program's call graph by relating the program counters of invocation statements to the identifiers of method bodies that their invocations may select. (Because constructors are dispatched statically in Java, a constructor invocation statement will select exactly one method body; invocation sites that use virtual dispatch may have more than one associated may-select relation.) The `unitActual/3` indicates actuals for a particular invocation site; in this case, local 34475 is passed as the this parameter (that is, the object being constructed), and local 34473 is passed as the first explicit actual.

Now consider the statement itself, as contained in the `stmt/2` structure. An "invocation statement" corresponds to an invocation with no return value or whose return value is ignored. Since constructors do not return values, all constructor invocations appear within `invokeStmt/1` clauses. (Invocation expressions with return values typically appear in assignment statements.) Finally, note that the invocation expression includes the explicit actual parameters in a Prolog list. Although the actuals are already represented by the `unitActual/3` relations, this redundancy increases the local coherence of assignment statement expressions.

The next statement in the method body assigns a local reference to a `Cell` to the `Cell.cdr` field of the newly-created object:

```
containsStmt(method(11316), unit(885)).
fallsThrough(unit(885)).
stmt(unit(885),
  assignStmt(
    instanceFieldRef(local(local(34475), method(11316), reftype('Cell')),
                     field(5222)),
    local(local(34474), method(11316), reftype('Cell')))).
```

The instance field reference contains a local variable (in this case, local 34475) and a field identifier. The local variable corresponds to the base object containing the field; the field identifier uniquely identifies a field by its type, name, and declaring class (the latter two correspond to ν and κ from Definition 2.3). We can identify field 5222 by consulting Figure 3.4, which shows that 5222 is ascribed to the `cdr` field declared in `Cell`. By the rules given in 3.7, we see that an assignment like this one – with a field reference on the left-hand-side and a local on the right-hand-side – corresponds to a heap store.

The final statement in `cons(int, Cell)` returns a reference to the newly-allocated cons cell object. Note that return statements do not fall through:

```
containsStmt(method(11316), unit(886)).
stmt(unit(886),
   returnStmt(local(local(34475), method(11316), reftype('Cell')))).
```

3.4 THE DIMPLE$^+$ ANALYSIS LANGUAGE

The DIMPLE$^+$ analysis language consists of two parts: a *statement processing language* and a *rule definition language*. The statement processing language describes *statement processing routines*, which convert from the DIMPLE$^+$ IR to an analysis-specific intermediate representation. Put another way, statement processing routines are subprograms that decide what derived relations should hold given the presence of certain program statements. The rule definition language facilitates descriptions of analysis rules in terms of these derived relations; analysis rules in the rule definition language are translated into code that will run on a particular query engine.

A DIMPLE$^+$ user will go through several steps when developing a new analysis:

1. Developing Prolog procedures to be used before the statement processing routine (as an additional preprocessor on the program text), or during the statement processing routine.

2. Specifying statement processing rules, indicating that certain derived relations should be added to the Prolog database if their free variables can be instantiated in some condition. DIMPLE$^+$ will then automatically generate a statement processing routine based on these rules.

3. Specifying analysis rules. These rules define analysis result relations as functions of derived relations. The DIMPLE$^+$ code generator automatically translates these into programs that run in the context of a particular query engine and generate exhaustive solutions to analysis problems.

Because the DIMPLE$^+$ analysis language is based on logic programming, it allows users to develop analyses in an *essentially declarative* style that is very similar to the formal definitions of analysis algorithms used in the literature. Unlike other tools for program analysis that support declarative abstractions, DIMPLE$^+$ also allows users to implement statement processing routines by providing declarative specifications for the translation from the DIMPLE$^+$ IR of an input program to an analysis-specific intermediate representation. In fact, the DIMPLE$^+$ statement processing language supports unrestricted Prolog in user-defined helper rules; consequently, Prolog's extralogical features, such as the cut, assignment, and assertion and retraction of relations and rules, are available to statement processing routines.

The DIMPLE$^+$ system supports generating analysis code that will run on multiple query engines with different capabilities and performance characteristics. In so doing, it allows users to trade off flexibility and expressivity for performance.

The built-in query engine is based on tabled Prolog; the tabled execution model allows for expressive analysis rules. In particular, tabled predicates may contain function symbols and — more generally — arbitrary finite structures. However, this flexibility comes at a price: if tabled search requires traversing a graph with very large strongly-connected components, it will use a great deal of memory, because subgoal solutions are stored at every node in the search space. Furthermore, the time complexity of top-down resolution with tabling depends on several implementation details and is hard to predict. As a consequence, tabled resolution may not be able to generate solutions to certain analyses queries on large inputs.

Different query engines based on different solvers can provide different performance characteristics, at the cost of some expressive power. A Datalog-based solver might exhibit much better memory performance than the tabled Prolog solver, due to bottom-up resolution; furthermore, there exist efficient methods for implementing Datalog queries, such as that presented by Liu and Stoller (2003), that also provide time and space complexity guarantees. However, a Datalog solver cannot handle the same kinds of relations and queries that a tabled Prolog system can: Datalog clauses cannot contain function symbols or lists. (Datalog also cannot treat Prolog's extralogical features, but DIMPLE$^+$ only supports these in statement processing routines, so they are not an issue for the query engine.) Liu et al. (1998) and Liu (2000) showed that it is possible to incrementally compute recursive rules with function symbols in rule heads; Liu and Stoller (2003) argue that it is possible to use this approach to transform some such programs to Datalog programs, but do not present an implementation.

The statement processing language

Once an analysis designer has decided how to map concrete program statements to elements of abstract domains, it is necessary to develop a routine to extract program statements of interest. The DIMPLE$^+$ *statement processing language* automatically generates such a statement processing routine given a set of rules describing when particular statements are interesting. These rules may simply refer to DIMPLE$^+$ IR statements: for example, asserting that a relation holds between two local variables when an assignment between them occurs in the program text. DIMPLE$^+$ statement processing rules may also refer to types and methods: for example, we may only be interested in variables of certain types, or statements that execute in particular methods. In fact, DIMPLE$^+$ affords analysis designers the opportunity to develop arbitrarily complex statement processing rules, since the statement processing routine may use unrestricted Prolog code and has access to a full representation of the program under analysis, including a class hierarchy and conservative call graph approximation.

Statement processing rules take the form `Head <-- Body`. *Head* must be a functor with at least one variable constituent; this variable must appear in *Body* as well. *Body* may contain relations about statements (e.g., `stmt/2`) as well as relations about the types of expressions within statements, relations about the class hierarchy, relations about a conservative method call graph, and user-defined relations. The default DIMPLE$^+$ statement processing routine will exhaustively identify where every *Body* relation holds, adding each unique corresponding *Head* to the database exactly once. The statement processing rules are executed in order, so relations added to the database by the head of some rule `r` are available to the body of every statement processing rule executed after `r`. The statement processing routine also keeps track of why each derived relation holds. As a result, an analysis designer can use a simple query to determine which source program statements and conditions led to a particular relation.

The case studies in sections 3.6 and 3.7 include representative examples of the statement processing language.

The rule definition language

As we have seen, DIMPLE$^+$'s statement processing routine interprets the user's specifications. As it runs, the statement processing routine generates an analysis-specific intermediate representation: it populates the database with relations describing the state of the program as is relevant to the current analysis. Statement processing rules act like filters over the program database. The statement processing routine finds all solutions to each, in order, but each rule is only solved for once.

DIMPLE$^+$ represents analysis rules in a similar way to preprocessor rules (analysis rules use the <== operator instead of <--), but the meaning of analysis rules is very different. The DIMPLE$^+$ code generator produces executable code for a particular query engine from analysis rules. Therefore, analysis rules are like procedures that may call each other. (Statement processing rules, on the other hand, may rely on the results of other Prolog procedures, DIMPLE$^+$ internal rules, or relations generated by statement processing rules that have already executed.)

Many specialized analyses and transformations depend on more general analyses. For example, analyses to determine whether or not an object is stack-allocable typically either incorporate or rely upon a points-to analysis. Other examples include constant propagation, partial evaluation, and some schemes for type inference – all of which depend on a reaching definitions analysis. Because DIMPLE$^+$ analyses and programs are stored in the same format, it is extremely straightforward to annotate programs with analysis results and serialize analysis results for use in later sessions. A user might develop an analysis, query for an exhaustive solution to the analysis rules, and save the relations derived from the analysis results to the database for use by the statement processing rules of more specialized analyses.

The case studies in sections 3.6 and 3.7 include representative examples of the rule definition language.

3.5 QUERY ENGINES

There are many excellent declarative frameworks that are well-suited to solving program analysis problems, but each presents different tradeoffs between expressivity and performance. DIMPLE$^+$ is designed for flexibility and supports interfacing with multiple, pluggable query engines. A DIMPLE$^+$ *query engine* consists of a solver, an interface to that solver, and Prolog procedures and support code to translate from rules in the rule definition language. We have implemented two query engines: one based on tabled Prolog, and one that uses the BDDBDDB system (Whaley and Lam 2004) as an external solver.

The tabled execution engine

Tabled evaluation provides several benefits. Many sorts of procedures enjoy improved execution efficiency when evaluated with tabling. Furthermore, tabled evaluation admits natural, declarative definitions of left-recursive procedures. However, these benefits come at two costs: a memory cost and a development cost.

In many cases, it is possible to address the memory cost. Tables for procedures may take up a great deal of memory, and the balance between space and execution time may not justify tabling certain predicates. As a result, DIMPLE$^+$ allows users to designate certain analysis rules as untabled; the code generator will simply generate standard Prolog rules for these.

The development cost presents a rather trickier problem. Typically, it is not possible to trace a tabled procedure, since it will be evaluated at most once for a given tuple of arguments. For many applications, the tradeoff between ease of debugging and execution time would be acceptable. However, we are interested in enabling researchers to prototype new analyses rapidly. As part of this process, a researcher may be interested in determining precisely *why* a particular spurious analysis result holds.

The code generator for the tabled query engine solves this problem by generating two versions of rules: *standard versions*, which are tabled and execute normally, and *tracing versions*, which are not tabled and can execute in a metainterpreter that produces a rule trace. Given a standard rule R, we generate the tracing version R$'$ as follows:

1. Give R$'$ a new name that does not belong to any extant procedures; create a relation in the database indicating that the name for R$'$ is the tracing version of the rule R.

2. Make a copy of R's body; this will become the body for R$'$. Map every call to a non-recursive procedure in this new body with a call to the tracing version of that procedure. Note that we do not generate tracing versions of relations generated by the statement processing routine. The code generator only replaces calls to non-recursive procedures in order to ensure that the metainterpreter will terminate when asked to explain any terminating relation.

Given these tracing versions, a user can drill-down to fully explain any individual analysis result. As an example, consider a user who wishes to explain why the relation `v_pt` holds with arguments `12` and `45`. When the user asks

DIMPLE$^+$ to explain why `v_pt(12,45)` holds, the metainterpreter will consult the tracing versions of the various clauses for `v_pt`. Assume that, for this example, the relevant clause of `v_pt` is recursive and depends on an `assign/2` relation generated by the statement processing routine: namely,

```
v_pt(Ref,Obj) <== assign(Ref,Int), v_pt(Int,Obj).
```

The generated tracing version of this clause would be very similar, except that it would not be declared as a tabled procedure. If there were calls to non-recursive analysis rules inside the body of this clause, then they would be replaced with calls to tracing versions. However, the call to `assign/2` would not change, since `assign/2` is a fact generated by the statement processing routine; the recursive call to `v_pt/2` would not change, since the code generator does not replace calls to recursive rules. Therefore, the tracing version of this clause, in standard Prolog syntax, would consist of:

```
dimpleTRACE_v_pt(Ref,Obj) :- assign(Ref,Int), v_pt(Int,Obj).
```

In this example, the metainterpreter might show that `assign(Ref,Int)` holds with `Ref = 12` and `Int = 14`; it will then consult the table for `v_pt/2` and see that `v_pt(Int,Obj)` holds with `Int = 14` and `Obj = 45`. Therefore, the user is able to see which rule led to the (perhaps spurious) result; if one wishes to trace further, e.g., to determine why `v_pt(14,45)` holds, one may iteratively query the metainterpreter for more details.

A BDD-based query engine

In tabled execution, partial solutions are stored for every subgoal, which can dramatically increase memory consumption when finding a solution to a large, interdependent system of rules. Disabling tabling on these predicates is sometimes an unattractive option, since doing so changes the semantics of solution search and may introduce non-termination. Therefore, some analysis problems — especially those that involve exploring very large search spaces or those with significant redundancies — may benefit from a more compact representation. We will begin with some background on such a representation, called ordered binary decision diagrams, before discussing a DIMPLE$^+$ query engine that exploits this representation by using an external solver.

Ordered binary decision diagrams,[3] due to Bryant (1992), represent boolean

[3] All of the binary decision diagrams we will discuss in this section are reduced ordered binary decision diagrams; we will refer to them as BDDs for simplicity.

x_0	x_1	x_2	Result
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Figure 3.6: Truth table for the ternary majority function

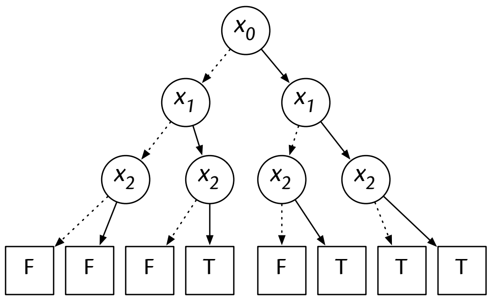


Figure 3.7: Decision tree for the ternary majority function

functions compactly as directed acyclic graphs. Figure 3.6 shows the truth table for the ternary majority function, which has the value shared by at least two of its arguments. Figure 3.7 shows a decision-tree representation of the ternary majority function, in which variables are nodes, each of which has a "false" child (denoted with a dashed line) and a "true" child (with a solid line). It should be clear that either of these explicit representations requires on the order of 2^n space, where n is the number of variables in the function.

The BDD representation exploits redundancies in the decision tree and generates a more compact graph representation in which the following are eliminated:

1. redundant leaf nodes, for example, every "true" leaf is replaced by one canonical "true" leaf, and likewise with "false;"

2. redundant roots, that is, those who correspond to the same variable and whose

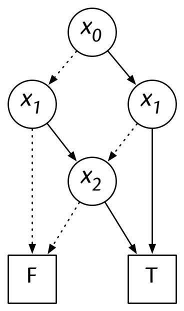


Figure 3.8: Reduced ordered binary decision diagram for the ternary majority function

inputs and outputs are the same; and

3. redundant tests, that is, those whose truth value does not alter the result of the function.

For functions of few variables, it suffices to repeatedly remove these redundancies until quiescence; this will result in a smaller representation, such as that in Figure 3.8, which represents the ternary majority function much more compactly than the decision tree or truth table. For larger functions, such an iterative approach is intractable and must be replaced with symbolic manipulation.

We should note that the variable ordering chosen for a particular BDD can dramatically impact the size of the fully reduced diagram; informally, the ordering can expose or hide redundancies in the function's result space. Bollig and Wegener (1996) showed that algorithmically choosing a "good or optimal" variable ordering for a BDD is NP-complete; since so doing can have a profound impact on the efficiency of Boolean manipulations, it is a nontrivial concern for BDD users. Furthermore, since BDDs are restricted to representing Boolean functions, a suitable BDD-based encoding of an analysis problem may not be immediately apparent. Finally, BDDs may not enable efficient symbolic manipulation of certain Boolean functions; as an example, Bryant (1992) shows that operations on a BDD representing integer multiplication are exponential in the best *and* worst cases. Nevertheless, the BDD representation has proven greatly beneficial to many problem domains, including program analysis.[4]

[4]Representative program analysis applications of BDDs include those of Marriott and Søndergaard (1993); Corbett (1994); Lagoon and Stuckey (2002); Berndl et al. (2003); Lhoták

The BDDBDDB system (Whaley and Lam 2004) implements Datalog using BDDs. While Datalog is strictly less expressive than tabled Prolog, it is expressive enough for a wide range of analysis problems, as many groups have demonstrated. (We cover representative applications in Section 3.8.) We have developed a DIMPLE$^+$ query engine that employs BDDBDDB as an external solver; in cases where the DIMPLE$^+$ rules for an analysis can be represented as a legal Datalog program, it translates the analysis and generates a set of rules for BDDBDDB to solve.

BDDBDDB represents a relation with at most 2^k elements as a k-ary Boolean function. As a consequence, one must place all entities from input relations (i.e. the analysis-specific IR) into fixed-size domains; one must also accurately estimate the upper bound on the size of the output relations. (If the size estimated for output domains is too small, the queries will fail; if it is too large, it may not be possible to issue complex queries.) BDDBDDB supports machine learning-based heuristics for selecting variable orderings, which relieves the user from the burden of choosing a good ordering, but may inhibit casual experimentation with new analysis rules. BDDBDDB does, however, feature support for debugging Datalog programs and inspecting analysis results.

3.6 CASE STUDY: ANDERSEN'S ANALYSIS

Consider a family of fundamental program analyses: *points-to* analyses, which provide answers to the question: "Which (abstract) memory locations *might* this reference-valued variable refer to at runtime?" Andersen's analysis (1994) is a points-to analysis; it provides a reasonable tradeoff between precision and worst-case execution time for many applications. Andersen's analysis also enjoys an intuitive, succinct specification in terms of constraints on a points-to graph: one could easily describe it to an implementer by writing the most important analysis rules on a cocktail napkin. However, producing a good imperative program that implements Andersen's analysis is a rather difficult task – early implementations were quite slow and did not scale. The first truly scalable implementation of an Andersen-style analysis, due to Heintze and Tardieu (2001), was reported *seven years* after Andersen's dissertation was published.

In this section, we present a DIMPLE$^+$ implementation of Andersen's points-

and Hendren (2004); Whaley and Lam (2004); Lam et al. (2005); Naik et al. (2006); Naik and Aiken (2007).

to analysis for Java.[5] We have adapted the analysis rules from the BDDBDDB implementation of a context-insensitive subset-based points-to analysis, due to Whaley and Lam (2004). We developed the statement processing rules in order to translate from DIMPLE$^+$ IR statements to the analysis-specificIR used by Whaley and Lam's specification.

Andersen's analysis is a *flow-* and *context-insensitive*, *inclusion-based* points-to analysis. This means that subprograms are treated as sets of statements (rather than as graphs of statements); that analysis results for a particular method are merged among all call sites of that method; and that a variable may refer to a subset of the locations that another variable refers to.[6]

We defined statement processing rules to extract relevant relations from the IR database. (Only a subset of all the DIMPLE$^+$ IR relations affect points-to information.) Figures 3.9 and 3.10 shows how we encoded these in preprocessing the program text:

1. Object allocation statements create a new abstract object, which we name by the program counter of the allocation site. Object allocation statements imply an *immediate points-to* (`pt/2`) relation between the local variable receiving the object reference and the newly-allocated object.

2. Assignment statements (including heap and array loads and stores) indicate that the variable or location on the left-hand side of the assignment may refer to a superset of the locations that the right-hand side of the assignment may refer to. Standard assignments between locals imply an `assign/2` relation; loads and stores of instance fields imply `load/3` and `store/3` relations. (We treat array accesses – not shown in the figure – as field accesses to a distinguished field name and static fields as global variables.)

3. Method invocations indicate that the local variables corresponding to formal parameters (indicated by `formal/3` relations) must refer to a superset of all abstract objects referred to by variables passed as actual parameters (indicated by `actual/3` relations). (Recall that we use a precomputed conservative approx-

[5] Andersen's analysis was initially designed for C; several groups (Berndl et al. 2003; Sridharan et al. 2005; Whaley and Lam 2004; Rountev et al. 2001) have refined it to take advantage of Java's type system and lack of unrestricted pointers; these various extensions are fundamentally similar.

[6] Readers who are interested in more information on categories of points-to analyses should refer to Hind and Pioli (2000) and Hind (2001) for a thorough overview of the field.

```
/* newly-allocated objects */
pt(La,Id) <--
   stmt(unit(Id), assignStmt(local(local(La),Ma,Ta)), newExpr(Loc,Type)).

/* assignments between locals */
assign(La,Lb) <--
   stmt(_, assignStmt(local(local(La),Ma,Ta), local(local(Lb),Mb,Tb))),
   reference_type(Ta).

/* static fields (i.e., globals) */
s_load(La,Field) <--
   stmt(unit(Id), assignStmt(local(local(La),Ma,Ta),
   staticFieldRef(Field))), reference_type(Ta).
s_store(Field, La) <--
   stmt(unit(Id), assignStmt(staticFieldRef(Field),
   local(local(La),Ma,Ta))), reference_type(Ta).

/* instance fields */

% La = Lb.F
load(La,F,Lb) <--
   stmt(unit(Id), assignStmt(local(local(La),Ma,Ta),
                instanceFieldRef(local(local(Lb),Mb,Tb), F))),
   reference_type(Ta).

% La.F = Lb
store(La,F,Lb) <--
   stmt(unit(Id), assignStmt(instanceFieldRef(local(local(La),Ma,Ta), F),
                local(local(Lb),Mb,Tb))),
   reference_type(Tb).

/* phi functions (SSA only) */

has_local(LHS, [value_unit(local(local(LHS),_,_),_)|Vunits]).
has_local(LHS, [_|Vunits]) :- has_local(LHS, Vunits).

info(X) <-- X = phi_assignments.

assign(La, Lb) <--
   stmt(_, assignStmt(local(local(La),_,_),
   phiExpr([H|T]))), has_local(Lb, [H|T]).
```

Figure 3.9: Andersen's analysis: select statement processing rules (rules treating arrays and exceptions are omitted)

```
/* formal and actual parameters */
formal(La, Index, Method) <--
    stmt(unit(Id), identityStmt(local(local(La),Ma,Ta),
    parameterRef(Method, Index))),
    reference_type(Ta).

formal(La, this, Method) <--
    stmt(unit(Id), identityStmt(local(local(La),Ma,Ta),
    thisRef(Method, Type))),
    reference_type(Ta).

actual(La, Index, Method) <--
    unitActual(Callsite, Index, local(local(La), Ma, Ta)),
    unitMaySelect(Callsite, Method), reference_type(Ta).

/* return values */
ret_caller(La, Method) <--
    stmt(unit(Id), assignStmt(local(local(La),Ma,Ta), X)), invocation(X,_),
    unitMaySelect(Id, Method), reference_type(Ta).

ret_callee(La, Method) <-- stmt(unit(Id), returnStmt(local(local(La),Ma,Ta))),
    reference_type(Ta), containsStmt(method(Method), unit(Id)).
```

Figure 3.10: Andersen's analysis: statement processing rules used by interprocedural transfer functions

imation of the dynamic call graph; the `unitMaySelect/2` relation indicates that a particular program counter may invoke a particular method.)

Let us now consider the actual analysis rules. Andersen's analysis is perhaps simpler to understand if we consider it as a graph problem. An exhaustive solution to the points-to question, as given by the set of all tuples under the `v_pt/2` relation, is simply the transitive closure of the assignment relation from abstract heap objects *Id* to reference variables *Ref*. With this in mind, we can consider the rules, as shown in Figure 3.11:

1. `v_pt(Ref,Id)` holds when *Ref* immediately points to *Id*.

2. `v_pt(Ref,Id)` holds when *Ref* has received a reference from some other variable *RefI*, and `v_pt(RefI,Id)` holds.

```
% case 1:
v_pt(Ref,Id) <== pt(Ref,Id).

% case 2:
v_pt(Ref,Id) <== assign(Ref, RefI), v_pt(RefI, Id).

% case 3:
v_pt(Ref, Id) <== s_load(Ref, F), s_store(F, RefI), v_pt(RefI, Id).

% case 4:
v_pt(Ref, Id) <== formal(Ref, I, M), actual(RefI, I, M), v_pt(RefI, Id).

% case 5:
h_pt(Obj1, F, Obj2) <==
    store(Ref1, F, Ref2), v_pt(Ref1, Obj1), v_pt(Ref2, Obj2).

% case 6:
v_pt(Ref, Id) <==
    load(Ref, F, Ref1), v_pt(Ref1, Id1), h_pt(Id1, F, Id).

% case 7:
v_pt(Ref, Id) <==
    ret_caller(Ref, Method), ret_callee(RefI, Method), v_pt(RefI, Id).
```

Figure 3.11: Andersen's analysis: complete analysis rules

3. `v_pt(Ref,Id)` holds when *Ref* may have been loaded from a static field *F*, *F* may have had some reference *RefI* stored to it, and `v_pt(RefI,Id)` holds.

4. `v_pt(Ref,Id)` holds when *Ref* is formal parameter *I* of some method *M*, *RefI* is actual parameter *I* of a call site that may invoke *M*, and `v_pt(RefI,Id)` holds.

5. `h_pt(Obj1,F,Obj2)` holds when some field *F* of an abstract heap object *Obj1* may refer to the abstract heap object *Obj2*.

6. `v_pt(Ref,Id)` holds when *Ref* may have been loaded from a field that has had *Id* stored to it.

7. `v_pt(Ref,Id)` holds when *Ref* may receive its value from a method that returns some *RefI* such that `v_pt(RefI,Id)` holds.

Benchmark	Memory	Run	CPU
antlr	564M	7.89	7.90
bloat	—	—	—
eclipse	1215M	18.55	21.89
hsqldb	2370M	37.37	65.41
jython	2584M	42.22	66.99
luindex	468M	7.46	7.47
lusearch	532M	8.05	8.06
pmd	790MB	11.27	11.27

Table 3.11: Performance results for Andersen's analysis with the tabled query engine. (`bloat` did not complete.)

Table 3.11 shows the performance results for executing Andersen's analysis on our subset of the DaCapo benchmarks, using DIMPLE$^+$ and the tabled execution engine. "Memory" indicates the total memory requirements for finding an exhaustive solution to the analysis problem. We report two timings: "run time" indicates query processing time exclusive of time spent managing memory (e.g. garbage collections and handling stack, heap, or trail space overflows), while "CPU time" indicates total execution time including memory management time. All timings include the time to load, parse, and index the analysis-specific intermediate representation.

3.7 CASE STUDY: EFFECTS INFERENCE

In this section, we present a simple *effects inference* analysis to aid program understanding. In so doing, we show an example of using DIMPLE$^+$ to extend an existing analysis. We also demonstrate how the interactive nature of DIMPLE$^+$ makes it easy to validate intuitions and apply these to improving an analysis. Before we introduce our particular analysis, we shall discuss the problem in more detail.

Standard type systems characterize the ranges of values that expressions may produce and that variables may assume. Effect systems (Lucassen and Gifford 1988) extend type systems to also characterize the computational effects (e.g., reads or writes to shared state) of expressions, statements, and methods. Thus, a type system might tell us that method `foo` takes an `int` parameter

```
public class Point {
   private float x, y;

   public Point(float x, float y) {
      this.x = x; this.y = y;
   }

   public void setX(float x) { this.x = x; }
   public void setY(float y) { this.y = y; }

   public float getX() { return this.x; }
   public float getY() { return this.y; }

   public void translate(float dx, float dy) {
      this.x = this.x + dx;
      this.y = this.y + dy;
   }
}
```

Figure 3.12: A simple `Point` class

and returns a reference to `Object`, but an effect system would also tell us that method `foo` may write values in regions X and Y and read values from regions X and Z. (A region is simply a subset of the heap.) Just as a type system may use explicit type annotations (as in Java or C) or infer types for variables and expressions (as in ML or Haskell), effect systems may either require user annotations of effecting behavior or infer this information.

There are many applications of effects inference analysis. Several notable examples include finding expression scheduling constraints (as in Lucassen and Gifford); automatically providing annotations for a model checker or specification language (Salcianu and Rinard); and, most commonly, improving *region-based memory management* (Tofte and Talpin 1997), in which object lifetimes are inferred at compile-time to enable a stack discipline for dynamic allocations, so that an entire region of objects may be deallocated at once. We present a more sophisticated effects inference analysis, designed to identify externally- pure and read-only methods and quiescing fields, in Chapter 4. The analysis we present in this section is designed to aid program understanding and debugging, so it does not calculate region lifetimes; rather, it answers the question: *given some method X, which abstract locations may X read or write?* We shall use the `Point` class declared in Figure 3.12 as an example as we introduce our analysis. Before we do so, we shall review some relevant Java features.

Side effects in Java

Java enforces a strict divide between heap data and stack data. In particular, Java supports references instead of unrestricted pointers. References may only refer to heap locations (objects or arrays); it is not possible to create a reference to a stack value.

Parameters are passed by value in Java. Therefore, the only data that can be shared between methods – the sort of data that we are interested in inferring effects on – are heap objects. The only way to access or modify heap data (thus, shared state) is via an array element reference or an object field reference.

Because Java is a typed language, it is not possible to refer to an object of type C via a reference of type D, where D is not a supertype of C. Doing so will result in either a compile-time or run-time error. At the IR (or bytecode) level, all field accesses are to fully-qualified names, including a field name and the name of its declaring class. Therefore, we can examine the IR for the `Point.setX` method and determine that it may only modify one abstract location: the `Point.x` field of some object that is an instance of `Point` or some subclass thereof. In the language of effect systems, we could say that `Point.setX` writes into the region `Point.x`.

We could devise a very basic effects inference for Java by preprocessing a program database to reject everything except heap reads and writes, method invocations, and the conservative call graph. We would write our analysis to indicate that a heap read had a READ effect on the field it accessed, that a heap write had a WRITE effect on the field that it accessed, and that a method invocation statement had the union of all effecting statements from every method that might be dispatched from that call site. Such a basic analysis would use Java's type system to provide extremely coarse memory disambiguation. Its results would be sound but perhaps not very useful – there would be no way to distinguish between writes to some field C.F through references X and Y, even if it were possible to statically guarantee that X and Y did not refer to the same object.

A simple effects inference analysis

We can develop a more precise analysis by building on the points-to analysis from Section 3.6. Java's type system provides an extremely coarse form of memory disambiguation; we can use a points-to analysis to discriminate among locations with finer granularity. We shall consider abstract objects to be regions.

```
public void foo() {
       Point pt = new Point(0,0);
       pt.translate(1,1);
}
```

Figure 3.13: `foo` exposes a shortcoming of the simple analysis

Therefore, an inferred effect will consist of READ or WRITE, an abstract object (or the distinguished location `global`), and a field name. (We discuss but do not show the DIMPLE$^+$ rules for the analysis we present in this section; many are substantially similar to the rules for the improved analysis that we present later in this section.)

This simple effects inference analysis proceeds as follows: First, it preprocesses the input database, extracting statements of interest. For the purposes of this analysis, we are exclusively interested in method invocations and reads and writes to heap locations. The analysis rules define the `reads/3` and `writes/3` relations to describe side effects, as follows:

reads(PC, Loc, F) holds when: (1) the statement at program counter *PC* reads from the instance field F from a reference R and `v_pt(R,Loc)` holds;[7] or (2) *Loc* is the atom `global`, and the statement at program counter *PC* reads the static field *F*; or (3) the statement at program counter *PC* may invoke a method that contains some statement *PC′* for which `reads(PC', Loc, F)` holds.

writes(PC, Loc, F) holds in analogous situations as does `reads/3`, except that `writes/3` holds when fields are written to instead of read from.

Andersen's analysis is a better discriminator between memory locations than is Java's type system, but it is limited by its insensitivity to contexts. Recall that Andersen's analysis merges analysis results for every context in which a method is invoked. (Here we construe "context" broadly to include call stack strings and receiver objects.) Specifically, consider the `foo` method from Figure 3.13.

A novice Java programmer would correctly note that the scope of `foo`'s side effects are confined to the object constructed in its first line. However,

[7]The rules treating array references are similar to those treating instance fields; thus, we have omitted them here as we did in Section 3.6.

because the simple effects analysis is using Andersen's analysis to disambiguate between memory locations, it will not fare as well. Since `foo` only consists of two method invocations (a constructor and `translate`), its READ and WRITE sets are the unions of those sets for all of the methods it invokes. Because of context-insensitivity, these sets may be very large. The constructor has WRITE effects for the `x` and `y` fields of every object that may be referred to by `this` – that is, every `Point` object that has been created with that constructor! The `translate` method likewise has READ and WRITE effects for the `x` and `y` fields of every object that is pointed to by a reference that has had the `translate` method invoked on it.

Parameterizing on receiver objects

It seems plausible that the READ or WRITE sets of most instance methods will contain primarily fields of the receiver object (i.e., `this`). Were this the case, we could parameterize our analysis so that when we encountered an effect involving `this`, instead of keeping track of all possible objects that might be referenced by `this`, the READ and WRITE sets would merely record an effect to `this`. Given this sort of parameterized analysis, for example, the WRITE sets for `Point.setX` would include only the `x` field of `this`. When our analysis encountered an invocation like `pt.setX(y)`, it could then calculate which objects `pt` might refer to, and generate WRITE sets for the method invocation by instantiating `this` in the WRITE set for `setX` with each such object.

Of course, a plausible intuition doesn't provide sufficient justification for building and validating a new analysis, no matter how straightforward it is to do so in a given framework. Fortunately, DIMPLE$^+$ makes it easy to collect empirical evidence for our intuitions: we may simply query the program database to see if certain conditions obtain.

In order to do so, we define an `isthis/1` predicate that holds when a variable is a `this` reference – that is, when it refers exclusively to the receiver object of the current method. (The DIMPLE$^+$ IR generator attempts to minimize the lifetimes of locals. Therefore, each method has one local variable that corresponds to `this` and possibly several variables that alias that local.) Then, we can use DIMPLE$^+$ to query the program text and determine whether our intuition is correct; viz., most instance field accesses are to `this` instead of to some other object.

For the benchmark programs we examined, this particular intuition happens to be correct. As a result, we know that it is probably worthwhile to

```
isthis(L) <-- formal(L,this,M).
isthis(L2) <--
    formal(L,this,M),
    assign(L2, L),
    all(Source,assign(L2, Source),[L]).

receiver(Id, L) <--
    unitActual(Id, this, local(local(L), _, _)).

/* globals */
read_global(Id, F) <--
    stmt(unit(Id), assignStmt(local(_,_,_), staticFieldRef(F))).

/* instance fields */
read_effect(Id, this, F) <--
    stmt(unit(Id), assignStmt(local(_,_,_),
    instanceFieldRef(local(local(L),_,_), F))),
    isthis(L).

read_effect(Id, L, F) <--
    stmt(unit(Id), assignStmt(local(_,_,_),
    instanceFieldRef(local(local(L),_,_), F))),
    \+ isthis(L).

/* invocations */
callgraph_edge(Id, Callee) <-- unitMaySelect(unit(Id), method(Callee)).

/* methods containing effecting statements */
in_method(Id, Method) <-- containsStmt(method(Method), unit(Id)),
    (read_effect(Id, Lr, Fr) ; write_effect(Id, Lw, Fw)).
in_method(Id, Method) <-- containsStmt(method(Method), unit(Id)),
    (read_global(Id, Fr) ; write_global(Id, Fw)).

method_contains(Method, Id) <--
        in_method(Id, Method).
```

Figure 3.14: Select statement processing rules for parameterized effects inference (WRITE effect rules are omitted)

```
reads(Id, global, F) <==
  read_global(Id, F).

reads(Id, O, F) <==
  read_effect(Id, L, F), v_pt(L, O).

reads(Id, this, F) <==
  read_effect(Id, this, F).

method_reads(Method, Loc, F) <==
  method_contains(Method, Id), reads(Id, Loc, F).

reads(Id, Loc, F) <==
   callgraph_edge(Id, Callee),
   method_reads(Callee, this, F),
      receiver(Id, V),
      v_pt(V, Loc).

reads(Id, Loc, F) <==
   callgraph_edge(Id, Callee),
   method_reads(Callee, Loc, F),
      Loc \= this.
```

Figure 3.15: Select analysis rules for parameterized effects inference (rules treating WRITE effects are omitted)

develop a new analysis that generates parameterized summaries of effects information for methods; our analysis can then instantiate these summaries at call sites to indicate that effects to `this` may only impact objects referenced by the receiver at the call site. Figure 3.14 contains the statement processing rules for this improved analysis, and Figure 3.15 contains the analysis rules. (We omit rules related to WRITE effects in the analysis rules presented here, as they are very similar to the rules for READ effects.)

3.8 RELATED WORK

Work related to the research we have reported here falls into two categories: declarative representations of programs and program analysis specifications, and techniques for efficiently solving declarative analysis queries. We discuss notable results in each of these areas that are most relevant to our work. We conclude by placing DIMPLE$^+$ in context in the field and recapitulating its contributions.

Declarative and relational frameworks for analysis

Dawson et al. (1996) argued that a general-purpose logic programming system (the XSB Prolog system) could be used to evaluate declarative formulations of program analysis problems. Their work demonstrated that the evaluation model of tabled Prolog is suitable for answering program analysis queries efficiently and completely. However, their evaluation was restricted to analyses of functional and logic programs consisting of tens to hundreds of lines of code and therefore did not evaluate the scalability of their techniques.

Codish et al. (1998) developed a system that used the tabled execution capability of XSB for developing abstract interpretation-based analyses of logic programs. Abstract interpreters can mirror the structure of concrete interpreters; therefore, their system enables extracting program analyses from fairly straightforward Prolog metainterpreters. Due to their use of tabling, their system is speed-competitive with specialized toolkits for analyzing logic programs.

Heintze (1992) demonstrated that many program properties could be faithfully approximated by sets – and, thus, that many program analyses could be formulated as a system of set constraints. Heintze and Jaffar (1994) present an overview of work in this area. Later in this section, we will describe some notable tools based on this abstraction.

Reps (1998) demonstrated that many interprocedural dataflow analyses could be formulated as reachability problems on context-free languages. Since there is a well-understood correspondence between context-free languages and declarative programs that recognize them, this approach implies declarative formulations of a large class of analysis problems. Reps et al. (2005) later generalized this approach to include analysis problems that could be specified as reachability problems on weighted pushdown systems.

Several researchers have investigated the class of problems that are expressible as CFL-reachability problems. Notably, Melski and Reps (1997) gave a general algorithm for translating from any CFL-reachability problem to a set-constraint satisfaction problem. Later, Kodumal and Aiken (2004) provided an algorithm for converting from *Dyck CFLs*, a subset of all context-free languages, to systems of set constraints; their algorithm is less general than the Melski-Reps reduction, but produces more efficient implementations in the special case of Dyck CFLs. Many static analysis problems can be expressed as Dyck CFLs; a representative example is given by Sridharan et al. (2005), who formulated demand-driven points-to analysis for Java with a Dyck CFL.

One disadvantage of many declarative frameworks for analysis — including DIMPLE$^+$ — is that they are designed for analyzing whole programs and are less well suited, at least out of the box, for modularized analyses. Besson et al. (2003) apply Datalog to *class analysis* or reaching-types analysis; that is, answering the question "what types of value may this expression evaluate to?" This analysis can be used for coarse memory disambiguation but also enables many profitable optimizations, such as virtual method resolution at monomorphic call sites (Sundaresan et al. 2000). The major innovation in Besson et al.'s work is that it supports developing modular analyses as open Datalog programs.

Hanbing Liu and J Strother Moore encoded the semantics for the Java Virtual Machine in the ACL2 theorem prover by developing an interpreter for the JVM in pure Lisp. As a consequence of their work, it is possible to use ACL2 to reason about Java programs. In a similar vein, Leroy (2006) developed a certified compiler for a C-like language in the Coq proof assistant. There is, of course, a strong and obvious analogy between using a theorem prover or proof assistant to reason about programs and using a logic programming system to reason about programs.

Solving declarative analysis queries

Prior to his work on interprocedural dataflow analysis as graph reachability, Reps (1994) showed how to automatically derive demand-driven versions of dataflow analysis algorithms – that is, analysis procedures that calculate exact results for a particular subset of the program or for a particular program point. His technique relied on the magic-sets transformation: applying this transformation to a logic program implementing an exhaustive analysis algorithm results in a demand-driven version of the algorithm.

Saha and Ramakrishnan (2005) adapted techniques for incremental and goal-driven evaluation of tabled Prolog in order to formulate incremental and demand-driven versions of program analyses. They evaluated their work on a version of Andersen's analysis for C, treating programs consisting of tens of thousands of preprocessed statements. As with the subset-based points-to analysis we present in Section 3.6, their analysis is flow-, field- and context-insensitive. (It is difficult to derive a sound field-sensitive analysis for C, since C admits many unsafe features. In contrast, the points-to analysis we presented is flow- and context-insensitive, but field-*sensitive*.) Their work focuses on applying techniques for improving the performance of logic programs to improving the utility of logic programming as a tool for program analysis.

Several special-purpose systems have been developed to allow declarative specifications of program analyses. The BANE toolkit (Fähndrich and Aiken 1997) and its successor BANSHEE (Kodumal and Aiken 2005) generate specialized solvers for program analyses specified as constraint-satisfaction problems. An analysis designer would use these tools by developing a preprocessor to extract relevant constraints from the program text and then declare the analysis itself in terms of constraints on terms or sets. BANSHEE enjoys a rich type system and static checking of type-safety for analyses. BANSHEE also achieves high performance – a BANSHEE implementation of Andersen's analysis for C analyzed hundreds of thousands of lines of preprocessed C code in seconds and millions of lines of preprocessed C code in under a minute. Like the work of Saha and Ramakrishnan, BANSHEE also provides support for incremental evaluation of analysis queries.

BDD-based solvers and approaches provide an attractive solution for managing scale in many kinds of program analyses. However, the performance (and tractability) of BDD-based analysis approaches depends on the size of the BDD graph, which in turn depends on the *variable ordering* that the BDD user has chosen. As we have discussed, choosing a good variable ordering for a BDD is NP-complete; a bad variable ordering can, for some functions, mean the difference between near-linear time and exponential time (Bryant 1992; see).

Lhoták and Hendren (2004) developed Jedd, which extends the syntax and semantics of Java with support for programming with relations. Jedd uses BDDs and propositional satisfiability to translate from a high-level relational programming model to low-level BDD operations. It requires users to develop a BDD variable ordering and provides support for profiling BDD size and performance given a particular ordering.

The BDDBDDB system (Whaley and Lam 2004; Lam et al. 2005) is a specialized implementation of Datalog based on *binary decision diagrams* (Bryant 1992); we reviewed the particulars of the BDD representation earlier in this chapter. Whaley and Lam (2004) demonstrate the performance of two BDDBDDB implementations of Andersen's analysis; their context-insensitive analysis is roughly comparable to other known techniques. Where the BDD-based representation excels, however, is in handling context-sensitive analyses – many of which would produce results too large to represent explicitly in a Prolog database. BDDBDDB requires the user to specify a variable ordering, but it features a machine learning-based process for automatically identifying a good ordering candidate from among several heuristically determined possibilities.

DIMPLE$^+$ *in context*

The work we discussed under the heading *Declarative and relational frameworks for analysis* primarily treats declarative formalisms for programs and program analysis problems. Obviously, the choice of a formalism for expressing analyses is somewhat subjective. However, in our opinion, the relational model of logic programming approaches (including DIMPLE$^+$) is rather easier to use than constraint- or set-based models.

In contrast, the work we mentioned relating to *Solving declarative analysis queries* treats implementation techniques for efficient tools to handle evaluating analysis problems expressed in terms of various formalisms. DIMPLE$^+$ already benefits from advances in logic programming system implementation; the efficient execution of a tabled Prolog system makes whole-program analyses feasible. However, users who wish to exploit capabilities not directly available in the underlying Prolog system could easily use DIMPLE$^+$ as a front-end for a specialized solver, in a manner similar to the way we have implemented a BDDBDDB-based query engine for the DIMPLE$^+$ backend. Such an approach would use the DIMPLE$^+$ IR, statement processing language, and analysis language, but would override the DIMPLE$^+$ code generator by writing a Prolog procedure that translates from user analysis rules to the format or formalism expected by an external tool.

Perhaps the best characterization of our work is that DIMPLE$^+$ provides an interface, framework, and language to facilitate using a logic programming system for program analysis. The related work under discussion either presents formalisms for program analysis or advances the state of the art of logic programming, perhaps with immediate application for program analysis problems. Since we have developed an application that exploits many of the features of an advanced logic programming system, contributions that improve logic programming systems are complementary to ours.

The DIMPLE$^+$ system enables analysis designers to rapidly prototype new program analyses in an interactive fashion. Unlike every other system under discussion, DIMPLE$^+$ enables analysis designers to use logic programming for every phase of the analysis development process. DIMPLE$^+$ is also unique in that the entire program text is available to the analysis designer for prototyping.

However, DIMPLE$^+$ enables users to discard irrelevant relations for efficient execution of production analyses.

While the other analysis frameworks we have mentioned represent excellent research contributions, no other system under discussion is as suitable for prototyping and interactive, exploratory development as DIMPLE$^+$. The DIMPLE$^+$ analysis developer need never leave the declarative world of Prolog and the DIMPLE$^+$ IR, whether preprocessing input programs, defining statement processing routines, or declaring and evaluating analysis rules. In contrast, other systems like BANSHEE and BDDBDDB require the user to develop a specialized preprocessor for source text that extracts relations of interest, to declare a set of rules or constraints, and then to feed preprocessed program text and the user-declared rules into a specialized solver. If there is an error in the preprocessor or rules, the user must start over; in DIMPLE$^+$, one may simply assert or retract additional rules or relations as necessary. Other factors that inhibit casual experimentation and rapid, interactive prototyping come from implementation details: BANSHEE analyses are C programs that link with a specialized solver library. BDDBDDB specifications require a good BDD variable ordering. Deciding on a good ordering is nontrivial and requires either a user with a profound understanding of the BDD abstraction and the problem domain or a user who is willing to wait for the BDDBDDB tool to automatically apply time-consuming heuristics and learning techniques to find a good ordering.

In conclusion, DIMPLE$^+$ is a framework that facilitates rapid prototyping, development, and implementation of program preprocessors and static analyses. Because the analysis designer can defer decisions about which program statements are relevant – or even which analysis rules are necessary – until the analysis is actually producing results, DIMPLE$^+$ encourages experimentation and interactive development and provides for a spectrum of executable analyses from flexible prototypes to efficient production implementations. More generally, we have confirmed prior work that has asserted the suitability of general-purpose logic programming systems for program analysis and processing tasks.

4 EFFECT INFERENCE FOR SAFE PARALLEL EXECUTION

The concept "cause," as it occurs in the works of most philosophers, is one which is apparently not used in any advanced science. But the concepts that are used have been developed from the primitive concept (which is that prevalent among philosophers), and the primitive concept, as I shall try to show, still has importance as the source of approximate generalisations and pre-scientific inductions, and as a concept which is valid when suitably limited.

— Bertrand Russell (1948)

Effect systems extend classical type systems with information about the computational effects exhibited by expressions, statements, and methods. Just as type signatures characterize the range of values an expression may assume, effect signatures can provide concise, useful summaries of the potential effects of a particular method invocation. Because of this capability, effect systems currently enjoy widespread application in several problem domains, including program analysis, semantics-preserving program transformation, software understanding, verification, and compile-time memory management; our goal in using effect signatures is to identify noninterfering computations on different objects in order to exploit implicit OLP.

In this chapter, we present two innovations that can increase the expressivity and precision of effect signatures. *Initialization effects* are writes that occur to the state of an object while it is being constructed but before it is available to the rest of the program; *quiescing fields* are instance variables of an object whose values remain constant after the dynamic lifetime of the object's constructor. We present these in the context of a fairly simple effects system for Java, but these novel features are based on concepts orthogonal to the underlying effects system and could be adapted to more expressive systems.

In Chapter 2, we introduced notions of function purity that exploits one engineering property of object-oriented programs: namely, that mutable state is typically accessed through the interface of the object that contains it. We refer to instance methods that are pure with respect to all mutable state outside of their receiver object as *externally pure*; we refer to methods that may read (but not write) external mutable state as *externally read-only*. In this chapter, we present analyses to infer such methods automatically.

Perhaps most surprisingly, we show that realistic Java programs exhibit a substantial degree of *mostly-functional* behavior. "Mostly-functional," due to Knight (1986), describes a programming discipline in which the presence and extent of computational effects are limited as much as possible. In the context of Java, this includes both accesses to quiescing fields — which are read-only after the object is available to the rest of the program — and the prevalence of externally-pure and externally read-only methods, whose updates to mutable state are only visible via an object's interface.

4.1 MOSTLY-FUNCTIONAL BEHAVIOR IN JAVA PROGRAMS

Knight (1986) coins the term "mostly-functional" to describe a programming style in which unnecessary side effects are limited as much as possible. Knight also notes that programming languages like Multilisp (Halstead 1985) exploit a similar style: namely, annotating an expression with future indicates either that it is side-effect free or that its side effects will not interfere with other parts of the program. The idea behind OLP, which we introduced in Chapter 2, is that many nontrivial Java programs are written in a similar kind of style: namely, that most of the side effects exhibited by a method will be benign, unobservable, or confined to the receiver object of that method.

Although he is describing a language based on Scheme that does not explicitly support objects and classes as language-level abstractions, Halstead (1985) describes a programming style that is very similar to that encouraged by contemporary object-oriented languages:

> [W]here side effects are used, as in maintaining a changing database, they can be encapsulated within a data abstraction that synchronizes concurrent operations on the data. The data abstraction can ensure that the data are only accessed according to the proper protocol.
>
> Multilisp thus supports a programming style in which most code is written without side effects, and data abstractions are used to encapsulate data on which side effects may be performed [....] The programmer's aim in using this style should be to produce a program whose side effects are compartmentalized carefully enough that any module may safely be invoked in parallel with any other. If this style is followed, the difficulties caused by the presence of side

effects will be isolated to small regions of the program and should therefore be reduced to manageable proportions.

Our intuition about object-oriented programs is that, as in mostly-functional programs, effects are often "compartmentalized." Unlike the effects exhibited by programs in mostly-functional languages, the effects exhibited by object-oriented programs are encapsulated within method, instance, or class boundaries rather than within module boundaries. Effects on ephemeral state, such as increments to a loop induction variable, are by definition invisible to other method activations. Effects on durable state, such as accesses to instance or static fields, often occur in a controlled way: some client of a particular object interacts with it through a well-defined interface.

In the remainder of this chapter, we will present analyses that can confirm this intuition about effecting behaviors in object-oriented programs. We will also refine the effects system for object-oriented programs that we initially sketched in Section 2.1; our goal will be to more precisely capture useful assertions about the effecting behavior of object-oriented programs by summarizing methods more accurately and masking effects that we can guarantee to be unobservable except from code executing in the methods on a particular object.

4.2 OBJECTS AND EFFECTS: BACKGROUND AND MOTIVATION

Recall that an *object-oriented effects system* is like a classical type-and-effect system in that it tracks not only the *what* — that is, the shapes and uses of values — but also the *how* — that is, the effects on durable state exhibited by a particular computation.[1] An object-oriented effects system is different from a classical type-and-effect system, though, in that it treats changes to the state of some particular object specially; it is able to distinguish between a method that may write an int to the f field of its receiver object and a method that may write an int to the f field of some other object. When applied to a typed, safe language like Java — in which a given memory location may only be described by one kind of field reference — an object-oriented effects system should also be able to identify that accesses to the f and g fields of arbitrary objects are disjoint.

Given a program with sound effects annotations (whether these are automatically derived or placed manually by the programmer), we can derive

[1] We initially defined these concepts at the end of Section 2.1, on pages 11–17.

scheduling constraints for parallel executions that will guarantee SEE (cf. Definition 2.8 on page 19) under the PIMA model, since methods with disjoint effects will commute. In this section, we will provide some background on object-oriented effects systems before introducing a simple system, which we shall successively refine by developing new features to improve its precision.

The Greenhouse-Boyland system

Greenhouse and Boyland (1999) developed an effects system for object-oriented languages like Java. Their effects system describes READ and WRITE effects that may occur in a hierarchy of regions:

1. The global region *All* contains all mutable state for an entire program,
2. *All* contains *static regions* that model the state of static fields and *instance regions* that contain part or all of the state of individual objects (an object may have several instance regions), and
3. individual instance regions contain regions corresponding to the state of individual instance fields.

The state of an object may contain the entire state of another object as part of its internal representation. For example, a dictionary object may contain a search tree object that is only accessible from the instance methods of the dictionary object. To address this possibility, Greenhouse and Boyland also provide an *unshared* annotation on reference-valued fields. This annotation indicates that any object referred to by an unshared field may only be referred to by that field and thus may be considered logically part of the state of its containing object.

Greenhouse and Boyland present an intraprocedural algorithm to check user-provided effects signatures of methods and to check user-provided *unshared* annotations on object fields, but they do not present an algorithm for reconstructing effect, region, and sharing information for unannotated programs.

4.3 A LIGHTWEIGHT OBJECT-ORIENTED EFFECTS SYSTEM

We now introduce a lightweight, straightforward object-oriented effects system. The basic ideas of our system are similar to that of Greenhouse and Boyland.

There are, however, nominal differences, semantic differences, and technical differences. The nominal differences are, by definition, straightforward: we call the global region ρ_ω, for example. The semantic differences are rather more interesting: notably, we do not support multiple instance regions per object, and we do not distinguish between instance regions and static regions. Instead, effects on static fields are merely effects on static field references in ρ_ω.

The technical differences between our approach and that of Greenhouse and Boyland are most prominent; some of these have to do with our implementation and the fact that we are inferring rather than checking effects:

1. Since our source language is Java bytecodes rather than Java source, and since our target language is not a dialect of Java with explicit support for effects, we provide an inference algorithm rather than a checker for user-supplied annotations.

2. As above, we support only one region for the nominal state of each object. However, we also support a hierarchy of regions based on an inclusion relation: field references are contained within abstract regions, which correspond to sets of abstract objects (taken from a may-alias analysis), or to the whole program.

Other technical details of our system represent novel contributions:

1. We introduce and infer a new kind of effect, *initializations*, which are writes to the state of an object while it is being constructed.

2. We automatically identify *quiescing fields*. A quiescing field is one whose value does not change after the object containing it has been constructed; READ effects on such fields can be safely masked when identifying interfering effects, since they represent accesses to run-time constants.

3. We automatically identify the *degree of purity* of a method. As we introduced in Definitions 2.6 and 2.7, methods may be "pure" (for a given definition of pure) or "read-only" both in an absolute sense and with respect to all of the state external to an object.

We present these developments in the context of a straightforward effects system, although they are orthogonal to the particular effects system in use and could benefit a more expressive system. We do this not only because it allows a more straightforward presentation, but also because it emphasizes that our novel features can dramatically increase the precision and expressivity of even a lightweight, simple effects system.

Relation	Description
$\mathsf{formal}(l, i, m)$	Holds when l is the formal parameter at position i (either this or a natural number) in method m.
$\mathsf{actual}(s, l, i)$	Holds when s invokes some method with l as the actual parameter at position i.
$\mathsf{assign}(l_l, l_r)$	Holds when the assignment $l_l = l_r$ occurs in the program.
$\mathsf{load}(s, l, l_h, \kappa.\nu)$	Holds s reads the value of the $\kappa.\nu$ instance field from the object referred to by l_h and copies it to l.
$\mathsf{load}(s, l, l_h, [])$	Holds when s loads l from an array referenced by l_h.
$\mathsf{load}(s, l, l_\omega, \kappa.\nu)$	Holds when s loads from the static field $\kappa.\nu$ into l.
$\mathsf{store}(s, l_h, \kappa.\nu, l)$	Holds when a heap store statement s replaces the value of the $\kappa.\nu$ field in the object referred to by l_h with the value of l.
$\mathsf{store}(s, l_h, [], l)$	Holds when s stores l into an array referenced by l_h.
$\mathsf{store}(s, l_\omega, \kappa.\nu, l)$	Holds when s stores the value of l into the static field $\kappa.\nu$.
$s \in m$	Holds when statement s is part of method body m.
$s \rightarrow m$	Holds when statement s contains a method invocation that may select m, viz., there is a call-graph edge from s to m.
$\vdash l : \tau$	Holds when local l has type τ. (Since locals have unique names, we need not consider typing environments.)
$\mathsf{pt}(l, \rho)$	Holds when ρ overapproximates the points-to set of l.

Table 4.1: Analysis-specific IR relations for a simple object-oriented effects system

We begin by defining an analysis-specific intermediate representation, which identifies heap loads and stores that execute in a particular method. In the presentation that follows, we assume the existence of the analysis-specific IR relations from Table 4.1, which includes conservative approximations of the call graph ($s \rightarrow m$) and the may-alias relation (pt); note that these are very similar to the analysis-specific IR presented in Section 3.6 except that we do not ignore operations on scalar values. Following Greenhouse and Boyland, we treat array loads and stores as accesses to a special field called `[]`. We treat static field loads and stores as accesses to instance fields of a distinguished local l_ω; since we record the declaring class and field name of all field accesses, we lose no precision by doing so. With the program distilled into this IR, we can begin developing a family of related effects inference algorithms of increasing sophistication.

The effects annotation on some statement, which we denote as $\varphi(s)$, consists of READ, WRITE, and (in our more sophisticated system) INIT sets of abstract locations. Abstract locations denote sets of concrete locations in which an effect may occur and consist of a pair $\langle \rho, \kappa.\nu \rangle$, where ρ is an abstract region in which an effect may occur and $\kappa.\nu$ describes a field reference qualified by the declaring class of the field.

Abstract regions consist of (possibly-empty) sets of abstract object identifiers (we leave the representation of these defined by a particular may-alias relation pt), the distinguished abstract region $\top$, which includes all possible abstract object identifiers, or special region variables ρ_{this} or $\rho_{0 \cdots n}$ denoting the regions reachable from formal parameters this or $0 \cdots n$; these variables are used to expand method summaries at call sites. We summarize the effects of methods on objects referred to by their parameters but lose precision for objects reachable from the fields of method parameters.

Effect annotations, as tuples of sets, form a lattice; since we are concerned with unifying effect locations, we only detail the join operation here. The join of two effect annotations consists of the READ set of abstract locations formed by unifying the READ sets from each annotation, the WRITE set formed by unifying the WRITE sets from each annotation, and the INIT set formed by unifying the INIT sets from each annotation. Unifying two sets of abstract locations $\mathcal{A}_1$ and $\mathcal{A}_2$, as in a READ, WRITE, or INIT set, proceeds as follows:

Divide each set $\mathcal{A}_i$ into the two disjoint sets $\mathcal{V}_i$ and $\mathcal{C}_i$ so that $\mathcal{V}_i$ is the set of all abstract locations from $\mathcal{A}_i$ whose regions are region variables, so that $\mathcal{C}_i$ is the set of all abstract locations from $\mathcal{A}_i$ whose regions are sets of abstract object identifiers or $\top$, and so that $\mathcal{V}_i \cup \mathcal{C}_i = \mathcal{A}_i$. $\mathcal{V}_1 \sqcup \mathcal{V}_2$ is defined simply as the union of the two sets. $\mathcal{C}_1 \sqcup \mathcal{C}_2$ consists of the union of the following:

1. The set of locations whose field identifiers appear in either $\mathcal{C}_1$ or $\mathcal{C}_2$, but not both:
$$\begin{aligned}\{\langle \rho, \kappa.\nu \rangle \ : \ & (\langle \rho, \kappa.\nu \rangle \in \mathcal{C}_1 \land \neg \exists \langle \rho', \kappa.\nu \rangle \in \mathcal{C}_2) \lor \\ & (\langle \rho, \kappa.\nu \rangle \in \mathcal{C}_2 \land \neg \exists \langle \rho', \kappa.\nu \rangle \in \mathcal{C}_1)\}\end{aligned}$$

2. The set of locations formed by unifying the regions of each abstract location whose field identifier appears in $\mathcal{C}_1$ and $\mathcal{C}_2$:
$$\{\langle \rho \cup \rho', \kappa.\nu \rangle \ : \ \langle \rho, \kappa.\nu \rangle \in \mathcal{C}_1 \land \langle \rho', \kappa.\nu \rangle \in \mathcal{C}_2\}$$

We can then define $\mathcal{A}_1 \sqcup \mathcal{A}_2$ as $\mathcal{V}_1 \cup \mathcal{V}_2 \cup (\mathcal{C}_1 \sqcup \mathcal{C}_2)$.

A simple, sound effects inference algorithm

Inferring sound but imprecise effects annotations for Java bytecodes is fairly straightforward. We know that heap reads and writes occur in a method if the load or store relations occur in its body or in the effects annotation of any method that it may transitively invoke. We also know what sort of data are being read or modified by field reads and writes, since Java `putfield` and `getfield` instructions (and their `putstatic` and `getstatic` analogues) include a *field reference*. A bytecode field reference is like an abstract field reference (Definition 2.3), but without any region information; it contains a declaring class name κ and a field name ν.

Abstract field references disambiguate the meaning of a field name by translating from a simple name (as would appear in Java source code, like `f`) to a name that is qualified with the name of the field's declaring class (like `Foo.f`). In so doing, however, abstract field references also impose constraints on the potential types of the base object. Because Java is a typed language, it is not possible to refer to an object of class c via a reference of type d, where d is not a supertype of c. (Doing so will result in either a compile-time or run-time error, depending on the particular circumstances of the cast.) Therefore, if a heap load accesses some field $\kappa.\nu$ from some object referred to by l, we can deduce that l refers to an object that is a subtype of κ.

Because Java's type system allows that a particular memory location may only be described by one field name, we can infer that a statement that exhibits an effect on some field $\kappa_1.\nu_1$ must not interfere with a statement that exhibits an effect on field $\kappa_2.\nu_2$ if ν_1 and ν_2 are distinct names or if κ_1 and κ_2 are distinct

$$\frac{\text{formal}(l, i, m)}{\text{rpt}(l, \rho_i)}\ \textsc{rpt-formal} \qquad \frac{}{\text{rpt}(l_\omega, \rho_\omega)}\ \textsc{rpt-global} \qquad \frac{l \neq l_\omega \quad \neg\text{formal}(l, i, m) \quad \text{pt}(l, \rho)}{\text{rpt}(l, \rho)}\ \textsc{rpt-other}$$

$$\frac{\text{load}(s, l, l_h, \kappa.v) \qquad \text{rpt}(l_h, \rho)}{\varphi(s) \sqsupseteq \textsc{read} : \{\langle \rho, \kappa.v \rangle\}}\ \textsc{simple-read} \qquad \frac{\text{store}(s, l, l_h, \kappa.v) \qquad \text{rpt}(l_h, \rho)}{\varphi(s) \sqsupseteq \textsc{write} : \{\langle \rho, \kappa.v \rangle\}}\ \textsc{simple-write}$$

$$\frac{s_0, \cdots, s_n \in m}{\varphi(m) \sqsupseteq \varphi(s_0) \sqcup \cdots \sqcup \varphi(s_n)}\ \textsc{simple-summary} \qquad \frac{s \to m'}{\varphi(s) \sqsupseteq \text{pmap}(s, \varphi(m'))}\ \textsc{simple-call}$$

Figure 4.1: Rules for reconstructing effects annotations in the straightforward effects system

classes. This alone provides a useful measure of memory disambiguation for field reads and writes.

We assume an input program with no effects annotations and present rules for reconstructing sound effects annotations in Figure 4.1. We denote an effects annotation on a particular statement as $\varphi(s)$; our inference rules identify the constraints on $\varphi(s)$ induced by each statement type. Each effect is a 2-tuple consisting of a *kind* (either READ or WRITE, in this system) and an abstract field reference.

Heap loads and stores exhibit READ and WRITE effects. We identify regions in which these effects may occur by the rpt relation. rpt relates a local variable to its associated region: a region variable for formal parameters, the global region $\top$ for globals, and the set of abstract objects aliased by the local in other cases.

The rules SIMPLE-READ and SIMPLE-WRITE, which establish lower bounds on the effect annotations for load and store statements, are straightforward. SUMMARY gives the annotation summary for a method body; it is this summary that is instantiated at call sites. Statements containing method invocations must have an effects signature that is greater than the union of the effects of all statements in any method that might be selected by a given call site, as given by SIMPLE-SUMMARY and SIMPLE-CALL.

The function pmap transforms effects annotations by substituting regions (or region variables) for region variables in a method summary at the point of a call to that method. pmap replaces every region variable with the region variable or explicit region associated with the local of the corresponding actual parameter, as given by the rpt relation.

4.4 INITIALIZERS AND INITIALIZATION EFFECTS

Type-and-effect systems identify READ and WRITE effects that code may exhibit upon shared state. (Some, but not all, type-and-effect systems also identify additional effects, such as allocation, exception raising, or taking references.) If the goal of effect systems is to identify potentially interfering computational effects, this taxonomy is rather impoverished, because it does not distinguish *initializations*, which are a special kind of WRITE that will not interfere with any other effects.

Background and definitions

We will introduce the notion of initializations with a simple example, but first we provide some background on some properties of Java programs and objects.

Java objects are created via the `new` operator, which performs three tasks before returning a reference to the newly-allocated object: memory allocation, zero-filling object fields, and constructor method invocation. Constructors may invoke other constructors declared in the same class (via the `this()` syntax) or in superclasses (implicitly or explicitly via the `super()` syntax), but there is no way to invoke a constructor on an object after the dynamic lifetime of its constructor invocation completes. There is also no way to create and use an object without invoking its constructor. (This is the case in Java source because `new`, which is the only way to create an object, includes both object allocation and constructor invocation. These tasks correspond to distinct Java bytecode instructions – `new` and `invokespecial` – but the Java Virtual Machine will signal an error if code attempts to access an object that has been allocated but not constructed.) As a consequence, each object will be constructed exactly once before it is accessible to the code that created it.

Consider the `String` class in the Java standard library. `String` is an *immutable* class; once an instance of `String` has been created, its contents cannot be modified. A constructor for the `String` class, in setting up the state of an individual instance, will exhibit WRITE effects on that object's fields. However, these WRITE effects will never interfere with other effects, since the only WRITE effects on a `String` will occur during its constructor and the code that creates a `String` will not be able to read its state until after the constructor completes.

Immutable classes present an extreme example, but WRITE effects on an object — even a mutable one — by its constructor will not interfere with other effects on that object that occur after the constructor completes. Classical type-and-effect systems do not discriminate between writes that occur to an object during its constructor and writes that occur after an object creation has completed. Such a system may spuriously identify WRITE effects occurring on an object during its creation as interfering with WRITE effects occurring on that object (or on other objects) that have already been created.

We will present a way to discriminate between WRITE effects to objects that have been created and initializations, which are writes that occur to an object while it is being constructed. However, we will first introduce the notion of an *initializer method* and present an algorithm for identifying which methods are initializers for given objects.

Initializer methods

Informally, an *initializer method* (or simply an *initializer*) on some object o is a method that executes on o during the dynamic lifetime of its constructor. Since we would like to use the notion of initializer methods to identify WRITE effects that are guaranteed to occur on an object while it is being constructed, we are not interested in any method that merely may initialize part of an object's state; rather, we are interested in methods that may *only* execute on an object during the dynamic lifetime of its constructor.

If we can assume a closed world, we can identify such methods with a simple extension to a conservative static approximation of the program's call graph. This extension annotates call-graph edges with information about a method's receiver; we thus term it a *receiver-sensitive call graph*.

Definition 4.1 A *receiver-sensitive call graph* (RSCG) is a set M of nodes corresponding to method bodies, a distinguished *start node* $m_{\mathsf{main}} \in M$, and a set C of labeled call-site edges. An edge is of the form $m \rightarrow_{\rho} m'$, indicating that m contains a call site that may invoke m' with a receiver of ρ, which is either this, indicating that m' is always invoked on the same object as m, or $\top$, indicating that m' may be invoked on some other method or is not an instance method.

We can now define the notion of initializer methods more formally:

Definition 4.2 An *initializer* is a method that may only execute on an object during the dynamic lifetime of its constructor. m is an initializer if and only if every path from the root of the RSCG to m goes through a constructor c_i^m, and every path from any c_i^m to m consists strictly of this-labeled edges.

We can define a conservative overapproximation of the initializer methods in a program inductively as follows:

1. m is an initializer on o if m is a constructor that may be executed on o (that is, a constructor declared in the class of o or in one of its superclasses).

2. m is an initializer on o if every edge to m in the RSCG is this-labeled and originates from an initializer on o.

In the remainder of this discussion we will assume a closed world. We note, however, that this technique is still applicable in an open-world situation — that is, in which the entire program and libraries are not available to be analyzed. It is still possible to identify initializers in an open world as long as the RSCG is constructed in such a way as to include conservative, sound assumptions about open parts of the program. For example, `private` methods could still soundly be identified as initializers even in an open world, since they can only be invoked from within their declaring class.

Initialization effects

An *initialization effect* is a write to an object's state that occurs during the dynamic lifetime of its constructor. Since we have already defined an initializer on some object o as a method that is only transitively invoked through zero or more this-edges in the RSCG from a constructor on o, we can identify initialization effects rather straightforwardly: An initialization effect is a WRITE effect that occurs from within an initializer and on some field of its receiver. We denote sets of initialization effects as an INIT set in an effects annotation and present updated inference rules for initializer methods and for WRITE and INIT effects in Figure 4.2. (The WRITE rule from Figure 4.2 supercedes that from Figure 4.1.)

If we can assume that an object will not be used until the after the dynamic lifetime of its constructor, then we can guarantee that INIT effects on some

$$\textsc{imeth-immed}\quad \frac{m \text{ is a constructor}}{\mathsf{imeth}(m)} \qquad \textsc{imeth-trans}\quad \frac{(\forall m', \rho) m' \rightarrow_\rho m \models \mathsf{imeth}(m') \land \rho = \rho_{\mathsf{this}}}{\mathsf{imeth}(m)}$$

$$\textsc{write}\quad \frac{\begin{array}{c}\mathsf{store}(s, l_h, \kappa.v, l) \qquad \mathsf{rpt}(l_h, \rho) \\ s \in m \qquad \neg(\rho = \rho_{\mathsf{this}} \land \mathsf{imeth}(m))\end{array}}{\varphi(s) \sqsupseteq \textsc{write} : \{\langle \rho, \kappa.v \rangle\}} \qquad \textsc{init}\quad \frac{\begin{array}{c}\mathsf{store}(s, l_h, \kappa.v, l) \qquad s \in m \\ \mathsf{imeth}(m) \qquad \mathsf{rpt}(l_h, \rho_{\mathsf{this}})\end{array}}{\varphi(s) \sqsupseteq \textsc{init} : \{\langle \rho_{\mathsf{this}}, \kappa.v \rangle\}}$$

Figure 4.2: Inference rules for initialization methods and initializer effects

object o will not interfere with other read or write effects on objects that are not statically distinguishable from o. This assumption — that uses of o by code outside of its constructor will come strictly after the dynamic lifetime of its constructor — is sound if no references to o may leak to code that might execute before the constructor for o has completed execution. We will make the (unsound) assumption that any such leak will not result in an object being used before the dynamic lifetime of its constructor completes. However, a variety of techniques are applicable to treating such leaks soundly; we sketch one such technique here.

A label-flow analysis could be used to indicate those constructors that might leak a reference to the constructed object.[2] Alternatively, a more expressive effects system that tracks reference accesses could be employed to find such leaky constructors; one such system is presented by Cherem and Rugina (2007). The classes containing such constructors could then be considered to not have initializers; as a consequence, WRITE effects occurring during the dynamic lifetime of a constructor on an object of such a class would be conservatively regarded as potentially interfering with `write` effects that occur strictly after the completion of an object's constructor.

Initialization effects are a useful addition to the expressivity of object-oriented effects systems. Since the initializations of a field during an object's creation will not interfere with any reads conducted after the dynamic lifetime

[2]Such analyses can be efficient enough for production use: Pratikakis, Foster, and Hicks (2006) present a framework for solving label-flow analysis problems via CFL-reachability; while the worst-case complexity of CFL-reachability was long understood to be cubic, Chaudhuri (2008) demonstrated that the problem has a subcubic worst-case bound.

of the object's constructor, initialization effects allow effect systems to statically identify a greater range of effects as noninterfering. As we shall see, inferring initialization effects also enables additional analyses, such as identifying quiescing fields.

4.5 QUIESCING FIELD INFERENCE

Some storage is mutable for its entire lifetime, but the lifetimes of many locations can be divided into two phases: an initialization phase, in which the contents of a location are mutable, and a read-only phase, in which the contents of a location will not change. We call such fields *quiescing fields* when the phase transition happens at a statically identifiable and semantically useful place. In this section, we introduce the concept of quiescing fields, explain how we can identify them, and describe why they are useful; compare quiescing fields to Java's `final` fields; and identify the static and dynamic prevalence of quiescing fields in the Java programs from the DaCapo benchmark suite.

Quiescing fields, defined and identified

We define a *quiescing field* as an instance field (i.e. an object member) that is mutable while its containing object is constructed but is immutable for the entire period of program execution strictly after the dynamic lifetime of its containing object's constructor. As a consequence, a quiescing field will have the same value for the entire period that the object containing the field is accessible to the code that created it (and to the rest of the program, modulo the no-leaks assumption of the previous section).

Because a quiescing field is guaranteed not to change after the object that contains it is fully constructed, quiescing fields represent a useful kind of run-time constant. If quiescing fields are prevalent in a program, identifying them can greatly simplify analyses and transformations that require accurate interprocedural data dependence information.

Given sound effects annotations including initialization effects, it is quite straightforward to identify quiescing fields: $\kappa.\nu$ is quiescing if and only if no effect annotation in the whole program contains an abstract location implicating $\kappa.\nu$ (e.g. $\langle\rho, \kappa.\nu\rangle$) in its WRITE set. (If $\kappa.\nu$ is not implicated in the INIT or WRITE sets of any effects annotation, then it is never written after allocation and is trivially a quiescing field.)

Because we need only examine every effect in the whole program once in order to determine which fields are implicated in WRITE effects — and we need not even unify method summaries at call sites in order to do so — quiescing field inference scales linearly with the number of statements in the program.

Quiescing fields compared to final fields

The Java language (Gosling et al. 2000) provides the `final` keyword and the semantic guarantee that instance variables declared as `final` will be assigned to exactly once for any given containing object. The `final` annotation provides useful documentation to human readers of a program and a useful constraint for use by compilers and analyses.

However, because the guarantee of finality is enforced by a rather coarse flow analysis (identifying "definite assignment," that is, that each final field is on the left-hand side of exactly one assignment along every possible path through each constructor of the object containing it), `final` is of limited applicability. To give one example, since all assignments to `final` fields must occur in the body of a constructor, it is impossible to share initialization code common to several constructors in a `private` instance method.

While it is often possible to restructure the code in a class so that a quiescing field meets the criteria for `final`, such a rewrite may be inconvenient. Furthermore, rewriting code so that a quiescing field is `final` may well obscure the clear meaning of the program for a human reader. Since many programmers will not immediately realize the benefits of having as many fields as possible declared `final`, manual code transformations to expose more fields as `final` are likely to be regarded as insufficiently profitable.

On the contrary, quiescing fields may be written arbitrarily many times during the dynamic lifetime of an object's constructor, not strictly in the static body of the constructor and exactly once along each path of each constructor. Quiescing fields may be read and written freely during the dynamic lifetime of their containing object's constructor, so long as they are not written to after their containing object is fully constructed. Finally, no programmer annotations are necessary to identify quiescing fields, since we present a straightforward and efficient technique for automatically inferring quiescing fields.

Static and dynamic prevalence of quiescing fields

We evaluated our definition of quiescing fields on seven of the programs from the DaCapo benchmark suite (Blackburn et al. 2006).

		Static		*Dynamic*	
Input	Time	% FF	% QF	% FF	% QF
antlr	3.11	19.89	49.25	3.65	24.13
bloat	3.16	22.30	53.01	64.05	70.05
eclipse	3.23	21.50	51.56	77.69	78.53
hsqldb	3.73	18.67	47.97	20.12	58.75
jython	3.61	18.74	52.99	19.17	50.30
luindex	3.06	20.82	51.06	43.87	47.43
pmd	3.35	19.48	48.47	0.78	24.93

Figure 4.3: Static and dynamic prevalence of final and quiescing fields in select DaCapo benchmarks. Time represents analysis time in seconds; *static* numbers show the percentage of fields implicated in at least one static effect that are final (FF) and quiescing (QF); *dynamic* numbers indicate the percentage of dynamic reads in a benchmark execution that are of final (FF) and quiescing (QF) fields.

We identified the *static prevalence* of `final` and quiescing fields by determining what percentage of all fields implicated in any effect were declared `final` and what percentage were inferred to be quiescing. (Since `final` fields are, by definition, quiescing, counts of quiescing fields include counts of `final` fields.) We also instrumented the Jikes RVM in order to get a trace of all instance field reads from a benchmark execution. From this trace, we derived the percentages of dynamic instance field reads that access `final` and quiescing fields; again, the count of quiescing field reads includes `final` field reads.

Figure 4.3 gives our complete results; in summary, we found that between 18.7% and 22.3% of fields implicated in any static effects annotation were declared `final`; between 48% and 53% of fields implicated in any static effects annotation were identifiable as quiescing. Between 0.78% and 77.7% of dynamic reads were from `final` fields, and between 24.13% and 78.53% of fields were from quiescing fields. The authors of the bloat, eclipse, and luindex benchmarks seem to have declared a high percentage of frequently-read quiescing fields as `final`; in the other benchmarks, the disparity between the number of dynamic reads of `final` and quiescing fields is much greater.

4.6 DEGREES OF PURITY

Methods may be *pure*. The classic definition identifies a method that exhibits no effects on mutable state as pure. However, this definition fails to admit idempotent methods that create and modify objects in order to complete their work.

A less restrictive definition, due to Leavens et al. (1998) and applied for static analysis by Salcianu and Rinard, characterizes a method as pure if and only if it does not modify any state that exists immediately before method entry. This definition of purity captures a notion of method purity as the absence of potential interference with other code: a method may have effects on mutable state that does not exist before it executes. Other definitions of purity are also possible; the concepts we present in this section are generally orthogonal to a base notion of purity and can be straightforwardly adapted to different definitions.

In accepting a definition of purity, we also decide which effects constitute "impure" behavior. Perhaps all side effects are "impure," as in the classical defintion. Alternatively, following Leavens et al., we could ignore certain READ or WRITE effects on objects that did not exist at a method's entry. (While our system does not provide for precisely tracking READ effects on newly-created objects or for tracking all WRITE effects on such objects, we can ignore many writes to newly-allocated objects by masking INIT effects.) We can then identify some methods as *read-only* – these are methods that may have "impure" READ effects (but not "impure" WRITE effects) on mutable state. (Note that all pure methods are also read-only methods.)

If we are to characterize the purity of methods in typical object-oriented programs, we may wish to characterize instance methods by the effects that they have on mutable state that exists outside of the receiver object. In Section 2.2, we introduced the concepts of *externally-pure* and *externally-read-only* methods (Definitions 2.6 and 2.7), which are methods that are pure or read-only with respect to all state external to their receiver object.

As we have said before, certain effects are benign or unobservable and could well be masked. The idea behind degrees of external purity is that it might be useful to categorize methods by where effects occur — whether or not these effects could be masked. We can combine the notion of external purity with initialization effects and quiescing fields by masking INIT effects (which represent writes to the state of objects that did not exist when the method began) and masking READ effects on quiescing fields. If we do so, we

		Externally	
Input	Time	% Pure	% RO
antlr	4.47	79.19	81.16
bloat	4.63	77.05	78.40
eclipse	4.63	79.21	80.73
hsqldb	5.44	76.87	78.26
jython	5.25	77.02	78.31
luindex	4.39	80.30	81.89
pmd	4.88	79.05	80.50

Figure 4.4: Percentage of all instance methods that are externally-pure or externally-read-only; INIT effects and effects on quiescing fields have been masked.

can identify a vast preponderance of instance methods as externally-pure or externally-read-only, as in Figure 4.4.

4.7 RELATED WORK

Work related to our contributions in this chapter falls into two broad categories: work on effects systems and work on inferring fields or memory locations that are immutable for at least some part of their lifetime.

Effect systems and applications

There is a substantial and broad canon of theoretical contributions and applications for effect systems. Several good overviews of research in this area are available, including Nielson and Nielson (1999) and Henglein, Makholm, and Niss (2005). We shall focus on work that imposes effect systems on object-oriented languages and on notable applications of effect systems: region-based memory management, static race detection and prevention, and verification.

Region-based memory management (Tofte and Talpin 1997) aims to achieve the convenience and correctness of automatically-managed memory while avoiding the performance penalties and heap fragmentation common to garbage-collected and manually-managed heaps.[3] It proposes that dynamically-allocated

[3]Garbage collection is, in particular, ill-suited for real-time systems.

storage should be organized as a stack of heaps, called *regions*,[4] and that new objects should be allocated into a region containing objects of comparable lifetimes. (Regions often, but not always, have lexically-scoped lifetimes.) When program execution proceeds to a point where no datum from a particular region will be required again, the contents of that entire region are deallocated at once. If region lifetimes are lexically scoped, then the end result is effective compile-time management of dynamic heap memory.

The central problem is that manual region annotations suffer in many of the same ways as other program annotations: they are tedious to apply, they can be verbose, and it can be difficult for a region checker to give useful feedback in the case of incorrect annotations. (If there is no region checker, then incorrect region annotations will have the same result as accessing memory that has been `free`d in a C program.) Therefore, we would like some way to infer correct region annotations. It is not enough, however, for region annotations to be correct. We would also like regions that minimize the amount of wasted heap space. That is, we would like the smallest reasonable region lifetimes. Effect systems can help identify when data are no longer necessary and eliminate dangling reference bugs by ensuring that references to data do not flow to longer-lived regions.

The Cyclone language (Grossman et al. 2002) added region-based memory management and automatic region inference (among many other useful features) to a C-like language. Boyapati et al. (2003) used an explicit type system based on ownership types to enable region-based memory management in Java programs. Chin et al. (2004) presented a polymorphic region inference algorithm for lexically-scoped regions in a subset of Java, targeting applications in real-time Java. The Jreg system, due to Cherem and Rugina (2004), extends full Java with support for automatic region-based memory management. Cherem and Rugina (2006) also developed Jfree, a system to identify and exploit opportunities for compile-time deallocation of individual objects.

Escape analysis and *object inlining transformations* are closely related to region inference. Escape analysis identifies objects whose lifetimes are bounded by the lifetime of the method in which they were created in order to enable allocating some heap objects on the stack or eliminating synchronization operations on certain objects. Object inlining transformations identify contained objects whose lifetime is bounded by the lifetime of a containing object in order

[4]Note that this usage is subtly different from the original definition of "region" in the context of effect systems, as a set of locations that might be aliased.

to subsume the fields and methods of contained objects into their containers.

Choi et al. (1999) present a subset-based dataflow analysis framework (with both flow-sensitive and flow-insensitive instantiations) to identify objects that do not escape the method or thread in which they were constructed. Vivien and Rinard (2001) developed an incrementalized version of a flow- and context-sensitive points-to and escape analysis.

Dolby and Chien (2000) presented an object inlining transformation and its evaluation; since it is based on value flow, it could be cast as a type inference problem.[5] Laud (2001) developed a slightly different definition of object inlinability. Lhoták and Hendren (2002) compare the definition of inlinability presented by Dolby and Chien with that of Laud and analyze dynamic traces of Java programs in order to identify opportunities for object inlining.

Lucassen and Gifford's 1988 paper on polymorphic effect systems focused on identifying scheduling constraints for execution of implicitly-parallel programs, as did Talpin and Jouvelot (1992), who presented a polymorphic type, region, and effect reconstruction algorithm. A related problem is ensuring that explicitly-parallel programs are free of race conditions. Effect systems and approaches inspired by effect systems have also been applied to this end.

Boyapati and Rinard (2001) presented a technique combining an ownership type system with effects for race- and deadlock-free execution of Java programs.[6] Choi et al. (2002) extend earlier work on escape analysis in order to precisely detect races in explicitly-parallel Java programs. Flanagan and Qadeer (2003) developed a type and effect system for identifying atomic methods, or those that can be assumed to execute serially (without interference from thread interleavings); they also argue that identifying atomicity is a better criterion for safe parallel executions than identifying the absence of data races. More recently, Naik, Aiken, and Whaley (2006) presented a system for static race detection based on the combination of effects and object-sensitive points-to analysis (Milanova et al. 2005).

Another notable application for effect systems is as a basis for other tools or analyses: whether automatically providing annotations for a model checker or specification language, or as the foundation for other interprocedural analyses. Salcianu and Rinard uses pointer and effects inference to identify pure methods (using "pure" in the sense of Leavens et al.); they do so by masking writes to `this` exhibited by a constructor. Cherem and Rugina (2007) developed a

[5] See also Dolby (1997) and Dolby and Chien (1998).

[6] See also Boyapati et al. (2002).

parameterized framework for compact effect signatures, which allows clients of effect annotations to trade precision for annotation size.

Skalka, Smith, and Horn (2005) present an effect system and inference algorithm for abstract interpretation of Featherweight Java (Igarashi et al. 1999). Their approach infers *history effects* or *trace effects*, which are labeled transition systems that approximate run-time traces of program behavior and can be used for model checking. They also show that their approach allows modular extensions to the base language (e.g. adding support for exceptions) by postprocessing effects annotations.

The natural compatibility of effects and objects has led to a great deal of excellent work. As we discussed in Section 4.2, Greenhouse and Boyland (1999) devised an idiomatic, object-oriented treatment of regions and effects, but did not provide an inference algorithm. Bierman and Parkinson (2003); Bierman et al. (2003) extended the work of Greenhouse and Boyland with a semantic treatment of effects and an effects inference algorithm for a subset of Java. However, their work left region annotations as the responsibility of the programmer. Skalka (2005), building on earlier work (Skalka et al. 2005), provides a detailed explanation of the interaction of trace effects and effect inference with object-oriented language features.

Given a notion of effects, it is possible to talk about the purity of functions. Barnett and Naumann (2004) formulate a system of invariants on object state that holds under ownership (in which state is encapsulated) and friendship (in which state is shared); their system identifies invariant dependence on states while ours provides an overapproximation of independence. Barnett et al. (2004) present several definitions of purity in the context of object-language methods that may appear in checkable specifications: observational purity (which admits memoization), strong purity (the classic definition), and weak purity (in which methods may modify newly-allocated state). Sălcianu and Rinard (2005) present an analysis to identify weakly-pure methods. Barnett et al. (2007) extend the Sălcianu-Rinard analysis to support iterators and the additional features, such as pass-by-reference, of the .NET runtime.

Our contributions — quiescing fields, initialization effects, and degrees of purity — are intended to enhance the expressivity and precision of effect systems and purity analyses and thus are orthogonal to the specific effects system or purity analysis being used. Our contributions could easily be adapted to benefit other effects systems; in fact, features of other systems could readily be added to the effects system of Section 4.3: for example, the parameter-leaking and borrowing effects of Cherem and Rugina could model information flow

```
public String foo() {
    // exhibits INIT effects on fields of s
    StringBuffer s = new StringBuffer("foo");

    // exhibits WRITE effects on fields of s
    s.append("bar");

    // exhibits INIT effects on fields of a new String
    return s.toString();
}
```

Figure 4.5: Example of a pure method that is not identified as such by our simple system

more accurately; one could use the results of a *field uniqueness* analysis, like that of Ghemawat et al. (2000) or Ma and Foster (2007) in order to automatically place *unshared* annotations. Conversely, we believe that initialization effects, quiescing fields, and external purity can be introduced to an extant effect system as crosscutting concerns.

As an example, we compare our pure-method inference to analyses built on a more expressive effect system. Our analysis can be used to identify the subset of weakly-pure methods that only exhibit INIT effects on newly-allocated objects simply by masking INIT effects; other analyses, like Sălcianu and Rinard (2005) and Barnett et al. (2007) can identify a broader range of weakly pure methods. Because they track points-to information precisely, their effects system will identify some pure methods that our technique (of masking INIT effects) will not: for example, a method like the one in Figure 4.5, which modifies a newly-created object after the dynamic lifetime of its constructor, is pure by the Leavens et al. definition, but will not be identified as pure by the rules we present in Section 4.6. However, because we track initialization effects separately from writes, we are able to readily identify quiescing fields; in addition, our notion of degrees of purity can aid client analyses, program understanding, and local reasoning by identifying methods whose effects are confined to their receiver object. In any case, our contributions could easily be added to such a system.

Inferring eventual immutablity

Porat et al. (2000) developed a flow-sensitive analysis for identifying that static fields — as well as the objects to which they refer — were immutable, even if the fields were not declared as `final`. In contrast, our approach merely identifies that the value of a field (i.e. a reference) will not change after the dynamic lifetime of its containing object's constructor; the state of an object referred to by a quiescing field may still change. Identifying unchanging references and identifying immutable objects represent complementary and orthogonal problems.

A different, but related, problem is that of checking and identifying *reference immutability*, which indicates that an object will not be modified via a particular reference. Tschantz and Ernst (2005) extended the Java language with type qualifiers to identify readonly references, which cannot be used to modify their referent,[7] and assignable fields, which are used for caching and may thus be written to via readonly methods. Zibin et al. (2007) built upon this work by using Java generics to provide object and reference immutability without syntax or type-system extensions. Finally, Quinonez et al. (2008) provided a tool and technique for precisely inferring readonly references in unannotated code. Reference immutability, like object immutability, is orthogonal to identifying quiescing fields.

Most directly related to our concept of quiescing fields is the *stationary fields* analysis of Unkel and Lam (2008), which we shall focus on in the remainder of this review. Unkel and Lam use a flow- and context-sensitive pointer analysis to identify fields for whom every dynamic read can be statically guaranteed to come after every dynamic write.

While both stationary fields and quiescing fields are capable of identifying eventually-immutable fields that are not declared `final`, and both identify about half of all fields in some set of realistic Java programs as eventually-immutable, there are several interesting differences between our approaches. First, our approach is substantially more lightweight: we use a flow- and context-insensitive analysis that exhibits linear time complexity and runs in seconds; their approach uses a flow- and context-sensitive analysis that takes between 7 and 106 minutes to analyze a realistic Java program. However, while our approach simply identifies quiescing fields, their approach identifies stationary fields and can also track their referents with greater precision. Both approaches

[7] This is similar to `const` pointers in C++.

can be used to improve the precision of effects and interprocedural def-use analyses; theirs is better-suited for also improving the precision of points-to analyses.

Perhaps more interestingly, though, is that the definitions of quiescing and stationary fields are subtly incompatible: while it seems most likely that the intersection of the sets of quiescing and stationary fields for any given program would be large, there are quiescing fields that are not stationary fields (e.g. those that might be read in the constructor before a write), and there are stationary fields that are not quiescing fields (e.g. those that are written after the dynamic lifetime of a constructor, but before any use of an object). We suspect that investigating the relationship between these two kinds of fields — and whether or not quiescing fields annotations could be used to improve the speed of stationary fields analysis — presents a fruitful avenue for future work.

5 RUNTIME EVALUATION AND SUPPORT

Philosophy is perfectly right in saying that life must be understood backwards. But then one forgets the other clause — that it must be lived forwards. The more one thinks through this clause, the more one concludes that life in temporality never becomes properly understandable, simply because never at any time does one get perfect repose to take a stance: backwards.

— Søren Kierkegaard (1843, trans. Hong)

We're not talking about the game. We're talking about practice.

— Allen Iverson (2004)

In Chapter 2, we presented object-level parallelism, a measure of implicit parallelism in object-oriented programs, and PIMA, a programming model designed to exploit OLP. In Chapter 4, we presented a type-and-effect system and analyses to identify mostly-functional behavior in Java programs: namely, quiescing fields and instance methods whose computational effects are confined to their receiver objects. (The analyses of Chapter 4 were implemented in our DIMPLE$^+$ framework, which we described in Chapter 3.) We also evaluated these analyses both in terms of their properties on static program texts and, in the case of the quiescing fields analysis, in terms of what it could tell us about dynamic program executions.

Therefore, we have a way to structure parallel programs and coordinate between parallel tasks, and a mechanism for identifying that two program fragments are noninterfering. These two are are necessary but not sufficient prerequisites for making the case that a particular kind of parallelism is manifest in realistic program executions, let alone for transforming serial programs into parallel ones. In this chapter, we characterize actual executions of the DaCapo benchmarks (Blackburn et al. 2006) under the Jikes RVM (Alpern et al. 1999, 2005) in order to answer questions about the actual dynamic behavior of programs and their implications for OLP: *how much time do programs spend in externally-pure and externally-read-only methods? How many individual externally-pure or externally-read-only method invocations are long enough to constitute plausible parallel tasks?* We present empirical results addressing these two questions and also consider the question of runtime support for exploiting OLP, which we first mentioned in Chapter 2: *What sort of virtual* *machine modifications and extensions would we need to exploit this implicit parallelism?*

The remainder of this chapter contains a description of our evaluation methodology (Section 5.1)and an evaluation of typical externally-pure and externally-read-only methods as suitable parallel tasks (Section 5.2). We then discuss the requirements for implementing virtual machine support for exploiting OLP (Section 5.3) before reviewing some of the most relevant related work on characterizing implicit parallelism in workloads.

5.1 EXPERIMENTAL ENVIRONMENT & EVALUATION METHODOLOGY

It is difficult to observe and characterize program behavior at a fine granularity, and any approach involves some compromises. Interrupting a program in order to inspect its state is likely to perturb its execution — especially on contemporary microprocessors, in which hundreds of instructions may be in flight at any given time. Instrumenting programs to generate traces allows tool or analysis designers to post-process information about program behavior in arbitrarily complex ways, but can also perturb executions, incurs nontrivial overheads, and can generate an enormous amount of data in a very short period. Running programs under simulation is flexible and can avoid observation effects since the behavior of the simulator may not be visible to the executed program, but simulation may be prohibitively slow to evaluate realistic programs. Before we introduce our approach for evaluating and characterizing the workloads presented by the DaCapo benchmarks, which is based on dynamic instrumentation and virtualized execution, we will briefly review some standard approaches. There is a great deal of excellent work on characterizing program performance and building tools to aid the same; our focus in this section is merely on reviewing the fundamental mechanisms and techniques for doing so.

Background

Simulation has long been an attractive option for performance projection (Lucas, Jr. 1971), or evaluating the performance impact of changing certain parameters of a system without actually implementing that system. In many domains, this was borne of necessity: it would be unfeasible, for example, to evaluate a new cache replacement policy by designing and fabricating a new microprocessor! More recently, simulation has been profitably applied not only to evaluating the parameters of systems and hardware but to evaluating and debugging application programs; Engblom et al. (2006) provide one example of using a commercial full-system simulator to evaluate and test software that is designed to run on embedded systems. The advantage to using simulation to evaluate application software is that it is possible to observe an application (and possibly also the virtual machine or operating system) in a way that is transparent to that application, because the simulator controls the application's view of the world. However, simulated execution can be orders of magnitude slower than execution on real hardware and thus may be unsuitable for evaluating entire executions of realistic programs.

Sample-based profilers, at their most basic, interrupt programs at specified intervals and inspect the stack; in so doing, they are able to provide a probabilistic estimate of which methods account for the highest percentage of program executions. Sample-based profiling is simple to implement, but it only provides coarse, probabilistic information about an entire program execution in the aggregate: it cannot describe how long individual method activations were atop the stack, for example, and thus loses precision since a method may be very expensive in certain contexts but not in others. Graham et al. (1982) extend the classic sampling concept with the `gprof` tool, which records call-graph information at the entry to each method so that it can ascribe *inclusive* timings to methods. Put another way, `gprof` can regard the time spent in callees of `m` as time spent in `m`. Instead of sampling when a timer quantum is exceeded, it is also possible to sample program execution when some quantum of another resource is exceeded, for example, when some number of floating-point exceptions or page faults has passed, as in Hall and Goldberg (1993). This approach has proven useful for full-system profiling — that is, including the application, the libraries or runtime, and the operating system kernel — Mousa et al. (2007) provide an example of this technique.

Other profilers do not sample program behavior and instead exclusively *instrument programs* to generate a trace of relevant events, stop and start timers, and count the number of times a high-level language statement or procedure is executed. Some such profilers insert event counters on basic blocks and branches; these are able to ascribe execution counts to control-flow graph edges or acyclic intraprocedural paths (Hall 1992; Ball and Larus 1992, 1996). Ammons et al. (1997) combine instrumentation with the idea of measuring resources other than time and with the concepts of flow- and context-sensitivity; their work uses hardware counters to ascribe microarchitectural events (such as cache misses or branch mispredictions) to program paths when executed in a particular context.

As we have seen, program instrumentation is used by basic block, call graph, and edge profilers. Program instrumentation can also be used to update counters, start and stop timers, and generate traces containing arbitrary events of interest. Of course, the overhead of executing instrumentation code may alter the performance characteristics of the program; furthermore, a trace generated of events that are deemed *a priori* potentially interesting may be prohibitively large. While instrumentation may be inserted statically — whether manually by programmers or automatically by program transformations — both the overheads of executing instrumentation and the quantity of traced events can be reduced by inserting and removing instrumentation *dynamically*, as shown by Hollingsworth et al. (1994). The dynamic instrumentation approach has been successful in problem domains ranging from automatic performance diagnosis of parallel programs (Miller et al. 1995) to low-overhead application debugging (Zhao et al. 2008).

Our approach

Dynamic instrumentation also forms the basis of our approach for characterizing the method executions in the DaCapo benchmarks. (Note that we used static instrumentation — albeit generated by a just-in-time compiler! — to characterize the dynamic prevalence of quiescing fields in Section 4.5.) Because we are running DaCapo under the Jikes RVM, a metacircular managed runtime that compiles the application code immediately before execution, along with standard library classes and substantial portion of its own support code, it is possible to examine the impact of code that is traditionally unavailable, like the allocator, garbage collector, and just-in-time compiler itself. However, this flexibility comes at a price: we do not have a traditional executable with debug symbols and routines in fixed locations and must therefore reconstruct a great deal of high-level information from the program's execution as it happens.

In order to do this, we built a specialized profiler on top of the Pin dynamic instrumentation framework (Luk et al. 2005). Pin combines aspects of dynamic instrumentation and simulation: it is able to insert arbitrary code specified in so-called *analysis routines* at points identified by *instrumentation routines*, and it does so by executing the application under a virtualized execution environment, dynamically recompiling the application with the analysis routines inserted in the application code. A program extending Pin by providing analysis and instrumentation routines is called a *pintool*.

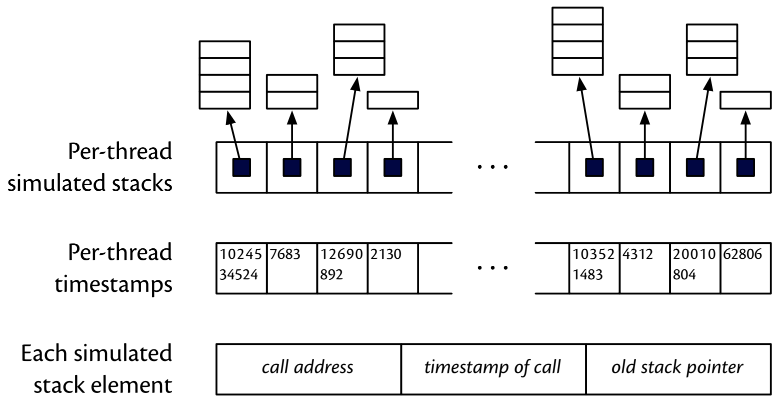
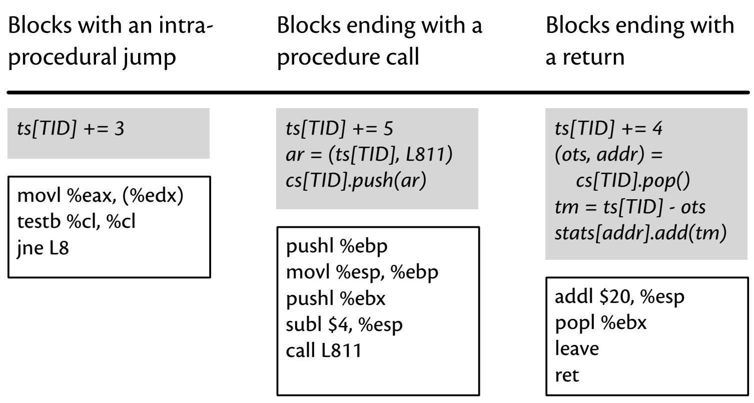


Figure 5.1: Per-thread state maintained by instrumentation code

The main goal of our pintool is to generate, for each method, aggregate statistics about the duration of its invocations: its invocation count, the duration of its shortest and longest invocations, and estimates for the mean and variance parameters for the distribution of invocation lengths. In order to do this, our pintool reads traces of program events and simulates program execution, maintaining some abstracted state for each virtual machine thread. (Note that a Java virtual machine may create many threads even while executing a single-threaded program – for example, the garbage collector and other runtime support code may run in a separate thread.)

We say that the state maintained by our pintool is abstract because it does not contain all of the state in a concrete execution: it does not track the values of architected registers, memory loads and stores, or the details of intraprocedural control flow, for example. Instead, it keeps for each thread a representation of the call stack, which records which methods (and `try` blocks) are currently executing, and a timestamp counter, which records the number of instructions executed in that thread. The values we track, however, are not abstract; that is, we know exactly how many application instructions have executed in each thread as well as the precise addresses of each method on the stack at a given time. Figure 5.1 shows the state that our pintool maintains for each thread.

In order to collect this information, we instrument every basic block in the Jikes RVM boot image and in all dynamically-compiled application and library code. (Because Pin handles dynamically-generated and self-modifying code, our pintool is able to soundly treat code that is recompiled by the virtual ma-


Blocks with an intra-procedural jump
Blocks ending with a procedure call
Blocks ending with a return

```
ts[TID] += 3

movl %eax, (%edx)
testb %cl, %cl
jne L8
```

```
ts[TID] += 5
ar = (ts[TID], L811)
cs[TID].push(ar)

pushl %ebp
movl %esp, %ebp
pushl %ebx
subl $4, %esp
call L811
```

```
ts[TID] += 4
(ots, addr) =
   cs[TID].pop()
tm = ts[TID] - ots
stats[addr].add(tm)

addl $20, %esp
popl %ebx
leave
ret
```

Figure 5.2: Example instrumentation code (gray) inserted for three kinds of basic blocks (white). (Old stack pointers are omitted from the simulated stack in this illustration.)

chine's adaptive optimization system.) Every basic block `b` is instrumented with code that increments the thread-specific timestamp counter for the currently-executing virtual machine thread by the number of instructions in `b`. We then treat different kinds of basic blocks in different ways:

1. Basic blocks that end with intraprocedural control-flow (say, for example, a simple branch) are not instrumented any further. That is, each such block is only instrumented with the timestamp counter increment;

2. Basic blocks that end with a procedure call instruction (whether calling an actual Java method, a subroutine, or entering a `try` block) are also instrumented with code to record an abstracted activation record, consisting of the called address, the current thread-specific timestamp (taking into account the number of instructions in this block), and the current stack pointer value;

3. Basic blocks that end with a return instruction are instrumented with code that updates the stack, determines which method (or subroutine, `try` block, etc.) the program is leaving, and updates the aggregate statistics for that method.

We give examples of each of these three kinds of basic blocks in Figure 5.2.

We are thus able to keep a running count of the number of calls for each method, as well as the minimum and maximum number of instructions executed in a single invocation of each method. We also keep updated estimates of the mean and variance for distribution of invocation durations for each method, using a technique due to West (1979). Finally, we modified the Jikes RVM to dump out a symbol table at the end of program; therefore, we are able to map call addresses to Java methods — and to the results of our purity analysis from Chapter 4.

5.2 METHOD INVOCATIONS AS POTENTIAL TASKS

In order to profitably execute a method asynchronously, we must be able to make several guarantees about its behavior:

1. It must not interfere with other methods that may execute at the same time;
2. It must represent a large enough computation to justify the communication costs of delegating it to another thread; and
3. There must be enough slack between the method's invocation and the point in the dynamic execution when its value will be required.

The first criterion is a safety consideration, but it is also a performance consideration: we must guarantee safety, either statically (e.g. by analysis), dynamically (e.g. by checking or software transactional memory), or by some combination of static and dynamic approaches. However, dynamic approaches for guaranteeing noninterference typically incur greater overheads in the event of conflicts, so we would like to limit the likelihood of conflict as much as possible.

We use the purity analysis from Chapter 4 as an overapproximation of noninterference; we note that externally-pure and externally-read only methods will not interfere with one another. (Although we do not do so, it is possible to statically identify impure methods that will not interfere.) In this section, we focus on the second criterion above by characterizing the durations of externally-pure, externally-read only, and impure method invocations; we leave identifying slack as an open question for future work.

Amdahl's law implies that it is not profitable to parallelize infrequently-executed code paths. In realistic programs, methods that are executed only once or merely a few times are often not characteristic of the majority of execution. As a consequence, we restrict our investigation to methods that are executed relatively more frequently: for each benchmark, we first identify those methods that are invoked frequently enough to account for more than one-one-hundredth of a percent of all method invocations. Table 5.1 lists the number of methods we examined for each benchmark.

	Externally		
Benchmark	Pure	Read-only	Impure
antlr	218	258	1245
bloat	99	105	659
eclipse	65	73	2157
jython	230	242	1044
luindex	156	173	928
pmd	34	36	461

Table 5.1: Counts of methods that represent more than 0.01% of total method invocations.

Because we are interested in finding relatively fine-grained parallel tasks to execute on chip multiprocessors, our ideal task size is large enough so that it will not be dominated by communication times (that is, accesses to a shared L2 or fast L3 cache), but not so large that the task is likely to cause destructive interference with concurrently executing code on a neighboring core. While different processors vary widely, we assume that task sizes of around one hundred instructions represent the smallest plausibly justifiable task on a processor with realistic intercore communication latencies. (This would correspond to a — very fast — latency of twenty or thirty cycles to access the L2 cache.)

Figures 5.3–5.8 present cumulative distribution plots of the mean number of dynamic instructions executed in individual invocations of each frequently-invoked method; note that the x-axis on each graph is scaled to the quantile function. Recall that we define a frequently-invoked method as one that accounts for at least one-one-hundredth of a percent of all method invocations. The impure methods for each benchmark include all methods we identified as pure and all methods that we did not analyze, including those methods that implement runtime support functionality (e.g. memory management and the adaptive optimization system) in the Jikes RVM.

For almost all of the benchmarks we analyzed, more than 50% of frequently executed, externally-pure and externally read-only methods have mean durations of approximately 100 instructions or more. (`pmd` is the notable exception.) Furthermore, a significant number of externally-pure and externally-read only methods are likely to execute for several hundred instructions or more. The bottom quartile of each plot typically features very short methods — we note

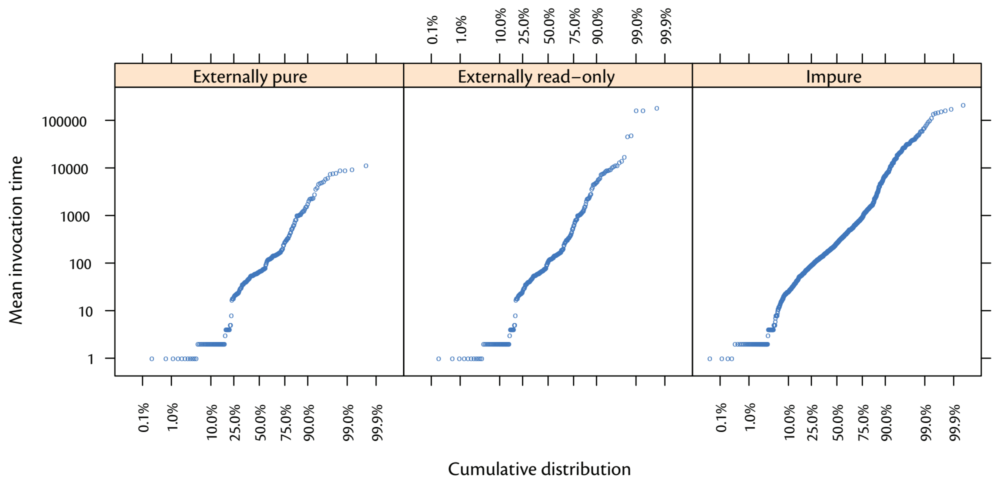


Figure 5.3: Mean method invocation durations for all frequently-executed methods in the `antlr` benchmark.

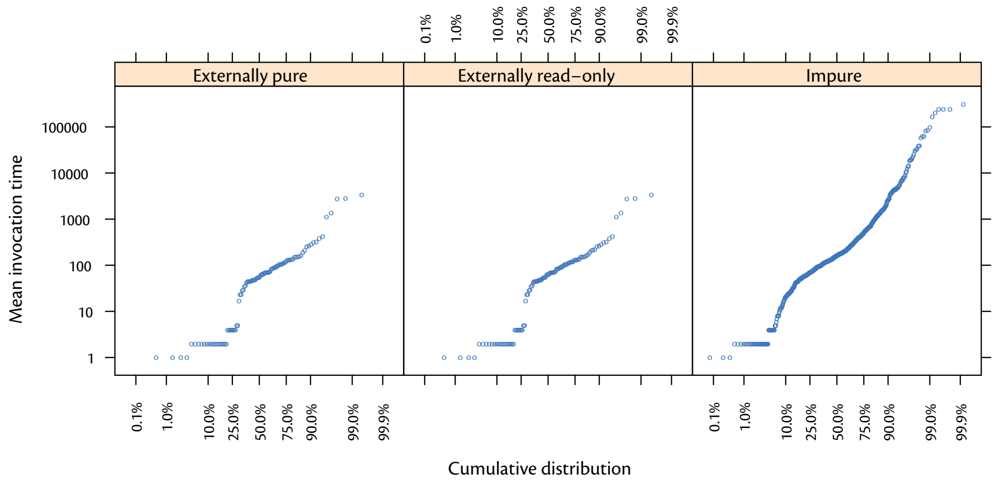


Figure 5.4: Mean method invocation durations for all frequently-executed methods in the `bloat` benchmark.

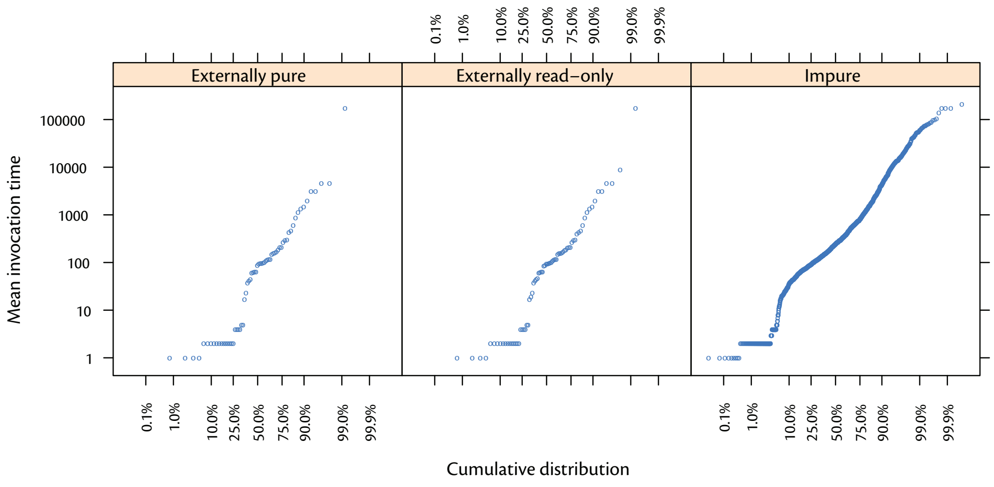


Figure 5.5: Mean method invocation durations for all frequently-executed methods in the `eclipse` benchmark.

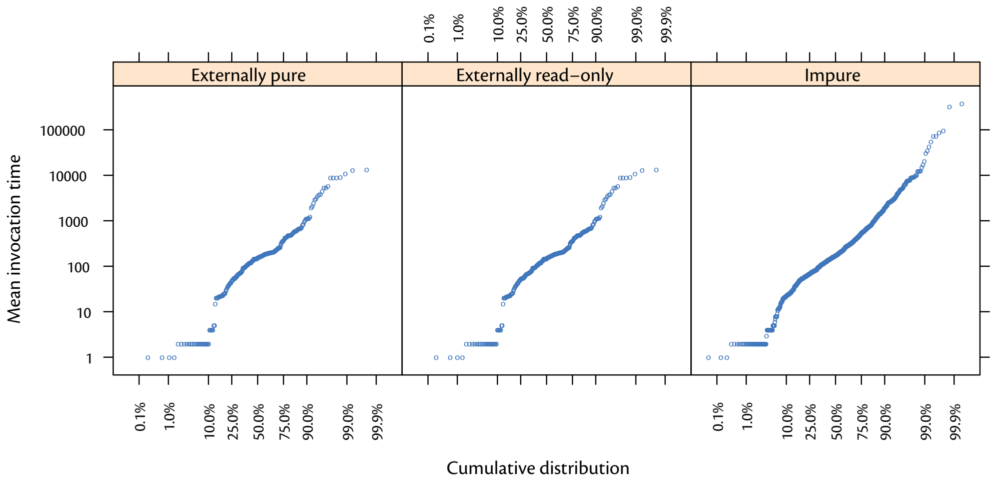


Figure 5.6: Mean method invocation durations for all frequently-executed methods in the `jython` benchmark.

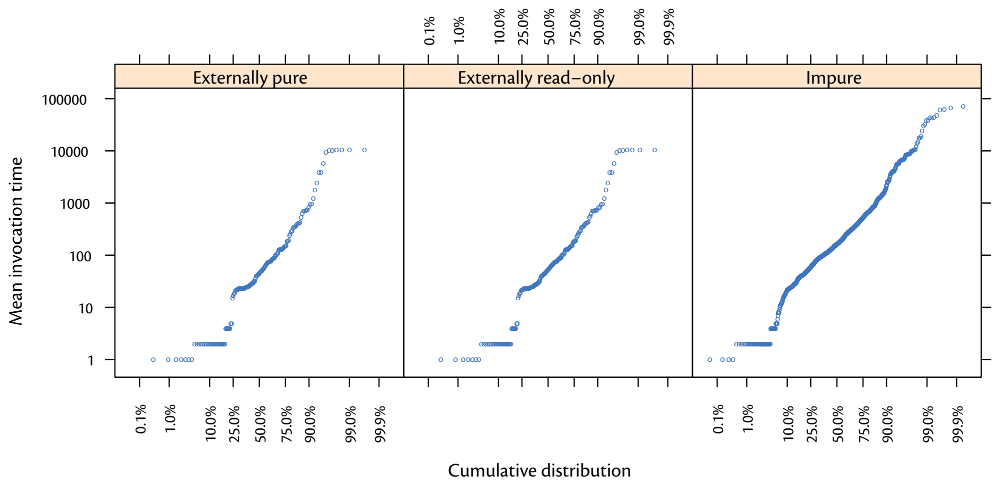


Figure 5.7: Mean method invocation durations for all frequently-executed methods in the `luindex` benchmark.

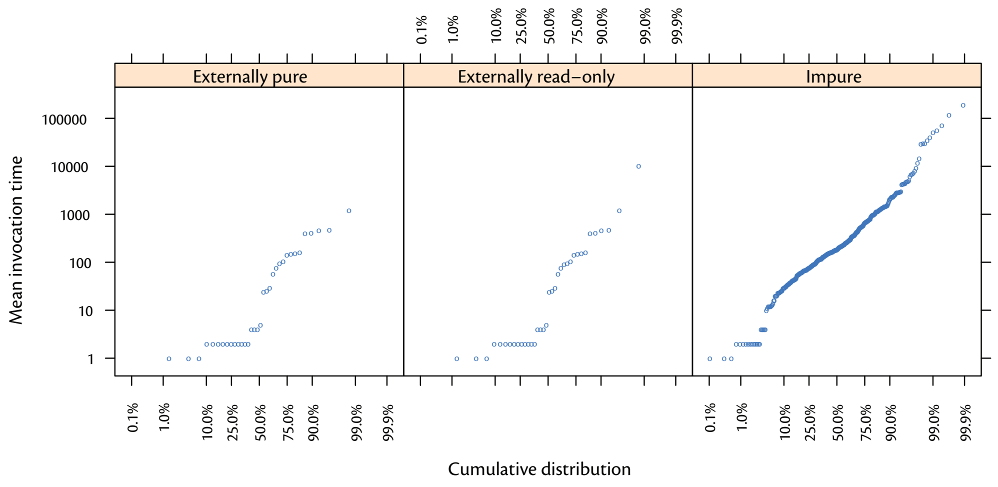


Figure 5.8: Mean method invocation durations for all methods frequently-executed in the `pmd` benchmark.

that short methods that are executed very frequently are likely to be inlined by the VM and thus will not appear at all — while the graphs for externally-pure and externally-read only methods taper off around methods of a few hundred instructions.

5.3 RUNTIME SUPPORT

Recall the main goals of runtime support for the PIMA model, as we sketched them in Chapter 2:

1. The runtime must be able to *demarcate* objects,

2. The runtime must be able to *delegate* methods on demarcated objects to run asynchronously on other *delegate threads*, and

3. The runtime must be able to synchronize between a delegate thread and the main thread when the main thread requires a value from a demarcated object.

We proposed using Java's virtual dispatch to maintain two copies of instance method bodies: one for normal execution, when an object is not demarcated, and one for when it is. The responsibilities of the method body for demarcated execution depend on the kind of method: methods that update object state but do not return a value require a method that proxies the invocation to an appropriate delegate thread; methods that return a value (ideally depending on the object's state) require a method that blocks until all other pending operations on this demarcated object have completed. (Refer to Figures 2.5, 2.6, and 2.7.)

While we will not evaluate an implementation of runtime support for PIMA and OLP, we can identify some additional goals that it should meet. Put another way, now that we have characterized the sorts of methods that we would like to be executing asynchronously on demarcated objects and examined more carefully the environment in which they will execute, we can suggest some additional constraints.

The first goal is related to a correctness concern. While many even very small methods are executed out of line, an optimizing runtime environment like the Jikes RVM may inline methods if a call site is likely to be monomorphic. (Because most contemporary virtual machines are able to invalidate assumptions when necessary — e.g. due to dynamic class loading or to a new type of receiver object reaching a call site — dynamic inlining can be even more

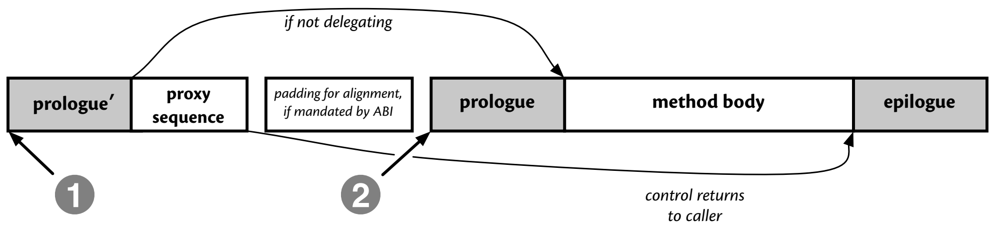


Figure 5.9: A more efficient proxy implementation.

aggressive than static inlining.) As a practical consequence, this means that it is possible for some very simple methods on a demarcated object to execute in the wrong thread under the simple virtual function table patching scheme we proposed in Chapter 2, so we need a scheme that will work in the face of inlining. There are several straightforward solutions to this problem. It would be possible to insert demarcation checks at the beginning of every compiled method, although this would introduce some overhead in the common case. Perhaps a better solution would be to generate a simple check at each inlined call site that might operate on a receiver other than `this`, executing the proxy code only if necessary.

The second goal is related to a performance concern: because methods — especially the sort of methods we might be delegating — are so short-lived, performance is crucial. We note that generating two copies of each method, one with proxy code and one without, is likely to increase memory and cache pressure. Instead, we propose generating both copies of the method in the same contiguous block of memory, as in Figure 5.9. In this approach, the virtual function table pointer either is at the location labeled **1** or **2**; **1** corresponds to execution on a (potentially) demarcated object and **2** corresponds to normal execution. In the demarcated case, the method will execute any necessary proxy code in the main thread before jumping to the epilogue and returning to the caller. If the receiver object is no longer demarcated, then control simply transfers to the instruction immediately following the second prologue and execution proceeds as normal in the calling thread. (Note that it may be necessary to pad the space between the delegate code so that both prologues are properly aligned to be the targets of `call` instructions.) In the non-demarcated case, execution simply proceeds normally.

5.4 RELATED WORK

The work described and proposed in this chapter falls into several categories: software tools for fine-grained performance evaluation, attempts to characterize latent parallelism, and efficient runtime support for concurrent object-oriented programs. We have reviewed results related to the first category already in this chapter (in Section 5.1) and contributions related to the third category after we first sketched our runtime support (see Section 2.4). In the remainder of this section, we will briefly review some notable efforts to establish limits on the upper bounds of implicit parallelism.

Historically, computer architects have devoted a great deal of research and engineering effort to understanding how much parallelism is implicit in realistic workloads. For many years, the primary focus of this work was on finding instruction-level parallelism (ILP) in order to motivate new advances in superscalar processor design. In identifying a limit to the amount of ILP possible, researchers also (implicitly or explicitly) identify their assumptions about what techniques for parallelism are possible: whether register or memory renaming, branch prediction, value prediction, etc., are available obviously have a huge impact on what parts of a program execution can be identified as "potentially parallel." There are many excellent results in this arena; we find especially notable the contribution of Austin and Sohi (1992), who present a technique for efficiently constructing dependency and dataflow graphs from traces of program executions, thus identifying the maximum possible ILP at different points in a program's execution.

More recently, following industry and research trends towards increased thread-level parallelism, some architects have examined implicit thread-level or task-level parallelism. This work is perhaps even more dependent on assumptions than work on finding implicit ILP, because it requires a researcher to identify both a model that puts constraints on what a task is and how it can be created (such as method-level parallelism or loop-body parallelism) as well as the technical constraints and capabilities of a particular implementation.

Chen and Olukotun (1998) investigated method-level parallelism in the context of the Hydra CMP chip; their technique is necessarily conservative with regard to methods with unpredictable return values or extensive side effects. Their later work on the Jrpm system Chen and Olukotun (2003) is targeted toward parallelizing loop-bound code, not idiomatic object-oriented programs. Their work depends on pipelining loop bodies — and thus depends, to some extent, on the ability of the programmer to alter methods so that their side effects occur in a limited portion of their execution, in order for method executions to be overlapped.

Warg and Stenström (2001) conducted a limit study of speculative *module-level parallelism* in Java and C. Their approach uses a *method-level parallelism* model that spawns a new thread for each method invocation; it is thus targeting slightly finer-grained parallelism than our model. Their proposal depends specialized hardware support: specifically, they expect that aggressive value prediction is available in order to enable speculative execution in the face of data dependences.

6 CONCLUSIONS AND FUTURE WORK

What has been is what will be, and what has been done is what will be done, and there is nothing new under the sun.

Is there a thing of which it is said, see, this is new? It has been already in the ages before us.

— Ecclesiastes 1:9–10 (esv)

Because of the many attempts to connect the past with the future one might be inclined to call this an Apollonian period. But the fury with which addicts of various schools fight for their theories presents rather a Dionysian aspect.

— Arnold Schoenberg (1948)

The thesis of this dissertation is that real-world client applications exhibit implicit thread-level parallelism because methods on distinct objects are often independent; furthermore, it is possible to identify this *object-level* parallelism statically via type-based program analysis and exploit it with lightweight run-time support. In an effort to argue this thesis, we have presented several contributions:

1. *Object-level parallelism* (olp), a measure of potential implicit parallelism analogous to instruction-level parallelism, and the pima programming model (in Chapter 2);

2. The dimple$^+$ declarative analysis framework for flexible, interactive, and scalable analysis of Java bytecode programs (in Chapter 3),

3. A novel object-oriented type-and-effect system, inference rules, and client analyses (implemented in dimple$^+$) to identify a large class of run-time constant fields and object instance methods that are good candidates for parallel execution (in Chapter 4); and

4. A characterization of dynamic opportunities to exploit latent olp in Java programs (in Chapter 5).

While we believe that these contributions are significant, we also believe that there are several interesting avenues for future work related to and enabled by the work described in this dissertation.

1. The DIMPLE$^+$ work presents a useful environment for prototyping and executing program analyses, but it is also a vehicle for research and development in logic programming systems. An early, unpublished version of DIMPLE$^+$ served as a test case for published work on the Yap system's performance on very large datasets.[1] Our work on DIMPLE$^+$ has raised interesting questions about characterizing the memory performance of tabled Prolog and the system-level and microarchitectural impact of declarative programming with large datasets. DIMPLE$^+$ could also be extended as a tool, perhaps through integration with automated reasoning systems.

2. The analyses of Chapter 4 likely have many applications beyond finding potential parallel tasks. We are interested in improving the precision of these analyses by incorporating them into more expressive effects systems and in finding new applications for them in software engineering, memory performance, and program understanding — to name a few areas.

3. Our results in Chapter 5 indicate that many of the methods that are invoked frequently in executions of these Java programs are likely to execute for long enough to justify asynchronous dispatch (and its prerequisite, intercore communication) on realistic processors. However, future work could present a more comprehensive limit study. There are research challenges involved in identifying exactly how to demarcate parts of a program trace into potential parallel tasks and obvious engineering challenges in managing and making sense of massive quantities of trace data.

4. Additionally, we note that at least two improvements are likely to have a positive impact on the number of methods that represent realistic parallel tasks: (1) *A more aggressive static analysis* that could identify more methods as externally-pure or read-only, would allow us to identify some methods that we currently regard as impure — and which, given our results, typically have longer mean runtimes — are actually unlikely to interfere with one another; and (2) *More sophisticated runtime support* that could enable us to execute even some impure methods asynchronously.

5. We have laid the analysis and programming-model foundations for an automated, round-trip system for finding and exploiting implicit OLP, which would include runtime support for asynchronous execution and static or dynamic

[1]See Costa (2007) and Costa, Sagonas, and Lopes (2007)

cost modeling to estimate profitability. Any one of these tasks is probably a worthy research program by itself.

Our initial goal in conducting this work was to bring some of the benefits of functional and declarative languages to "mainstream" object-oriented software; to bridge the gap between some good old ideas and a contemporary language climate that is not amenable — or even hospitable — to them. In this way, our work is analogous to type-based analysis, which imposes a parallel, implicit type system that encodes an analysis problem onto a language — and which can be applied profitably even to untyped languages. Even though many aspects of the Java language (and extant Java software) are hostile to declarative-style concurrency, we believe we have shown that it is possible to find and exploit implicit parallelism in realistic Java programs.

However, we hope that the next generation of programmers will not be fully employed finding parallelism in yesterday's bytecodes. Instead, we expect that programs will eventually be written in languages that make pervasive, aggressive concurrency possible. We believe that such languages will need to account for a landscape in which computers have a large number of CPUs, each with several cores or threads. Therefore, they must make explicit concurrency easier (by imposing mostly-functional constraints on code, enforcing modularity, and limiting sharing) and implicit concurrency possible (by being amenable to aggressive program analysis). We are fairly certain that the most fruitful future work growing out of this dissertation will focus upon applying the lessons of scheduling serial code to run as parallel tasks to the problem of finding a way to express programs in a parallel tomorrow.

A CONVENTIONS

This document employs many naming and typographical conventions in order to allow a clearer presentation of the material. In this chapter, we describe the conventions that apply throughout the document; locally-applicable conventions will be introduced when they appear.

A.1 NAMING CONVENTIONS

When we discuss Java programs at the source or intermediate representation level, we will deal with entities in several different domains. In order to simplify the presentation of analysis rules or properties of Java programs, we will name metavariables according to certain conventions. For example, a variable beginning with c will typically range over classes; $\mathcal{C}$ will denote the set of all classes. Table A.1 shows the set of domains and metavariable naming conventions.

Metavariable		Domain	Domain description
c, d, κ	$\in$	**Class**	Java classes
e	$\in$	**Exp**	Expressions
h	$\in$	**Heap**	Heap storage; field or array element references
k	$\in$	**Imm**	Constant or immediate values
l, x, y	$\in$	**Local**	Local variables
m	$\in$	**Method**	Method bodies
n, ν	$\in$	**Name**	Names
p	$\in$	**Param**	Parameter indices (i.e. natural numbers for explicit parameters or this for the implicit parameter in instance methods)
s	$\in$	**Stmt**	Statements
t, τ	$\in$	**Type**	Types

Table A.1: Conventions for metavariable domains

```
public class Point {
       private int x; // this is a discrete point
       private int y;

       public Point(int x, int y) { /* ... */  }

       public int getX() { return x; }
       public int getY() { return y; }
}
```

Figure A.1: An example Java program listing

```
member(X,[X|T]).
member(X,[_|T]) :- member(X,T).
```

Figure A.2: An example Prolog program listing

A.2 TYPOGRAPHICAL CONVENTIONS

We will use *emphasis* to denote inline definitions. Longer definitions, or those that will be cross-referenced, will be given in numbered paragraphs like the following:

Definition A.1 A *numbered definition* defines a term and provides a number for future references in the text.

When referring inline to programming language entities – including classes, methods, or keywords – or to external executable programs, we will use a `typewriter` face to set the term. For example, we might refer to the `String` class in the `java.lang` package, to the `jsr` Java bytecode instruction, or to the UNIX `ls` command. When presenting Java or Prolog source listings, we will use the conventions established in Figure A.1 and Figure A.2. Note also that, as in Figure A.1, we will often elide unimportant or obvious parts of program listings, and we will typically denote such an elision by placing an ellipsis within a comment. We use the standard nomenclature to refer to Prolog structures;

we will refer to a structure named `foo` of arity 2 as `foo/2`.

B RELATIONS IN THE DIMPLE IR

`addExpr(L1,L2)` Represents integer or floating-point addition with operands `L1` and `L2`.

`analyzed(M)` Indicates that method `M` has been processed by the DIMPLE$^+$ front-end. This relation is useful because it lets analysis designers develop analyses that correctly treat open programs.

`andExpr(L1,L2)` Represents logical or bitwise AND with operands `L1` and `L2`.

`arrayRef(L1,L2)` Represents the `L2`th element of the array referred to by `L1`.

`arraytype(T,I)` Represents the type of an `I`-dimensional array with a base type of `T`. For example, the type of a one-dimensional array of `int` would be denoted as `arraytype(primtype('int'),1)`

`assignStmt(L1,L2)` Represents an assignment from `L2` to `L1`.

`assignStmt(L1,H)` Represents a load from a heap location `H` to a local `L1`.

`assignStmt(H,L1)` Represents a store to `H` of the value stored in `L1`.

`branches(S)` Indicates that statement `S` may branch (and not merely fall through to the next statement).

`castExpr(L,T)` Represents a cast of local `L` to type `T`.

`caughtExceptionRef(T)` Represents the exception object of type `T` caught by a particular exception handler.

`cg_main(M)` Represents the root of the call graph; that is, the application `main` method.

`class(C)` Represents a member of the class domain.

`class(C,N)` Represents a class declaration with class ID `C` and name `N`.

`cmpExpr(L1,L2)` Represents a `cmp` bytecode with operands `L1` and `L2`.

`cmpgExpr(L1,L2)` Represents a `cmpg` bytecode with operands `L1` and `L2`.

`cmplExpr(L1,L2)` Represents a `cmpl` bytecode with operands `L1` and `L2`.

`concreteClass(C)` Indicates that `C` is a concrete class.

`containsStmt(M,S)` Indicates that `M` contains `S`.

`divExpr(L1,L2)` Represents division with operands `L1` and `L2`.

`doubleConstant(V)` Represents an immediate `double` value of `V`.

`enterMonitorStmt(L)` Represents entering the monitor for the object referred to by `L`.

`eqExpr(L1,L2)` Represents equality test with operands `L1` and `L2`.

`exitMonitorStmt(L)` Represents leaving the monitor for the object referred to by `L`.

`fallsThrough(S)` Indicates that `S` may fall through to the next statement.

`field(F)` Represents a member of the field domain.

`field_decl(F, NS, T, NQ, C, Vis, instance)` Represents an instance field `F` of type `T`, declared in class `C`. `F` has short name `NS`, qualified name `NQ`, and visibility specifier `Vis`.

`field_decl(F, NS, T, NQ, C, Vis, static)` Represents a static field `F` of type `T`, declared in class `C`. `F` has short name `NS`, qualified name `NQ`, and visibility specifier `Vis`.

`final(F)` Indicates that `F` is final.

`floatConstant(V)` Represents an immediate `float` value of `V`.

`geExpr(L1, L2)` Represents a greater-than-or-equal expression with operands `L1` and `L2`.

`gotoStmt(S)` Represents an unconditional branch to `S`.

`gtExpr(L1, L2)` Represents a strictly-greater-than expression with operands `L1` and `L2`.

`identityStmt(L,Exp)` Represents the initial (and sole) assignment to `L`, which models a formal parameter. `Exp` is either a `thisRef/2` or a `parameterRef/2`.

`ifStmt(L,S)` Represents a conditional branch.

`immedExtends(C1,C2)` Indicates that `C1` *immediately* extends `C2`. The transitive closure of this relation is the subclassing relationship.

`immedImplements/2` Indicates that `C1` *immediately* implements `C2`; `C2` must be an interface.

`instanceFieldRef(L,F)` Represents an instance field reference from the base object referred to by `L`.

`instanceOfExpr(L,C)` Represents a Java `instanceof` expression.

`intConstant(V)` Represents an immediate `int` value of `V`. (Also used to represent immediate values of narrower integral types.)

`interface(C)` Indicates that `C` is an interface (rather than a class).

`interfaceInvokeExpr(M,[P|Ps])` Describes a method invocation of `M` using interface-style dispatch.

`invokeStmt(Exp)` A statement invoking a method but ignoring its return value.

`leExpr(L1,L2)` Represents a less-than-or-equal expression with operands `L1` and `L2`.

`lengthExpr(L)` Represents an array-length inspection.

`local(N)` Denotes a member of the local domain.

`local(L,M,T)` Denotes a local `L` in method `M` with type `T`.

`longConstant(V)` Represents an immediate `long` value of `V`.

`ltExpr(L1,L2)` Represents a strictly-less-than expression with operands `L1` and `L2`.

`mainclass(C)` Indicates that `C` holds the `main` method for the program that this database represents.

`method(N)` Denotes a member of the method domain.

`method_decl(M, N, C, SSig, V, IS, [Mod|Mods])`

`mulExpr(L1,L2)` Represents multiplication with operands `L1` and `L2`.

`neExpr(L1,L2)` Represents inequality with operands `L1` and `L2`.

`negExpr(L)` Represents unary negation.

`newArrayExpr(Source,T)` Represents an array allocation of base type `T`; `Source` represents the location in the program text of this allocation.

`newExpr(Source,T)` Represents an object allocation of type `T`; `Source` represents the location in the program text of this allocation.

`newMultiArrayExpr(Source,T)` Represents a multidimensional array allocation of base type `T`; `Source` represents the location in the program text of this allocation.

`nullConstant(_)` Represents `null`.

`offset(M,V)` Represents the bytecode at offset `V` from the beginning of method `M`.

`orExpr(L1,L2)` Represents logical or bitwise OR with operands `L1` and `L2`.

`param_type(M,I,T)` Indicates that parameter `I` of method `M` has type `T`.

`parameterRef(M,I)` A reference to parameter `I` of method `M`.

`phiExpr([value_unit(L,S)|VUs])` A ϕ-expression, with a list of `value_unit/2` structures.

`primtype(Name)` Describes a primitive type.

`publicClass(C)` Indicates that `C` is a `public` class.

`reftype(Name)` Describes a reference type.

`remExpr(L1,L2)` Represents modulus with operands `L1` and `L2`.

`returnStmt(L)` Represents returning a value.

`returnVoidStmt(_)` Represents return from a `void` method.

`return_type(M,T)` Indicates that the return type of `M` is `T`.

`shlExpr(L,Exp)` Represents a bitwise left shift of `L` by `Exp`, which may be a local or an integer constant.

`shrExpr(L,Exp)` Represents a bitwise right shift of `L` by `Exp`, which may be a local or an integer constant.

`specialInvokeExpr(M,[P|Ps])` Describes a method invocation of `M` using special dispatch (that is, static dispatch for an instance method).

`staticFieldRef(F)` Represents a static field reference.

`staticInvokeExpr(M,[P|Ps])` Describes a method invocation of `M` using static dispatch.

`stmt(S,SStruct)` Represents a statement of `SStruct` at program counter `S`.

`stmtContainedBy(S,M)` Indicates that `S` is contained in `M`.

`stmt_offset(S, offset(M,V))` Indicates the offset of `S` in `M`.

`stringConstant(Val)` Describes a string constant.

`subExpr(L1,L2)` Represents integer or floating-point subtraction with operands `L1` and `L2`.

`subsig(SSig)` Represents a method subsignature.

`succ(S1,S2)` Indicates that `S2` may follow `S1`.

`thisRef(M,T)` Represents a reference to `this` in `M`.

`throwStmt(L)` Represents a `throw` statement.

`type(void)` Represents `void`.

`unit(S)` Represents a member of the statement domain.

`unitActual(S,I,Exp)` Indicates that the call site at `S` passes `Exp` as parameter `I`; `Exp` can be either an immediate value or a local.

`unitMaySelect(S,M)` Represents an edge in the call-graph

`ushrExpr(L,Exp)` Represents a bitwise unsigned right shift of `L` by `Exp`, which may be a local or an integer constant.

`value_unit(L,S)` Used in ϕ-expressions; indicates that `L` is the appropriate value if control has transfered from `S`.

`virtualInvokeExpr(M,[P|Ps])` Describes a method invocation of `M` using virtual dispatch.

`xorExpr(L1,L2)` Represents bitwise exclusive-OR with operands `L1` and `L2`.

DISCARD THIS PAGE

COLOPHON

This document is set in Adobe Warnock Pro 12 point; the sans-serif face is Adobe Cronos Pro, and the mathematical and symbol font is AMS Euler. It was set by the author on a Macintosh using pdfTeX with the `microtype` package, the `memoir` class, a locally-modified version of the `bringhurst` chapter style, and the `natbib` bibliography package. Prof. John Owens' `otfinst` utility was essential for converting OpenType fonts for use with LaTeX. All figures were created in OmniGraffle, Adobe Illustrator CS3, and R.

REFERENCES

Abelson, H., R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. Adams Iv, D. P. Friedman, E. Kohlbecker, G. L. Steele, Jr., D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, and M. Wand. 1998. Revised^5 report on the algorithmic language scheme. *Higher Order Symbolic Computation* 11(1):7–105.

Aldrich, Jonathan, Valentin Kostadinov, and Craig Chambers. 2002. Alias annotations for program understanding. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 311–330. New York, NY, USA:ACM Press.

Alpern, B., S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, V. Sarkar, and M. Trapp. 2005. The Jikes Research Virtual Machine project: Building an open-source research community. *IBM Systems Journal* 44(2):399–417.

Alpern, Bowen, C. Richard Attanasio, John J. Barton, Anthony Cocchi, Susan Flynn Hummel, Derek Lieber, Ton Ngo, Mark F. Mergen, Janice C. Shepherd, and Stephen Smith. 1999. Implementing jalapeno in Java. In *OOPSLA '99*, 314–324.

Ammons, Glenn, Thomas Ball, and James R. Larus. 1997. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, 85–96. New York, NY, USA:ACM Press.

Andersen, Lars Ole. 1994. Program analysis and specialization for the C programming language. Ph.D. thesis, DIKU, University of Copenhagen.

Andrews, Jeff, and Nick Baker. 2006. Xbox 360 system architecture. *IEEE Micro* 26(2):25–37.

Austin, Todd M., and Gurindar S. Sohi. 1992. Dynamic dependency analysis of ordinary programs. In *ISCA '92: Proceedings of the 19th Annual International Symposium on Computer Architecture*, 342–351. New York, NY, USA:ACM Press.

Baker, Henry G., and Carl Hewitt. 1977. The incremental garbage collection of processes. In *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*, vol. 12.

Ball, Thomas, and James R. Larus. 1992. Optimally profiling and tracing programs. In *ACM Transactions on Programming Languages and Systems*, 59–70.

———. 1996. Efficient path profiling. In *In Proceedings of the 29th Annual International Symposium on Microarchitecture*, 46–57.

Barnett, Michael, and David A. Naumann. 2004. Friends need a bit more: Maintaining invariants over shared state. In *MPC*, ed. Dexter Kozen and Carron Shankland, vol. 3125 of *Lecture Notes in Computer Science*, 54–84. Springer.

Barnett, Mike, Manuel Fändrich, Diego Garbervetsky, and Francesco Logozzo. 2007. Annotations for (more) precise points-to analysis. In *IWACO 2007: ECOOP International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming.*

Barnett, Mike, David A. Naumann, Wolfram Schulte, and Qi Sun. 2004. 99.44% pure: Useful abstractions in specifications. In *In ECOOP Workshop on Formal Techniques for Java-like Programs*, 11–19.

Benton, William C., and Charles N. Fischer. 2007. Interactive, scalable, declarative program analysis: from prototype to implementation. In *PPDP '07: Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, 13–24. New York, NY, USA:ACM Press.

———. 2009. Mostly-functional behavior in java programs. In *VMCAI '09: Proceedings of the Tenth International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer-Verlag.

Berndl, Marc, Ondřej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. 2003. Points-to analysis using BDDs. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, 103–114. ACM Press.

Besson, Frederic, Thomas Jensen, and Fausto Spoto. 2003. Modular class analysis with Datalog. In *10th International Symposium on Static Analysis, LNCS 2694*. Springer-Verlag.

Bierman, Gavin M., and Matthew J. Parkinson. 2003. Effects and effect inference for a core Java calculus. In *Electronic Notes in Theoretical Computer Science*, ed. Viviana Bono and Michele Bugliesi, vol. 82. Elsevier.

Bierman, G.M., M.J. Parkinson, and A.M. Pitts. 2003. MJ: An imperative core calculus for Java and Java with... Tech. Rep., University of Cambridge.

Blackburn, S. M., R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*. New York, NY, USA:ACM Press.

Bollig, Beate, and Ingo Wegener. 1996. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers* 45(9):993–1002.

Boyapati, Chandrasekhar, Robert Lee, and Martin C. Rinard. 2002. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 211–230. New York, NY, USA:ACM Press.

Boyapati, Chandrasekhar, and Martin C. Rinard. 2001. A parameterized type system for race-free Java programs. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN Conference on Object oriented programming, systems, languages, and applications*, 56–69. New York, NY, USA:ACM Press.

Boyapati, Chandrasekhar, Alexandru Salcianu, Jr. William Beebee, and Martin C. Rinard. 2003. Ownership types for safe region-based memory management in real-time Java. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, 324–337. New York, NY, USA:ACM Press.

Bryant, Randal E. 1992. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys* 24(3):293–318.

Caromel, Denis. 1993. Toward a method of object-oriented concurrent programming. *Communications of the ACM* 36(9):90–102.

Chaudhuri, Swarat. 2008. Subcubic algorithms for recursive state machines. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 159–169. New York, NY, USA:ACM Press.

Chen, Michael, and Kunle Olukotun. 1998. Exploiting method-level parallelism in single-threaded Java programs. In *Proceedings of PACT '98*.

———. 2003. The jrpm system for dynamically parallelizing Java programs. In *Proceedings of the 30th International Symposium on Computer Architecture (ISCA-30)*. San Diego, California.

Chen, Weidong, and David S. Warren. 1996. Tabled evaluation with delaying for general logic programs. *Journal of the ACM* 43(1):20–74.

Cherem, Sigmund, and Radu Rugina. 2004. Region analysis and transformation for Java programs. In *Proceedings of the 2004 International Symposium on Memory Management* . Vancouver, Canada.

———. 2006. Compile-time deallocation of individual objects. In *Proceedings of the 2006 International Symposium on Memory Management* . Ottawa, Canada.

———. 2007. A practical escape and effect analysis for building lightweight method summaries. In *16th International Conference on Compiler Construction (CC 2007)* . Braga, Portugal.

Chin, Wei-Ngan, Florin Craciun, Shengchao Qin, and Martin C. Rinard. 2004. Region inference for an object-oriented language. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, 243–254. New York, NY, USA:ACM Press.

Choi, J.-D., M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. 1999. Escape analysis for Java. In *1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA99)*. Denver, CO.

Choi, J.-D., K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. 2002. Efficient and precise datarace detection for multithreaded object-oriented programs. In *In Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation.*

Codish, Michael, Bart Demoen, and Konstantinos Sagonas. 1998. Semantics-based program analysis for logic-based languages using XSB. *Springer International Journal of Software Tools for Technology Transfer* 2(1):29–45.

Corbett, James C. 1994. An empirical evaluation of three methods for deadlock analysis of ada tasking programs. In *ISSTA '94: Proceedings of the 1994 ACM SIGSOFT International symposium on Software testing and analysis*, 204–215. New York, NY, USA:ACM Press.

Costa, Vítor Santos. 2007. Prolog performance on larger datasets. In *Proceedings of the 9th International Symposium on Practical Aspects of Declarative Languages*, 185–199.

Costa, Vitor Santos, Luis Damas, Rogerio Reis, and Ruben Azevedo. 2000. *YAP user's manual*. Universidade do Porto.

Costa, Vítor Santos, Konstantinos F. Sagonas, and Ricardo Lopes. 2007. Demand-driven indexing of prolog clauses. In *Proceedings of the 23rd International Conference Logic Programming*, 395–409.

Dahl, Ole-Johan, Bjørn Myhrhaug, and Kristen Nygaard. 1970. *SIMULA information: Common base language*. S-22, Oslo:Norwegian Computing Center.

Dawson, Steven, C. R. Ramakrishnan, and David S. Warren. 1996. Practical program analysis using general purpose logic programming systems–a case study. In *In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, vol. 31, 117–126. New York, NY, USA:ACM Press.

Diwan, Amer, Kathryn S. McKinley, and J. Eliot B. Moss. 2001. Using types to analyze and optimize object-oriented programs. *Programming Languages and Systems* 23(1):30–72.

Dolby, Julian. 1997. Automatic inline allocation of objects. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, 7–17. New York, NY, USA:ACM Press.

Dolby, Julian, and Andrew Chien. 2000. An automatic object inlining optimization and its evaluation. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, 345–357. New York, NY, USA:ACM Press.

Dolby, Julian, and Andrew A. Chien. 1998. An evaluation of automatic object inline allocation techniques. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1–20. New York, NY, USA:ACM Press.

Engblom, Jakob, Guillaume Girard, and Bengt Werner. 2006. Testing embedded software using simulated hardware. In *Proceedings of ERTS 2006: Embedded Real-Time Software*.

Esparza, Javier, and Andreas Podelski. 2000. Efficient algorithms for pre* and post* on interprocedural parallel flow graphs. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1–11. New York, NY, USA:ACM Press.

Fähndrich, Manuel, and Alexander Aiken. 1997. Program analysis using mixed term and set constraints. In *Static Analysis Symposium*, 114–126.

Flanagan, Cormac, and Shaz Qadeer. 2003. A type and effect system for atomicity. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, 338–349. New York, NY, USA:ACM Press.

Friedman, Daniel P., and David S. Wise. In *Proceedings of the 1976 International Conference on Parallel Processing*, ed. Philip H. Enslow. Detroit, MI.

Ghemawat, Sanjay, Keith H. Randall, and Daniel J. Scales. 2000. Field analysis: getting useful and low-cost interprocedural information. In *ACM SIGPLAN '00 Conference on Programming Language Design and Implementation (PLDI).*, vol. 35, 334–344.

Gosling, James, Bill Joy, Guy Steele, and Gilad Bracha. 2000. *The Java language specification second edition*. Boston, Mass.:Addison-Wesley.

Graham, Susan L., Peter B. Kessler, and Marshall K. Mckusick. 1982. Gprof: A call graph execution profiler. *SIGPLAN Not.* 17(6):120–126.

Greenhouse, Aaron, and John Boyland. 1999. An object-oriented effects system. In *ECOOP '99: Proceedings of the 13th European Conference on Object-Oriented Programming*, 205–229. London, UK:Springer-Verlag.

Grossman, Dan, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. 2002. Region-based memory management in cyclone. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 282–293. New York, NY, USA:ACM Press.

Hall, Robert J. 1992. Call path profiling. In *ICSE '92: Proceedings of the 14th International Conference on Software engineering*, 296–306. New York, NY, USA:ACM Press.

Hall, Robert J., and Aaron J. Goldberg. 1993. Call path profiling of monotonic program resources in UNIX. In *Usenix-stc'93: Proceedings of the USENIX Summer 1993 Technical Conference*, 1–19. Berkeley, CA, USA:USENIX Association.

Halstead, Robert H. 1985. MULTILISP: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 7(4):501–538.

Hanbing Liu, and J Strother Moore. In *Proceedings of 17th International TPHOLS 2004*, ed. Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan, 184–200.

Heintze, Nevin. 1992. Set-based program analysis. Ph.D. thesis, Pittsburgh, PA, USA.

Heintze, Nevin, and Joxan Jaffar. 1994. Set constraints and set-based analysis. In *Principles and Practice of Constraint Programming*, 281–298.

Heintze, Nevin, and Olivier Tardieu. 2001. Ultra-fast aliasing analysis using CLA: a million lines of c code in a second. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, 254–263. New York, NY, USA:ACM Press.

Hendren, Laurie J., C. Donawa, Maryam Emami, Guang R. Gao, Justiani, and B. Sridharan. 1993. Designing the McCAT compiler based on a family of structured intermediate representations. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, 406–420. London, UK:Springer-Verlag.

Henglein, Fritz, Henning Makholm, and Henning Niss. 2005. Effect type systems and region-based memory management. In *Advanced Topics In Types And Programming Languages*, ed. Benjamin C. Pierce, chap. 3, 87–133. The MIT Press.

Hind, Michael. 2001. Pointer analysis: Haven't we solved this problem yet? In *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01).* Snowbird, UT.

Hind, Michael, and Anthony Pioli. 2000. Which pointer analysis should i use? In *International Symposium on Software Testing and Analysis*, 113–123.

Hoare, C. Anthony R. 1974. Monitors: An operating system structuring concept. *Communications of the ACM* 17(10):549–557.

Hollingsworth, Jeffrey K., Barton P. Miller, and Jon Cargille. 1994. Dynamic program instrumentation for scalable performance tools. In *Scalable High-Performance Computing Conference*, 841–850.

Igarashi, Atshushi, Benjamin C. Pierce, and Philip Wadler. 1999. Featherweight Java: a minimal core calculus for Java and GJ. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 132–146. New York, NY, USA:ACM Press.

Kennan, Kent Wheeler. 1987. *Counterpoint: Based on eighteenth-century practice.* 3rd ed. Prentice Hall.

Knight, Tom. 1986. An architecture for mostly functional languages. In *LFP '86: Proceedings of the 1986 ACM Conference on LISP and functional programming*, 105–112. New York, NY, USA:ACM Press.

Kodumal, John, and Alex Aiken. 2004. The set constraint/CFL reachability connection in practice. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, 207–218. New York, NY, USA:ACM Press.

Kodumal, John, and Alexander Aiken. 2005. Banshee: A scalable constraint-based analysis toolkit. In *Proceedings of 12th International Static Analysis Symposium*, 218–234.

Lagoon, Vitaly, and Peter J. Stuckey. 2002. Precise pair-sharing analysis of logic programs. In *PPDP '02: Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, 99–108. New York, NY, USA:ACM Press.

Lam, Monica S., John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. 2005. Context-sensitive program analysis as database queries. In *Proceedings of the Twenty-fourth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. ACM Press.

Laud, Peeter. 2001. Analysis for object inlining in Java. In *In Proceedings of the JOSES Workshop*, 1–8.

Lea, Doug. 2004. Concurrency utilities. Tech. Rep. JSR166, Sun Microsystems.

Leavens, Gary T., Albert L. Baker, and Clyde Ruby. 1998. Preliminary design of JML: a behavioral interface specification language for Java. Tech. Rep. TR98-06, Department of Computer Science, Iowa State University, Ames, Iowa.

———. 2006. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Software Engineering Notes* 31(3):1–38.

Leroy, Xavier. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 42–54. New York, NY, USA:ACM Press.

Lhoták, Ondrej, and Laurie Hendren. 2002. Run-time evaluation of opportunities for object inlining in Java. In *JGI '02: Proceedings of the 2002 joint ACM-ISCOPE Conference on Java Grande*, 175–184. New York, NY, USA:ACM Press.

Lhoták, Ondřej, and Laurie Hendren. 2004. Jedd: A BDD-based relational extension of Java. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*. ACM Press.

Lindholm, Tim, and Frank Yellin. 1999. *Java virtual machine specification*. Addison-Wesley Longman Publishing Co., Inc.

Liu, Yanhong A. 2000. Efficiency by incrementalization: An introduction. *Higher Order Symbolic Computation* 13(4):289–313.

Liu, Yanhong A., and Scott D. Stoller. 2003. From datalog rules to efficient programs with time and space guarantees. In *PPDP '03: Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, 172–183. New York, NY, USA:ACM Press.

Liu, Yanhong A., Scott D. Stoller, and Tim Teitelbaum. 1998. Static caching for incremental computation. *ACM Transactions on Programming Languages and Systems* 20(3):546–585.

Lucas, Jr., Henry. 1971. Performance evaluation and monitoring. *ACM Computing Surveys* 3(3):79–91.

Lucassen, J. M., and D. K. Gifford. 1988. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 47–57. ACM Press.

Luk, Chi-Keung, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay J. Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, vol. 40, 190–200. New York, NY, USA:ACM Press.

Ma, Kin-Keung, and Jeffrey S. Foster. 2007. Inferring aliasing and encapsulation properties for java. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN Conference on Object oriented programming systems and applications*, 423–440. New York, NY, USA:ACM Press.

Makinson, David Clement. 1965. The paradox of the preface. *Analysis* 25(6): 205–207.

Marriott, Kim, and Harald Søndergaard. 1993. Precise and efficient groundness analysis for logic programs. *ACM Letters on Programming Languages and Systems* 2(1-4):181–196.

Melski, David, and Thomas Reps. 1997. Interconvertbility of set constraints and context-free language reachability. In *PEPM '97: Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, 74–89. New York, NY, USA:ACM Press.

Meyer, Bertrand. 1988. *Object Oriented Software Construction*. Computer Science, Prentice Hall.

———. 1993. Systematic concurrent object-oriented programming. *Communications of the ACM* 36:56–80.

Milanova, Ana, Atanas Rountev, and Barbara G. Ryder. 2005. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering Methodology* 14(1):1–41.

Miller, Barton P., Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. 1995. The Paradyn parallel performance measurement tool. *IEEE Computer* 28(11):37–46.

Moreau, Luc. 1996. The semantics of scheme with future. In *International Conference on Functional Programming*, 146–156.

Mousa, Hussam, Chandra Krintz, Lamia Youseff, and Richard Wolski. 2007. Viprof: Vertically integrated full-system performance profiler. In *Proceedings of the 21th International Parallel and Distributed Processing Symposium (IPDPS 2007), 26-30 march 2007, long beach, california, USA*, 1–6. IEEE.

Naik, Mayur, and Alex Aiken. 2007. Conditional must not aliasing for static race detection. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 327–338. New York, NY, USA:ACM Press.

Naik, Mayur, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 308–319. New York, NY, USA:ACM Press.

Nayfeh, Basem A., Lance Hammond, and Kunle Olukotun. 1996. Evaluation of design alternatives for a multiprocessor microprocessor. In *23rd Annual International Symposium on Computer Architecture*. Philadelphia, PA.

Nielson, Flemming, and Hanne Riis Nielson. 1999. Type and effect systems. In *Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the occasion of his retirement from his professorship at the University of Kiel)*, 114–136. London, UK:Springer-Verlag.

Olukotun, Kunle, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. 1996. The case for a single-chip multiprocessor. In *ASPLOS-VII: Proceedings of the seventh International Conference on Architectural support for programming languages and operating systems*, 2–11. New York, NY, USA:ACM Press.

Palsberg, Jens. 2001. Type-based analysis and applications. In *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '01)*, 20–27. Snowbird, UT.

Perlis, Alan J. 1982. Epigrams on programming. *SIGPLAN Notices* 17(9):7–13.

Permandla, Pratibha, Michael Roberson, and Chandrasekhar Boyapati. 2007. A type system for preventing data races and deadlocks in the java virtual machine language: 1. *SIGPLAN Not.* 42(7):10.

Porat, Sara, Marina Biberstein, Larry Koved, and Bilha Mendelson. 2000. Automatic detection of immutable fields in Java. In *CASCON '00: Proceedings of the 2000 Conference of the Centre for Advanced Studies on Collaborative research*, 10. IBM Press.

Pratikakis, Polyvios, Jeffrey S. Foster, and Michael Hicks. 2006. Existential label flow inference via CFL reachability. In *Proceedings of the Static Analysis Symposium (SAS)*.

Pratikakis, Polyvios, Jaime Spacco, and Michael Hicks. 2004. Transparent proxies for Java futures. In *Proceedings of the ACM Conference on Object-Oriented Programming Languages and Systems (OOPSLA)*.

Quinonez, Jaime, Matthew S. Tschantz, and Michael D. Ernst. 2008. Inference of reference immutability. In *ECOOP 2008 — Object-Oriented Programming, 22nd European Conference*, 616–641. Paphos, Cyprus.

Reps, Thomas. 1998. Program analysis via graph reachability. *Information and Software Technology* 40(11–12):701–726.

Reps, Thomas, Stefan Schwoon, Somesh Jha, and David Melski. 2005. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Science of Computer Programming* 58(1-2):206–263.

Reps, Thomas W. 1994. Solving demand versions of interprocedural analysis problems. In *CC '94: Proceedings of the 5th International Conference on Compiler Construction*, 389–403. London, UK:Springer-Verlag.

Rocha, Ricardo, Fernando Silva, and Vítor Santos Costa. 2005. On Applying Or-Parallelism and Tabling to Logic Programs. *Theory and Practice of Logic Programming Systems* 5(1-2):161–205.

Rountev, Atanas, Ana Milanova, and Barbara G. Ryder. 2001. Points-to analysis for Java using annotated constraints. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN Conference on Object oriented programming, systems, languages, and applications*, 43–55. New York, NY, USA:ACM Press.

Saha, Diptikalyan, and C. R. Ramakrishnan. 2005. Incremental and demand-driven points-to analysis using logic programming. In *Principles and Practice of Declarative Programming*. Lisbon, Portugal:ACM Press.

Salcianu, Alexandru, and Martin C. Rinard. Tech. Rep.

Skalka, Christian. 2005. Trace effects and object orientation. In *Proceedings of the ACM Conference on Principles and Practice of Declarative Programming*.

Skalka, Christian, Scott Smith, and David Van Horn. 2005. A type and effect system for flexible abstract interpretation of Java. In *Proceedings of the ACM Workshop on Abstract Interpretation of Object Oriented Languages*. Electronic Notes in Theoretical Computer Science.

Smith, James E., and Gurindar S. Sohi. 1995. The microarchitecture of superscalar processors. *Proceedings of the IEEE* 83(12):1609–1624.

Sridharan, Manu, Denis Gopan, Lexin Shan, and Rastislav Bod&#237;k. 2005. Demand-driven points-to analysis for Java. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications*, 59–76. New York, NY, USA:ACM Press.

Steensgaard, Bjarne. 1996. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 32–41. New York, NY, USA:ACM Press.

Sălcianu, Alexandru, and Martin C. Rinard. 2005. Purity and side effect analysis for Java programs. In *VMCAI*, ed. Radhia Cousot, vol. 3385 of *Lecture Notes in Computer Science*, 199–215. Springer.

Sundaresan, Vijay, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Pat rick Lam, Etienne Gagnon, and Charles Godin. 2000. Practical virtual method call resolution for Java. *ACM SIGPLAN Notices* 35(10):264–280.

Talpin, Jean-Pierre, and Pierre Jouvelot. 1992. Polymorphic type, region and effect inference. *Journal of Functional Programming* 2:245–271.

Tofte, Mads, and Jean-Pierre Talpin. 1997. Region-based memory management. *Information and Computation*.

Tomb, Aaron, and Cormac Flanagan. 2005. Automatic type inference via partial evaluation. In *PPDP '05: Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, 106–116. New York, NY, USA:ACM Press.

Tschantz, Matthew S., and Michael D. Ernst. 2005. Javari: Adding reference immutability to Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2005)*, 211–230. San Diego, CA, USA.

Tullsen, Dean M., Susan J. Eggers, and Henry M. Levy. 1995. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. Santa Margherita Ligure, Italy.

Unkel, Christopher, and Monica S. Lam. 2008. Automatic inference of stationary fields: a generalization of java's final fields. In *POPL*, ed. George C. Necula and Philip Wadler, 183–195. ACM Press.

Vallée-Rai, Raja, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative research*, 13. IBM Press.

Van Roy, Peter. 2008. The challenges and opportunities of multiple processors: Why multi-core processors are easy and internet is hard. In *International Computer Music Conference*.

Vivien, Frédéric, and Martin C. Rinard. 2001. Incrementalized pointer and escape analysis. In *SIGPLAN Conference on Programming Language Design and Implementation*, 35–46.

Warg, F., and P. Stenström. 2001. Limits on speculative module-level parallelism in imperative and object-oriented programs on CMP platforms. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '01)*. IEEE.

Welc, Adam, Suresh Jagannathan, and Antony Hosking. 2005. Safe futures for Java. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN Conference on Object oriented programming, systems, languages, and applications*, 439–453. New York, NY, USA:ACM Press.

West, D. H. D. 1979. Updating mean and variance estimates: an improved method. *Communications of the ACM* 22(9):532–535.

Whaley, John, and Monica S. Lam. 2004. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the Conference on Programming Language Design and Implementation*. ACM Press.

van Wijngarden, Adriaan, Cornelis H. A. Koster B. J. Mailloux, J. E. L. Peck, Michel Sintzoff, C. H. Lindsey, Lambert G. L. T. Meertens, and R. G. Fisker. 1975. Revised report on the algorithmic language ALGOL 68. *Acta Informatica* 5:1–236.

Zhao, Qin, Rodric M. Rabbah, Saman P. Amarasinghe, Larry Rudolph, and Weng-Fai Wong. 2008. How to do a million watchpoints: Efficient debugging using dynamic instrumentation. In *CC*, ed. Laurie J. Hendren, vol. 4959 of *Lecture Notes in Computer Science*, 147–162. Springer.

Zibin, Yoav, Alex Potanin, Mahmood Ali, Shay Artzi, Adam Kieżun, and Michael D. Ernst. 2007. Object and reference immutability using Java generics. In *ESEC/FSE 2007: Proceedings of the 11th European Software Engineering Conference and the 15th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 75–84. Dubrovnik, Croatia.